CKAN documentation

Release 2.11.0a0

CKAN contributors

Mar 13, 2024

CONTENTS

1 User guide **1**
1.1 What is CKAN? . . . 1
1.2 Using CKAN . . . 2

2 Sysadmin guide **17**
2.1 Creating a sysadmin account . . . 17
2.2 Customizing look and feel . . . 18
2.3 Managing organizations and datasets . . . 19
2.4 Permanently deleting datasets, organizations and groups . . . 20
2.5 Managing users . . . 20

3 Maintainer's guide **23**
3.1 CKAN releases . . . 23
3.2 Installing CKAN . . . 24
3.3 Upgrading CKAN . . . 42
3.4 Getting started . . . 47
3.5 Database Management . . . 48
3.6 Command Line Interface (CLI) . . . 50
3.7 Organizations and authorization . . . 61
3.8 Data preview and visualization . . . 63
3.9 FileStore and file uploads . . . 75
3.10 DataStore extension . . . 77
3.11 Apps & Ideas . . . 93
3.12 Tag Vocabularies . . . 93
3.13 Form Integration . . . 94
3.14 Linked Data and RDF . . . 96
3.15 Background jobs . . . 96
3.16 Email notifications . . . 103
3.17 Page View Tracking . . . 104
3.18 Multilingual Extension . . . 107
3.19 Stats Extension . . . 108
3.20 Configuration Options . . . 108

4 API guide **165**
4.1 Legacy APIs . . . 166
4.2 Making an API request . . . 171
4.3 Example: Importing datasets with the CKAN API . . . 172
4.4 API versions . . . 173
4.5 Authentication and API tokens . . . 173
4.6 GET-able API functions . . . 174

4.7 JSONP support . . . 175
4.8 API Examples . . . 175
4.9 Action API reference . . . 176

5 Extending guide **217**
5.1 Writing extensions tutorial . . . 217
5.2 Using custom config settings in extensions . . . 228
5.3 Making configuration options runtime-editable . . . 230
5.4 Testing extensions . . . 234
5.5 Best practices for writing extensions . . . 240
5.6 Customizing dataset and resource metadata fields using IDatasetForm . . . 244
5.7 Plugin interfaces reference . . . 255
5.8 Plugins toolkit reference . . . 282
5.9 Validator functions reference . . . 296
5.10 Internationalizing strings in extensions . . . 307
5.11 Migration from Pylons to Flask . . . 310
5.12 Signals . . . 311
5.13 Customizing the DataStore Data Dictionary Form . . . 314

6 Theming guide **319**
6.1 Customizing CKAN's templates . . . 320
6.2 Adding static files . . . 339
6.3 Customizing CKAN's CSS . . . 341
6.4 Adding CSS and JavaScript files using Webassets . . . 345
6.5 Customizing CKAN's JavaScript . . . 347
6.6 Creating dynamic user interfaces with htmx . . . 361
6.7 Best practices for writing CKAN themes . . . 365
6.8 Custom Jinja2 tags reference . . . 368
6.9 Variables and functions available to templates . . . 368
6.10 Objects and methods available to JavaScript modules . . . 370
6.11 Template helper functions reference . . . 370
6.12 Template snippets reference . . . 382
6.13 JavaScript sandbox reference . . . 383
6.14 JavaScript API client reference . . . 383
6.15 CKAN jQuery plugins reference . . . 383

7 Contributing guide **385**
7.1 Reporting issues . . . 385
7.2 Translating CKAN . . . 386
7.3 Testing CKAN . . . 389
7.4 Writing commit messages . . . 394
7.5 Making a pull request . . . 395
7.6 Reviewing and merging a pull request . . . 396
7.7 Writing documentation . . . 397
7.8 Projects for beginner CKAN developers . . . 408
7.9 CKAN code architecture . . . 408
7.10 CSS coding standards . . . 413
7.11 HTML coding standards . . . 416
7.12 JavaScript coding standards . . . 419
7.13 Python coding standards . . . 426
7.14 String internationalization . . . 433
7.15 Unicode handling . . . 439
7.16 Testing coding standards . . . 442
7.17 Frontend development guidelines . . . 467

7.18 Database migrations . . . 493
7.19 Upgrading CKAN's dependencies . . . 494
7.20 Doing a CKAN release . . . 495

8 Changelog **505**
8.1 v.2.10.4 2024-03-13 . . . 505
8.2 v.2.10.3 2023-12-13 . . . 506
8.3 v.2.10.2 . . . 507
8.4 v.2.10.1 2023-05-24 . . . 507
8.5 v.2.10.0 2023-02-15 . . . 509
8.6 v.2.9.11 2024-03-13 . . . 516
8.7 v.2.9.10 2023-12-13 . . . 517
8.8 v.2.9.9 2023-05-24 . . . 517
8.9 v.2.9.8 2023-02-15 . . . 518
8.10 v.2.9.7 2022-10-26 . . . 518
8.11 v.2.9.6 2022-09-28 . . . 519
8.12 v.2.9.5 2022-01-19 . . . 520
8.13 v.2.9.4 2021-09-22 . . . 521
8.14 v.2.9.3 2021-05-19 . . . 522
8.15 v.2.9.2 2021-02-10 . . . 523
8.16 v.2.9.1 2020-10-21 . . . 524
8.17 v.2.9.0 2020-08-05 . . . 525
8.18 v.2.8.12 2022-10-26 . . . 529
8.19 v.2.8.11 2022-09-28 . . . 529
8.20 v.2.8.10 2022-01-19 . . . 529
8.21 v.2.8.9 2021-09-22 . . . 530
8.22 v.2.8.8 2021-05-19 . . . 530
8.23 v.2.8.7 2021-02-10 . . . 530
8.24 v.2.8.6 2020-10-21 . . . 531
8.25 v.2.8.5 2020-08-05 . . . 531
8.26 v.2.8.4 2020-04-15 . . . 532
8.27 v.2.8.3 2019-07-03 . . . 533
8.28 v.2.8.2 2018-12-12 . . . 534
8.29 v.2.8.1 2018-07-25 . . . 534
8.30 v.2.8.0 2018-05-09 . . . 535
8.31 v.2.7.12 2021-09-22 . . . 538
8.32 v.2.7.11 2021-05-19 . . . 538
8.33 v.2.7.10 2021-02-10 . . . 539
8.34 v.2.7.9 2020-10-21 . . . 539
8.35 v.2.7.8 2020-08-05 . . . 540
8.36 v.2.7.7 2020-04-15 . . . 540
8.37 v.2.7.6 2019-07-03 . . . 541
8.38 v2.7.5 2018-12-12 . . . 542
8.39 v2.7.4 2018-05-09 . . . 542
8.40 v2.7.3 2018-03-15 . . . 542
8.41 v2.7.2 2017-09-28 . . . 543
8.42 v2.7.1 2017-09-27 . . . 543
8.43 v2.7.0 2017-08-02 . . . 544
8.44 v.2.6.9 2020-04-15 . . . 546
8.45 v.2.6.8 2019-07-03 . . . 547
8.46 v2.6.7 2018-12-12 . . . 547
8.47 v2.6.6 2018-05-09 . . . 547
8.48 v2.6.5 2018-03-15 . . . 548
8.49 v2.6.4 2017-09-27 . . . 548

8.50 v2.6.3 2017-08-02 . . . 548
8.51 v2.6.2 2017-03-22 . . . 549
8.52 v2.6.1 2017-02-22 . . . 549
8.53 v2.6.0 2016-11-02 . . . 550
8.54 v2.5.9 2018-05-09 . . . 552
8.55 v2.5.8 2018-03-15 . . . 552
8.56 v2.5.7 2017-09-27 . . . 553
8.57 v2.5.6 2017-08-02 . . . 553
8.58 v2.5.5 2017-03-22 . . . 553
8.59 v2.5.4 2017-02-22 . . . 554
8.60 v2.5.3 2016-11-02 . . . 554
8.61 v2.5.2 2016-03-31 . . . 555
8.62 v2.5.1 2015-12-17 . . . 555
8.63 v2.5.0 2015-12-17 . . . 556
8.64 v2.4.9 2017-09-27 . . . 556
8.65 v2.4.8 2017-08-02 . . . 557
8.66 v2.4.7 2017-03-22 . . . 557
8.67 v2.4.6 2017-02-22 . . . 557
8.68 v2.4.5 2017-02-22 . . . 558
8.69 v2.4.4 2016-11-02 . . . 558
8.70 v2.4.3 2016-03-31 . . . 558
8.71 v2.4.2 2015-12-17 . . . 559
8.72 v2.4.1 2015-09-02 . . . 559
8.73 v2.4.0 2015-07-22 . . . 559
8.74 v2.3.5 2016-11-02 . . . 561
8.75 v2.3.4 2016-03-31 . . . 561
8.76 v2.3.3 2015-12-17 . . . 561
8.77 v2.3.2 2015-09-02 . . . 562
8.78 v2.3.1 2015-07-22 . . . 562
8.79 v2.3 2015-03-04 . . . 562
8.80 v2.2.4 2015-12-17 . . . 568
8.81 v2.2.3 2015-07-22 . . . 568
8.82 v2.2.2 2015-03-04 . . . 569
8.83 v2.2.1 2014-10-15 . . . 569
8.84 v2.2 2014-02-04 . . . 570
8.85 v2.1.6 2015-12-17 . . . 574
8.86 v2.1.5 2015-07-22 . . . 574
8.87 v2.1.4 2015-03-04 . . . 574
8.88 v2.1.3 2014-10-15 . . . 574
8.89 v2.1.2 2014-02-04 . . . 575
8.90 v2.1.1 2013-11-8 . . . 576
8.91 v2.1 2013-08-13 . . . 576
8.92 v2.0.8 2015-12-17 . . . 578
8.93 v2.0.7 2015-07-22 . . . 578
8.94 v2.0.6 2015-03-04 . . . 578
8.95 v2.0.5 2014-10-15 . . . 579
8.96 v2.0.4 2014-02-04 . . . 579
8.97 v2.0.3 2013-11-8 . . . 579
8.98 v2.0.2 2013-08-13 . . . 580
8.99 v2.0.1 2013-06-11 . . . 580
8.100 v2.0 2013-05-10 . . . 581
8.101 v1.8.2 2013-08-13 . . . 583
8.102 v1.8.1 2013-05-10 . . . 584
8.103 v1.8 2012-10-19 . . . 584

8.104 v1.7.4 2013-08-13 . . . 585
8.105 v1.7.3 2013-05-10 . . . 585
8.106 v1.7.2 2012-10-19 . . . 585
8.107 v1.7.1 2012-06-20 . . . 586
8.108 v1.7 2012-05-09 . . . 586
8.109 v1.6 2012-02-24 . . . 587
8.110 v1.5.1 2012-01-04 . . . 588
8.111 v1.5 2011-11-07 . . . 589
8.112 v1.4.3.1 2011-09-30 . . . 590
8.113 v1.4.3 2011-09-13 . . . 591
8.114 v1.4.2 2011-08-05 . . . 591
8.115 v1.4.1 2011-06-27 . . . 592
8.116 v1.4 2011-05-19 . . . 592
8.117 v1.3.3 2011-04-08 . . . 593
8.118 v1.3.2 2011-03-15 . . . 593
8.119 v1.3 2011-02-18 . . . 594
8.120 v1.2 2010-11-25 . . . 594
8.121 v1.1 2010-08-10 . . . 595
8.122 v1.0.2 2010-08-27 . . . 595
8.123 v1.0.1 2010-06-23 . . . 595
8.124 v1.0 2010-05-11 . . . 596
8.125 v0.11 2010-01-25 . . . 596
8.126 v0.10 2009-09-30 . . . 597
8.127 v0.9 2009-07-31 . . . 598
8.128 v0.8 2009-04-10 . . . 598
8.129 v0.7 2008-10-31 . . . 598
8.130 v0.6 2008-07-08 . . . 598
8.131 v0.5 2008-01-22 . . . 599
8.132 v0.4 2007-07-04 . . . 599
8.133 v0.3 2007-04-12 . . . 599
8.134 v0.2 2007-02 . . . 599
8.135 v0.1 2006-05 . . . 600

Python Module Index **601**

Index **603**

CHAPTER

ONE

USER GUIDE

This user guide covers using CKAN's web interface to organize, publish and find data. CKAN also has a powerful API (machine interface), which makes it easy to develop extensions and links with other information systems. The API is documented in *API guide*.

Some web UI features relating to site administration are available only to users with sysadmin status, and are documented in *Sysadmin guide*.

1.1 What is CKAN?

CKAN is a tool for making open data websites. (Think of a content management system like WordPress - but for data, instead of pages and blog posts.) It helps you manage and publish collections of data. It is used by national and local governments, research institutions, and other organizations who collect a lot of data.

Once your data is published, users can use its faceted search features to browse and find the data they need, and preview it using maps, graphs and tables - whether they are developers, journalists, researchers, NGOs, citizens, or even your own staff.

1.1.1 Datasets and resources

For CKAN purposes, data is published in units called "datasets". A dataset is a parcel of data - for example, it could be the crime statistics for a region, the spending figures for a government department, or temperature readings from various weather stations. When users search for data, the search results they see will be individual datasets.

A dataset contains two things:

- Information or "metadata" about the data. For example, the title and publisher, date, what formats it is available in, what license it is released under, etc.
- A number of "resources", which hold the data itself. CKAN does not mind what format the data is in. A resource can be a CSV or Excel spreadsheet, XML file, PDF document, image file, linked data in RDF format, etc. CKAN can store the resource internally, or store it simply as a link, the resource itself being elsewhere on the web. A dataset can contain any number of resources. For example, different resources might contain the data for different years, or they might contain the same data in different formats.

Note: On early CKAN versions, datasets were called "packages" and this name has stuck in some places, specially internally and on API calls. Package has exactly the same meaning as "dataset".

1.1.2 Users, organizations and authorization

CKAN users can register user accounts and log in. Normally (depending on the site setup), login is not needed to search for and find data, but is needed for all publishing functions: datasets can be created, edited, etc by users with the appropriate permissions.

Normally, each dataset is owned by an "organization". A CKAN instance can have any number of organizations. For example, if CKAN is being used as a data portal by a national government, the organizations might be different government departments, each of which publishes data. Each organization can have its own workflow and authorizations, allowing it to manage its own publishing process.

An organization's administrators can add individual users to it, with different roles depending on the level of authorization needed. A user in an organization can create a dataset owned by that organization. In the default setup, this dataset is initially private, and visible only to other users in the same organization. When it is ready for publication, it can be published at the press of a button. This may require a higher authorization level within the organization.

Datasets cannot normally be created except within organizations. It is possible, however, to set up CKAN to allow datasets not owned by any organization. Such datasets can be edited by any logged-in user, creating the possibility of a wiki-like datahub.

Note: The user guide covers all the main features of the web user interface (UI). In practice, depending on how the site is configured, some of these functions may be slightly different or unavailable. For example, in an official CKAN instance in a production setting, the site administrator will probably have made it impossible for users to create new organizations via the UI. You can try out all the features described at http://demo.ckan.org.

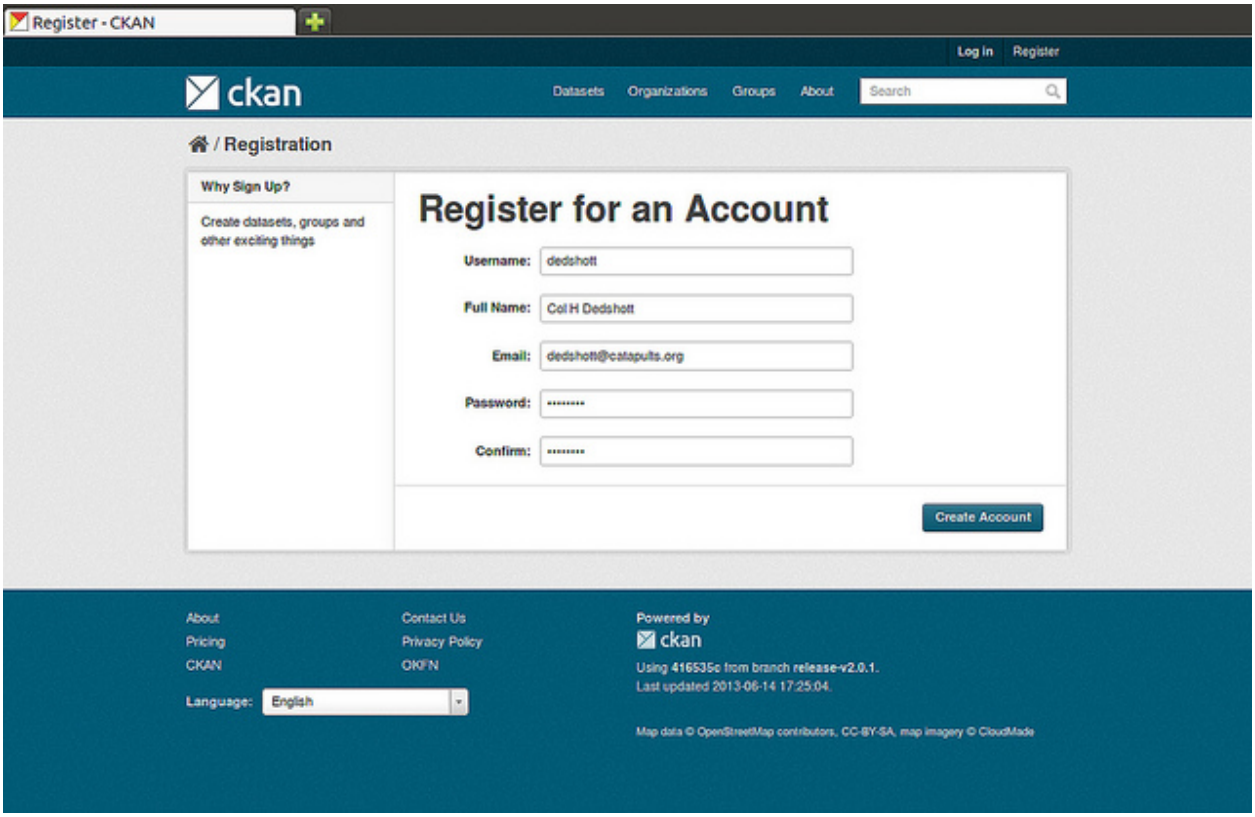
1.2 Using CKAN

1.2.1 Registering and logging in

Note: Registration is needed for most publishing features and for personalization features, such as "following" datasets. It is not needed to search for and download data.

To create a user ID, use the "Register" link at the top of any page. CKAN will ask for the following:

- *Username* – choose a username using only letters, numbers, - and _ characters. For example, "jbloggs" or "joe_bloggs93".
- *Full name* – to be displayed on your user profile
- *E-mail address* – this will not be visible to other users
- *Password* – enter the same password in both boxes



If there are problems with any of the fields, CKAN will tell you the problem and enable you to correct it. When the fields are filled in correctly, CKAN will create your user account and automatically log you in.

Note: It is perfectly possible to have more than one user account attached to the same e-mail address. For this reason, choose a username you will remember, as you will need it when logging in.

1.2.2 Features for publishers

Adding a new dataset

Note: You may need to be a member of an organization in order to add and edit datsets. See the section *Creating an organization* below. On https://demo.ckan.org, you can add a dataset without being in an organization, but dataset features relating to authorization and organizations will not be available.

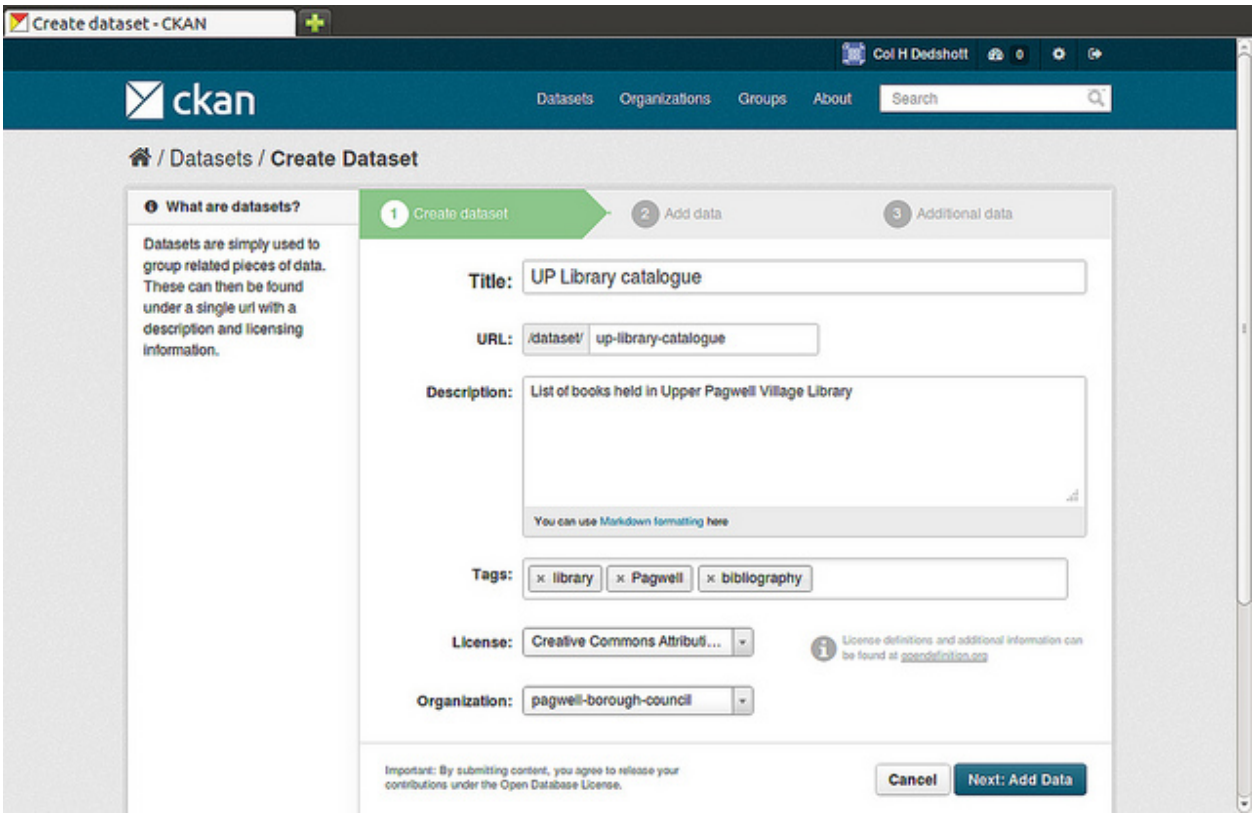
Step 1. You can access CKAN's "Create dataset" screen in two ways.

a) Select the "Datasets" link at the top of any page. From this, above the search box, select the "Add Dataset" button.

b) Alternatively, select the "organizations" link at the top of a page. Now select the page for the organization that should own your new dataset. Provided that you are a member of this organization, you can now select the "Add Dataset" button above the search box.

Step 2. CKAN will ask for the following information about your data. (The actual data will be added in step 4.)

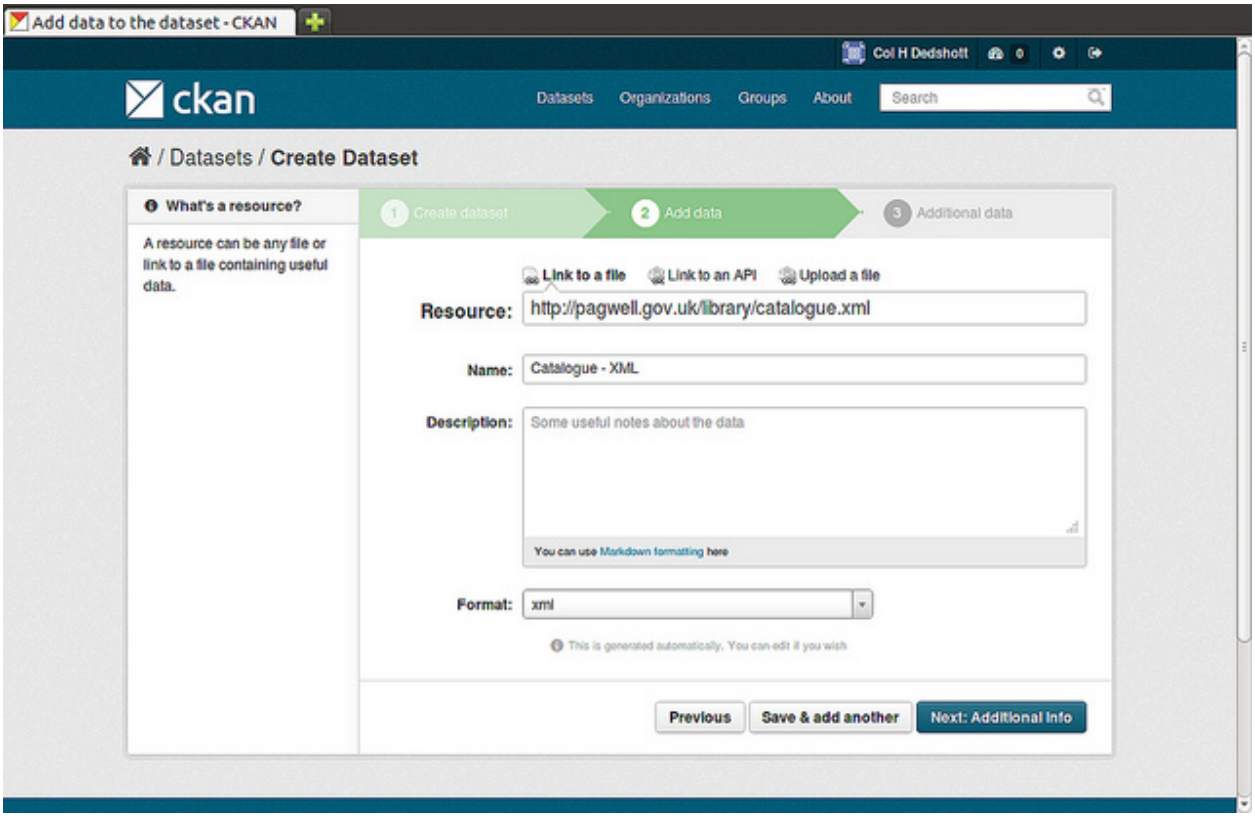
- *Title* – this title will be unique across CKAN, so make it brief but specific. E.g. "UK population density by region" is better than "Population figures".
- *Description* – You can add a longer description of the dataset here, including information such as where the data is from and any information that people will need to know when using the data.
- *Tags* – here you may add tags that will help people find the data and link it with other related data. Examples could be "population", "crime", "East Anglia". Hit the <return> key between tags. If you enter a tag wrongly, you can use its delete button to remove it before saving the dataset.
- *License* – it is important to include license information so that people know how they can use the data. This field should be a drop-down box. If you need to use a license not on the list, contact your site administrator.
- *Organization* - If you are a member of any organizations, this drop-down will enable you to choose which one should own the dataset. Ensure the default chosen is the correct one before you proceed. (Probably most users will be in only one organization. If this is you, CKAN will have chosen your organization by default and you need not do anything.)



Note: By default, the only required field on this page is the title. However, it is good practice to include, at the minimum, a short description and, if possible, the license information. You should ensure that you choose the correct organization for the dataset, since at present, this cannot be changed later. You can edit or add to the other fields later.

Step 3. When you have filled in the information on this page, select the "Next: Add Data" button. (Alternatively select "Cancel" to discard the information filled in.)

Step 4. CKAN will display the "Add data" screen.



This is where you will add one or more "resources" which contain the data for this dataset. Choose a file or link for your data resource and select the appropriate choice at the top of the screen:

- If you are giving CKAN a link to the data, like `http://example.com/mydata.csv`, then select "Link to a file" or "Link to an API". (If you don't know what an API is, you don't need to worry about this option - select "Link to a file".)
- If the data to be added to CKAN is in a file on your computer, select "Upload a file". CKAN will give you a file browser to select it.

Step 5. Add the other information on the page. CKAN does not require this information, but it is good practice to add it:

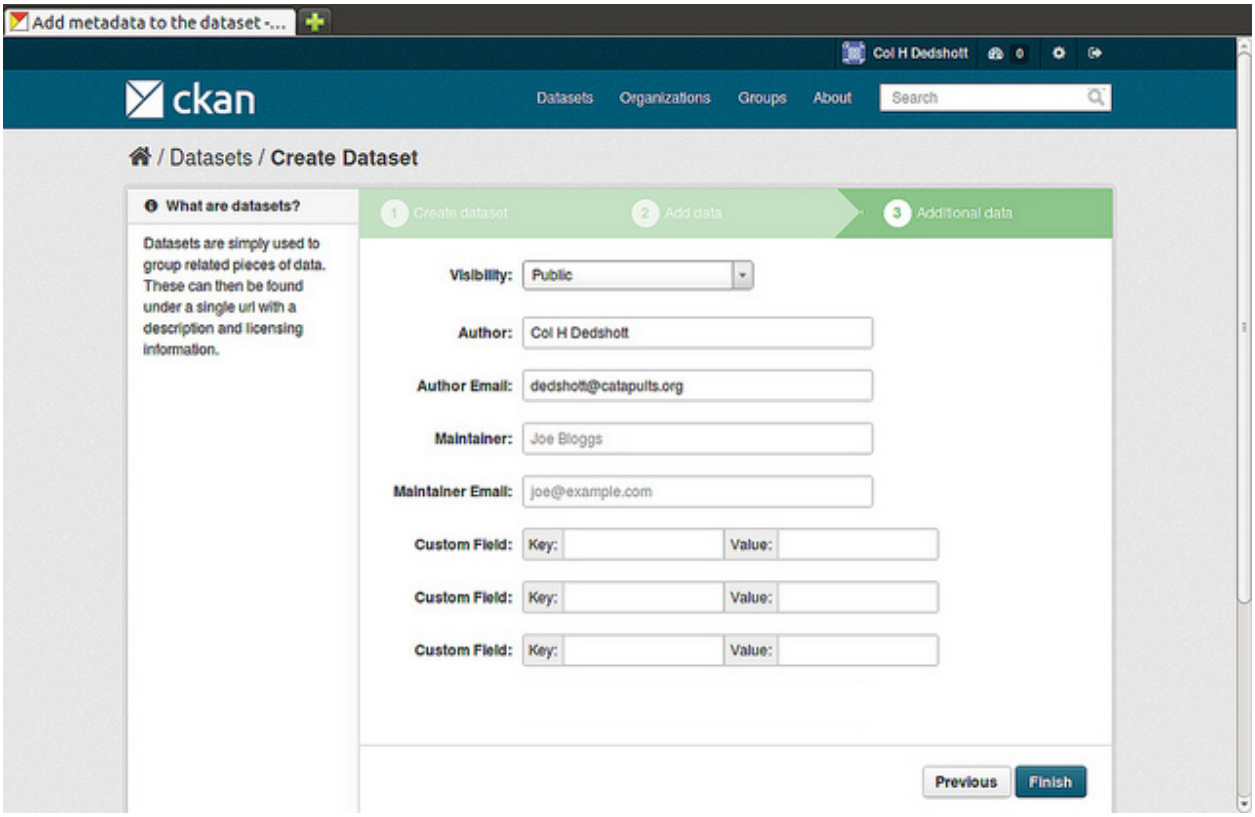
- *Name* – a name for this resource, e.g. "Population density 2011, CSV". Different resources in the dataset should have different names.
- *Description* – a short description of the resource.
- *Format* – the file format of the resource, e.g. CSV (comma-separated values), XLS, JSON, PDF, etc.

Step 6. If you have more resources (files or links) to add to the dataset, select the "Save & add another" button. When you have finished adding resources, select "Next: Additional Info".

Step 7. CKAN displays the "Additional data" screen.

- *Visibility* – a `Public` dataset is public and can be seen by any user of the site. A `Private` dataset can only be seen by members of the organization owning the dataset and will not show up in searches by other users.
- *Author* – The name of the person or organization responsible for producing the data.
- *Author e-mail* – an e-mail address for the author, to which queries about the data should be sent.
- *Maintainer / maintainer e-mail* – If necessary, details for a second person responsible for the data.

- *Custom fields* – If you want the dataset to have another field, you can add the field name and value here. E.g. "Year of publication". Note that if there is an extra field that is needed for a large number of datasets, you should talk to your site administrator about changing the default schema and dataset forms.



Note: Everything on this screen is optional, but you should ensure the "Visibility" is set correctly. It is also good practice to ensure an Author is named.

Changed in version 2.2: Previous versions of CKAN used to allow adding the dataset to existing groups in this step. This was changed. To add a dataset to an existing group now, go to the "Group" tab in the Dataset's page.

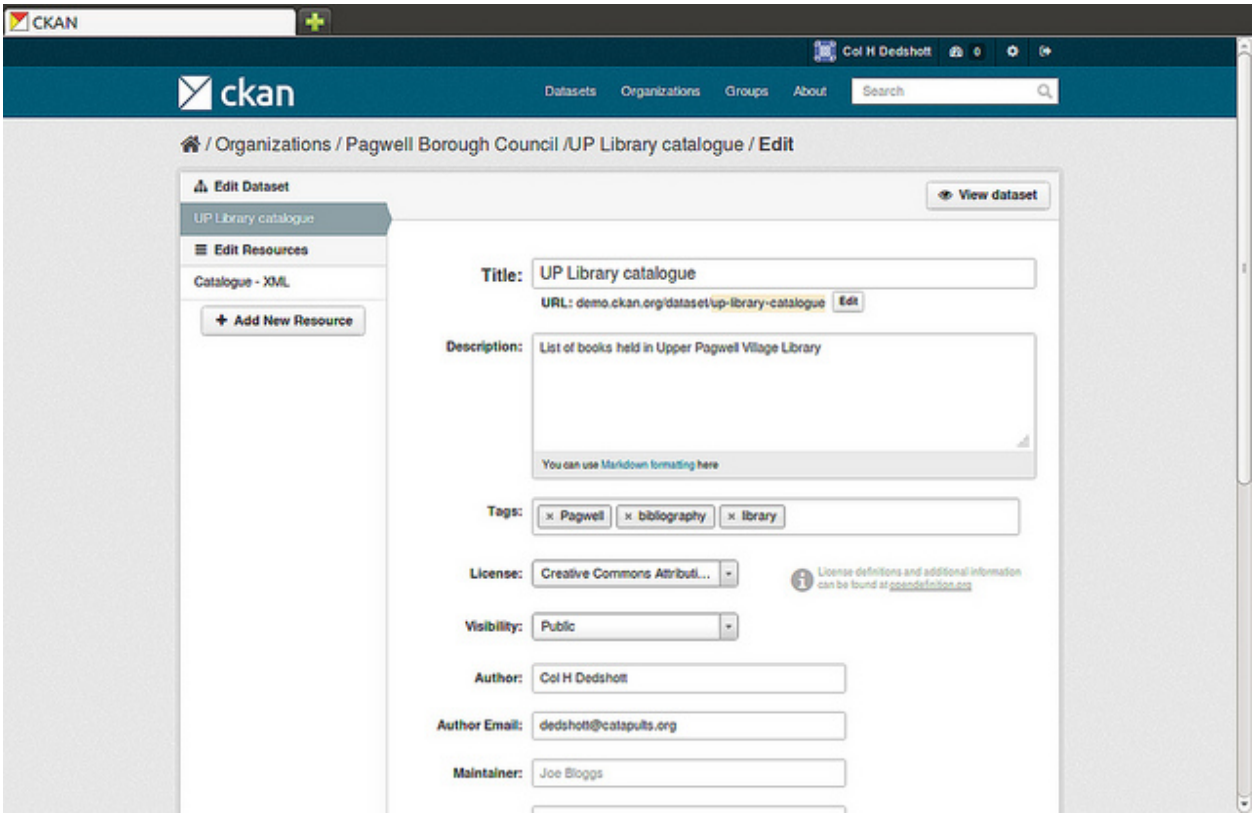
Step 8. Select the 'Finish' button. CKAN creates the dataset and shows you the result. You have finished!

You should be able to find your dataset by typing the title, or some relevant words from the description, into the search box on any page in your CKAN instance. For more information about finding data, see the section *Finding data*.

Editing a dataset

You can edit the dataset you have created, or any dataset owned by an organization that you are a member of. (If a dataset is not owned by any organization, then any registered user can edit it.)

1. Go to the dataset's page. You can find it by entering the title in the search box on any page.
2. Select the "Edit" button, which you should see above the dataset title.
3. CKAN displays the "Edit dataset" screen. You can edit any of the fields (Title, Description, Dataset, etc), change the visibility (Private/Public), and add or delete tags or custom fields. For details of these fields, see *Adding a new dataset*.
4. When you have finished, select the "Update dataset" button to save your changes.



Adding, deleting and editing resources

1. Go to the dataset's "Edit dataset" page (steps 1-2 above).
2. In the left sidebar, there are options for editing resources. You can select an existing resource (to edit or delete it), or select "Add new resource".
3. You can edit the information about the resource or change the linked or uploaded file. For details, see steps 4-5 of "Adding a new resource", above.
4. When you have finished editing, select the button marked "Update resource" (or "Add", for a new resource) to save your changes. Alternatively, to delete the resource, select the "Delete resource" button.

Deleting a dataset

1. Go to the dataset's "Edit dataset" page (see "Editing a dataset", above).
2. Select the "Delete" button.
3. CKAN displays a confirmation dialog box. To complete deletion of the dataset, select "Confirm".

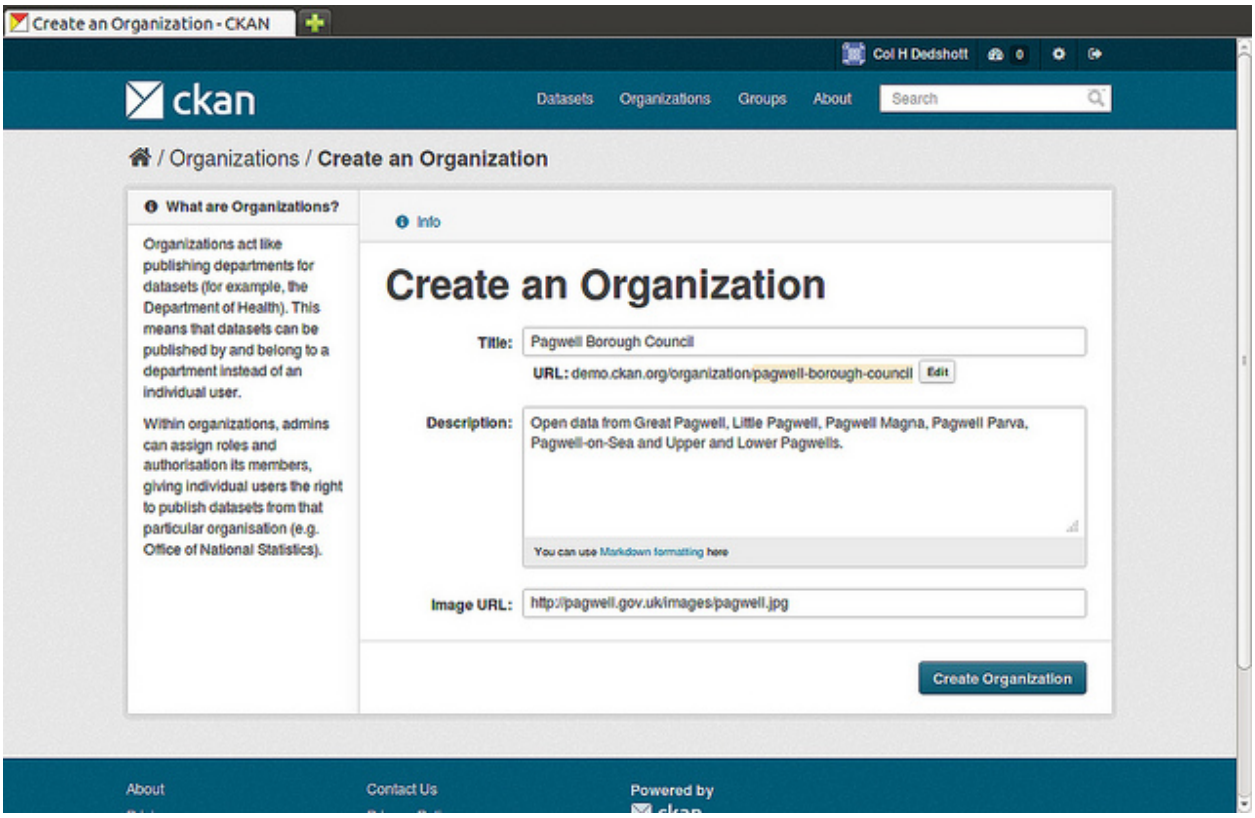
Note: The "Deleted" dataset is not completely deleted. It is hidden, so it does not show up in any searches, etc. However, by visiting the URL for the dataset's page, it can still be seen (by users with appropriate authorization), and "undeleted" if necessary. If it is important to completely delete the dataset, contact your site administrator.

Creating an organization

In general, each dataset is owned by one organization. Each organization includes certain users, who can modify its datasets and create new ones. Different levels of access privileges within an organization can be given to users, e.g. some users might be able to edit datasets but not create new ones, or to create datasets but not publish them. Each organization has a home page, where users can find some information about the organization and search within its datasets. This allows different data publishing departments, bodies, etc to control their own publishing policies.

To create an organization:

1. Select the "Organizations" link at the top of any page.
2. Select the "Add Organization" button below the search box.
3. CKAN displays the "Create an Organization" page.
4. Enter a name for the organization, and, optionally, a description and image URL for the organization's home page.
5. Select the "Create Organization" button. CKAN creates your organization and displays its home page. Initially, of course, the organization has no datasets.



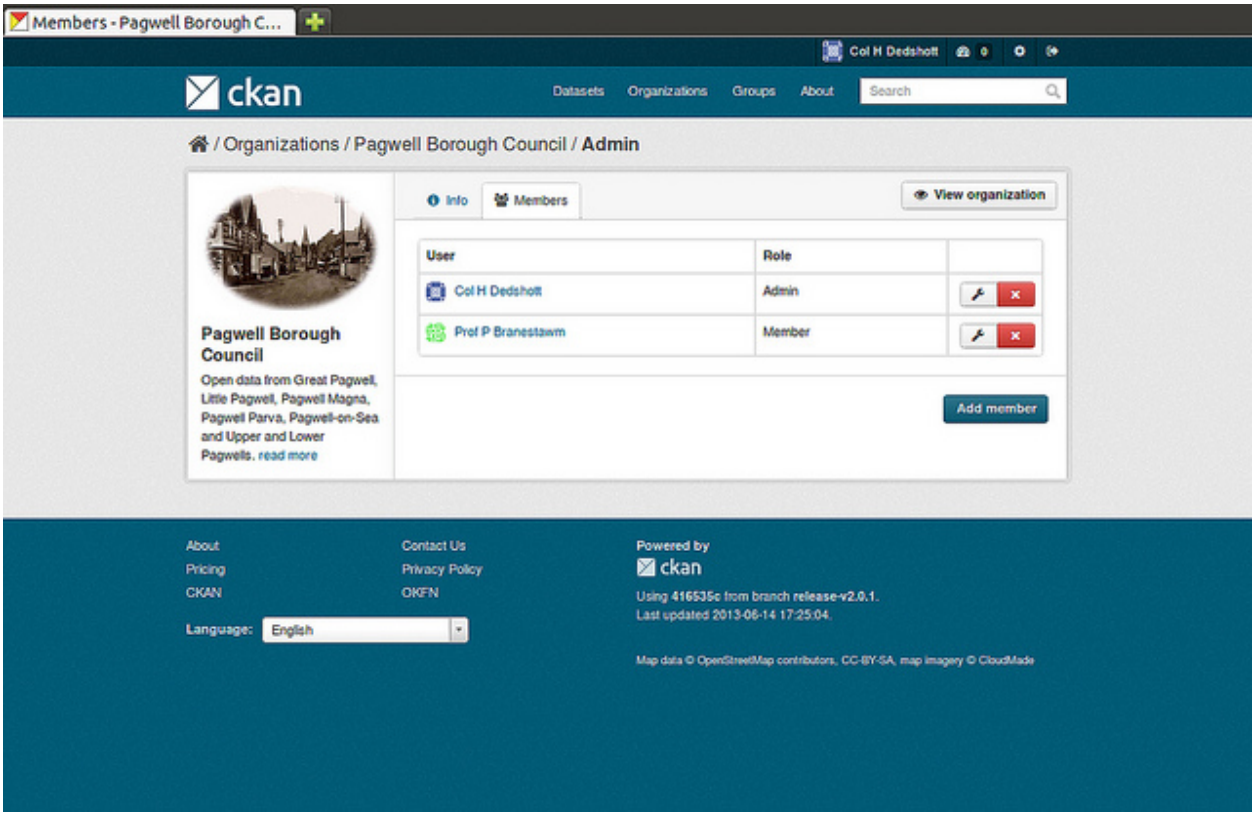
You can now change the access privileges to the organization for other users - see *Managing an organization* below. You can also create datasets owned by the organization; see *Adding a new dataset* above.

Note: Depending on how CKAN is set up, you may not be authorized to create new organizations. In this case, if you need a new organization, you will need to contact your site administrator.

Managing an organization

When you create an organization, CKAN automatically makes you its "Admin". From the organization's page you should see an "Admin" button above the search box. When you select this, CKAN displays the organization admin page. This page has two tabs:

- *Info* – Here you can edit the information supplied when the organization was created (title, description and image).
- *Members* – Here you can add, remove and change access roles for different users in the organization. Note: you will need to know their username on CKAN.



By default CKAN allows members of organizations with three roles:

- *Member* – can see the organization's private datasets
- *Editor* – can edit and publish datasets
- *Admin* – can add, remove and change roles for organization members

1.2.3 Finding data

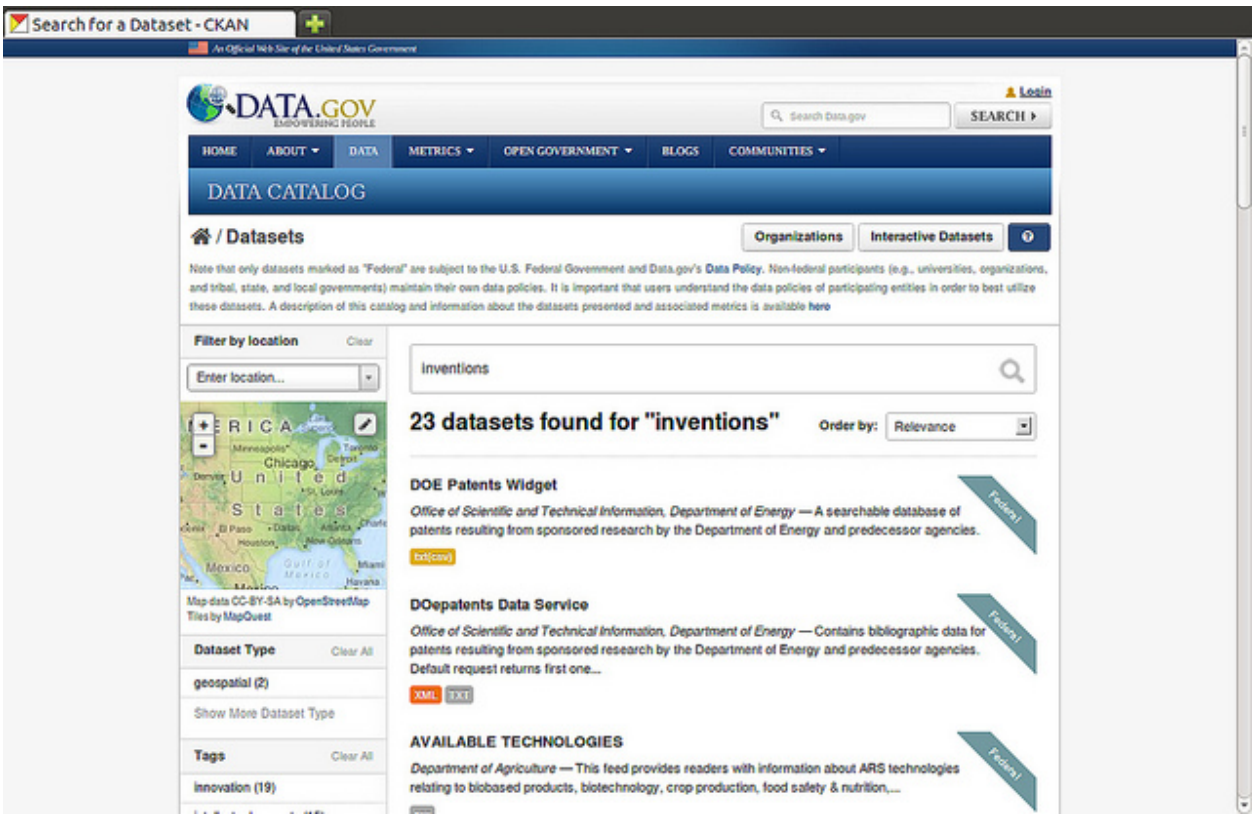
Searching the site

To find datasets in CKAN, type any combination of search words (e.g. "health", "transport", etc) in the search box on any page. CKAN displays the first page of results for your search. You can:

- View more pages of results
- Repeat the search, altering some terms
- Restrict the search to datasets with particular tags, data formats, etc using the filters in the left-hand column

If there are a large number of results, the filters can be very helpful, since you can combine filters, selectively adding and removing them, and modify and repeat the search with existing filters still in place.

If datasets are tagged by geographical area, it is also possible to run CKAN with an extension which allows searching and filtering of datasets by selecting an area on a map.



Searching within an organization

If you want to look for data owned by a particular organization, you can search within that organization from its home page in CKAN.

1. Select the "Organizations" link at the top of any page.
2. Select the organization you are interested in. CKAN will display your organization's home page.
3. Type your search query in the main search box on the page.

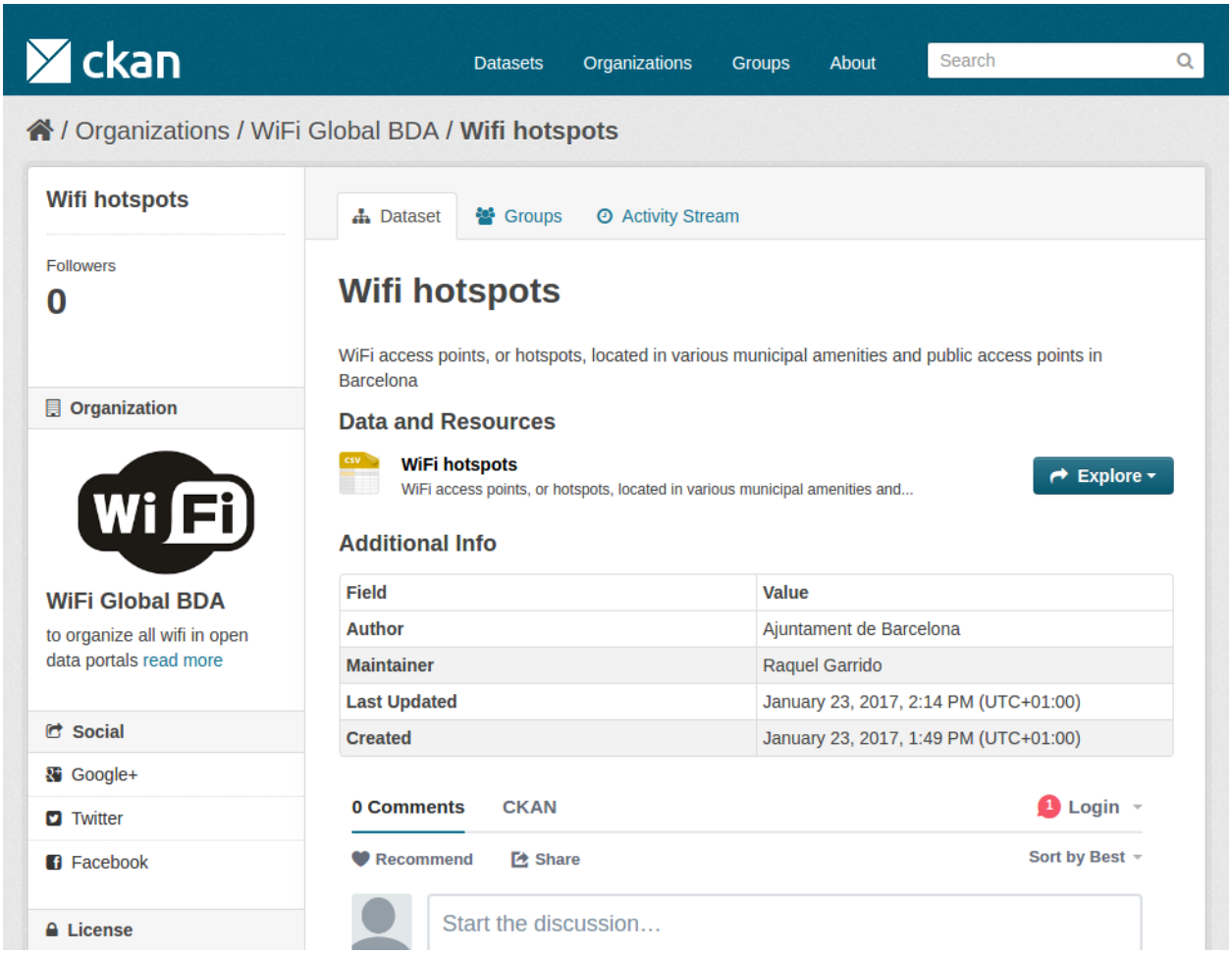
CKAN will return search results as normal, but restricted to datasets from the organization.

If the organization is of interest, you can opt to be notified of changes to it (such as new datasets and modifications to datasets) by using the "Follow" button on the organization page. See the section *Managing your news feed* below. You must have a user account and be logged in to use this feature.

Exploring datasets

When you have found a dataset you are interested and selected it, CKAN will display the dataset page. This includes

- The name, description, and other information about the dataset
- Links to and brief descriptions of each of the resources



The resource descriptions link to a dedicated page for each resource. This resource page includes information about the resource, and enables it to be downloaded. Many types of resource can also be previewed directly on the resource page. .CSV and .XLS spreadsheets are previewed in a grid view, with map and graph views also available if the data is suitable. The resource page will also preview resources if they are common image types, PDF, or HTML.

The dataset page also has two other tabs:

- *Activity stream* – see the history of recent changes to the dataset
- *Groups* – see any group associated with this dataset.

If the dataset is of interest, you can opt to be notified of changes to it by using the "Follow" button on the dataset page. See the section *Managing your news feed* below. You must have a user account and be logged in to use this feature.

1.2.4 Search in detail

CKAN supports two search modes, both are used from the same search field. If the search terms entered into the search field contain no colon (":") CKAN will perform a simple search. If the search expression does contain at least one colon (":") CKAN will perform an advanced search.

Simple Search

CKAN defers most of the search to Solr and by default it uses the DisMax Query Parser that was primarily designed to be easy to use and to accept almost any input without returning an error.

The search words typed by the user in the search box defines the main "query" constituting the essence of the search. The + and - characters are treated as **mandatory** and **prohibited** modifiers for terms. Text wrapped in balanced quote characters (for example, "San Jose") is treated as a phrase. By default, all words or phrases specified by the user are treated as **optional** unless they are preceded by a "+" or a "-".

Note: CKAN will search for the **complete** word and when doing simple search are wildcards are not supported.

Simple search examples:

- `census` will search for all the datasets containing the word "census" in the query fields.
- `census +2019` will search for all the datasets contaning the word "census" and filter only those matching also "2019" as it is treated as mandatory.
- `census -2019` will search for all the datasets containing the word "census" and will exclude "2019" from the results as it is treated as prohibited.
- `"european census"` will search for all the datasets containing the phrase "european census".

Solr applies some preprocessing and stemming when searching. Stemmers remove morphological affixes from words, leaving only the word stem. This may cause, for example, that searching for "testing" or "tested" will show also results containing the word "test".

- `Testing` will search for all the datasets containing the word "Testing" and also "Test" as it is the stem of "Testing".

Note: If the Name of the dataset contains words separated by "-" it will consider each word independently in the search.

Advanced Search

If the query has a colon in it it will be considered a fielded search and the query syntax of Solr will be used to search. This will allow us to use wildcards "*", proximity matching "~" and general features described in Solr docs. The basic syntax is `field:term`.

Advanced Search Examples:

- `title:european` this will look for all the datasets containing in its title the word "european".
- `title:europ*` this will look for all the datasets containing in its title a word that starts with "europ" like "europe" and "european".
- `title:europe || title:africa` will look for datasets containing "europe" or "africa" in its title.

- `title: "european census" ~ 4` A proximity search looks for terms that are within a specific distance from one another. This example will look for datasets which title contains the words "european" and "census" within a distance of 4 words.
- `author:powell~` CKAN supports fuzzy searches based on the Levenshtein Distance, or Edit Distance algorithm. To do a fuzzy search use the "~" symbol at the end of a single-word term. In this example words like "jowell" or "pomell" will also be found.

Note: Field names used in advanced search may differ from Datasets Attributes, the mapping rules are defined in the `schema.xml` file. You can use `title` to search by the dataset name and `text` to look in a catch-all field that includes author, license, mantainer, tags, etc.

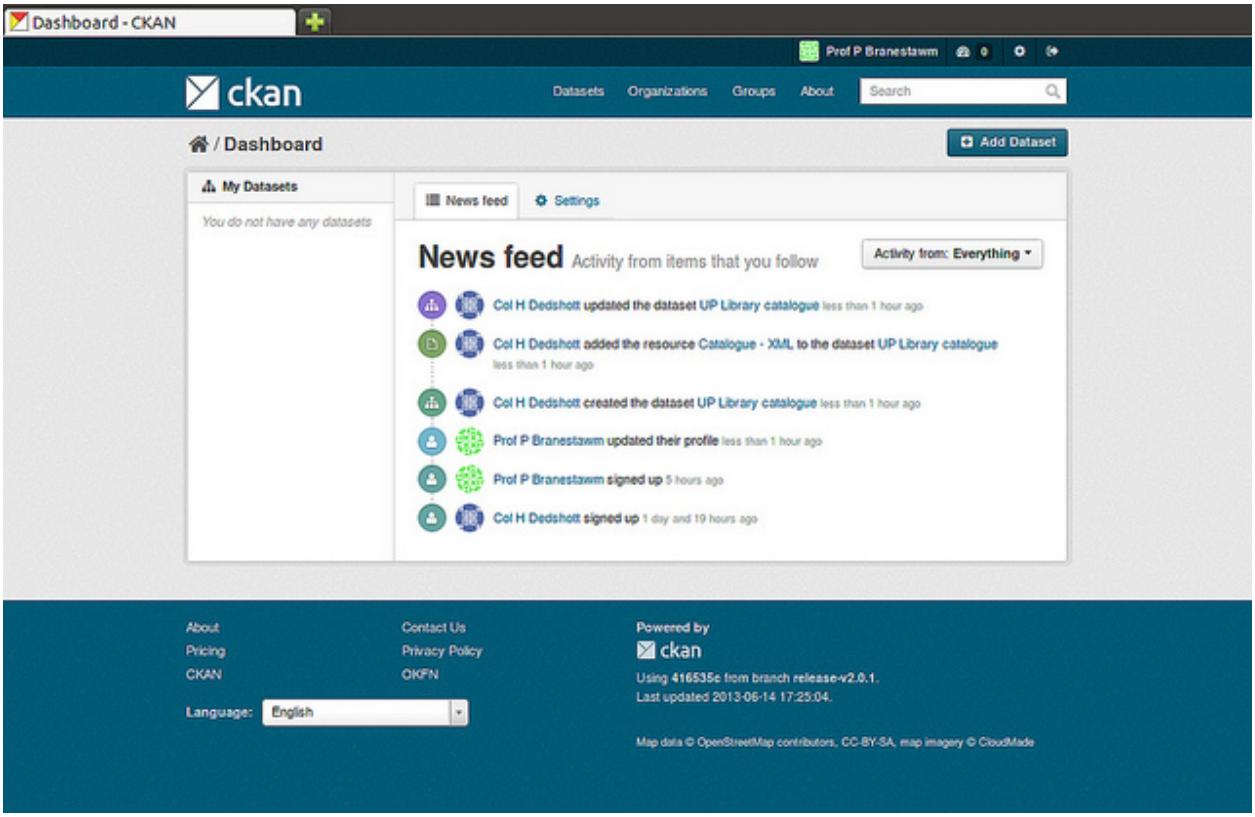
Note: CKAN uses Apache Solr as its search engine. For further details check the Solr documentation. Please note that CKAN sometimes uses different values than what is mentioned in that documentation. Also note that not the whole functionality is offered through the simplified search interface in CKAN or it can differ due to extensions or local development in your CKAN instance.

1.2.5 Personalization

CKAN provides features to personalize the experience of both searching for and publishing data. You must be logged in to use these features.

Managing your news feed

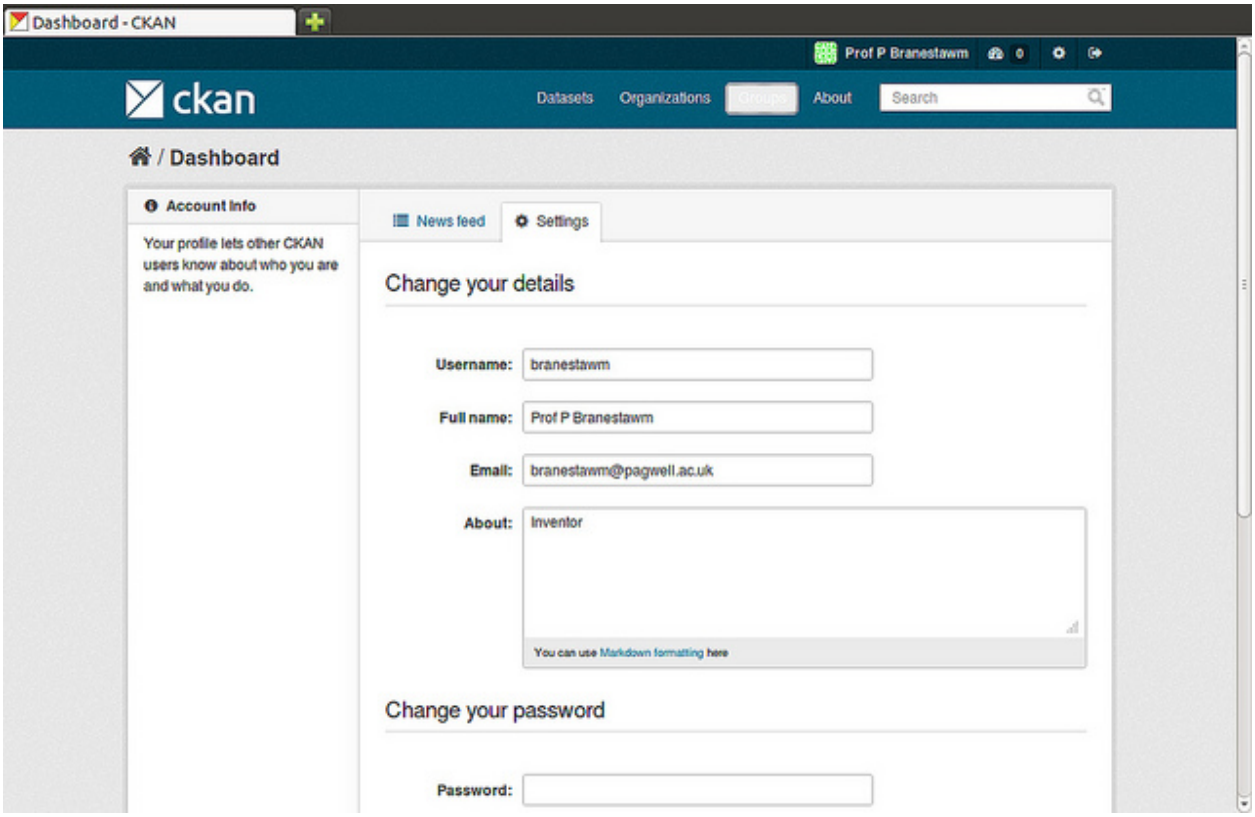
At the top of any page, select the dashboard symbol (next to your name). CKAN displays your News feed. This shows changes to datasets that you follow, and any changed or new datasets in organizations that you follow. The number by the dashboard symbol shows the number of new notifications in your News feed since you last looked at it. As well as datasets and organizations, it is possible to follow individual users (to be notified of changes that they make to datasets).



If you want to stop following a dataset (or organization or user), go to the dataset's page (e.g. by selecting a link to it in your News feed) and select the "Unfollow" button.

Managing your user profile

You can change the information that CKAN holds about you, including what other users see about you by editing your user profile. (Users are most likely to see your profile when you edit a dataset or upload data to an organization that they are following.) To do this, select the gearwheel symbol at the top of any page.



CKAN displays the user settings page. Here you can change:

- Your full name
- Your e-mail address (note: this is not displayed to other users)
- Your profile text - an optional short paragraph about yourself
- Your password

Make the changes you require and then select the "Update Profile" button.

CHAPTER

TWO

SYSADMIN GUIDE

This guide covers the administration features of CKAN 2.0, such as managing users and datasets. These features are available via the web user interface to a user with sysadmin rights. The guide assumes familiarity with the *User guide*.

Certain administration tasks are not available through the web UI but need access to the server where CKAN is installed. These include the range of configuration options using the site's "config" file, documented in *Configuration Options*, and those available via *Command Line Interface (CLI)*.

Warning: A sysadmin user can access and edit any organizations, view and change user details, and permanently delete datasets. You should carefully consider who has access to a sysadmin account on your CKAN system.

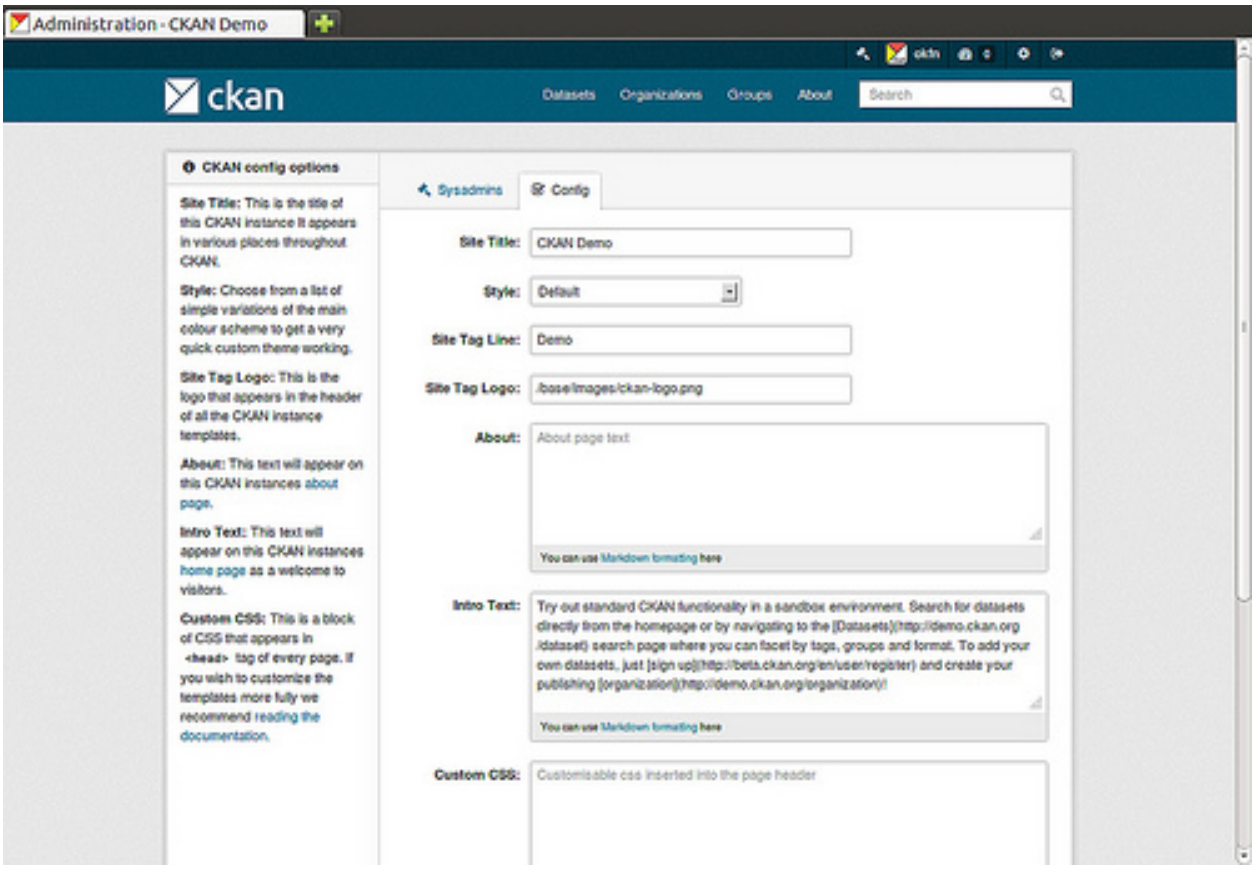
2.1 Creating a sysadmin account

Normally, a sysadmin account is created as part of the process of setting up CKAN. If one does not already exist, you will need to create a sysadmin user, or give sysadmin rights to an existing user. To do this requires access to the server; see *Creating a sysadmin user* for details. If another organization is hosting CKAN, you will need to ask them to create a sysadmin user.

Adding more sysadmin accounts is done in the same way. It cannot be done via the web UI.

2.2 Customizing look and feel

Some simple customizations to customize the 'look and feel' of your CKAN site are available via the UI, at `http://<my-ckan-url>/ckan-admin/config/`.



Here you can edit the following:

Site title
: This title is used in the HTML <title> of pages served by CKAN (which may be displayed on your browser's title bar). For example if your site title is "CKAN Demo", the home page is called "Welcome - CKAN Demo". The site title is also used in a few other places, e.g. in the alt-text of the main site logo.

Style
: Choose one of five colour schemes for the default theme.

Site tag line
: This is not used in CKAN's current default themes, but may be used in future.

Site tag logo
: A URL for the site logo, used at the head of every page of CKAN.

About
: Text that appears on the "about" page, `http://<my-ckan-url>/about`. You can use Markdown here. If it is left empty, a standard text describing CKAN will appear.

Intro text
: This text appears prominently on the home page of your site.

Custom CSS
: For simple style changes, you can add CSS code here which will be added to the `<head>` of every page.

2.3 Managing organizations and datasets

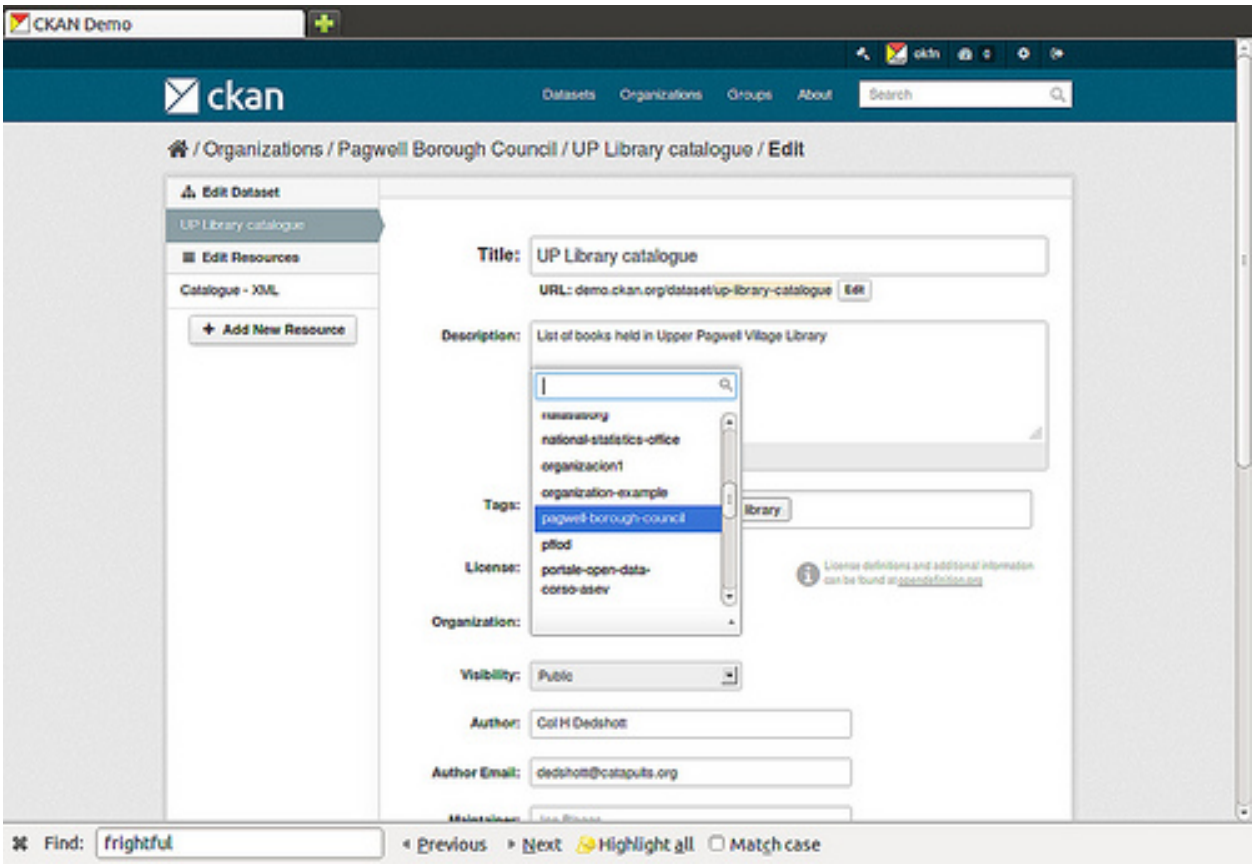
A sysadmin user has full access to user accounts, organizations and datasets. For example, you have access to every organization as if you were a member of that organization. Thus most management operations are done in exactly the same way as in the normal web interface.

For example, to add or delete users to an organization, change a user's role in the organization, delete the organization or edit its description, etc, visit the organization's home page. You will see the 'Admin' button as if you were a member of the organization. You can use this to perform all organization admin functions. For details, see the *User guide*.

Similarly, to edit, update or delete a dataset, go to the dataset page and use the 'Edit' button. As an admin user you can see all datasets including those that are private to an organization. They will show up when doing a dataset search.

2.3.1 Moving a dataset between organizations

To move a dataset between organizations, visit the dataset's Edit page. Choose the appropriate entry from the "organization" drop-down list, and press "Save".



2.4 Permanently deleting datasets, organizations and groups

A dataset, organization or group which has been deleted is not permanently removed from CKAN; it is simply marked as 'deleted' and will no longer show up in search, etc. The assigned URL cannot be re-used for a new entity.

To permanently delete ("purge") an entity:

- Navigate to the dataset's "Edit" page, and delete it.
- Visit `http://<my-ckan-url>/ckan-admin/trash/`.

This page shows all deleted datasets, organizations and groups and allows you to delete them permanently.

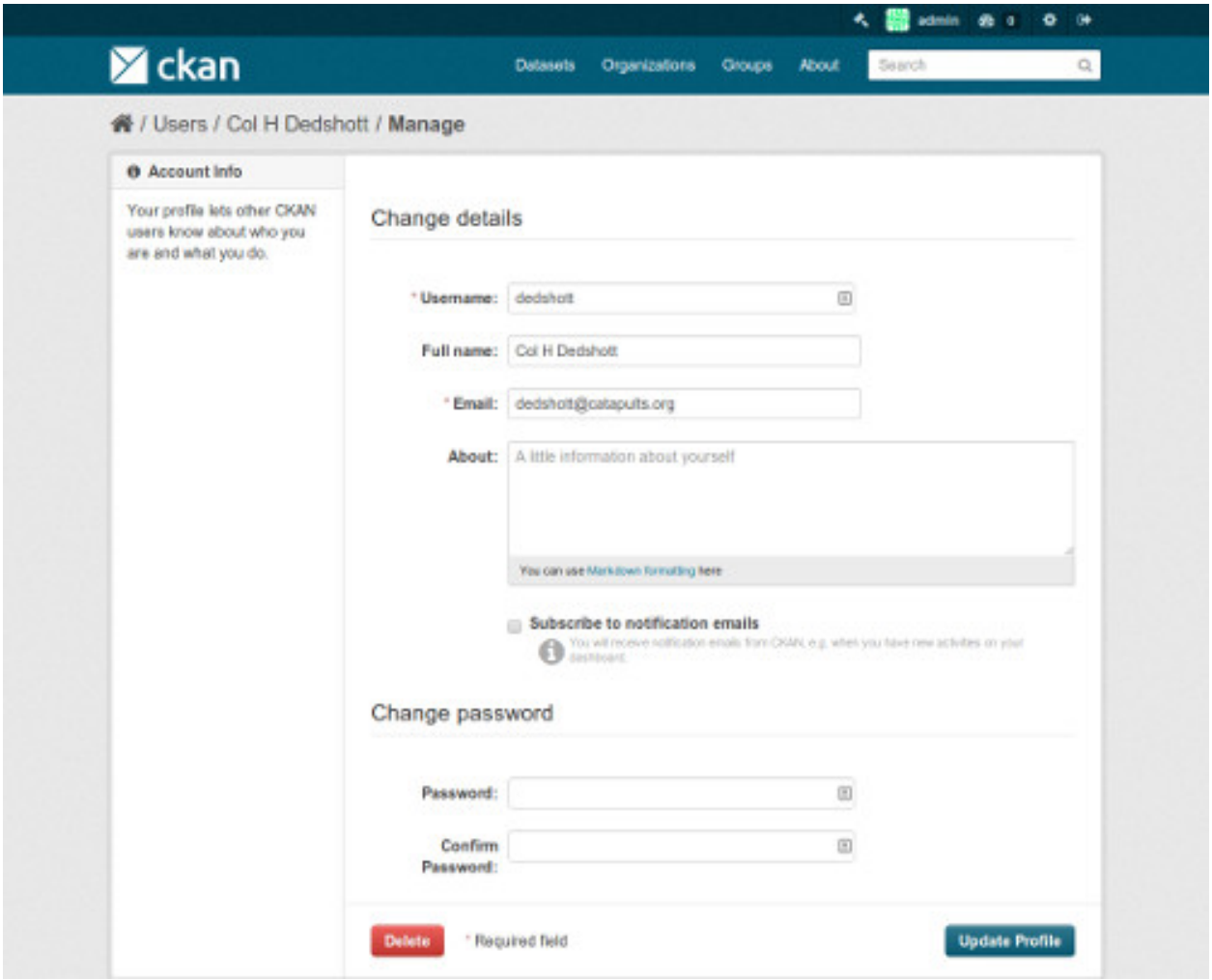
Warning: This operation cannot be reversed!

2.5 Managing users

To find a user's profile, go to `http://<my-ckan-url>/user/`. You can search for users in the search box provided.

You can search by any part of the user profile, including their e-mail address. This is useful if, for example, a user has forgotten their user ID. For non-sysadmin users, the search on this page will only match public parts of the profile, so they cannot search by e-mail address.

On their user profile, you will see a "Manage" button. CKAN displays the user settings page. You can delete the user or change any of its settings, including their name and password.



New in version 2.2: Previous versions of CKAN didn't allow you to delete users through the web interface.

CHAPTER

THREE

MAINTAINER'S GUIDE

The sections below document how to setup and maintain a CKAN site, including installing, upgrading and configuring CKAN and its features and extensions.

3.1 CKAN releases

This document describes the different types of CKAN releases, and explains which releases are officially supported at any given time.

Note: The currently supported CKAN version is **CKAN 2.10.4**

Security and performance fixes are also provided for **CKAN 2.9.11**.

Read more about *officially supported versions*

3.1.1 Release types

CKAN follows a predictable release cycle so that users can depend on stable releases of CKAN, and can plan their upgrades to new releases.

Each release has a version number of the form `M.m` (eg. 2.1) or `M.m.p` (eg. 1.8.2), where `M` is the **major version**, `m` is the **minor version** and `p` is the **patch version** number. There are three types of release:

Major Releases
Major releases, such as CKAN 1.0 and CKAN 2.0, increment the major version number. These releases contain major changes in the CKAN code base, with significant refactorings and breaking changes, for instance in the API or the templates. These releases are very infrequent.

Minor Releases
Minor releases, such as CKAN 2.9 and CKAN 2.10, increment the minor version number. These releases are not as disruptive as major releases, but they *may* include some backwards-incompatible changes. The *Changelog* will document any breaking changes. We aim to release a minor version of CKAN roughly twice a year.

Patch Releases
Patch releases, such as CKAN 2.9.5 or CKAN 2.10.1, increment the patch version number. These releases do not break backwards-compatibility, they include only bug fixes for security and performance issues. Patch releases do not contain:

- Database schema changes or migrations (unless addressing security issues)
- Solr schema changes

- Function interface changes
- Plugin interface changes
- New dependencies (unless addressing security issues)
- Big refactorings or new features in critical parts of the code

Note: Outdated patch releases will no longer be supported after a newer patch release has been released. For example once CKAN 2.9.2 has been released, CKAN 2.9.1 will no longer be supported.

Releases are announced on the ckan-announce mailing list, a low-volume list that CKAN instance maintainers can subscribe to in order to be up to date with upcoming releases.

3.1.2 Supported versions

At any one time, the CKAN Tech Team will support the latest patch release of the last released minor version plus the last patch release of the previous minor version.

The previous minor version will only receive security and bug fixes. If a patch does not clearly fit in these categories, it is up to the maintainers to decide if it can be backported to a previous version.

The latest patch releases are the only ones officially supported. Users should always run the latest patch release for the minor release they are on, as they contain important bug fixes and security updates. Running CKAN in an unsupported version puts your site and data at risk.

Because patch releases don't include backwards incompatible changes, the upgrade process (as described in *Upgrading a CKAN 2 package install to a new patch release*) should be straightforward.

Extension maintainers can decide at their discretion to support older CKAN versions.

See also:

Changelog
The changelog lists all CKAN releases and the main changes introduced in each release.

Doing a CKAN release
Documentation of the process that the CKAN developers follow to do a CKAN release.

3.2 Installing CKAN

Note: The currently supported CKAN version is **CKAN 2.10.4**

Security and performance fixes are also provided for **CKAN 2.9.11**.

Read more about *officially supported versions*

CKAN 2.10 supports Python 3.7 to Python 3.10.

Before you can use CKAN on your own computer, you need to install it. There are three ways to install CKAN:

1. Install from an operating system package
2. Install from source
3. Install from Docker Compose

Additional deployment tips can be found in our wiki, such as the recommended Hardware Requirements.

3.2.1 Package install

Installing from package is the quickest and easiest way to install CKAN, but it requires Ubuntu 20.04 64-bit or Ubuntu 22.04 64-bit.

You should install CKAN from package if:

- You want to install CKAN on an Ubuntu 20.04 or 22.04, 64-bit server, *and*
- You only want to run one CKAN website per server

See *Installing CKAN from package*.

3.2.2 Source install

You should install CKAN from source if:

- You want to install CKAN on a 32-bit computer, *or*
- You want to install CKAN on a different version of Ubuntu, not 20.04 or 22.04, *or*
- You want to install CKAN on another operating system (eg. RHEL, CentOS, OS X), *or*
- You want to run multiple CKAN websites on the same server, *or*
- You want to install CKAN for development

See *Installing CKAN from source*.

3.2.3 Docker Compose install

The ckan-docker repository contains the necessary scripts and images to install CKAN using Docker Compose. It provides a clean and quick way to deploy a standard CKAN instance pre-configured with the *Filestore* and *DataStore extension*. It also allows the addition (and customization) of extensions. The emphasis leans more towards a Development environment, however the base install can be used as the foundation for progressing to a Production environment. Please note that a fully-fledged CKAN Production system using Docker containers is beyond the scope of the provided setup.

You should install CKAN from Docker Compose if:

- You want to install CKAN with less effort than a source install and more flexibility than a package install, **or**
- You want to run or even develop extensions with the minimum setup effort, **or**
- You want to see whether and how CKAN, Docker and your respective infrastructure will fit together.

To install CKAN using Docker Compose, follow the links below:

- Configuration and setup files to run a CKAN site.
- Official Docker images for CKAN.

If you've already setup a CKAN website and want to upgrade it to a newer version of CKAN, see *Upgrading CKAN*.

Installing CKAN from package

This section describes how to install CKAN from package. This is the quickest and easiest way to install CKAN, but it requires **Ubuntu 20.04 or 22.04 64-bit**. If you're not using any of these Ubuntu versions, or if you're installing CKAN for development, you should follow *Installing CKAN from source* instead.

At the end of the installation process you will end up with two running web applications, CKAN itself and the DataPusher, a separate service for automatically importing data to CKAN's *DataStore extension*. Additionally, there will be a process running the worker for running *Background jobs*. All these processes will be managed by Supervisor.

For Python 3 installations, the minimum Python version required is 3.9.

Host ports requirements:

Service	Port	Used for
NGINX	80	Proxy
uWSGI	8080	Web Server
uWSGI	8800	DataPusher
Solr	8983	Search
PostgreSQL	5432	Database
Redis	6379	Search

1. Install the CKAN package

On your Ubuntu system, open a terminal and run these commands to install CKAN:

1. Update Ubuntu's package index:

```
sudo apt update
```

2. Install the Ubuntu packages that CKAN requires (and 'git', to enable you to install CKAN extensions):

```
sudo apt install -y libpq5 redis-server nginx supervisor
```

3. Download the CKAN package:

- On Ubuntu 20.04:

```
wget https://packaging.ckan.org/python-ckan_2.10-focal_amd64.deb
```

- On Ubuntu 22.04:

```
wget https://packaging.ckan.org/python-ckan_2.10-jammy_amd64.deb
```

1. Install the CKAN package:

- On Ubuntu 20.04:

```
sudo dpkg -i python-ckan_2.10-focal_amd64.deb
```

- On Ubuntu 22.04:

```
sudo dpkg -i python-ckan_2.10-jammy_amd64.deb
```

2. Install and configure PostgreSQL

Tip: You can install PostgreSQL and CKAN on different servers. Just change the *sqlalchemy.url* setting in your /etc/ckan/default/ckan.ini file to reference your PostgreSQL server.

Note: The commands mentioned below are tested in Ubuntu

orphan

Install PostgreSQL required packages:

```
sudo apt install -y postgresql
```

Note: If you are facing a problem in case postgresql is not running, execute the command `sudo service postgresql start`

Check that PostgreSQL was installed correctly by listing the existing databases:

```
sudo -u postgres psql -l
```

Check that the encoding of databases is `UTF8`, if not you might find issues later on with internationalisation. Since changing the encoding of PostgreSQL may mean deleting existing databases, it is suggested that this is fixed before continuing with the CKAN install.

Next you'll need to create a database user if one doesn't already exist. Create a new PostgreSQL user called ckan_default, and enter a password for the user when prompted. You'll need this password later:

```
sudo -u postgres createuser -S -D -R -P ckan_default
```

Create a new PostgreSQL database, called ckan_default, owned by the database user you just created:

```
sudo -u postgres createdb -O ckan_default ckan_default -E utf-8
```

Note: If PostgreSQL is run on a separate server, you will need to edit *postgresql.conf* and *pg_hba.conf*. On Ubuntu, these files are located in *etc/postgresql/{Postgres version}/main*.

Uncomment the *listen_addresses* parameter and specify a comma-separated list of IP addresses of the network interfaces PostgreSQL should listen on or '*' to listen on all interfaces. For example,

```
listen_addresses = 'localhost,192.168.1.21'
```

Add a line similar to the line below to the bottom of *pg_hba.conf* to allow the machine running the web server to connect to PostgreSQL. Please change the IP address as desired according to your network settings.

```
host all all 192.168.1.22/32 md5
```

Edit the *sqlalchemy.url* option in your *CKAN configuration file* (/etc/ckan/default/ckan.ini) file and set the correct password, database and database user.

3. Install and configure Solr

Tip: You can install Solr and CKAN on different servers. Just change the *solr_url* setting in your /etc/ckan/default/ckan.ini /etc/ckan/default/production.ini file to reference your Solr server.

orphan

CKAN uses Solr as its search engine, and uses a customized Solr schema file that takes into account CKAN's specific search needs. Now that we have CKAN installed, we need to install and configure Solr.

Warning: CKAN supports **Solr 9** (recommended version) and Solr 8. Starting from CKAN 2.10 these are the only Solr version supported. CKAN 2.9 can run with Solr 9 and 8 as long as it is patched to at least 2.9.5.

There are two supported ways to install Solr.

1. Using CKAN's official Docker images. This is generally the easiest one and the recommended one if you are developing CKAN locally
2. Installing Solr locally and configuring it with the CKAN schema. You can use this option if you can't or don't want to use Docker.

Installing Solr using Docker

You will need to have Docker installed. Please refer to its installation documentation for details.

There are pre-configured Docker images for Solr for each CKAN version. Make sure to pick the image tag that matches your CKAN version (they are named `ckan/ckan-solr:<Major version>.<Minor version>`). To start a local Solr service you can run:

```
docker run --name ckan-solr -p 8983:8983 -d ckan/ckan-solr:2.10-solr9
```

You can now jump to the *Next steps* section.

Installing Solr manually

The following instructions have been tested in Ubuntu 22.04 and are provided as a guidance only. For a Solr production setup is it recommended that you follow the official Solr documentation.

1. Install the OS dependencies:

   ```
   sudo apt-get install openjdk-11-jdk
   ```

2. Download the latest supported version from the Solr downloads page. CKAN supports Solr version 9.x (recommended) and 8.x.
3. Extract the install script file to your desired location (adjust the Solr version number to the one you are using):

   ```
   tar xzf solr-9.2.1.tgz solr-9.2.1/bin/install_solr_service.sh --strip-components=2
   ```

4. Run the installation script as `root`:

```
sudo bash ./install_solr_service.sh solr-9.2.1.tgz
```

5. Check that Solr started running:

```
sudo service solr status
```

6. Create a new core for CKAN:

```
sudo -u solr /opt/solr/bin/solr create -c ckan
```

7. Replace the standard schema with the CKAN one:

```
sudo -u solr wget -O /var/solr/data/ckan/conf/managed-schema https://raw.
↪githubusercontent.com/ckan/ckan/dev-v2.10/ckan/config/solr/schema.xml
```

8. Restart Solr:

```
sudo service solr restart
```

Next steps with Solr

To check that Solr started you can visit the web interface at http://localhost:8983/solr

> **Warning:** The two installation methods above will leave you with a setup that is fine for local development, but Solr should never be exposed publicly in a production site. Pleaser refer to the Solr documentation to learn how to secure your Solr instance.

If you followed any of the instructions above, the CKAN Solr core will be available at http://localhost:8983/solr/ckan. If for whatever reason you ended up with a different one (eg with a different port, host or core name), you need to change the *solr_url* setting in your *CKAN configuration file* (/etc/ckan/default/ckan.ini) to point to your Solr server, for example:

```
solr_url=http://my-solr-host:8080/solr/ckan-2.10
```

4. Set up a writable directory

CKAN needs a directory where it can write certain files, regardless of whether you are using the *FileStore and file uploads* or not (if you do want to enable file uploads, set the *ckan.storage_path* configuration option in the next section).

1. Create the directory where CKAN will be able to write files:

   ```
   sudo mkdir -p /var/lib/ckan/default
   ```

2. Set the permissions of this directory. For example if you're running CKAN with Nginx, then the Nginx's user (`www-data` on Ubuntu) must have read, write and execute permissions on it:

   ```
   sudo chown www-data /var/lib/ckan/default
   sudo chmod u+rwx /var/lib/ckan/default
   ```

5. Update the configuration and initialize the database

1. Edit the *CKAN configuration file* (/etc/ckan/default/ckan.ini) to set up the following options:

   site_id
   Each CKAN site should have a unique `site_id`, for example:

   ```
   ckan.site_id = default
   ```

   site_url
   Provide the site's URL. For example:

   ```
   ckan.site_url = http://demo.ckan.org
   ```

2. Initialize your CKAN database by running this command in a terminal:

   ```
   sudo ckan db init
   ```

3. Optionally, setup the DataStore and DataPusher by following the instructions in *DataStore extension*.
4. Also optionally, you can enable file uploads by following the instructions in *FileStore and file uploads*.

6. Start the Web Server and restart Nginx

Reload the Supervisor daemon so the new processes are picked up:

```
sudo supervisorctl reload
```

After a few seconds run the following command to check the status of the processes:

```
sudo supervisorctl status
```

You should see three processes running without errors:

```
ckan-datapusher:ckan-datapusher-00   RUNNING   pid 1963, uptime 0:00:12
ckan-uwsgi:ckan-uwsgi-00             RUNNING   pid 1964, uptime 0:00:12
ckan-worker:ckan-worker-00           RUNNING   pid 1965, uptime 0:00:12
```

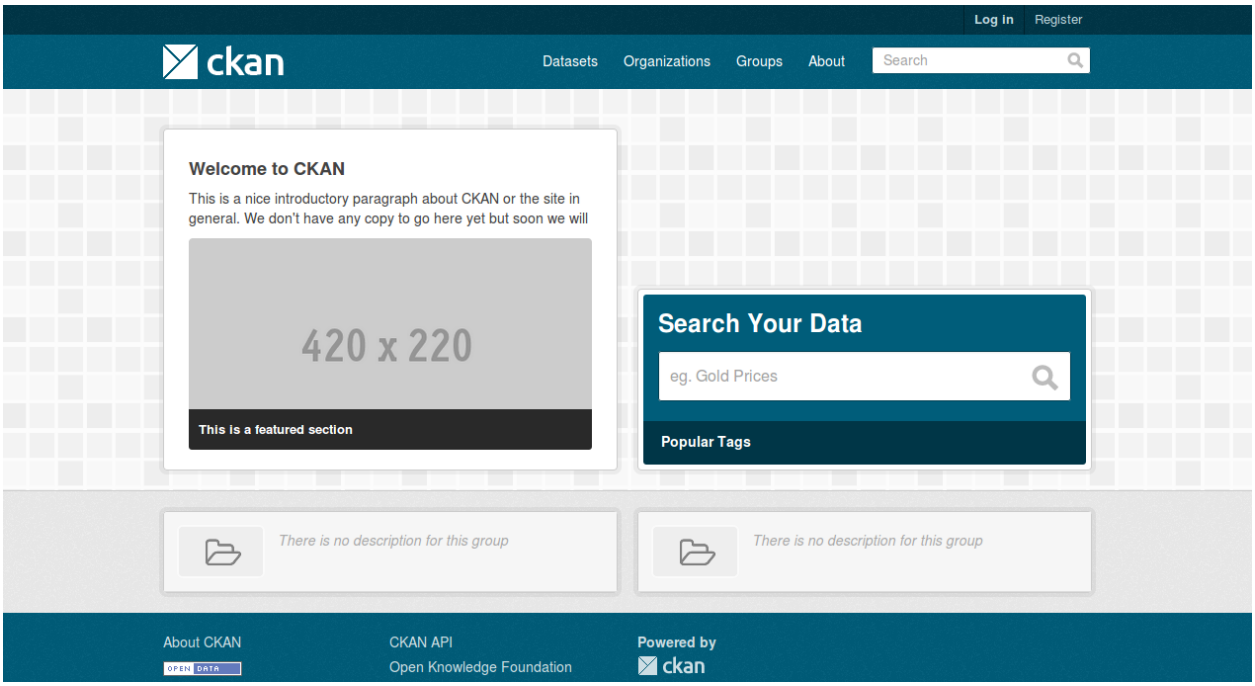
If some of the processes reports an error, make sure you've run all the previous steps and check the logs located in `/var/log/ckan` for more details.

Restart Nginx by running this command:

```
sudo service nginx restart
```

7. You're done!

Open http://localhost in your web browser. You should see the CKAN front page, which will look something like this:



You can now move on to *Getting started* to begin using and customizing your CKAN site.

Note: The default authorization settings on a new install are deliberately restrictive. Regular users won't be able to create datasets or organizations. You should check the *Organizations and authorization* documentation, configure CKAN accordingly and grant other users the relevant permissions using the *sysadmin account*.

Note: There may be a `PermissionError: [Errno 13] Permission denied:` message when restarting supervisor or accessing CKAN via a browser for the first time. This happens when a different user is used to execute the web server process than the user who installed CKAN and the support software. A workaround would be to open up the permissions on the `/usr/lib/ckan/default/src/ckan/ckan/public/base/i18n/` directory so that this user could write the .js files into it. Accessing CKAN will generate these files for a new install, or you could run `ckan -c /etc/ckan/default/ckan.ini translation js` to explicitly generate them.

Installing CKAN from source

CKAN is a Python application that requires three main services: PostgreSQL, Solr and Redis.

This section describes how to install CKAN from source. Although *Installing CKAN from package* is simpler, it requires Ubuntu 20.04 64-bit or Ubuntu 22.04 64-bit. Installing CKAN from source works with other versions of Ubuntu and with other operating systems (e.g. RedHat, Fedora, CentOS, OS X). If you install CKAN from source on your own operating system, please share your experiences on our How to Install CKAN wiki page.

The minimum Python version required is 3.9

From source is also the right installation method for developers who want to work on CKAN.

1. Install the required packages

If you're using a Debian-based operating system (such as Ubuntu) install the required packages with this command:

```
sudo apt-get install python3-dev libpq-dev python3-pip python3-venv git-core redis-
↪server libmagic1
```

If you're not using a Debian-based operating system, find the best way to install the following packages on your operating system (see our How to Install CKAN wiki page for help):

Package	Description
Python	The Python programming language, v3.9 or newer
PostgreSQL	The PostgreSQL database system, v10 or newer
libpq	The C programmer's interface to PostgreSQL
pip	A tool for installing and managing Python packages
python3-venv	The Python3 virtual environment builder (or for Python 2 use 'virtualenv' instead)
Git	A distributed version control system
Apache Solr	A search platform
Jetty	An HTTP server (used for Solr).
OpenJDK JDK	The Java Development Kit (used by Jetty)
Redis	An in-memory data structure store

2. Install CKAN into a Python virtual environment

Tip: If you're installing CKAN for development and want it to be installed in your home directory, you can symlink the directories used in this documentation to your home directory. This way, you can copy-paste the example commands from this documentation without having to modify them, and still have CKAN installed in your home directory:

```
mkdir -p ~/ckan/lib
sudo ln -s ~/ckan/lib /usr/lib/ckan
mkdir -p ~/ckan/etc
sudo ln -s ~/ckan/etc /etc/ckan
```

a. Create a Python virtual environment (virtualenv) to install CKAN into, and activate it:

```
sudo mkdir -p /usr/lib/ckan/default
sudo chown `whoami` /usr/lib/ckan/default
python3 -m venv /usr/lib/ckan/default
. /usr/lib/ckan/default/bin/activate
```

Important: The final command above activates your virtualenv. The virtualenv has to remain active for the rest of the installation and deployment process, or commands will fail. You can tell when the virtualenv is active because its name appears in front of your shell prompt, something like this:

```
(default) $ _
```

For example, if you logout and login again, or if you close your terminal window and open it again, your virtualenv will no longer be activated. You can always reactivate the virtualenv with this command:

```
. /usr/lib/ckan/default/bin/activate
```

b. Install an up-to-date pip:

```
pip install --upgrade pip
```

c. Install the CKAN source code into your virtualenv.

To install the latest stable release of CKAN (CKAN 2.10.4), run:

```
pip install -e 'git+https://github.com/ckan/ckan.git@ckan-2.10.4
↪#egg=ckan[requirements]'
```

If you're installing CKAN for development, you may want to install the latest development version (the most recent commit on the master branch of the CKAN git repository). In that case, run this command instead:

```
pip install -e 'git+https://github.com/ckan/ckan.git#egg=ckan[requirements,dev]'
```

> **Warning:** The development version may contain bugs and should not be used for production websites! Only install this version if you're doing CKAN development.

d. Deactivate and reactivate your virtualenv, to make sure you're using the virtualenv's copies of commands like `ckan` rather than any system-wide installed copies:

```
deactivate
. /usr/lib/ckan/default/bin/activate
```

3. Setup a PostgreSQL database

orphan

Install PostgreSQL required packages:

```
sudo apt install -y postgresql
```

Note: If you are facing a problem in case postgresql is not running, execute the command `sudo service postgresql start`

Check that PostgreSQL was installed correctly by listing the existing databases:

```
sudo -u postgres psql -l
```

Check that the encoding of databases is `UTF8`, if not you might find issues later on with internationalisation. Since changing the encoding of PostgreSQL may mean deleting existing databases, it is suggested that this is fixed before continuing with the CKAN install.

Next you'll need to create a database user if one doesn't already exist. Create a new PostgreSQL user called ckan_default, and enter a password for the user when prompted. You'll need this password later:

```
sudo -u postgres createuser -S -D -R -P ckan_default
```

Create a new PostgreSQL database, called ckan_default, owned by the database user you just created:

```
sudo -u postgres createdb -O ckan_default ckan_default -E utf-8
```

Note: If PostgreSQL is run on a separate server, you will need to edit *postgresql.conf* and *pg_hba.conf*. On Ubuntu, these files are located in *etc/postgresql/{Postgres version}/main*.

Uncomment the *listen_addresses* parameter and specify a comma-separated list of IP addresses of the network interfaces PostgreSQL should listen on or '*' to listen on all interfaces. For example,

```
listen_addresses = 'localhost,192.168.1.21'
```

Add a line similar to the line below to the bottom of *pg_hba.conf* to allow the machine running the web server to connect to PostgreSQL. Please change the IP address as desired according to your network settings.

```
host all all 192.168.1.22/32 md5
```

4. Create a CKAN config file

Create a directory to contain the site's config files:

```
sudo mkdir -p /etc/ckan/default
sudo chown -R `whoami` /etc/ckan/
```

Create the CKAN config file:

```
ckan generate config /etc/ckan/default/ckan.ini
```

Edit the `ckan.ini` file in a text editor, changing the following options:

sqlalchemy.url

This should refer to the database we created in *3. Setup a PostgreSQL database* above:

```
sqlalchemy.url = postgresql://ckan_default:pass@localhost/ckan_default
```

Replace `pass` with the password that you created in *3. Setup a PostgreSQL database* above.

Tip: If you're using a remote host with password authentication rather than SSL authentication, use:

```
sqlalchemy.url = postgresql://ckan_default:pass@<remotehost>/ckan_default?
↪sslmode=disable
```

site_id

Each CKAN site should have a unique `site_id`, for example:

```
ckan.site_id = default
```

site_url

Provide the site's URL (used when putting links to the site into the FileStore, notification emails etc). For example:

```
ckan.site_url = http://demo.ckan.org
```

Do not add a trailing slash to the URL.

5. Setup Solr

orphan

CKAN uses Solr as its search engine, and uses a customized Solr schema file that takes into account CKAN's specific search needs. Now that we have CKAN installed, we need to install and configure Solr.

> **Warning:** CKAN supports **Solr 9** (recommended version) and Solr 8. Starting from CKAN 2.10 these are the only Solr version supported. CKAN 2.9 can run with Solr 9 and 8 as long as it is patched to at least 2.9.5.

There are two supported ways to install Solr.

1. Using CKAN's official Docker images. This is generally the easiest one and the recommended one if you are developing CKAN locally
2. Installing Solr locally and configuring it with the CKAN schema. You can use this option if you can't or don't want to use Docker.

Installing Solr using Docker

You will need to have Docker installed. Please refer to its installation documentation for details.

There are pre-configured Docker images for Solr for each CKAN version. Make sure to pick the image tag that matches your CKAN version (they are named `ckan/ckan-solr:<Major version>.<Minor version>`). To start a local Solr service you can run:

```
docker run --name ckan-solr -p 8983:8983 -d ckan/ckan-solr:2.10-solr9
```

You can now jump to the *Next steps* section.

Installing Solr manually

The following instructions have been tested in Ubuntu 22.04 and are provided as a guidance only. For a Solr production setup is it recommended that you follow the official Solr documentation.

1. Install the OS dependencies:

   ```
   sudo apt-get install openjdk-11-jdk
   ```

2. Download the latest supported version from the Solr downloads page. CKAN supports Solr version 9.x (recommended) and 8.x.
3. Extract the install script file to your desired location (adjust the Solr version number to the one you are using):

   ```
   tar xzf solr-9.2.1.tgz solr-9.2.1/bin/install_solr_service.sh --strip-components=2
   ```

4. Run the installation script as `root`:

   ```
   sudo bash ./install_solr_service.sh solr-9.2.1.tgz
   ```

5. Check that Solr started running:

   ```
   sudo service solr status
   ```

6. Create a new core for CKAN:

```
sudo -u solr /opt/solr/bin/solr create -c ckan
```

7. Replace the standard schema with the CKAN one:

```
sudo -u solr wget -O /var/solr/data/ckan/conf/managed-schema https://raw.
↪githubusercontent.com/ckan/ckan/dev-v2.10/ckan/config/solr/schema.xml
```

8. Restart Solr:

```
sudo service solr restart
```

Next steps with Solr

To check that Solr started you can visit the web interface at http://localhost:8983/solr

Warning: The two installation methods above will leave you with a setup that is fine for local development, but Solr should never be exposed publicly in a production site. Pleaser refer to the Solr documentation to learn how to secure your Solr instance.

If you followed any of the instructions above, the CKAN Solr core will be available at http://localhost:8983/solr/ckan. If for whatever reason you ended up with a different one (eg with a different port, host or core name), you need to change the *solr_url* setting in your *CKAN configuration file* (/etc/ckan/default/ckan.ini) to point to your Solr server, for example:

```
solr_url=http://my-solr-host:8080/solr/ckan-2.10
```

6. Setup Redis

If you installed it locally on the first step, make sure you have a Redis instance running in the *6379* port.

If you have Docker installed, you can setup a default Redis instance by running:

```
docker run --name ckan-redis -p 6379:6379 -d redis
```

7. Create database tables

Now that you have a configuration file that has the correct settings for your database, you can *create the database tables*:

```
cd /usr/lib/ckan/default/src/ckan
ckan -c /etc/ckan/default/ckan.ini db init
```

You should see `Initialising DB: SUCCESS`.

Tip: If the command prompts for a password it is likely you haven't set up the `sqlalchemy.url` option in your CKAN configuration file properly. See *4. Create a CKAN config file*.

8. Set up the DataStore

Note: Setting up the DataStore is optional. However, if you do skip this step, the *DataStore features* will not be available and the DataStore tests will fail.

Follow the instructions in *DataStore extension* to create the required databases and users, set the right permissions and set the appropriate values in your CKAN config file.

Once you have set up the DataStore, you may then wish to configure either the DataPusher or XLoader extensions to add data to the DataStore. To install DataPusher refer to this link: https://github.com/ckan/datapusher and to install XLoader refer to this link: https://github.com/ckan/ckanext-xloader

9. Create CKAN user

To create, remove, list and manage users, you can follow the steps at Create and Manage Users.

10. You're done!

You can now run CKAN from the command-line. This is a simple and lightweight way to serve CKAN that is useful for development and testing:

```
cd /usr/lib/ckan/default/src/ckan
ckan -c /etc/ckan/default/ckan.ini run
```

Open http://127.0.0.1:5000/ in a web browser, and you should see the CKAN front page.

Now that you've installed CKAN, you should:

- Run CKAN's tests to make sure that everything's working, see *Testing CKAN*.
- If you want to use your CKAN site as a production site, not just for testing or development purposes, then deploy CKAN using a production web server such as uWSGI or Nginx. See *Deploying a source install*.
- Begin using and customizing your site, see *Getting started*.

Note: The default authorization settings on a new install are deliberately restrictive. Regular users won't be able to create datasets or organizations. You should check the *Organizations and authorization* documentation, configure CKAN accordingly and grant other users the relevant permissions using the *sysadmin account*.

Source install troubleshooting

Solr setup troubleshooting

Solr requests and errors are logged in the web server log files.

- For Jetty servers, the log files are:

  ```
  /var/log/jetty/<date>.stderrout.log
  ```

- For Tomcat servers, they're:

```
/var/log/tomcat6/catalina.<date>.log
```

AttributeError: 'module' object has no attribute 'css/main.debug.css'

This error is likely to show up when *debug* is set to *True*. To fix this error, install frontend dependencies. See *Frontend development guidelines*.

After installing the dependencies, run `npm run build` and then start ckan server again.

If you do not want to compile CSS, you can also copy the main.css to main.debug.css to get CKAN running:

```
cp /usr/lib/ckan/default/src/ckan/ckan/public/base/css/main.css \
/usr/lib/ckan/default/src/ckan/ckan/public/base/css/main.debug.css
```

ImportError: No module named 'flask_debugtoolbar'

This may show up if you have enabled debug mode in the config file. Simply install the development requirements:

```
pip install -r /usr/lib/ckan/default/src/ckan/dev-requirements.txt
```

Deploying a source install

Once you've installed CKAN from source by following the instructions in *Installing CKAN from source*, you can follow these instructions to deploy your CKAN site using a rudimentary web server, so that it's available to the Internet.

Because CKAN uses WSGI, a standard interface between web servers and Python web applications, CKAN can be used with a number of different web server and deployment configurations, however the CKAN project has now standardized on one NGINX with uwsgi

This guide explains how to deploy CKAN using a uwsgi web server and proxied with NGINX on an Ubuntu server. These instructions have been tested on Ubuntu 18.04.

1. Install Nginx

Install NGINX (a web server) which will proxy the content from one of the WSGI Servers and add a layer of caching:

```
sudo apt-get install nginx
```

2. Create the WSGI script file

The WSGI script file can be copied from the CKAN distribution: `sudo cp /usr/lib/ckan/default/src/ckan/wsgi.py /etc/ckan/default/`

Here is the file:

```
# -- coding: utf-8 --

import os
from ckan.config.middleware import make_app
from ckan.cli import CKANConfigLoader
```

```
from logging.config import fileConfig as loggingFileConfig
config_filepath = os.path.join(
    os.path.dirname(os.path.abspath(__file__)), 'ckan.ini')
abspath = os.path.join(os.path.dirname(os.path.abspath(__file__)))
loggingFileConfig(config_filepath)
config = CKANConfigLoader(config_filepath).get_config()
application = make_app(config)
```

The WSGI Server (configured next) will redirect requests to this WSGI script file. The script file then handles those requests by directing them on to your CKAN instance (after first configuring the Python environment for CKAN to run in).

3. Create the WSGI Server

Make sure you have activated the Python virtual environment before running this command:

```
. /usr/lib/ckan/default/bin/activate
```

uwsgi

Run `pip install uwsgi` The uwsgi configuration file can be copied from the CKAN distribution: `sudo cp /usr/lib/ckan/default/src/ckan/ckan-uwsgi.ini /etc/ckan/default/`

Here is the file:

```
[uwsgi]

http            =  127.0.0.1:8080
uid             =  www-data
gid             =  www-data
wsgi-file       =  /etc/ckan/default/wsgi.py
virtualenv      =  /usr/lib/ckan/default
module          =  wsgi:application
master          =  true
pidfile         =  /tmp/%n.pid
harakiri        =  50
max-requests    =  5000
vacuum          =  true
callable        =  application
strict          =  true
```

If you notice database connection issues in the uwsgi log, try adding the following configurations to resolve them:

```
enable-threads  = true
lazy-apps       = true
```

4. Install Supervisor for the uwsgi

Install Supervisor (a Process Control System) used to control starting, stopping the uwsgi or gunicorn servers:

```
sudo apt-get install supervisor
sudo service supervisor restart
```

uwsgi

Create the `/etc/supervisor/conf.d/ckan-uwsgi.conf` file

```
[program:ckan-uwsgi]

command=/usr/lib/ckan/default/bin/uwsgi -i /etc/ckan/default/ckan-uwsgi.ini

; Start just a single worker. Increase this number if you have many or
; particularly long running background jobs.
numprocs=1
process_name=%(program_name)s-%(process_num)02d

; Log files - change this to point to the existing CKAN log files
stdout_logfile=/etc/ckan/default/uwsgi.OUT
stderr_logfile=/etc/ckan/default/uwsgi.ERR

; Make sure that the worker is started on system start and automatically
; restarted if it crashes unexpectedly.
autostart=true
autorestart=true

; Number of seconds the process has to run before it is considered to have
; started successfully.
startsecs=10

; Need to wait for currently executing tasks to finish at shutdown.
; Increase this if you have very long running tasks.
stopwaitsecs = 600

; Required for uWSGI as it does not obey SIGTERM.
stopsignal=QUIT
```

5. Install an email server

If one isn't installed already, install an email server to enable CKAN's email features (such as sending traceback emails to sysadmins when crashes occur, or sending new activity *email notifications* to users). For example, to install the Postfix email server, do:

```
sudo apt-get install postfix
```

When asked to choose a Postfix configuration, choose *Internet Site* and press return.

6. Create the NGINX config file

Create your site's NGINX config file at /etc/nginx/sites-available/ckan, with the following contents:

```
proxy_temp_path /tmp/nginx_proxy 1 2;

server {
    client_max_body_size 100M;
    location / {
        proxy_pass http://127.0.0.1:8080/;
        proxy_set_header X-Forwarded-For $remote_addr;
        proxy_set_header Host $host;
    }

}
```

To prevent conflicts, disable your default nginx sites and restart:

```
sudo rm -vi /etc/nginx/sites-enabled/default
sudo ln -s /etc/nginx/sites-available/ckan /etc/nginx/sites-enabled/ckan
sudo service nginx restart
```

7. Access your CKAN site

You should now be able to visit your server in a web browser and see your new CKAN instance.

8. Setup a worker for background jobs

CKAN uses asynchronous *Background jobs* for long tasks. These jobs are executed by a separate process which is called a *worker*.

To run the worker in a robust way, *install and configure Supervisor*.

Deployment changes for CKAN 2.9

This section describes how to update your deployment for CKAN 2.9 or later, if you have an existing deployment of CKAN 2.8 or earlier. This is necessary, whether you continue running CKAN on Python 2 or Python 3, because the WSGI entry point for running CKAN has changed. If your existing deployment is different to that described in the official CKAN 2.8 deployment instructions (apache2 + mod_wsgi + nginx) then you'll need to adapt these instructions to your setup.

We now recommend you activate the Python virtual environment in a different place, compared to earlier CKAN versions. For the WSGI server, activation is done in the uwsgi server config file (/etc/ckan/default/ckan-uwsgi.ini).

(In CKAN 2.8.x and earlier, the virtual environment was activated in the WSGI script file.)

3.3 Upgrading CKAN

This document explains how to upgrade a site to a newer version of CKAN. It will walk you through the steps to upgrade your CKAN site to a newer version of CKAN.

Note: The currently supported CKAN version is **CKAN 2.10.4**

Security and performance fixes are also provided for **CKAN 2.9.11**.

Read more about *officially supported versions*

3.3.1 1. Prepare the upgrade

- Before upgrading your version of CKAN you should check that any custom templates or extensions you're using work with the new version of CKAN. For example, you could install the new version of CKAN in a new virtual environment and use that to test your templates and extensions.
- You should also read the *Changelog* to see if there are any extra notes to be aware of when upgrading to the new version.

Warning: You should always **backup your CKAN database** before upgrading CKAN. If something goes wrong with the CKAN upgrade you can use the backup to restore the database to its pre-upgrade state. See *Backup your CKAN database*

3.3.2 2. Upgrade CKAN

The process of upgrading CKAN differs depending on whether you have a package install or a source install of CKAN, and whether you're upgrading to a *major, minor or patch release* of CKAN. Follow the appropriate one of these documents:

Upgrading a CKAN 2 package install to a new patch release

Note: Before upgrading CKAN you should check the compatibility of any custom themes or extensions you're using, check the changelog, and backup your database. See *Upgrading CKAN*.

Patch releases are distributed in the same package as the minor release they belong to, so for example CKAN `2.0`, `2.0.1`, `2.0.2`, etc. will all be installed using the CKAN `2.0` package (`python-ckan_2.0_amd64.deb`):

1. Download the CKAN package:

   ```
   wget https://packaging.ckan.org/python-ckan_2.0_amd64.deb
   ```

   You can check the actual CKAN version from a package running the following command:

   ```
   dpkg --info python-ckan_2.0_amd64.deb
   ```

   Look for the `Version` field in the output:

```
...
Package: python-ckan
Version: 2.0.1-3
...
```

2. Install the package with the following command:

```
sudo dpkg -i python-ckan_2.0_amd64.deb
```

Your CKAN instance should be upgraded straight away.

Note: If you have changed the Apache, Nginx or `who.ini` configuration files, you will get a prompt like the following, asking whether to keep your local changes or replace the files. You generally would like to keep your local changes (option `N`, which is the default), but you can look at the differences between versions by selecting option `D`:

```
Configuration file `/etc/apache2/sites-available/ckan_default'
 ==> File on system created by you or by a script.
 ==> File also in package provided by package maintainer.
   What would you like to do about it ?  Your options are:
    Y or I  : install the package maintainer's version
    N or O  : keep your currently-installed version
      D     : show the differences between the versions
      Z     : start a shell to examine the situation
 The default action is to keep your current version.
*** ckan_default (Y/I/N/O/D/Z) [default=N] ?
```

Your local CKAN configuration file in /etc/ckan/default will not be replaced.

Note: The install process will uninstall any existing CKAN extensions or other libraries located in the `src` directory of the CKAN virtualenv. To enable them again, the installation process will iterate all folders in the `src` directory, reinstall the requirements listed in `pip-requirements.txt` and `requirements.txt` files and run `python setup.py develop` for each. If you are using a custom extension which does not use this requirements file names or is located elsewhere, you will need to manually reenable it.

3. Finally, restart uWSGI and Nginx:

```
sudo supervisorctl restart ckan-uwsgi:*
sudo service nginx restart
```

4. You're done!

You should now be able to visit your CKAN website in your web browser and see that it's running the new version of CKAN.

Upgrading a CKAN 2 package install to a new minor release

Note: Before upgrading CKAN you should check the compatibility of any custom themes or extensions you're using, check the changelog, and backup your database. See *Upgrading CKAN*.

Each *minor release* is distributed in its own package, so for example CKAN `2.0.X` and `2.1.X` will be installed using the `python-ckan_2.0_amd64.deb` and `python-ckan_2.1_amd64.deb` packages respectively.

1. Download the CKAN package for the new minor release you want to upgrade to (replace the version number with the relevant one):

```
wget https://packaging.ckan.org/python-ckan_2.1_amd64.deb
```

2. Install the package with the following command:

```
sudo dpkg -i python-ckan_2.1_amd64.deb
```

Note: If you have changed the Apache, Nginx or `who.ini` configuration files, you will get a prompt like the following, asking whether to keep your local changes or replace the files. You generally would like to keep your local changes (option `N`, which is the default), but you can look at the differences between versions by selecting option `D`:

```
Configuration file `/etc/apache2/sites-available/ckan_default'
 ==> File on system created by you or by a script.
 ==> File also in package provided by package maintainer.
   What would you like to do about it ?  Your options are:
    Y or I  : install the package maintainer's version
    N or O  : keep your currently-installed version
      D     : show the differences between the versions
      Z     : start a shell to examine the situation
 The default action is to keep your current version.
*** ckan_default (Y/I/N/O/D/Z) [default=N] ?
```

Your local CKAN configuration file in /etc/ckan/default will not be replaced.

Note: The install process will uninstall any existing CKAN extensions or other libraries located in the `src` directory of the CKAN virtualenv. To enable them again, the installation process will iterate over all folders in the `src` directory, reinstall the requirements listed in `pip-requirements.txt` and `requirements.txt` files and run `python setup.py develop` for each. If you are using a custom extension which does not use this requirements file name or is located elsewhere, you will need to manually reinstall it.

3. If there have been changes in the database schema (check the *Changelog* to find out) you need to *upgrade your database schema*.

4. If there have been changes in the Solr schema (check the *Changelog* to find out) you need to restart Jetty for the changes to take effect:

```
sudo service jetty restart
```

5. If you have any CKAN extensions installed from source, you may need to checkout newer versions of the extensions that work with the new CKAN version. Refer to the documentation for each extension. We recommend

disabling all extensions on your ini file and re-enable them one by one to make sure they are working fine.

6. If new configuration options have been introduced (check the *Changelog* to find out) then check whether you need to change them from their default values. See *Configuration Options* for details.
7. Rebuild your search index by running the `ckan search-index rebuild` command:

   ```
   sudo ckan search-index rebuild -r
   ```

   See *search-index: Rebuild search index* for details of the `ckan search-index rebuild` command.
8. Finally, restart the web server and Nginx, eg for a CKAN package install running uWSGI:

   ```
   sudo supervisorctl restart ckan-uwsgi:*
   sudo service nginx restart
   ```

Upgrading a source install

Note: Before upgrading CKAN you should check the compatibility of any custom themes or extensions you're using, check the changelog, and backup your database. See *Upgrading CKAN*.

The process for upgrading a source install is the same, no matter what type of CKAN release you're upgrading to:

1. Check the *Changelog* for changes regarding the required 3rd-party packages and their minimum versions (e.g. web, database and search servers) and update their installations if necessary.
2. Activate your virtualenv and switch to the ckan source directory, e.g.:

   ```
   . /usr/lib/ckan/default/bin/activate
   cd /usr/lib/ckan/default/src/ckan
   ```
3. Checkout the new CKAN version from git, for example:

   ```
   git fetch
   git checkout ckan-2.10.4
   ```

   If you have any CKAN extensions installed from source, you may need to checkout newer versions of the extensions at this point as well. Refer to the documentation for each extension.

   As of CKAN 2.6 branch naming has changed. See *Doing a CKAN release* for naming conventions. Specific patches and minor versions can be checked-out using tags.
4. Update CKAN's dependencies:

   ```
   pip install --upgrade -r requirements.txt
   ```
5. Register any new or updated plugins:

   ```
   python setup.py develop
   ```
6. If there have been changes in the Solr schema (check the *Changelog* to find out) you need to restart Jetty for the changes to take effect:

   ```
   sudo service jetty restart
   ```
7. If there have been changes in the database schema (check the *Changelog* to find out) you need to *upgrade your database schema*.

8. If new configuration options have been introduced (check the *Changelog* to find out) then check whether you need to change them from their default values. See *Configuration Options* for details.
9. Rebuild your search index by running the `ckan search-index rebuild` command:

   ```
   ckan -c /path/to/ckan.ini search-index rebuild -r --config=/etc/ckan/default/ckan.
   ↪ini
   ```

   See *search-index: Rebuild search index* for details of the `ckan search-index rebuild` command.
10. Finally, restart your web server. For example if you have deployed CKAN using a package install, run this command:

    ```
    sudo supervisorctl restart ckan-uwsgi:*
    ```
11. You're done!

You should now be able to visit your CKAN website in your web browser and see that it's running the new version of CKAN.

Upgrading a CKAN install from Python 2 to Python 3

These instructions describe how to upgrade a source install of CKAN 2.9 from Python 2 to Python 3, which is necessary because Python 2 is end of life, as of January 1st, 2020.

Preparation

- Backup your CKAN source, virtualenv and databases, just in case.
- Upgrade to CKAN 2.9, if you've not done already.

Upgrade

You'll probably need to deactivate your existing virtual environment:

```
deactivate
```

The existing setup has the virtual environment here: /usr/lib/ckan/default and the CKAN source code underneath in */usr/lib/ckan/default/src*. We'll move that aside in case we need to roll-back:

```
sudo mv /usr/lib/ckan/default /usr/lib/ckan/py2
```

From this doc: *Installing CKAN from source* you need to do these sections:

- 1. Install the required packages
- 2. Install CKAN into a Python virtual environment
- 6. Link to who.ini

Note: For changes about CKAN deployment see: *Installing CKAN from source* and specifically the changes with CKAN 2.9: *Deployment changes for CKAN 2.9*.

See also:

CKAN releases
Information about the different CKAN releases and the officially supported versions.

Changelog
The changelog lists all CKAN releases and the main changes introduced in each release.

Doing a CKAN release
Documentation of the process that the CKAN developers follow to do a CKAN release.

3.4 Getting started

Once you've finished *installing CKAN*, this section will walk you through getting started with your new CKAN website, including creating a CKAN sysadmin user, some test data, and the basics of configuring your CKAN site.

3.4.1 Creating a sysadmin user

You have to use CKAN's command line interface to create your first sysadmin user, and it can also be useful to create some test data from the command line. For full documentation of CKAN's command line interface (including troubleshooting) see *Command Line Interface (CLI)*.

Note: CKAN commands are executed using the `ckan` command on the server that CKAN is installed on. Before running the ckan commands below, you need to make sure that your virtualenv is activated and that you're in your ckan source directory. For example:

```
. /usr/lib/ckan/default/bin/activate
cd /usr/lib/ckan/default/src/ckan
```

You have to create your first CKAN sysadmin user from the command line. For example, to create a new user called `seanh` and make him a sysadmin:

```
ckan -c /etc/ckan/default/ckan.ini sysadmin add seanh email=seanh@localhost name=seanh
```

You'll be prompted to enter a password during account creation.

Or, if you already have an existing user, you could promote him to a sysadmin:

```
ckan -c /etc/ckan/default/ckan.ini sysadmin add seanh
```

For a list of other command line commands for managing sysadmins, run:

```
ckan -c /etc/ckan/default/ckan.ini sysadmin --help
```

Read the *Sysadmin guide* to learn what you can do as a CKAN sysadmin.

3.4.2 Creating test data

It can be handy to have some test data to start with, to quickly check that everything works. You can add a random set of test data to your site from the command line with the following `generate fake-data` commands:

```
ckan -c /etc/ckan/default/ckan.ini generate fake-data organization
# check the output and save the ID of organization into variable:
owner_org=<Organization ID from the previous command>

ckan -c /etc/ckan/default/ckan.ini generate fake-data dataset --owner_org=$owner_org
```

If you later want to delete this test data and start again with an empty database, you can use the *db clean* command.

For a short description of this subcommand, run:

```
ckan -c /etc/ckan/default/ckan.ini generate fake-data --help
```

3.4.3 Config file

All of the options that can be set in the admin page and many more can be set by editing CKAN's config file. By default, from CKAN 2.9 the config file is located at /etc/ckan/default/ckan.ini. (For older versions, the config file is located at /etc/ckan/default/development.ini or /etc/ckan/default/production.ini). The config file can be edited in any text editor. For example, to change the title of your site you would find the `ckan.site_title` line in your config file and edit it:

```
ckan.site_title = Masaq Data Hub
```

Make sure the line is not commented-out (lines in the config file that begin with # are considered comments, so if there's a # at the start of a line you've edited, delete it), save the file, and then restart your web server for the changes to take effect. For example, if using a CKAN package install:

```
sudo supervisorctl restart ckan-uwsgi:*
```

For full documentation of CKAN's config file and all the options you can set, see *Configuration Options*.

Note: If the same option is set in both the config file and in the admin page, the admin page setting takes precedence. You can use the *Reset* button on the admin page to clear your settings, and allow settings from the config file to take effect.

3.5 Database Management

Note: See *Command Line Interface (CLI)* for details on running the `ckan` commands mentioned below.

3.5.1 Initialization

Before you can run CKAN for the first time, you need to run `db init` to initialize your database:

```
ckan -c /etc/ckan/default/ckan.ini db init
```

If you forget to do this you'll see this error message in your web browser:

> 503 Service Unavailable: This site is currently off-line. Database is not initialised.

3.5.2 Cleaning

Warning: This will delete all data from your CKAN database!

You can delete everything in the CKAN database, including the tables, to start from scratch:

```
ckan -c /etc/ckan/default/ckan.ini db clean
```

After cleaning the database you must do either *initialize it* or *import a previously created dump*.

3.5.3 Import and Export

Dumping and Loading databases to/from a file

PostgreSQL offers the command line tools pg_dump and pg_restore for dumping and restoring a database and its content to/from a file.

For example, first dump your CKAN database:

```
sudo -u postgres pg_dump --format=custom -d ckan_default > ckan.dump
```

Warning: The exported file is a complete backup of the database, and includes API keys and other user data which may be regarded as private. So keep it secure, like your database server.

Note: If you've chosen a non-default database name (i.e. *not* `ckan_default`) then you need to adapt the commands accordingly.

Then restore it again:

```
ckan -c /etc/ckan/default/ckan.ini db clean
sudo -u postgres pg_restore --clean --if-exists -d ckan_default < ckan.dump
```

If you're importing a dump from an older version of CKAN you must *upgrade the database schema* after the import.

Once the import (and a potential upgrade) is complete you should *rebuild the search index*.

Exporting Datasets to JSON Lines

You can export all of your CKAN site's datasets from your database to a JSON Lines file using ckanapi:

```
ckanapi dump datasets -c /etc/ckan/default/ckan.ini --all -O my_datasets.jsonl
```

This is useful to create a simple public listing of the datasets, with no user information. Some simple additions to the Apache config can serve the dump files to users in a directory listing. To do this, add these lines to your virtual Apache config file (e.g. /etc/apache2/sites-available/ckan_default.conf):

```
Alias /dump/ /home/okfn/var/srvc/ckan.net/dumps/

# Disable the mod_python handler for static files
<Location /dump>
    SetHandler None
    Options +Indexes
</Location>
```

Warning: Don't serve an SQL dump of your database (created using the `pg_dump` command), as those contain private user information such as email addresses and API keys.

Exporting User Accounts to JSON Lines

You can export all of your CKAN site's user accounts from your database to a JSON Lines file using ckanapi:

```
ckanapi dump users -c /etc/ckan/default/ckan.ini --all -O my_database_users.jsonl
```

3.5.4 Upgrading

> **Warning:** You should *create a backup of your database* before upgrading it.
>
> To avoid problems during the database upgrade, comment out any plugins that you have enabled in your ini file. You can uncomment them again when the upgrade finishes.

If you are upgrading to a new CKAN *major release* update your CKAN database's schema using the `ckan db upgrade` command:

```
ckan -c /etc/ckan/default/ckan.ini db upgrade
```

This command applies all CKAN core migrations and all unapplied migrations from enabled plugins. `--skip-core` and `--skip-plugins` flags can be used to run either only core migration, or only migrations from enabled plugins.

3.6 Command Line Interface (CLI)

Note: From CKAN 2.9 onwards the CKAN configuration file is named 'ckan.ini'. Previous names: 'production.ini' and 'development.ini' (plus others) may also still appear in documentation and the software. These legacy names will eventually be phased out.

Note: From CKAN 2.9 onwards, the `paster` command used for common CKAN administration tasks has been replaced with the `ckan` command.

If you have trouble running 'ckan' CLI commands, see *Troubleshooting ckan Commands* below.

Note: Once you activate your CKAN virtualenv the "ckan" command is available from within any location within the host environment.

To run a ckan command without activating the virtualenv first, you have to give the full path the ckan script within the virtualenv, for example:

```
/usr/lib/ckan/default/bin/ckan -c /etc/ckan/default/ckan.ini user list
```

In the example commands below, we assume you're running the commands with your virtualenv activated and from your ckan directory.

The general form of a CKAN `ckan` command is:

```
ckan --config=/etc/ckan/default/ckan.ini command
```

The `` –config`` option tells CKAN where to find your config file, which it reads for example to know which database it should use. As you'll see in the examples below, this option can be given as `-c` for short.

The config file (ckan.ini) will generally be located in the `/etc/ckan/default/` directory however it can be located in any directory on the host machine

command should be replaced with the name of the CKAN command that you wish to execute. Most commands have their own subcommands and options.

Note: You may also specify the location of your config file using the CKAN_INI environment variable. You will no longer need to use –config= or -c to tell ckan where the config file is:

```
export CKAN_INI=/etc/ckan/default/ckan.ini
```

Note: You can run the ckan command in the same directory as the CKAN config file when the config file is named 'ckan.ini'. You will not be required to use –config or -c in this case. For backwards compatibility, the config file can be also named 'development.ini', but this usage is deprecated and will be phased out in a future CKAN release.

`cd /usr/lib/ckan/default/src/ckan; ckan command`

Commands and Subcommands

`ckan -c /etc/ckan/default/ckan.ini user list`

(Here `user` is the name of the CKAN command you're running, and `list` is a subcommand of `user`.)

For a list of all available commands, see *CKAN Commands Reference*.

Each command has its own help text, which tells you what subcommands and options it has (if any). To print out a command's help text, run the command with the `--help` option, for example:

`ckan -c /etc/ckan/default/ckan.ini user --help`

3.6.1 CLI command: ckan shell

The main goal to execute a ckan shell command is IPython session with the application loaded for easy debugging and dynamic coding.

There are three variables already populated into the namespace of the shell:

- **app** containing the Flask application
- **config** containing the CKAN config dictrionary
- **model** module to access to the database using SQLAlchemy syntax

command:

```
$ ckan shell
```

Example 1:

```
$ ckan shell
Python 3.9.13 (main, Dec 11 2022, 15:23:12)
Type 'copyright', 'credits' or 'license' for more information

In [1]: model.User.all()
Out[1]:
[<User id=f48287e2-6fac-41a9-9170-fc25ddbcc2d7 name=default password=$pbkdf2- sha512
↪$25000$4rzXure2NkYoBeA8h5DyHg$yKMLOBZCtY.bA5XYq/qhzXfNCO7QOHGuRSkvCjkE2wThE.
↪km/2L6GwQbY4p4lFXyyRMYXnACLxXvR27rVDq/yw fullname=None email=None␣
↪apikey=46a0b1cc-28f3-4f96-9cf2-f0479fd3f200 created=2022-06-08 12:54:20.344765 reset_
↪key=None about=None last_active=None activity_streams_email_notifications=False␣
↪sysadmin=True state=active image_url=None plugin_extras=None>]

In [2]: from ckan.logic.action.get import package_show

In [3]: package_show({"model": model}, {"id": "api-package-1"})
Out[3]:
{'author': None,
'author_email': None,
'creator_user_id': 'f0c04c11-4369-4cf1-9da4-69d9aae06a2e',
'id': '922f3a91-c9ed-4e19-a722-366671b7d72c',
'isopen': False,
'license_id': None,
'license_title': None,
'maintainer': None,
'maintainer_email': None,
'metadata_created': '2022-06-16T14:13:37.736125',
'metadata_modified': '2022-06-16T14:20:19.639665',
'name': 'api-package-1',
'notes': 'Update from API:10000',
'num_resources': 0,
'num_tags': 0,
'organization': None,
'owner_org': None,
'private': False,
'state': 'active',
'title': 'api-package-1',
'type': 'dataset',
'url': None,
'version': None,
'resources': [],
'tags': [],
'extras': [],
'groups': [],
'relationships_as_subject': [],
'relationships_as_object': []}
```

Example 2:

```
In [7]: from ckanext.activity.logic import action
In [8]: before = datetime.fromisoformat('2022-06-16T14:14:00.627446').timestamp()
In [9]: %timeit action.package_activity_list({}, {'id': 'api-package-1', 'before':␣
↪before})3.17 ms ± 11.9 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
In [10]: %timeit action.package_activity_list({}, {'id': 'api-package-1', 'offset': 9000}
↪)25.3 ms ± 504 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

3.6.2 Troubleshooting ckan Commands

Permission Error

If you receive 'Permission Denied' error, try running ckan with sudo.

```
sudo /usr/lib/ckan/default/bin/ckan -c /etc/ckan/default/ckan.ini db clean
```

Virtualenv not activated, or not in ckan dir

Most errors with ckan commands can be solved by remembering to **activate your virtual environment** and **change to the ckan directory** before running the command:

```
. /usr/lib/ckan/default/bin/activate
cd /usr/lib/ckan/default/src/ckan
```

Error messages such as the following are usually caused by forgetting to do this:

- **Command 'foo' not known** (where *foo* is the name of the command you tried to run)
- **The program 'ckan' is currently not installed**
- **Command not found: ckan**
- **ImportError: No module named webassets** (or other `ImportErrors`)

Running ckan commands provided by extensions

If you're trying to run a CKAN command provided by an extension that you've installed and you're getting an error like **Command 'foo' not known** even though you've activated your virtualenv, make sure that you have added the relevant plugin to the *ckan.plugins* setting in the ini file.

Wrong config file path

AssertionError: Config filename development.ini does not exist
This means you forgot to give the `--config` or `-c` option to tell CKAN where to find your config file. (CKAN looks for a config file named `development.ini` in your current working directory by default.)

ConfigParser.MissingSectionHeaderError: File contains no section headers
This happens if the config file that you gave with the `-c` or `--config` option is badly formatted, or if you gave the wrong filename.

IOError: [Errno 2] No such file or directory: '...'
This means you gave the wrong path to the `--config` or `-c` option (you gave a path to a file that doesn't exist).

3.6.3 ckan Commands Reference

The following ckan commands are supported by CKAN:

asset	WebAssets commands.
config	Search, validate, describe config options
config-tool	Tool for editing options in a CKAN config file
datapusher	Perform commands in the datapusher.
dataset	Manage datasets.
datastore	Perform commands to set up the datastore.
db	Perform various tasks on the database.
generate	Generate empty extension files to expand CKAN
jobs	Manage background jobs
sass	Compile all root sass documents into their CSS counterparts
notify	Send out modification notifications.
plugin-info	Provide info on installed plugins.
profile	Code speed profiler.
run	Start Development server.
search-index	Creates a search index for all datasets
sysadmin	Gives sysadmin rights to a named user.
tracking	Update tracking statistics.
translation	Translation helper functions
user	Manage users.
views	Create views on relevant resources

asset: WebAssets commands

Usage

```
ckan asset build            - Builds bundles, regardless of whether they are changed or
↪not
ckan asset watch            - Start a daemon which monitors source files, and rebuilds
↪bundles
ckan asset clean            - Will clear out the cache, which after a while can grow
↪quite large
```

config: Search, validate, describe config options

Usage

```
ckan config declaration [PLUGIN...]  - Print declared config options for the given
↪plugins.
ckan config describe [PLUGIN..]      - Print out config declaration for the given
↪plugins.
ckan config search [PATTERN]         - Print all declared config options that match
↪pattern.
ckan config undeclared               - Print config options that has no declaration.
ckan config validate                 - Validate global configuration object against
↪declaration.
```

config-tool: Tool for editing options in a CKAN config file

Usage

```
ckan config-tool --section (-s)  - Section of the config file
ckan config-tool --edit (-e)     - Checks the option already exists in the config file
ckan config-tool --file (-f)     - Supply an options file to merge in
```

Examples

```
ckan config-tool /etc/ckan/default/ckan.ini sqlalchemy.url=123 'ckan.site_title=ABC'
ckan config-tool /etc/ckan/default/ckan.ini -s server:main -e port=8080
ckan config-tool /etc/ckan/default/ckan.ini -f custom_options.ini
```

datapusher: Perform commands in the datapusher

Usage

```
ckan datapusher resubmit    - Resubmit udated datastore resources
ckan datapusher submit      - Submits resources from package
```

dataset: Manage datasets

Usage

```
ckan dataset DATASET_NAME|ID            - shows dataset properties
ckan dataset show DATASET_NAME|ID       - shows dataset properties
ckan dataset list                       - lists datasets
ckan dataset delete [DATASET_NAME|ID]   - changes dataset state to 'deleted'
ckan dataset purge [DATASET_NAME|ID]    - removes dataset from db entirely
```

datastore: Perform commands in the datastore

Make sure that the datastore URLs are set properly before you run these commands.

Usage

```
ckan datastore set-permissions  - generate SQL for permission configuration
ckan datastore dump             - dump a datastore resource
ckan datastore purge            - purge orphaned datastore resources
```

db: Manage databases

```
ckan db clean               - Clean the database
ckan db downgrade           - Downgrade the database
ckan db duplicate_emails    - Check users email for duplicate
ckan db init                - Initialize the database
ckan db pending-migrations  - List all sources with unapplied migrations.
ckan db upgrade             - Upgrade the database
ckan db version             - Returns current version of data schema
```

See *Database Management*.

generate: Scaffolding for regular development tasks

Usage

```
ckan generate config        -  Create a ckan.ini file.
ckan generate extension     -  Create empty extension.
ckan generate fake-data     -  Generate random entities of the given category.
ckan generate migration     -  Create new alembic revision for DB migration.
```

jobs: Manage background jobs

```
ckan jobs cancel      - cancel a specific job.
ckan jobs clear       - cancel all jobs.
ckan jobs list        - list jobs.
ckan jobs show        - show details about a specific job.
ckan jobs test        - enqueue a test job.
ckan jobs worker      - start a worker
```

The `jobs` command can be used to manage *Background jobs*.

New in version 2.7.

Run a background job worker

`ckan -c /etc/ckan/default/ckan.ini jobs worker [--burst] [QUEUES]`

Starts a worker that fetches job from the *job queues* and executes them. If no queue names are given then it listens to the default queue. This is equivalent to

`ckan -c /etc/ckan/default/ckan.ini jobs worker default`

If queue names are given then the worker listens to those queues and only those:

`ckan -c /etc/ckan/default/ckan.ini jobs worker my-custom-queue another-special-queue`

Hence, if you want the worker to listen to the default queue and some others then you must list the default queue explicitly

`ckan -c /etc/ckan/default/ckan.ini jobs worker default my-custom-queue`

If the `--burst` option is given then the worker will exit as soon as all its queues are empty. Otherwise it will wait indefinitely until a new job is enqueued (this is the default).

Note: In a production setting you should *use a more robust way of running background workers*.

List enqueued jobs

```
ckan -c /etc/ckan/default/ckan.ini jobs list [QUEUES]
```

Lists the currently enqueued jobs from the given *job queues*. If no queue names are given then the jobs from all queues are listed.

Show details about a job

```
ckan -c /etc/ckan/default/ckan.ini jobs show ID
```

Shows details about the enqueued job with the given ID.

Cancel a job

```
ckan -c /etc/ckan/default/ckan.ini jobs cancel ID
```

Cancels the enqueued job with the given ID. Jobs can only be canceled while they are enqueued. Once a worker has started executing a job it cannot be aborted anymore.

Clear job queues

```
ckan -c /etc/ckan/default/ckan.ini jobs clear [QUEUES]
```

Cancels all jobs on the given *job queues*. If no queues are given then *all* queues are cleared.

Enqueue a test job

```
ckan -c /etc/ckan/default/ckan.ini jobs test [QUEUES]
```

Enqueues a test job. If no *job queues* are given then the job is added to the default queue. If queue names are given then a separate test job is added to each of the queues.

sass: Compile all root sass documents into their CSS counterparts

Usage

```
sass
```

notify: Send out modification notifications

Usage

```
ckan notify replay    - send out modification signals. In "replay" mode,
                      an update signal is sent for each dataset in the database.
```

plugin-info: Provide info on installed plugins

As the name suggests, this commands shows you the installed plugins (based on the .ini file) , their description, and which interfaces they implement

profile: Code speed profiler

Provide a ckan url and it will make the request and record how long each function call took in a file that can be read by runsnakerun.

Usage

```
ckan profile URL
```

The result is saved in profile.data.search. To view the profile in runsnakerun:

```
runsnakerun ckan.data.search.profile
```

You may need to install the cProfile python module.

run: Start Development server

Usage

```
ckan run --host (-h)                  - Set Host
ckan run --port (-p)                  - Set Port
ckan run --disable-reloader (-r)      - Use reloader
ckan run --passthrough_errors         - Crash instead of handling fatal errors
ckan run --disable-debugger           - Disable the default debugger
```

Use `--passthrough-errors` to enable pdb

Exceptions are caught and handled by CKAN. Sometimes, user needs to disable this error handling, to be able to use pdb or the debug capabilities of the most common IDE. This allows to use breakpoints, inspect the stack frames and evaluate arbitrary Python code. Running CKAN with `--passthrough-errors` will automatically disable CKAN reload capabilities and run everything in a single process, for the sake of simplicity.

Example:

> python -m pdb ckan run –passthrough-errors

Use `--disable-debugger` for external debugging

CKAN uses the run_simple function from the werkzeug package, which enables hot reloading and debugging amongst other things. If we wish to use external debugging tools such as debugpy (for remote, container-based debugging), we must disable the default debugger for CKAN.

Example:

> python -m pdb ckan run –disable-debugger

search-index: Search index commands

Usage

```
ckan search-index check                    - Check search index
ckan search-index clear                    - Clear the search index
ckan search-index rebuild                  - Rebuild search index
ckan search-index rebuild-fast             - Reindex with multiprocessing
ckan search-index show                     - Show index of a dataset
```

search-index: Rebuild search index

Rebuilds the search index. This is useful to prevent search indexes from getting out of sync with the main database.

For example

`ckan -c /etc/ckan/default/ckan.ini search-index rebuild`

This default behaviour will refresh the index keeping the existing indexed datasets and rebuild it with all datasets. If you want to rebuild it for only one dataset, you can provide a dataset name

`ckan -c /etc/ckan/default/ckan.ini search-index rebuild test-dataset-name`

Alternatively, you can use the *-o* or *–only-missing* option to only reindex datasets which are not already indexed

`ckan -c /etc/ckan/default/ckan.ini search-index rebuild -o`

There is also an option available which works like the refresh option but tries to use all processes on the computer to reindex faster

`ckan -c /etc/ckan/default/ckan.ini search-index rebuild-fast`

There is also an option to clear the whole index first and then rebuild it with all datasets:

`ckan -c /etc/ckan/default/ckan.ini search-index rebuild --clear`

There are other search related commands, mostly useful for debugging purposes

```
ckan search-index check                  - checks for datasets not indexed
ckan search-index show DATASET_NAME      - shows index of a dataset
ckan search-index clear [DATASET_NAME]   - clears the search index for the provided␣
↪dataset or for the whole ckan instance
```

sysadmin: Give sysadmin rights

Usage

```
ckan sysadmin add       - convert user into a sysadmin
ckan sysadmin list      - list sysadmins
ckan sysadmin remove    - removes user from sysadmins
```

For example, to make a user called 'admin' into a sysadmin

`ckan -c /etc/ckan/default/ckan.ini sysadmin add admin`

tracking: Update tracking statistics

Starting CKAN 2.11 tracking command is only available if the extension es enabled.

Usage

```
ckan tracking update [start_date]       - update tracking stats
ckan tracking export FILE [start_date]  - export tracking stats to a csv file
```

translation: Translation helper functions

Usage

```
ckan translation js          - generate the JavaScript translations
ckan translation mangle      - mangle the zh_TW translations for testing
ckan translation check-po    - check po files for common mistakes
```

Note: Since version 2.7 the JavaScript translation files are automatically regenerated if necessary when CKAN is started. Hence you usually do not need to run `ckan translation js` manually.

user: Create and manage users

Lets you create, remove, list and manage users.

Usage

```
ckan user add         - add new user
ckan user list        - list all users
ckan user remove      - remove user
ckan user setpass     - set password for the user
ckan user show        - show user
```

For example, to create a new user called 'admin'

```
ckan -c /etc/ckan/default/ckan.ini user add admin email=admin@localhost
```

Note: You can use password=test1234 option if "non-interactive" usage is a requirement.

To delete the 'admin' user

```
ckan -c /etc/ckan/default/ckan.ini user remove admin
```

views: Create views on relevant resources

Usage

```
ckan views clean      - permanently delete views for all types no...
ckan views clear      - permanently delete all views or the ones with...
ckan views create     - create views on relevant resources.

ckan views --dataset (-d)        - Set Dataset
ckan views --no-default-filters
ckan views --search (-s)         - Set Search
ckan views --yes (-y)
```

3.7 Organizations and authorization

CKAN's authorization system controls which users are allowed to carry out which actions on the site. All actions that users can carry out on a CKAN site are controlled by the authorization system. For example, the authorization system controls who can register new user accounts, delete user accounts, or create, edit and delete datasets, groups and organizations.

Authorization in CKAN can be controlled in four ways:

1. Organizations
2. Dataset collaborators
3. Configuration file options
4. Extensions

The following sections explain each of the four methods in turn.

Note: An **organization admin** in CKAN is an administrator of a particular organization within the site, with control over that organization and its members and datasets. A **sysadmin** is an administrator of the site itself. Sysadmins can always do everything, including adding, editing and deleting datasets, organizations and groups, regardless of the organization roles and configuration options described below.

3.7.1 Organizations

Organizations are the primary way to control who can see, create and update datasets in CKAN. Each dataset can belong to a single organization, and each organization controls access to its datasets.

Datasets can be marked as public or private. Public datasets are visible to everyone. Private datasets can only be seen by logged-in users who are members of the dataset's organization. Private datasets are not shown in dataset searches unless the logged in user (or the user identified via an API key) has permission to access them.

When a user joins an organization, an organization admin gives them one of three roles: member, editor or admin.

A **member** can:

- View the organization's private datasets.

An **editor** can do everything a **member** can plus:

- Add new datasets to the organization

- Edit or delete any of the organization's datasets
- Make datasets public or private.

An organization **admin** can do everything as **editor** plus:

- Add users to the organization, and choose whether to make the new user a member, editor or admin
- Change the role of any user in the organization, including other admin users
- Remove members, editors or other admins from the organization
- Edit the organization itself (for example: change the organization's title, description or image)
- Delete the organization

When a user creates a new organization, they automatically become the first admin of that organization.

3.7.2 Dataset collaborators

Changed in version 2.9: Dataset collaborators were introduced in CKAN 2.9

In addition to traditional organization-based permissions, CKAN instances can also enable the dataset collaborators feature, which allows dataset-level authorization. This provides more granular control over who can access and modify datasets that belong to an organization, or allows authorization setups not based on organizations. It works by allowing users with appropriate permissions to give permissions to other users over individual datasets, regardless of what organization they belong to.

Dataset collaborators are not enabled by default, you need to activate it by setting *ckan.auth.allow_dataset_collaborators* to `True`.

By default, only Administrators of the organization a dataset belongs to can add collaborators to a dataset. When adding them, they can choose between two roles: member and editor.

A **member** can:

- View the dataset if it is private.

An **editor** can do everything a **member** can plus:

- Make the dataset public or private.
- Edit or delete the dataset (including assigning it to an organization)

In addition, if *ckan.auth.allow_admin_collaborators* is set to `True`, collaborators can have another role: admin.

An **admin** collaborator can do everything an **editor** can plus:

- Add collaborators to the dataset, and choose whether to make them a member, editor or admin (if enabled)
- Change the role of any collaborator in the dataset, including other admin users
- Remove collaborators of any role from the dataset

If the `ckan.auth.allow_admin_collaborators` setting is turned off in a site where admin collaborators have already been created, existing collaborators with role **admin** will no longer be able to manage collaborators, but they will still be able to edit and access the datasets that they are assigned to (ie they will have the same permissions as an **editor**.

If the global `ckan.auth.allow_dataset_collaborators` setting is turned off in a site where collaborators have already been created, collaborators will no longer have permissions on the datasets they are assigned to, and normal organization-based permissions will be in place.

> **Warning:** When turning off this setting, you must reindex all datasets to update the permission labels, in order to prevent access to private datasets to the previous collaborators.

By default, collaborators can not change the owner organization of a dataset unless they are admins or editors in both the source and destination organizations. To allow collaborators to change the owner organization even if they don't belong to the source organization, set *ckan.auth.allow_collaborators_to_change_owner_org* to `True`.

Dataset collaborators can be used with other authorization settings to create custom authentication scenarios. For instance, on instances where datasets don't need to belong to an organization (both *ckan.auth.create_dataset_if_not_in_organization* and *ckan.auth.create_unowned_dataset* are `True`), the user that originally created a dataset can also add collaborators to it (allowing admin collaborators or not depending on the `ckan.auth.allow_admin_collaborators` setting). Note that in this case though, if the dataset is assigned to an organization, the original creator might no longer be able to access and edit, as organization permissions take precedence over collaborators ones.

3.7.3 Configuration File Options

See *Authorization Settings*.

3.7.4 Extensions

CKAN extensions can implement custom authorization rules by overriding the authorization functions that CKAN uses. This is done by implementing the `IAuthFunctions` plugin interface.

Dataset visibility is determined by permission labels stored in the search index. Implement the `IPermissionLabels` plugin interface then *rebuild your search index* to change your dataset visibility rules. There is no no need to override the `package_show` auth function, it will inherit these changes automatically.

To get started with writing CKAN extensions, see *Extending guide*.

3.8 Data preview and visualization

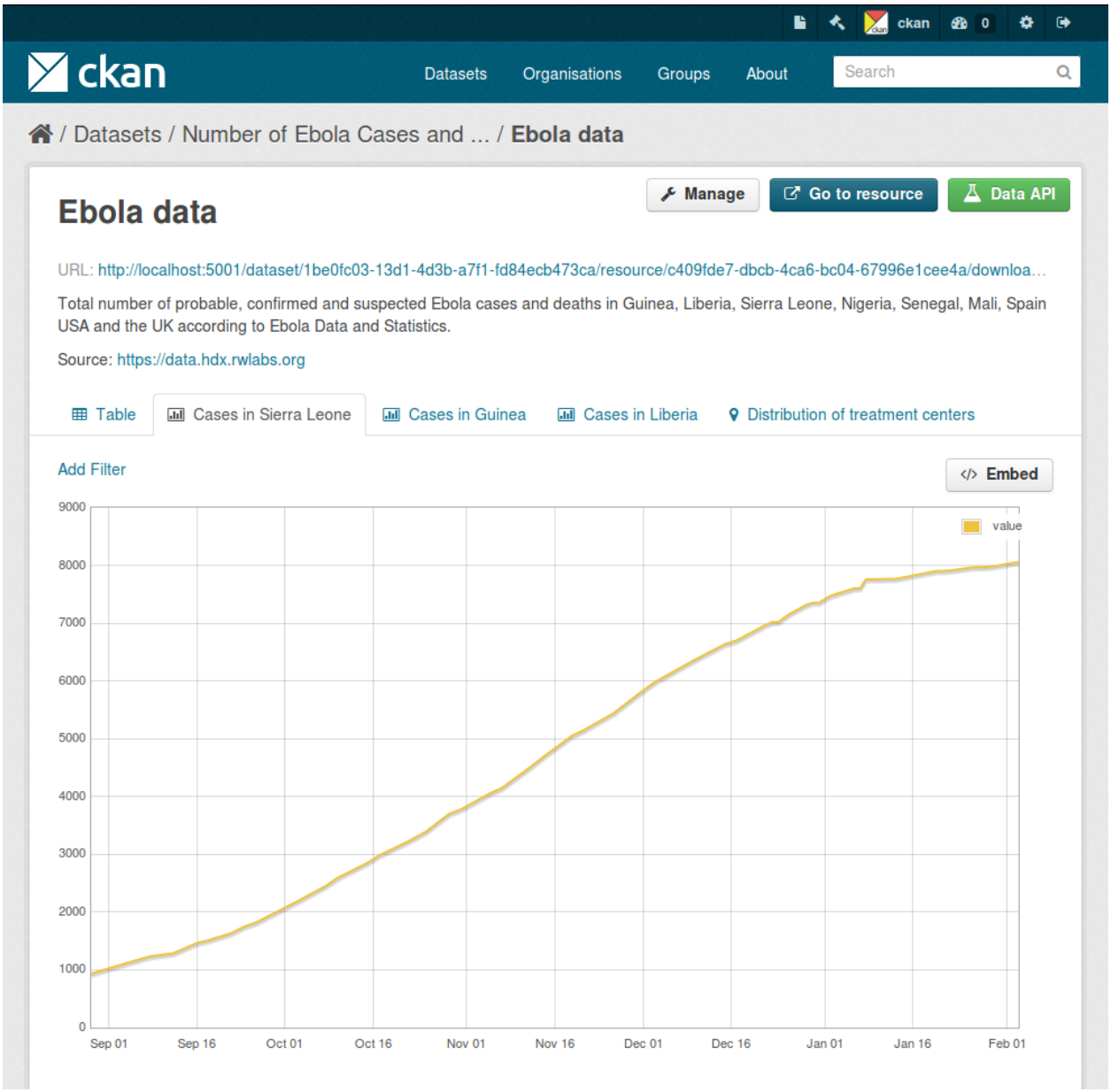
Contents

- *Data preview and visualization*
  - *Overview*
  - *Managing resource views*
  - *Defining views to appear by default*
  - *Available view plugins*
    - *DataTables view*
    - *Text view*
    - *Image view*
    - *Video view*
    - *Audio view*

    * *Web page view*
  - *Other view plugins*
  - *Resource Proxy*
  - *Migrating from previous CKAN versions*
  - *Command line interface*

3.8.1 Overview

The CKAN resource page can contain one or more visualizations of the resource data or file contents (a table, a bar chart, a map, etc). These are commonly referred to as *resource views*.



The main features of resource views are:

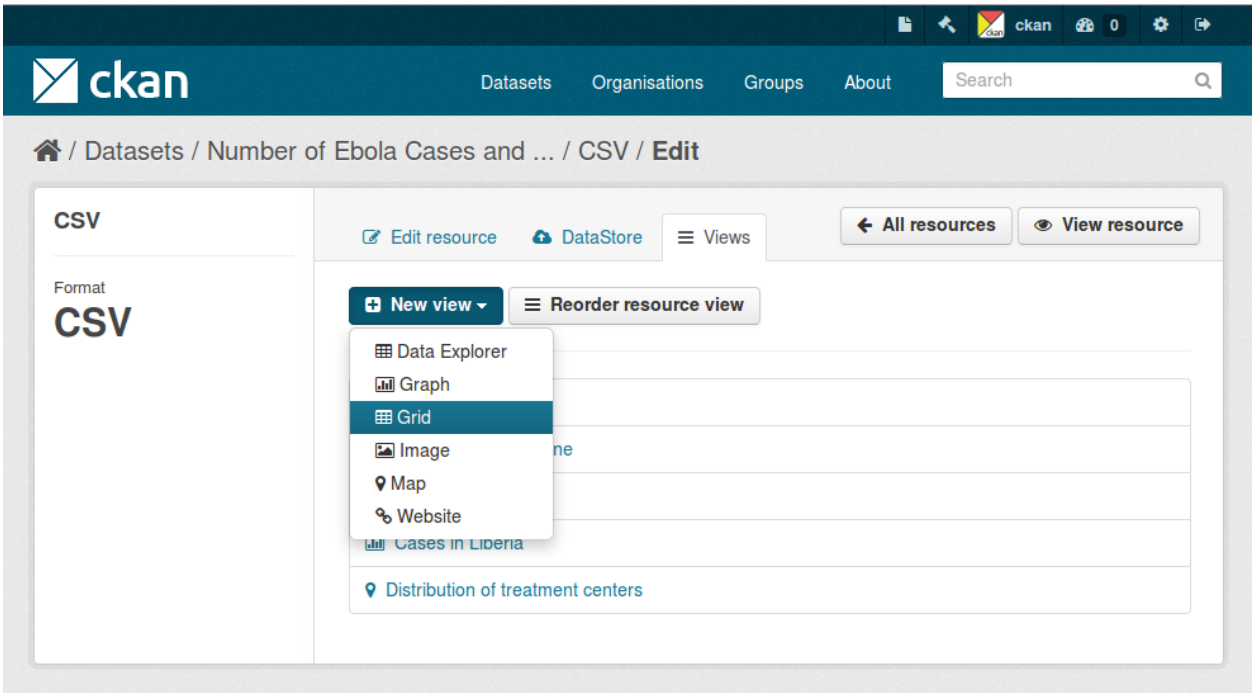
- One resource can have multiple views of the same data (for example a grid and some graphs for tabular data).
- Dataset editors can choose which views to show, reorder them and configure them individually.
- Individual views can be embedded on external sites.

Different view types are implemented via custom plugins, which can be activated on a particular CKAN site. Once these plugins are added, instance administrators can decide which views should be created by default if the resource is suitable (for instance a table on resources uploaded to the DataStore, a map for spatial data, etc.).

Whether a particular resource can be rendered by the different view plugins is decided by the view plugins themselves. This is generally done checking the resource format or whether its data is on the *DataStore extension* or not.

3.8.2 Managing resource views

Users who are allowed to edit a particular dataset can also manage the views for its resources. To access the management interface, click on the *Manage* button on the resource page and then on the *Views* tab. From here you can create new views, update or delete existing ones and reorder them.



The *New view* dropdown will show the available view types for this particular resource. If the list is empty, you may need to add the relevant view plugins to the *ckan.plugins* setting on your configuration file, eg:

```
ckan.plugins = ... image_view datatables_view pdf_view
```

3.8.3 Defining views to appear by default

From the management interface you can create and edit views manually, but in most cases you will want views to be created automatically on certain resource types, so data can be visualized straight away after uploading or linking to a file.

To do so, you define a set of view plugins that should be checked whenever a dataset or resource is created or updated. For each of them, if the resource is a suitable one, a view will be created.

This is configured with the *ckan.views.default_views* setting. In it you define the view plugins that you want to be created as default:

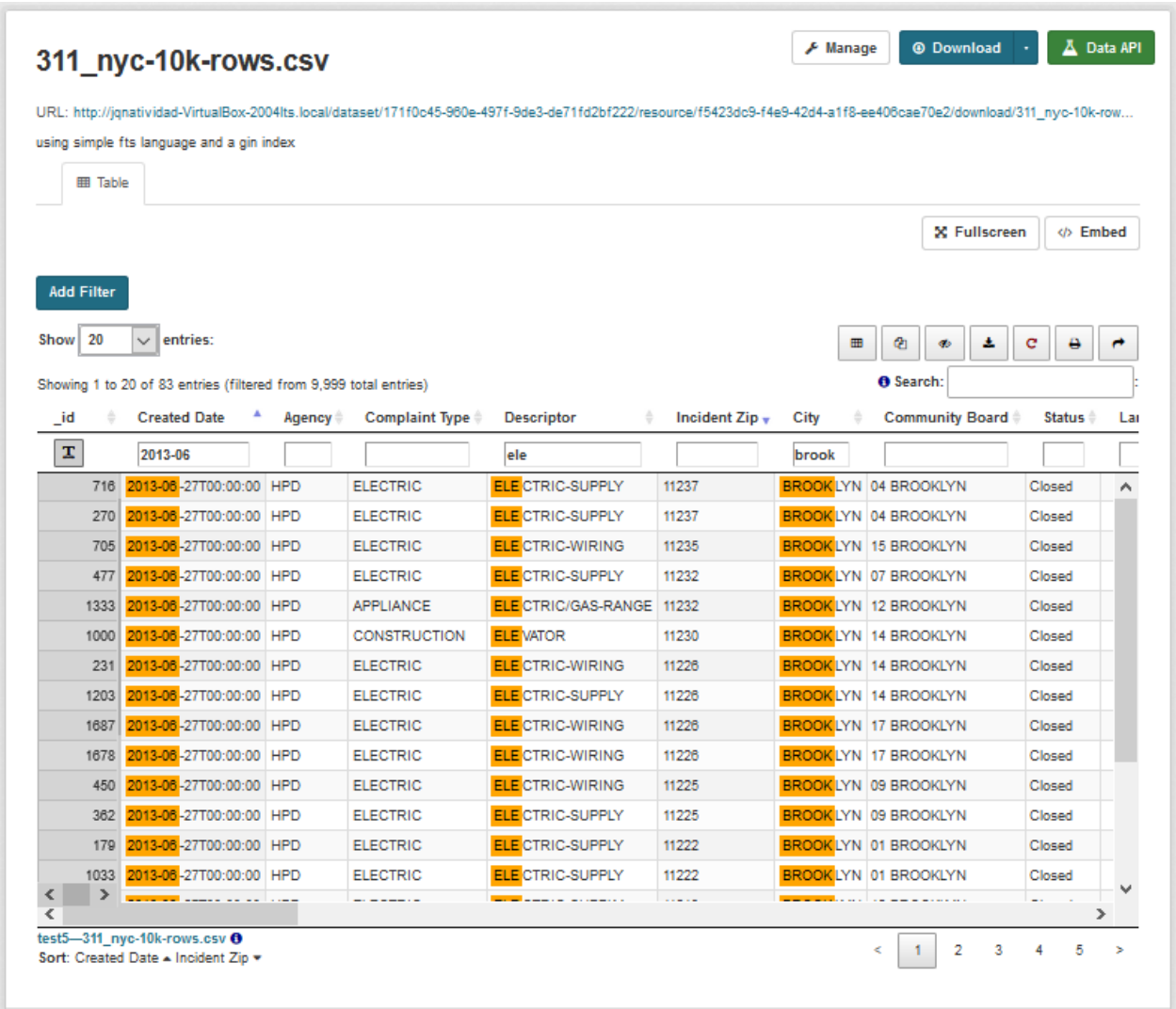
```
ckan.views.default_views = datatables_view pdf_view geojson_view
```

This configuration does not mean that each new resource will get all of these views by default, but that for instance if the uploaded file is a PDF file, a PDF viewer will be created automatically and so on.

3.8.4 Available view plugins

Some view plugins for common formats are included in the main CKAN repository. These don't require further setup and can be directly added to the *ckan.plugins* setting.

DataTables view



View plugin: `datatables_view`

Displays a filterable, sortable, table view of structured data using the DataTables jQuery plugin, with the following features.

- Search highlighting
- Column Filters
- Multi-column sorting
- Two view modes (table/list). Table shows the data in a typical grid with horizontal scrolling. List displays the data in a responsive mode, with a Record Details view.
- Filtered Downloads
- Column Visibility control
- Copy to clipboard and Printing of filtered results and row selection/s
- Drag-and-drop column reordering

- State Saving - saves search keywords, column order/visibility, row selections and page settings between session, with the ability to share saved searches.
- Data Dictionary Integration
- Automatic "linkification" of URLs
- Automatic creation of zoomable thumbnails when a cell only contains a URL to an image.
- Available automatic, locale-aware date formatting to convert raw ISO-8601 timestamps to a user-friendly date format

It is designed not only as a data viewer, but also as a simple ad-hoc report generator - allowing users to quickly find an actionable subset of the data they need from inside the resource view, without having to first download the dataset.

It's also optimized for embedding datasets and saved searches on external sites - with a backlink to the portal and automatic resizing.

This plugin requires data to be in the DataStore.

Text view

View plugin: `text_view`

Displays files in XML, JSON or plain text based formats with the syntax highlighted. The formats detected can be configured using the *ckan.preview.xml_formats*, *ckan.preview.json_formats* and *ckan.preview.text_formats* configuration options respectively.

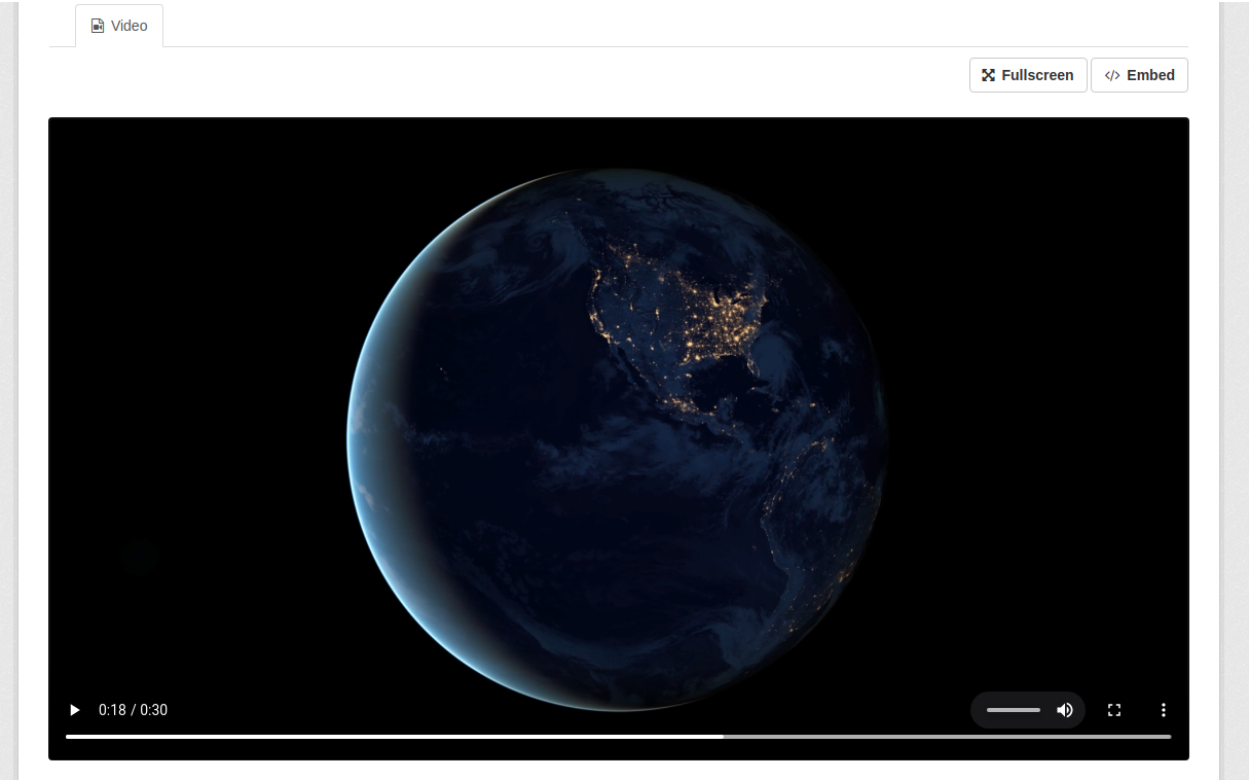
If you want to display files that are hosted in a different server from your CKAN instance (eg that haven't been uploaded to CKAN) you will need to enable the *Resource Proxy* plugin.

Image view

View plugin: `image_view`

If the resource format is a common image format like PNG, JPEG or GIF, it adds an `<img>` tag pointing to the resource URL. You can provide an alternative URL on the edit view form. The available formats can be configured using the *ckan.preview.image_formats* configuration option.

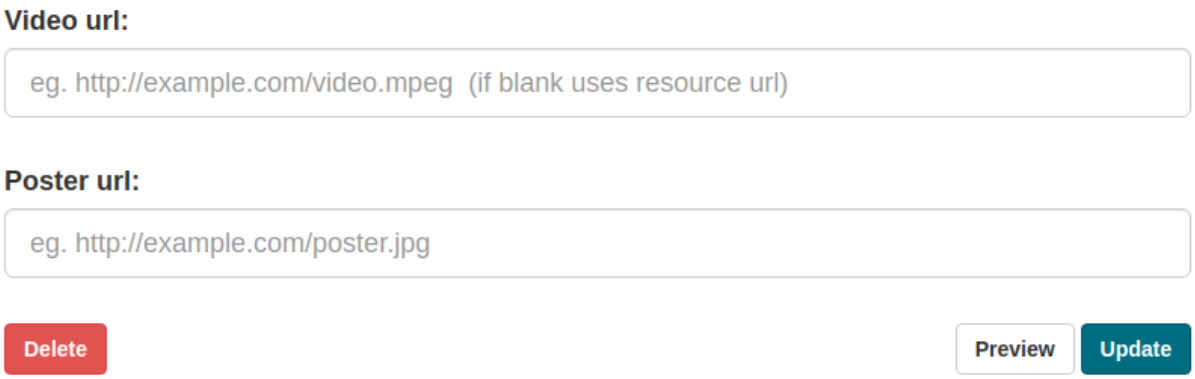
Video view



View plugin: `video_view`

This plugin uses the HTML5 <video> tag to embed video content into a page, such as movie clip or other video streams.

There are three supported video formats: MP4, WebM, and OGG.



You can provide an alternative URL on the edit view form. Otherwise, the resource link will be used.

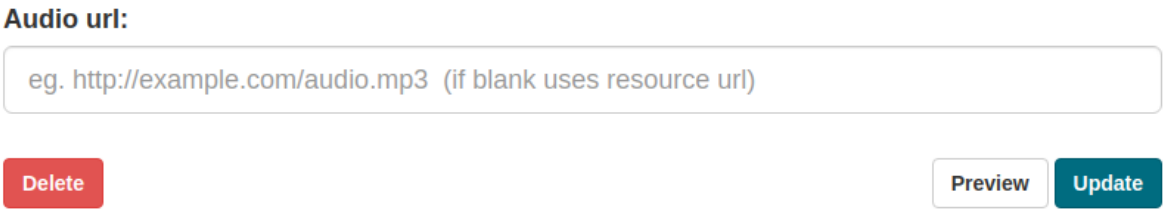
Also, you can provide a poster image URL. The poster image will be shown while the video is downloading, or until the user hits the play button. If this is not provided, the first frame of the video will be used instead.

Audio view

View plugin: `audio_view`

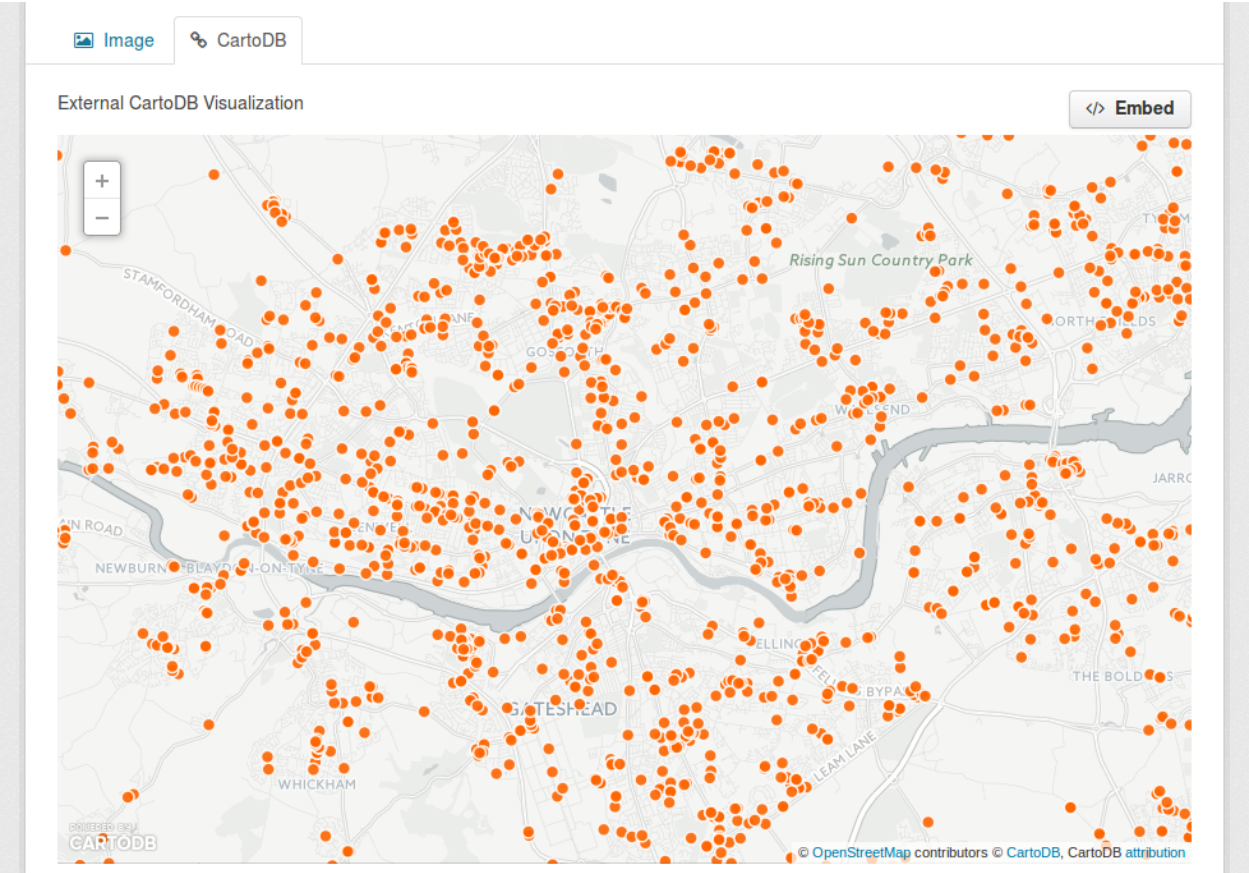
This plugin uses the HTML5 audio tag to embed an audio player on the page.

Since we rely on HTML5 <audio> tag, there are three supported audio formats: MP3, WAV, and OGG. Notice. Browsers don't all support the same file types and audio codecs.



You can provide an alternative URL on the edit view form. Otherwise, the resource link will be used.

Web page view



View plugin: `webpage_view`

Adds an `<iframe>` tag to embed the resource URL. You can provide an alternative URL on the edit view form.

> **Warning:** Do not activate this plugin unless you trust the URL sources. It is not recommended to enable this view type on instances where all users can create datasets.

3.8.5 Other view plugins

There are many more view plugins developed by the CKAN community, which are hosted on separate repositories. Some examples include:

- React Data explorer: A modern data explorer, maintained by Datopian.
- Ckanext Visualize: An extension to easily create user visualization from data in the DataStore, maintained by Keitaro.
- Dashboard: Allows to combine multiple views into a single dashboard.
- PDF viewer: Allows to render PDF files on the resource page.
- Geo viewer: Renders various spatial formats like GeoJSON, WMS or shapefiles in an interactive map.
- Choropleth map: Displays data on the DataStore on a choropleth map.

- Basic charts: Provides alternative graph types and renderings.

If you want to add another view type to this list, edit this file by sending a pull request on GitHub.

New plugins to render custom view types can be implemented using the `IResourceView` interface.

Todo: Link to a proper tutorial for writing custom views

3.8.6 Resource Proxy

As resource views are rendered on the browser, if the file they are accessing is located in a different domain than the one CKAN is hosted, the browser will block access to it because of the same-origin policy. For instance, files hosted on *www.example.com* won't be able to be accessed from the browser if CKAN is hosted on *data.catalog.com*.

To allow view plugins access to external files you need to activate the `resource_proxy` plugin on your configuration file:

```
ckan.plugins = resource_proxy ...
```

This will request the file on the server side and serve it from the same domain as CKAN.

You can modify the maximum allowed size for proxied files using the *ckan.resource_proxy.max_file_size* configuration setting.

3.8.7 Migrating from previous CKAN versions

If you are upgrading an existing instance running CKAN version 2.2.x or lower to CKAN 2.3 or higher, you need to perform a migration process in order for the resource views to appear. If the migration does not take place, resource views will only appear when creating or updating datasets or resources, but not on existing ones.

The migration process involves creating the necessary view objects in the database, which can be done using the `ckan views create` command.

Note: The `ckan views create` command uses the search API to get all necessary datasets and resources, so make sure your search index *is up to date* before starting the migration process.

The way the `ckan views create` commands works is getting all or a subset of the instance datasets from the search index, and for each of them checking against a list of view plugins if it is necessary to create a view object. This gets determined by each of the individual view plugins depending on the dataset's resources fields.

Before each run, you will be prompted with the number of datasets affected and asked if you want to continue (unless you pass the `-y` option):

```
You are about to check 3336 datasets for the following view plugins: ['image_view',
↪'datatables_view', 'text_view']
 Do you want to continue? [Y/n]
```

Note: On large CKAN instances the migration process can take a significant time if using the default options. It is worth planning in advance and split the process using the search parameters to only check relevant datasets. The following documentation provides guidance on how to do this.

If no view types are provided, the default ones are used (check *Defining views to appear by default* to see how these are defined):

```
ckan -c |ckan.ini| views create
```

Specific view types can be also provided:

```
ckan -c |ckan.ini| views create image_view datatables_view pdf_view
```

For certain view types (the ones with plugins included in the main CKAN core), default filters are applied to the search to only get relevant resources. For instance if `image_view` is defined, filters are added to the search to only get datasets with resources that have image formats (png, jpg, etc).

You can also provide arbitrary search parameters like the ones supported by `package_search()`. This can be useful for instance to only include datasets with resources of a certain format:

```
ckan -c |ckan.ini| views create geojson_view -s '{"fq": "res_format:GEOJSON"}'
```

To instead avoid certain formats you can do:

```
ckan -c |ckan.ini| views create -s '{"fq": "-res_format:HTML"}'
```

Of course this is not limited to resource formats, you can filter out or in using any field, as in a normal dataset search:

```
ckan -c |ckan.ini| views create -s '{"q": "groups:visualization-examples"}'
```

Tip: If you set the `ckan_logger` level to `DEBUG` on your configuration file you can see the full search parameters being sent to Solr.

For convenience, there is also an option to create views on a particular dataset or datasets:

```
ckan -c |ckan.ini| views create -d dataset_id

ckan -c |ckan.ini| views create -d dataset_name -d dataset_name
```

3.8.8 Command line interface

The `ckan views` command allows to create and remove resource views objects from the database in bulk.

Check the command help for the full options:

```
ckan -c |ckan.ini| views create -h
```

Todo: Tutorial for writing custom view types.

3.9 FileStore and file uploads

When enabled, CKAN's FileStore allows users to upload data files to CKAN resources, and to upload logo images for groups and organizations. Users will see an upload button when creating or updating a resource, group or organization.

New in version 2.2: Uploading logo images for groups and organizations was added in CKAN 2.2.

Changed in version 2.2: Previous versions of CKAN used to allow uploads to remote cloud hosting but we have simplified this to only allow local file uploads (see *Migration from 2.1 to 2.2* for details on how to migrate). This is to give CKAN more control over the files and make access control possible.

See also:

DataStore extension

Resource files linked-to from CKAN or uploaded to CKAN's FileStore can also be pushed into CKAN's DataStore, which then enables data previews and a data API for the resources.

3.9.1 Setup file uploads

To setup CKAN's FileStore with local file storage:

1. Create the directory where CKAN will store uploaded files:

   `sudo mkdir -p /var/lib/ckan/default`

2. Add the following line to your CKAN config file, after the `[app:main]` line:

   `ckan.storage_path = /var/lib/ckan/default`

3. Set the permissions of your *ckan.storage_path* directory. For example if you're running CKAN with Nginx, then the Nginx's user (`www-data` on Ubuntu) must have read, write and execute permissions for the *ckan.storage_path*:

   ```
   sudo chown www-data /var/lib/ckan/default
   sudo chmod u+rwx /var/lib/ckan/default
   ```

4. Restart your web server, for example to restart uWSGI on a package install:

   ```
   sudo supervisorctl restart ckan-uwsgi:*
   ```

3.9.2 FileStore API

Changed in version 2.2: The FileStore API was redesigned for CKAN 2.2. The previous API has been deprecated.

Files can be uploaded to the FileStore using the `resource_create()` and `resource_update()` action API functions. You can post multipart/form-data to the API and the key, value pairs will be treated as if they are a JSON object. The extra key `upload` is used to actually post the binary data.

For example, to create a new CKAN resource and upload a file to it using curl:

```
curl -H'Authorization: your-api-key' 'http://yourhost/api/action/resource_create' --form␣
↪upload=@filetoupload --form package_id=my_dataset
```

(Curl automatically sends a `multipart-form-data` heading with you use the `--form` option.)

To create a new resource and upload a file to it using the Python library requests:

```
import requests
requests.post('http://0.0.0.0:5000/api/action/resource_create',
              data={"package_id":"my_dataset"},
              headers={"Authorization": "21a47217-6d7b-49c5-88f9-72ebd5a4d4bb"},
              files=[('upload', open('/path/to/file/to/upload.csv', 'rb'))])
```

(Requests automatically sends a `multipart-form-data` heading when you use the `files=` parameter.)

To overwrite an uploaded file with a new version of the file, post to the *`resource_update()`* action and use the `upload` field:

```
curl -H'Authorization: your-api-key' 'http://yourhost/api/action/resource_update' --form␣
↪upload=@newfiletoupload --form id=resourceid
```

To replace an uploaded file with a link to a file at a remote URL, use the `clear_upload` field:

```
curl -H'Authorization: your-api-key' 'http://yourhost/api/action/resource_update' --form␣
↪url=http://expample.com --form clear_upload=true --form id=resourceid
```

3.9.3 Migration from 2.1 to 2.2

If you are using pairtree local file storage then you can keep your current settings without issue. The pairtree and new storage can live side by side but you are still encouraged to migrate. If you change your config options to the ones specified in this doc you will need to run the migration below.

If you are running remote storage then all previous links will still be accessible but if you want to move the remote storage documents to the local storage you will run the migration also.

In order to migrate make sure your CKAN instance is running as the script will request the data from the instance using APIs. You need to run the following on the command line to do the migration:

```
ckan -c |ckan.ini| db migrate-filestore
```

This may take a long time especially if you have a lot of files remotely. If the remote hosting goes down or the job is interrupted it is saved to run it again and it will try all the unsuccessful ones again.

3.9.4 Custom Internet media types (MIME types)

New in version 2.2.

CKAN uses the default Python library mimetypes to detect the media type of an uploaded file. If some particular format is not included in the ones guessed by the `mimetypes` library, a default `application/octet-stream` value will be returned.

Users can still register a more appropriate media type by using the `mimetypes` library. A good way to do so is to use the `IConfigurer` interface so the custom types get registered on startup:

```
import mimetypes
import ckan.plugins as p

class MyPlugin(p.SingletonPlugin):

    p.implements(p.IConfigurer)
```

(continues on next page)

(continued from previous page)

```
    def update_config(self, config):

        mimetypes.add_type('application/json', '.geojson')

        # ...
```

3.10 DataStore extension

The CKAN DataStore extension provides an *ad hoc* database for storage of structured data from CKAN resources. Data can be pulled out of resource files and stored in the DataStore.

When a resource is added to the DataStore, you get:

- Automatic data previews on the resource's page, using for instance the *DataTables view extension*
- *The Data API*: search, filter and update the data, without having to download and upload the entire data file

The DataStore is integrated into the *CKAN API* and authorization system.

The DataStore is generally used alongside the DataPusher, which will automatically upload data to the DataStore from suitable files, whether uploaded to CKAN's FileStore or externally linked.

- *Relationship to FileStore*
- *Setting up the DataStore*
- *DataPusher: Automatically Add Data to the DataStore*
- *Data Dictionary*
- *Downloading Resources*
- *The Data API*
- *Extending DataStore*

3.10.1 Relationship to FileStore

The DataStore is distinct but complementary to the FileStore (see *FileStore and file uploads*). In contrast to the FileStore which provides 'blob' storage of whole files with no way to access or query parts of that file, the DataStore is like a database in which individual data elements are accessible and queryable. To illustrate this distinction, consider storing a spreadsheet file like a CSV or Excel document. In the FileStore this file would be stored directly. To access it you would download the file as a whole. By contrast, if the spreadsheet data is stored in the DataStore, one would be able to access individual spreadsheet rows via a simple web API, as well as being able to make queries over the spreadsheet contents.

3.10.2 Setting up the DataStore

Changed in version 2.6: Previous CKAN (and DataStore) versions were compatible with earlier versions of PostgreSQL.

1. Enable the plugin

Add the `datastore` plugin to your CKAN config file:

```
ckan.plugins = datastore
```

2. Set-up the database

Warning: Make sure that you follow the steps in *Set Permissions* below correctly. Wrong settings could lead to serious security issues.

The DataStore requires a separate PostgreSQL database to save the DataStore resources to.

List existing databases:

```
sudo -u postgres psql -l
```

Check that the encoding of databases is `UTF8`, if not internationalisation may be a problem. Since changing the encoding of PostgreSQL may mean deleting existing databases, it is suggested that this is fixed before continuing with the datastore setup.

Create users and databases

Tip: If your CKAN database and DataStore databases are on different servers, then you need to create a new database user on the server where the DataStore database will be created. As in *Installing CKAN from source* we'll name the database user ckan_default:

```
sudo -u postgres createuser -S -D -R -P -l ckan_default
```

Create a database_user called datastore_default. This user will be given read-only access to your DataStore database in the *Set Permissions* step below:

```
sudo -u postgres createuser -S -D -R -P -l datastore_default
```

Create the database (owned by ckan_default), which we'll call datastore_default:

```
sudo -u postgres createdb -O ckan_default datastore_default -E utf-8
```

Set URLs

Now, uncomment the *ckan.datastore.write_url* and *ckan.datastore.read_url* lines in your CKAN config file and edit them if necessary, for example:

```
ckan.datastore.write_url = postgresql://ckan_default:pass@localhost/datastore_default
ckan.datastore.read_url = postgresql://datastore_default:pass@localhost/datastore_default
```

Replace `pass` with the passwords you created for your ckan_default and datastore_default database users.

Set permissions

Once the DataStore database and the users are created, the permissions on the DataStore and CKAN database have to be set. CKAN provides a ckan command to help you correctly set these permissions.

If you are able to use the `psql` command to connect to your database as a superuser, you can use the `datastore set-permissions` command to emit the appropriate SQL to set the permissions.

For example, if you can connect to your database server as the `postgres` superuser using:

```
sudo -u postgres psql
```

Then you can use this connection to set the permissions:

```
ckan -c /etc/ckan/default/ckan.ini datastore set-permissions | sudo -u postgres␣
↪psql --set ON_ERROR_STOP=1
```

Note: If you performed a package install, you will need to replace all references to 'ckan -c /etc/ckan/default/ckan.ini …' with 'sudo ckan …' and provide the path to the config file, e.g.:

```
sudo ckan datastore set-permissions | sudo -u postgres psql --set ON_ERROR_STOP=1
```

If your database server is not local, but you can access it over SSH, you can pipe the permissions script over SSH:

```
ckan -c /etc/ckan/default/ckan.ini datastore set-permissions | ssh dbserver sudo -u␣
↪postgres psql --set ON_ERROR_STOP=1
```

If you can't use the `psql` command in this way, you can simply copy and paste the output of:

```
ckan -c /etc/ckan/default/ckan.ini datastore set-permissions
```

into a PostgreSQL superuser console.

3. Test the set-up

The DataStore is now set-up. To test the set-up, (re)start CKAN and run the following command to list all DataStore resources:

```
curl -X GET "http://127.0.0.1:5000/api/3/action/datastore_search?resource_id=_table_
↪metadata"
```

This should return a JSON page without errors.

To test the whether the set-up allows writing, you can create a new DataStore resource. To do so, run the following command:

```
curl -X POST http://127.0.0.1:5000/api/3/action/datastore_create -H "Authorization:
↪{YOUR-API-KEY}" -d '{"resource": {"package_id": "{PACKAGE-ID}"}, "fields": [ {"id": "a
↪"}, {"id": "b"} ], "records": [ { "a": 1, "b": "xyz"}, {"a": 2, "b": "zzz"} ]}'
```

Replace `{YOUR-API-KEY}` with a valid API key and `{PACKAGE-ID}` with the id of an existing CKAN dataset.

A table named after the resource id should have been created on your DataStore database. Visiting this URL should return a response from the DataStore with the records inserted above:

```
http://127.0.0.1:5000/api/3/action/datastore_search?resource_id={RESOURCE_ID}
```

Replace `{RESOURCE-ID}` with the resource id that was returned as part of the response of the previous API call.

You can now delete the DataStore table with:

```
curl -X POST http://127.0.0.1:5000/api/3/action/datastore_delete -H "Authorization:
↪{YOUR-API-KEY}" -d '{"resource_id": "{RESOURCE-ID}"}'
```

To find out more about the Data API, see *The Data API*.

3.10.3 DataPusher: Automatically Add Data to the DataStore

Often, one wants data that is added to CKAN (whether it is linked to or uploaded to the *FileStore*) to be automatically added to the DataStore. This requires some processing, to extract the data from your files and to add it to the DataStore in the format the DataStore can handle.

This task of automatically parsing and then adding data to the DataStore is performed by the DataPusher, a service that runs asynchronously and can be installed alongside CKAN.

To install this please look at the docs here: https://github.com/ckan/datapusher

Note: The DataPusher only imports the first worksheet of a spreadsheet. It also does not support duplicate column headers. That includes blank column headings.

3.10.4 Data Dictionary

DataStore columns may be described with a Data Dictionary. A Data Dictionary tab will appear when editing any resource with a DataStore table. The Data Dictionary form allows entering the following values for each column:

- **Type Override:** the type to be used the next time DataPusher is run to load data into this column
- **Label:** a human-friendly label for this column
- **Description:** a full description for this column in markdown format

Extension developers may add new fields to this form by overriding the default Data Dictionary form template `datastore/snippets/dictionary_form.html`.

The Data Dictionary is set through the API as part of the *Fields* passed to `datastore_create()` and returned from `datastore_search()`.

3.10.5 Downloading Resources

A DataStore resource can be downloaded in the CSV file format from `{CKAN-URL}/datastore/dump/{RESOURCE-ID}`.

For an Excel-compatible CSV file use `{CKAN-URL}/datastore/dump/{RESOURCE-ID}?bom=true`.

Other formats supported include tab-separated values (`?format=tsv`), JSON (`?format=json`) and XML (`?format=xml`). E.g. to download an Excel-compatible tab-separated file use `{CKAN-URL}/datastore/dump/{RESOURCE-ID}?format=tsv&bom=true`.

A number of parameters from *`datastore_search()`* can be used:

`offset`, `limit`, `filters`, `q`, `full_text`, `distinct`, `plain`, `language`, `fields`, `sort`

3.10.6 The Data API

The CKAN DataStore offers an API for reading, searching and filtering data without the need to download the entire file first. The DataStore is an ad hoc database which means that it is a collection of tables with unknown relationships. This allows you to search in one DataStore resource (a *table* in the database) as well as queries across DataStore resources.

Data can be written incrementally to the DataStore through the API. New data can be inserted, existing data can be updated or deleted. You can also add a new column to an existing table even if the DataStore resource already contains some data.

Triggers may be added to enforce validation, clean data as it is loaded or even record histories. Triggers are PL/pgSQL functions that must be created by a sysadmin.

You will notice that we tried to keep the layer between the underlying PostgreSQL database and the API as thin as possible to allow you to use the features you would expect from a powerful database management system.

A DataStore resource can not be created on its own. It is always required to have an associated CKAN resource. If data is stored in the DataStore, it can automatically be previewed by a *preview extension*.

Making a Data API request

Making a Data API request is the same as making an Action API request: you post a JSON dictionary in an HTTP POST request to an API URL, and the API also returns its response in a JSON dictionary. See the *API guide* for details.

API reference

Note: Lists can always be expressed in different ways. It is possible to use lists, comma separated strings or single items. These are valid lists: `['foo', 'bar']`, `'foo, bar'`, `"foo", "bar"` and `'foo'`. Additionally, there are several ways to define a boolean value. `True`, `on` and `1` are all vaid boolean values.

Note: The table structure of the DataStore is explained in *Internal structure of the database*.

`ckanext.datastore.logic.action.`**`datastore_create`**(*context: Context*, *data_dict: dict[str, Any]*)

Adds a new table to the DataStore.

The datastore_create action allows you to post JSON data to be stored against a resource. This endpoint also supports altering tables, aliases and indexes and bulk insertion. This endpoint can be called multiple times to initially insert more data, add/remove fields, change the aliases or indexes as well as the primary keys.

To create an empty datastore resource and a CKAN resource at the same time, provide `resource` with a valid `package_id` and omit the `resource_id`.

If you want to create a datastore resource from the content of a file, provide `resource` with a valid `url`.

See *Fields* and *Records* for details on how to lay out records.

Parameters

- **resource_id** (*string*) – resource id that the data is going to be stored against.
- **force** (*bool (optional, default: False)*) – set to True to edit a read-only resource
- **resource** (*dictionary*) – resource dictionary that is passed to *resource_create()*. Use instead of `resource_id` (optional)
- **aliases** (*list or comma separated string*) – names for read only aliases of the resource. (optional)
- **fields** (*list of dictionaries*) – fields/columns and their extra metadata. (optional)
- **delete_fields** (*bool (optional, default: False)*) – set to True to remove existing fields not passed
- **records** (*list of dictionaries*) – the data, eg: [{"dob": "2005", "some_stuff": ["a", "b"]}] (optional)
- **primary_key** (*list or comma separated string*) – fields that represent a unique key (optional)
- **indexes** (*list or comma separated string*) – indexes on table (optional)
- **triggers** (*list of dictionaries*) – trigger functions to apply to this table on update/insert. functions may be created with *datastore_function_create()*. eg: [{"function": "trigger_clean_reference"}, {"function": "trigger_check_codes"}]
- **calculate_record_count** (*bool (optional, default: False)*) – updates the stored count of records, used to optimize datastore_search in combination with the *total_estimation_threshold* parameter. If doing a series of requests to change a resource, you only need to set this to True on the last request.

Please note that setting the `aliases`, `indexes` or `primary_key` replaces the existing aliases or constraints. Setting `records` appends the provided records to the resource. Setting `fields` without including all existing fields will remove the others and the data they contain.

Results:

Returns
The newly created data object, excluding `records` passed.

Return type
dictionary

See *Fields* and *Records* for details on how to lay out records.

`ckanext.datastore.logic.action.`**`datastore_run_triggers`**(*context: Context*, *data_dict: dict[str, Any]*) → int

update each record with trigger

The datastore_run_triggers API action allows you to re-apply existing triggers to an existing DataStore resource.

Parameters
resource_id (*string*) – resource id that the data is going to be stored under.

Results:

Returns
The rowcount in the table.

Return type
int

`ckanext.datastore.logic.action.datastore_upsert(context: Context, data_dict: dict[str, Any])`

Updates or inserts into a table in the DataStore

The datastore_upsert API action allows you to add or edit records to an existing DataStore resource. In order for the *upsert* and *update* methods to work, a unique key has to be defined via the datastore_create action. The available methods are:

upsert
Update if record with same key already exists, otherwise insert. Requires unique key or _id field.

insert
Insert only. This method is faster that upsert, but will fail if any inserted record matches an existing one. Does *not* require a unique key.

update
Update only. An exception will occur if the key that should be updated does not exist. Requires unique key or _id field.

Parameters

- **resource_id** (*string*) – resource id that the data is going to be stored under.
- **force** (*bool (optional, default: False)*) – set to True to edit a read-only resource
- **records** (*list of dictionaries*) – the data, eg: [{"dob": "2005", "some_stuff": ["a","b"]}] (optional)
- **method** (*string*) – the method to use to put the data into the datastore. Possible options are: upsert, insert, update (optional, default: upsert)
- **calculate_record_count** (*bool (optional, default: False)*) – updates the stored count of records, used to optimize datastore_search in combination with the *total_estimation_threshold* parameter. If doing a series of requests to change a resource, you only need to set this to True on the last request.
- **dry_run** (*bool (optional, default: False)*) – set to True to abort transaction instead of committing, e.g. to check for validation or type errors.

Results:

Returns
The modified data object.

Return type
dictionary

`ckanext.datastore.logic.action.datastore_info(context: Context, data_dict: dict[str, Any]) → dict[str, Any]`

Returns detailed metadata about a resource.

Parameters
resource_id (*string*) – id or alias of the resource we want info about.

Results:

Return type
dictionary

Returns

meta: resource metadata dictionary with the following keys:

- aliases - aliases (views) for the resource
- count - row count
- db_size - size of the datastore database (bytes)
- id - resource id (useful for dereferencing aliases)
- idx_size - size of all indices for the resource (bytes)
- size - size of resource (bytes)
- table_type - BASE TABLE, VIEW, FOREIGN TABLE or MATERIALIZED VIEW

fields: A list of dictionaries based on *Fields*, with an additional nested dictionary per field called **schema**, with the following keys:

- native_type - native database data type
- index_name
- is_index
- notnull
- uniquekey

`ckanext.datastore.logic.action.`**`datastore_delete`**(*context: Context*, *data_dict: dict[str, Any]*)

Deletes a table or a set of records from the DataStore. (Use `datastore_records_delete()` to keep tables intact)

Parameters

- **`resource_id`** (*`string`*) – resource id that the data will be deleted from. (optional)
- **`force`** (*`bool (optional, default: False)`*) – set to True to edit a read-only resource
- **`filters`** (*`dictionary`*) – *Filters* to apply before deleting (eg {"name": "fred"}). If missing delete whole table and all dependent views. (optional)
- **`calculate_record_count`** (*`bool (optional, default: False)`*) – updates the stored count of records, used to optimize datastore_search in combination with the *total_estimation_threshold* parameter. If doing a series of requests to change a resource, you only need to set this to True on the last request.

Results:

Returns
Original filters sent.

Return type
dictionary

`ckanext.datastore.logic.action.`**`datastore_records_delete`**(*context: Context*, *data_dict: dict[str, Any]*)

Deletes records from a DataStore table but will never remove the table itself.

Parameters

- **`resource_id`** (*`string`*) – resource id that the data will be deleted from. (required)

- **force** (*bool (optional, default: False)*) – set to True to edit a read-only resource
- **filters** (*dictionary*) – *Filters* to apply before deleting (eg {"name": "fred"}). If {} delete all records. (required)
- **calculate_record_count** (*bool (optional, default: False)*) – updates the stored count of records, used to optimize datastore_search in combination with the *total_estimation_threshold* parameter. If doing a series of requests to change a resource, you only need to set this to True on the last request.

Results:

Returns
Original filters sent.

Return type
dictionary

`ckanext.datastore.logic.action.`**`datastore_search`**(*context: Context*, *data_dict: dict[str, Any]*)

Search a DataStore resource.

The datastore_search action allows you to search data in a resource. By default 100 rows are returned - see the *limit* parameter for more info.

A DataStore resource that belongs to a private CKAN resource can only be read by you if you have access to the CKAN resource and send the appropriate authorization.

Parameters

- **resource_id** (*string*) – id or alias of the resource to be searched against
- **filters** (*dictionary*) – *Filters* for matching conditions to select, e.g {"key1": "a", "key2": "b"} (optional)
- **q** (*string or dictionary*) – full text query. If it's a string, it'll search on all fields on each row. If it's a dictionary as {"key1": "a", "key2": "b"}, it'll search on each specific field (optional)
- **full_text** (*string*) – full text query. It search on all fields on each row. This should be used in replace of `q` when performing string search accross all fields
- **distinct** (*bool*) – return only distinct rows (optional, default: false)
- **plain** (*bool*) – treat as plain text query (optional, default: true)
- **language** (*string*) – language of the full text query (optional, default: english)
- **limit** (*int*) – maximum number of rows to return (optional, default: `100`, unless set in the site's configuration `ckan.datastore.search.rows_default`, upper limit: `32000` unless set in site's configuration `ckan.datastore.search.rows_max`)
- **offset** (*int*) – offset this number of rows (optional)
- **fields** (*list or comma separated string*) – fields to return (optional, default: all fields in original order)
- **sort** (*string*) – comma separated field names with ordering e.g.: "fieldname1, fieldname2 desc nulls last"
- **include_total** (*bool*) – True to return total matching record count (optional, default: true)
- **total_estimation_threshold** (*int or None*) – If "include_total" is True and "total_estimation_threshold" is not None and the estimated total (matching record count) is above the "total_estimation_threshold" then this datastore_search will return an *estimate* of the total, rather than a precise one. This is often good enough, and saves computationally

expensive row counting for larger results (e.g. >100000 rows). The estimated total comes from the PostgreSQL table statistics, generated when Express Loader or DataPusher finishes a load, or by autovacuum. NB Currently estimation can't be done if the user specifies 'filters' or 'distinct' options. (optional, default: None)

- **records_format** (*controlled list*) – the format for the records return value: 'objects' (default) list of {fieldname1: value1, ...} dicts, 'lists' list of [value1, value2, ...] lists, 'csv' string containing comma-separated values with no header, 'tsv' string containing tab-separated values with no header

Setting the `plain` flag to false enables the entire PostgreSQL full text search query language.

A listing of all available resources can be found at the alias `_table_metadata`.

If you need to download the full resource, read *Downloading Resources*.

Results:

The result of this action is a dictionary with the following keys:

Return type
A dictionary with the following keys

Parameters

- **fields** (*list of dictionaries*) – fields/columns and their extra metadata
- **offset** (*int*) – query offset value
- **limit** (*int*) – queried limit value (if the requested `limit` was above the `ckan.datastore.search.rows_max` value then this response `limit` will be set to the value of `ckan.datastore.search.rows_max`)
- **filters** (*list of dictionaries*) – query filters
- **total** (*int*) – number of total matching records
- **total_was_estimated** (*bool*) – whether or not the total was estimated
- **records** (*depends on records_format value passed*) – list of matching results

`ckanext.datastore.logic.action.`**datastore_search_sql**(*context: Context*, *data_dict: dict[str, Any]*)

Execute SQL queries on the DataStore.

The datastore_search_sql action allows a user to search data in a resource or connect multiple resources with join expressions. The underlying SQL engine is the PostgreSQL engine. There is an enforced timeout on SQL queries to avoid an unintended DOS. The number of results returned is limited to 32000, unless set in the site's configuration `ckan.datastore.search.rows_max` Queries are only allowed if you have access to the all the CKAN resources in the query and send the appropriate authorization.

Note: This action is not available by default and needs to be enabled with the *ckan.datastore.sqlsearch.enabled* setting.

Note: When source data columns (i.e. CSV) heading names are provided in all UPPERCASE you need to double quote them in the SQL select statement to avoid returning null results.

Parameters
sql (*string*) – a single SQL select statement

Results:

The result of this action is a dictionary with the following keys:

Return type
A dictionary with the following keys

Parameters

- **fields** (*list of dictionaries*) – fields/columns and their extra metadata
- **records** (*list of dictionaries*) – list of matching results
- **records_truncated** (*bool*) – indicates whether the number of records returned was limited by the internal limit, which is 32000 records (or other value set in the site's configuration `ckan.datastore.search.rows_max`). If records are truncated by this, this key has value True, otherwise the key is not returned at all.

`ckanext.datastore.logic.action.`**`set_datastore_active_flag`**(*context: Context*, *data_dict: dict[str, Any]*, *flag: bool*)

Set appropriate datastore_active flag on CKAN resource.

Called after creation or deletion of DataStore table.

`ckanext.datastore.logic.action.`**`datastore_function_create`**(*context: Context*, *data_dict: dict[str, Any]*)

Create a trigger function for use with datastore_create

Parameters

- **name** (*string*) – function name
- **or_replace** (*bool*) – True to replace if function already exists (default: False)
- **rettype** (*string*) – set to 'trigger' (only trigger functions may be created at this time)
- **definition** (*string*) – PL/pgSQL function body for trigger function

`ckanext.datastore.logic.action.`**`datastore_function_delete`**(*context: Context*, *data_dict: dict[str, Any]*)

Delete a trigger function

Parameters
name (*string*) – function name

Fields

Fields define the column names and the type of the data in a column. A field is defined as follows:

```
{
    "id":  # the column name (required)
    "type":  # the data type for the column
    "info": {
        "label":  # human-readable label for column
        "notes":  # markdown description of column
        "type_override":  # type for datapusher to use when importing data
        ...:  # free-form user-defined values
    }
    ...:  # values defined and validated with IDataDictionaryForm
}
```

Field types not provided will be guessed based on the first row of provided data. Set the types to ensure that future inserts will not fail because of an incorrectly guessed type. See *Field types* for details on which types are valid.

See also:

For more on custom field values and customizing the Data Dictionary form, see *Customizing the DataStore Data Dictionary Form*.

Records

A record is the data to be inserted in a DataStore resource and is defined as follows:

```
{
    column_1_id: value_1,
    columd_2_id: value_2,
    ...
}
```

Example:

```
[
    {
        "code_number": 10,
        "description": "Submitted successfully"
    },
    {
        "code_number": 42,
        "description": "In progress"
    }
]
```

Field types

The DataStore supports all types supported by PostgreSQL as well as a few additions. A list of the PostgreSQL types can be found in the type section of the documentation. Below you can find a list of the most common data types. The `json` type has been added as a storage for nested data.

In addition to the listed types below, you can also use array types. They are defines by prepending a `_` or appending `[]` or `[n]` where n denotes the length of the array. An arbitrarily long array of integers would be defined as `int[]`.

text
: Arbitrary text data, e.g. `Here's some text`.

json
: Arbitrary nested json data, e.g `{"foo": 42, "bar": [1, 2, 3]}`. Please note that this type is a custom type that is wrapped by the DataStore.

date
: Date without time, e.g `2012-5-25`.

time
: Time without date, e.g `12:42`.

timestamp
: Date and time, e.g `2012-10-01T02:43Z`.

int
Integer numbers, e.g `42`, `7`.

float
Floats, e.g. `1.61803`.

bool
Boolean values, e.g. `true`, `0`

You can find more information about the formatting of dates in the date/time types section of the PostgreSQL documentation.

Filters

Filters define the matching conditions to select from the DataStore. A filter is defined as follows:

```
  {
      "resource_id":  # the resource ID (required)
      "filters": {
          # column name: # field value
          # column name: # List of field values
          ...:  # other user-defined filters
}
  }
```

Filters must be supplied as a dictonary. Filters are used as *WHERE* statements. The filters have to be valid key/value pairs. The key must be a valid column name and the value must match the respective column type. The value may be provided as a List of multiple matching values. See *Field types* for details on which types are valid.

Example (single filter values, used as *WHERE* = statements):

```
{
    "resource_id":  "5f38da22-7d55-4312-81ce-17f1a9e84788",
    "filters": {
        "name": "Fred",
        "dob":  "1994-7-07"
    }
}
```

Example (multiple filter values, used as *WHERE IN* statements):

```
{
    "resource_id":  "5f38da22-7d55-4312-81ce-17f1a9e84788",
    "filters": {
        "name": ["Fred", "Jones"],
        "dob":  ["1994-7-07", "1992-7-27"]
    }
}
```

Resource aliases

A resource in the DataStore can have multiple aliases that are easier to remember than the resource id. Aliases can be created and edited with the `datastore_create()` API endpoint. All aliases can be found in a special view called `_table_metadata`. See *Internal structure of the database* for full reference.

Comparison of different querying methods

The DataStore supports querying with two API endpoints. They are similar but support different features. The following list gives an overview of the different methods.

	`datastore_search()`	`datastore_search_sql()`
Ease of use	Easy	Complex
Flexibility	Low	High
Query language	Custom (JSON)	SQL
Join resources	No	Yes

Internal structure of the database

The DataStore is a thin layer on top of a PostgreSQL database. Each DataStore resource belongs to a CKAN resource. The name of a table in the DataStore is always the resource id of the CKAN resource for the data.

As explained in *Resource aliases*, a resource can have mnemonic aliases which are stored as views in the database.

All aliases (views) and resources (tables respectively relations) of the DataStore can be found in a special view called `_table_metadata`. To access the list, open `http://{YOUR-CKAN-INSTALLATION}/api/3/action/datastore_search?resource_id=_table_metadata`.

`_table_metadata` has the following fields:

_id
: Unique key of the relation in `_table_metadata`.

alias_of
: Name of a relation that this alias point to. This field is `null` iff the name is not an alias.

name
: Contains the name of the alias if alias_of is not null. Otherwise, this is the resource id of the CKAN resource for the DataStore resource.

oid
: The PostgreSQL object ID of the table that belongs to name.

3.10.7 Extending DataStore

Starting from CKAN version 2.7, backend used in DataStore can be replaced with custom one. For this purpose, custom extension must implement *ckanext.datastore.interfaces.IDatastoreBackend*, which provides one method - *register_backends*. It should return dictonary with names of custom backends as keys and classes, that represent those backends as values. Each class supposed to be inherited from *ckanext.datastore.backend.DatastoreBackend*.

Note: Example of custom implementation can be found at *ckanext.example_idatastorebackend*

`ckanext.datastore.backend.get_all_resources_ids_in_datastore()` → list[str]

Helper for getting id of all resources in datastore.

Uses *get_all_ids* of active datastore backend.

exception `ckanext.datastore.backend.DatastoreException`

class `ckanext.datastore.backend.DatastoreBackend`

Base class for all datastore backends.

Very simple example of implementation based on SQLite can be found in *ckanext.example_idatastorebackend*. In order to use it, set datastore.write_url to 'example-sqlite://///tmp/database-name-on-your-choice'

Prop _backend
mapping(schema, class) of all registered backends

Prop _active_backend
current active backend

classmethod `register_backends()`

Register all backend implementations inside extensions.

classmethod `set_active_backend(`*config: CKANConfig*`)`

Choose most suitable backend depending on configuration

Parameters
`config` – configuration object

Return type
ckan.common.CKANConfig

classmethod `get_active_backend()`

Return currently used backend

`configure(`*config: CKANConfig*`)`

Configure backend, set inner variables, make some initial setup.

Parameters
`config` – configuration object

Returns
config

Return type
CKANConfig

`create(`*context: Context*, *data_dict: dict[str, Any]*, *plugin_data: dict[int, dict[str, Any]]*`)` → Any

Create new resourct inside datastore.

Called by *datastore_create*.

Parameters
`data_dict` – See *ckanext.datastore.logic.action.datastore_create*

Returns
The newly created data object

Return type
dictonary

upsert(*context: Context*, *data_dict: dict[str, Any]*) → Any

Update or create resource depending on data_dict param.

Called by *datastore_upsert*.

Parameters
data_dict – See *ckanext.datastore.logic.action.datastore_upsert*

Returns
The modified data object

Return type
dictonary

delete(*context: Context*, *data_dict: dict[str, Any]*) → Any

Remove resource from datastore.

Called by *datastore_delete*.

Parameters
data_dict – See *ckanext.datastore.logic.action.datastore_delete*

Returns
Original filters sent.

Return type
dictonary

search(*context: Context*, *data_dict: dict[str, Any]*) → Any

Base search.

Called by *datastore_search*.

Parameters

- **data_dict** – See *ckanext.datastore.logic.action.datastore_search*
- **fields** (*list of dictionaries*) – fields/columns and their extra metadata
- **offset** (*int*) – query offset value
- **limit** (*int*) – query limit value
- **filters** (*list of dictionaries*) – query filters
- **total** (*int*) – number of total matching records
- **records** (*list of dictionaries*) – list of matching results

Return type
dictonary with following keys

search_sql(*context: Context*, *data_dict: dict[str, Any]*) → Any

Advanced search.

Called by *datastore_search_sql*. :param sql: a single seach statement :type sql: string

Return type
dictonary

Parameters

- **fields** (*list of dictionaries*) – fields/columns and their extra metadata
- **records** (*list of dictionaries*) – list of matching results

resource_exists(*id: str*) → bool

Define whether resource exists in datastore.

resource_fields(*id: str*) → Any

Return dictonary with resource description.

Called by *datastore_info*. :returns: A dictionary describing the columns and their types.

resource_info(*id: str*) → Any

Return DataDictonary with resource's info - #3414

resource_id_from_alias(*alias: str*) → Any

Convert resource's alias to real id.

Parameters
alias (*string*) – resource's alias or id

Returns
real id of resource

Return type
string

get_all_ids() → list[str]

Return id of all resource registered in datastore.

Returns
all resources ids

Return type
list of strings

create_function(**args: Any*, ***kwargs: Any*) → Any

Called by *datastore_function_create* action.

drop_function(**args: Any*, ***kwargs: Any*) → Any

Called by *datastore_function_delete* action.

3.11 Apps & Ideas

The old "Apps & Ideas" functionality to allow users to provide information on apps, ideas, visualizations, articles etc that are related to a specific dataset has been moved to a separate extension: ckanext-showcase.

3.12 Tag Vocabularies

New in version 1.7.

CKAN sites can have *tag vocabularies*, which are a way of grouping related tags together into custom fields.

For example, if you were making a site for music datasets. you might use a tag vocabulary to add two fields *Genre* and *Composer* to your site's datasets, where each dataset can have one of the values *Avant-Garde*, *Country* or *Jazz* in its genre field, and one of the values *Beethoven*, *Wagner*, or *Tchaikovsky* in its composer field. In this example, genre and composer would be vocabularies and the values would be tags:

- Vocabulary: Genre
  - Tag: Avant-Garde

  - Tag: Country
  - Tag: Jazz
- Vocabulary: Composer
  - Tag: Beethoven
  - Tag: Wagner
  - Tag: Tchaikovsky

Ofcourse, you could just add Avant-Garde, Beethoven, etc. to datasets as normal CKAN tags, but using tag vocabularies lets you define Avant-Garde, Country and Jazz as genres and Beethoven, Wagner and Tchaikovsky as composers, and lets you enforce restrictions such as that each dataset must have a genre and a composer, and that no dataset can have two genres or two composers, etc.

Another example use-case for tag vocabularies would be to add a *Country Code* field to datasets defining the geographical coverage of the dataset, where each dataset is assigned a country code such as *en*, *fr*, *de*, etc. See `ckanext/example_idatasetform` for a working example implementation of country codes as a tag vocabulary.

3.12.1 Properties of Tag Vocabularies

- A CKAN website can have any number of vocabularies.
- Each vocabulary has an ID and name.
- Each tag either belongs to a vocabulary, or can be a *free tag* that doesn't belong to any vocabulary (i.e. a normal CKAN tag).
- A dataset can have more than one tag from the same vocabulary, and can have tags from more than one vocabulary.

3.12.2 Using Vocabularies

To add a tag vocabulary to a site, a CKAN sysadmin must:

1. Call the `vocabulary_create()` action of the CKAN API to create the vocabulary and tags. See *API guide*.
2. Implement an `IDatasetForm` plugin to add a new field for the tag vocabulary to the dataset schema. See *Extending guide*.
3. Provide custom dataset templates to display the new field to users when adding, updating or viewing datasets in the CKAN web interface. See *Theming guide*.

See `ckanext/example_idatasetform` for a working example of these steps.

3.13 Form Integration

CKAN allows you to integrate its Edit Dataset and New Dataset forms into an external front-end. To that end, CKAN also provides a simple way to redirect these forms back to the external front-end upon submission.

3.13.1 Redirecting CKAN Forms

It is obviously simple enough for an external front-end to link to CKAN's Edit Dataset and New Dataset forms, but once the forms are submitted, it would be desirable to redirect the user back to the external front-end, rather than CKAN's dataset read page.

This is achieved with a parameter to the CKAN URL. The 'return URL' can be specified in two places:

1. Passed as a URL-encoded value with the parameter `return_to` in the link to CKAN's form page.
2. Specified in the CKAN config keys *package_new_return_url* and *package_edit_return_url*.

(If the 'return URL' is supplied in both places, then the first takes precedence.)

Since the 'return URL' may need to include the dataset name, which could be changed by the user, CKAN replaces a known placeholder `<NAME>` with this value on redirect.

Note: Note that the downside of specifying the 'return URL' in the CKAN config is that the CKAN web interface becomes less usable on its own, since the user is hampered by the redirects to the external interface.

Example

An external front-end displays a dataset 'ontariolandcoverv100' here:

```
http://datadotgc.ca/dataset/ontariolandcoverv100
```

It displays a link to edit this dataset using CKAN's form, which without the redirect would be:

```
http://ca.ckan.net/dataset/edit/ontariolandoverv100
```

At first, it may seem that the return link should be `http://datadotgc.ca/dataset/ontariolandcoverv100`. But when the user edits this dataset, the name may change. So the return link needs to be:

```
http://datadotgc.ca/dataset/<NAME>
```

And this is URL-encoded to become:

```
http%3A%2F%2Fdatadotgc.ca%2Fdataset%2F%3CNAME%3E
```

So, in summary, the edit link becomes:

```
http://ca.ckan.net/dataset/edit/ontariolandoverv100?return_to=http%3A%2F%2Fdatadotgc.ca
↪%2Fdataset%2F%3CNAME%3E
```

During editing the dataset, the user changes the dataset name to *canadalandcover*, presses 'preview' and finally 'commit'. The user is now redirected back to the external front-end at:

```
http://datadotgc.ca/dataset/canadalandcover
```

The same functionality could be achieved by this line in the config file (`ca.ckan.net.ini`):

```
...

[app:main]
package_edit_return_url = http://datadotgc.ca/dataset/<NAME>
```

(continues on next page)

(continued from previous page)

```
...
```

3.14 Linked Data and RDF

Linked data and RDF features for CKAN are provided by the ckanext-dcat extension:

https://github.com/ckan/ckanext-dcat

These features include the RDF serializations of CKAN datasets based on DCAT, that used to be generated using templates hosted on the main CKAN repo, eg:

- https://demo.ckan.org/dataset/newcastle-city-council-payments-over-500.xml
- https://demo.ckan.org/dataset/newcastle-city-council-payments-over-500.ttl
- https://demo.ckan.org/dataset/newcastle-city-council-payments-over-500.n3
- https://demo.ckan.org/dataset/newcastle-city-council-payments-over-500.jsonld

ckanext-dcat offers many more features, including catalog-wide endpoints and harvesters to import RDF data into CKAN. Please check its documentation to know more about

As of CKAN 2.5, the RDF templates have been moved out of CKAN core in favour of the ckanext-dcat customizable endpoints. Note that previous CKAN versions can still use the ckanext-dcat RDF representations, which will override the old ones served by CKAN core.

3.15 Background jobs

CKAN allows you to create jobs that run in the 'background', i.e. asynchronously and without blocking the main application. Such jobs can be created in *Extensions* or in core CKAN.

Background jobs can be essential to providing certain kinds of functionality, for example:

- Creating web-hooks that notify other services when certain changes occur (for example a dataset is updated)
- Performing processing or validation or on data (as done by the Archiver and DataStorer Extensions)

Basically, any piece of work that takes too long to perform while the main application is waiting is a good candidate for a background job.

Note: The current background job system is based on RQ and was introduced in CKAN 2.7. See *Migrating from CKAN's previous background job system* for details on how to migrate your jobs from the previous system introduced in CKAN 1.5.

3.15.1 Writing and enqueuing background jobs

Note: This section is only relevant for developers working on CKAN or an extension.

The core of a background job is a regular Python function. For example, here's a very simply job function that logs a message:

```
import logging

def log_job(msg, level=logging.INFO, logger=u'ckan'):
    u'''
    Background job to log a message.
    '''
    logger = logging.getLogger(logger)
    logger.log(level, msg)
```

And that's it. Your job function can use all the usual Python features. Just keep in mind that your function will be run in a separate process by a *worker*, so your function should not depend on the current state of global variables, etc. Ideally your job function should receive all the information it needs via its arguments.

In addition, the module that contains your job function must be importable by the worker, which must also be able to get the function from its module. This means that nested functions, lambdas and instance methods cannot be used as job functions. While class methods of top-level classes can be used it's best to stick to ordinary module-level functions.

Note: Background jobs do not support return values (since they run asynchronously there is no place to return those values to). If your job function produces a result then it needs to store that result, for example in a file or in CKAN's database.

Once you have a job function, all you need to do is to use `ckan.lib.jobs.enqueue` to create an actual job out of it:

```
import ckan.lib.jobs as jobs

jobs.enqueue(log_job, [u'My log message'])
```

This will place a job on the *job queue* where it can be picked up and executed by a worker.

Note: Extensions should use `ckan.plugins.toolkit.enqueue_job()` instead. It's the same function but accessing it via *`ckan.plugins.toolkit` decouples your code from CKAN's internal structure*.

The first argument to `enqueue` is the job function to use. The second is a list of the arguments which should be passed to the function. You can omit it in which case no arguments will be passed. You can also pass keyword arguments in a dict as the third argument:

```
jobs.enqueue(log_job, [u'My log message'], {u'logger': u'ckanext.foo'})
```

You can also give the job a title which can be useful for identifying it when *managing the job queue*:

```
jobs.enqueue(log_job, [u'My log message'], title=u'My log job')
```

A timeout can also be set on a job iwth the `timeout` keyword argument:

```
jobs.enqueue(log_job, [u'My log message'], rq_kwargs={"timeout": 3600})
```

The default background job timeout is 180 seconds. This is set in the ckan config `.ini` file under the `ckan.jobs.timeout` item.

Accessing the database from background jobs

Code running in a background job can access the CKAN database like any other CKAN code.

In particular, using the action functions to modify the database from within a background job is perfectly fine. Just keep in mind that while your job is running in the background, the CKAN main process or other background jobs may also modify the database. Hence a single call to an action function is atomic from your job's view point, but between multiple calls there may be foreign changes to the database.

Special care has to be taken if your background job needs low-level access to the database, for example to modify SQLAlchemy model instances directly without going through an action function. Each background job runs in a separate process and therefore has its own SQLAlchemy session. Your code has to make sure that the changes it makes are properly contained in transactions and that you refresh your view of the database to receive updates where necessary. For these (and other) reasons it is recommended to *use the action functions to interact with the database*.

3.15.2 Running background jobs

Jobs are placed on the *job queue*, from which they can be retrieved and executed. Since jobs are designed to run asynchronously that happens in a separate process called a *worker*.

After it has been started, a worker listens on the queue until a job is enqueued. The worker then removes the job from the queue and executes it. Afterwards the worker waits again for the next job to be enqueued.

Note: Executed jobs are discarded. In particular, no information about past jobs is kept.

Workers can be started using the *Run a background job worker* command:

```
ckan -c /etc/ckan/default/ckan.ini jobs worker
```

The worker process will run indefinitely (you can stop it using `CTRL+C`).

Note: You can run multiple workers if your setup uses many or particularly long background jobs.

Using Supervisor

In a production setting, the worker should be run in a more robust way. One possibility is to use Supervisor.

First install Supervisor:

```
sudo apt-get install supervisor
```

Next copy the configuration file template:

```
sudo cp /usr/lib/ckan/default/src/ckan/ckan/config/supervisor-ckan-worker.conf /etc/
↪supervisor/conf.d
```

Next make sure the `/var/log/ckan/` directory exists, if not then it needs to be created:

```
sudo mkdir /var/log/ckan
```

Open `/etc/supervisor/conf.d/supervisor-ckan-worker.conf` in your favourite text editor and make sure all the settings suit your needs. If you installed CKAN in a non-default location (somewhere other than `/usr/lib/ckan/default`) then you will need to update the paths in the config file (see the comments in the file for details).

Restart Supervisor:

```
sudo service supervisor restart
```

The worker should now be running. To check its status, use

```
sudo supervisorctl status
```

You can restart the worker via

```
sudo supervisorctl restart ckan-worker:*
```

To test that background jobs are processed correctly you can enqueue a test job via

```
ckan -c |ckan.ini| jobs test
```

The worker's log files (`/var/log/ckan/ckan-worker.stdout.log` and/or `/var/log/ckan/ckan-worker.stderr.log`) should then show how the job was processed by the worker.

In case you run into problems, make sure to check the logs of Supervisor and the worker:

```
cat /var/log/supervisor/supervisord.log
cat /var/log/ckan/ckan-worker.stdout.log
cat /var/log/ckan/ckan-worker.sterr.log
```

3.15.3 Managing background jobs

Once they are enqueued, background jobs can be managed via the *ckan command* and the *web API*.

List enqueues jobs

- *ckan jobs list*
- *`ckan.logic.action.get.job_list()`*

Show details about a job

- *ckan jobs show*
- *`ckan.logic.action.get.job_show()`*

Cancel a job

A job that hasn't been processed yet can be canceled via

- *ckan jobs cancel*
- `ckan.logic.action.delete.job_cancel()`

Clear all enqueued jobs

- *ckan jobs clear*
- `ckan.logic.action.delete.job_clear()`

Logging

Information about enqueued and processed background jobs is automatically logged to the CKAN logs. You may need to update your logging configuration to record messages at the *INFO* level for the messages to be stored.

3.15.4 Background job queues

By default, all functionality related to background jobs uses a single job queue that is specific to the current CKAN instance. However, in some situations it is useful to have more than one queue. For example, you might want to distinguish between short, urgent jobs and longer, less urgent ones. The urgent jobs should be processed even if a long and less urgent job is already running.

For such scenarios, the job system supports multiple queues. To use a different queue, all you have to do is pass the (arbitrary) queue name. For example, to enqueue a job at a non-default queue:

```
jobs.enqueue(log_job, [u"I'm from a different queue!"],
             queue=u'my-own-queue')
```

Similarly, to start a worker that only listens to the queue you just posted a job to:

```
ckan -c |ckan.ini| jobs worker my-own-queue
```

See the documentation of the various functions and commands for details on how to use non-standard queues.

Note: If you create a custom queue in your extension then you should prefix the queue name using your extension's name. See *Avoid name clashes*.

Queue names are internally automatically prefixed with the CKAN site ID, so multiple parallel CKAN instances are not a problem.

3.15.5 Testing code that uses background jobs

Due to the asynchronous nature of background jobs, code that uses them needs to be handled specially when writing tests.

A common approach is to use the mock package to replace the `ckan.plugins.toolkit.enqueue_job` function with a mock that executes jobs synchronously instead of asynchronously:

```
import unittest.mock as mock

from ckan.tests import helpers


def synchronous_enqueue_job(job_func, args=None, kwargs=None, title=None):
    '''
    Synchronous mock for ``ckan.plugins.toolkit.enqueue_job``.
    '''
    args = args or []
    kwargs = kwargs or {}
    job_func(*args, **kwargs)


class TestSomethingWithBackgroundJobs(helpers.FunctionalTestBase):

    @mock.patch('ckan.plugins.toolkit.enqueue_job',
                side_effect=synchronous_enqueue_job)
    def test_something(self, enqueue_job_mock):
        some_function_that_enqueues_a_background_job()
        assert something
```

Depending on how the function under test calls `enqueue_job` you might need to adapt where the mock is installed. See mock's documentation for details.

3.15.6 Migrating from CKAN's previous background job system

Before version 2.7 (starting from 1.5), CKAN offered a different background job system built around Celery. As of CKAN 2.8, that system is no longer available. You should therefore update your code to use the new system described above.

Migrating existing job functions is easy. In the old system, a job function would look like this:

```
@celery.task(name=u'my_extension.echofunction')
def echo(message):
    print message
```

As *described above*, under the new system the same function would be simply written as

```
def echo(message):
    print message
```

There is no need for a special decorator. In the new system there is also no need for registering your tasks via `setup.py`.

Migrating the code that enqueues a task is also easy. Previously it would look like this:

```
celery.send_task(u'my_extension.echofunction', args=[u'Hello World'],
                 task_id=str(uuid.uuid4()))
```

With the new system, it looks as follows:

```
import ckan.lib.jobs as jobs

jobs.enqueue(ckanext.my_extension.plugin.echo, [u'Hello World'])
```

As you can see, the new system does not use strings to identify job functions but uses the functions directly instead. There is also no need for creating a job ID, that will be done automatically for you.

Supporting both systems at once

It might make sense to support both the RQ and the old Celery-based job system.

The easiest way to do that is to use ckanext-rq, which provides a back-port of the new system to older CKAN versions.

If you are unable to use *ckanext-rq* then you will need to write your code in such a way that it works on both systems. This could looks as follows. First split your Celery-based job functions into the job itself and its Celery handler. That is, change

```
@celery.task(name=u'my_extension.echofunction')
def echo(message):
    print message
```

to

```
def echo(message):
    print message

@celery.task(name=u'my_extension.echofunction')
def echo_celery(*args, **kwargs):
  echo(*args, **kwargs)
```

That way, you can call `echo` using the new system and use the name for Celery.

Then use the new system if it is available and fall back to Celery otherwise:

```
def compat_enqueue(name, fn, args=None):
    u'''
    Enqueue a background job using Celery or RQ.
    '''
    try:
        # Try to use RQ
        from ckan.plugins.toolkit import enqueue_job
        enqueue_job(fn, args=args)
    except ImportError:
        # Fallback to Celery
        import uuid
        from ckan.lib.celery_app import celery
        celery.send_task(name, args=args, task_id=str(uuid.uuid4()))
```

Use that function as follows for enqueuing a job:

```
compat_enqueue(u'my_extension.echofunction',
               ckanext.my_extension.plugin.echo,
               [u'Hello World'])
```

3.16 Email notifications

CKAN can send email notifications to users, for example when a user has new activities on her dashboard. Once email notifications have been enabled by a site admin, each user of a CKAN site can turn email notifications on or off for herself by logging in and editing her user preferences. To enable email notifications for a CKAN site, a sysadmin must:

1. Setup a cron job or other scheduled job on a server to call CKAN's `send_email_notifications` API action at regular intervals (e.g. hourly) and send any pending email notifications to users.

   On most UNIX systems you can setup a cron job by running `crontab -e` in a shell to edit your crontab file, and adding a line to the file to specify the new job. For more information run `man crontab` in a shell.

   CKAN's `send_email_notifications` API action can be called via the cli's `ckan notify send_emails` command. For example, here is a crontab line to send out CKAN email notifications hourly:

   ```
   @hourly echo '{}' | ckan -c path-to-your-ckan.ini notify send_emails > /dev/null
   ```

   The `@hourly` can be replaced with `@daily`, `@weekly` or `@monthly`.

   > **Warning:** CKAN will not send email notifications for events older than the time period specified by the `ckan.email_notifications_since` config setting (default: 2 days), so your cron job should run more frequently than this. `@hourly` and `@daily` are good choices.

   > **Note:** Since `send_email_notifications` is an API action, it can be called from a machine other than the server on which CKAN is running, simply by POSTing an HTTP request to the CKAN API (you must be a sysadmin to call this particular API action). See *API guide*.

2. CKAN will not send out any email notifications, nor show the email notifications preference to users, unless the *ckan.activity_streams_email_notifications* option is set to `True`, so put this line in the `[app:main]` section of your CKAN config file:

   ```
   ckan.activity_streams_email_notifications = True
   ```

3. Make sure that *ckan.site_url* is set correctly in the `[app:main]` section of your CKAN configuration file. This is used to generate links in the bodies of the notification emails. For example:

   ```
   ckan.site_url = http://publicdata.eu
   ```

4. Make sure that *smtp.mail_from* is set correctly in the `[app:main]` section of your CKAN configuration file. This is the email address that CKAN's email notifications will appear to come from. For example:

   ```
   smtp.mail_from = mailman@publicdata.eu
   ```

   This is combined with your *ckan.site_title* to form the `From:` header of the email that are sent, for example:

   ```
   From: PublicData.eu <mailmain@publicdata.eu>
   ```

If you would like to use an alternate reply address, such as a "no-reply" address, set *smtp.reply_to* in the `[app:main]` section of your CKAN configuration file. For example:

```
smtp.reply_to = noreply@example.com
```

5. If you do not have an SMTP server running locally on the machine that hosts your CKAN instance, you can change the *Email settings* to send email via an external SMTP server. For example, these settings in the `[app:main]` section of your configuration file will send emails using a gmail account (not recommended for production websites!):

```
smtp.server = smtp.gmail.com:587
smtp.starttls = True
smtp.user = your_username@gmail.com
smtp.password = your_gmail_password
smtp.mail_from = your_username@gmail.com
```

6. You need to restart the web server for the new configuration to take effect. For example, if you are using a CKAN package install, run this command in a shell:

   `sudo supervisorctl restart ckan-uwsgi:*`

3.17 Page View Tracking

CKAN has a core extension already installed that allows the system to anonymously track visits to pages of your site. You ca use this tracking data to:

- Sort datasets by popularity
- Highlight popular datasets and resources
- Show view counts next to datasets and resources
- Show a list of the most popular datasets
- Export page-view data to a CSV file

See also:

ckanext-googleanalytics
A CKAN extension that integrates Google Analytics into CKAN.

Note: CKAN 2.10 and older versions had tracking integrated into the core and this instructions no longer apply. Checkout the 2.10 documentation for more information.

3.17.1 Enabling Page View Tracking Extension

To enable page view tracking:

1. Add the *tracking* extension to your CKAN configuration file (e.g. /etc/ckan/default/ckan.ini):

```
[app:main]
ckan.plugins = tracking
```

   Save the file and restart your web server. CKAN will now record raw page view tracking data in your CKAN database as pages are viewed.

2. Setup a cron job to update the tracking summary data.

   For operations based on the tracking data CKAN uses a summarised version of the data, not the raw tracking data that is recorded "live" as page views happen. The `ckan tracking update` and `ckan search-index rebuild` commands need to be run periodicially to update this tracking summary data.

   You can setup a cron job to run these commands. On most UNIX systems you can setup a cron job by running `crontab -e` in a shell to edit your crontab file, and adding a line to the file to specify the new job. For more information run `man crontab` in a shell. For example, here is a crontab line to update the tracking data and rebuild the search index hourly:

   ```
   @hourly ckan -c /etc/ckan/default/ckan.ini tracking update  && ckan -c /etc/ckan/
   ↪default/ckan.ini search-index rebuild -r
   ```

   Replace `/usr/lib/ckan/bin/` with the path to the `bin` directory of the virtualenv that you've installed CKAN into, and replace '/etc/ckan/default/ckan.ini' with the path to your CKAN configuration file.

   The `@hourly` can be replaced with `@daily`, `@weekly` or `@monthly`.

3.17.2 Retrieving Tracking Data

When the extension is enabled, tracking summary data for datasets and resources is available in the dataset and resource dictionaries returned by, for example, the `package_show()` API:

```
"tracking_summary": {
    "recent": 5,
    "total": 15
},
```

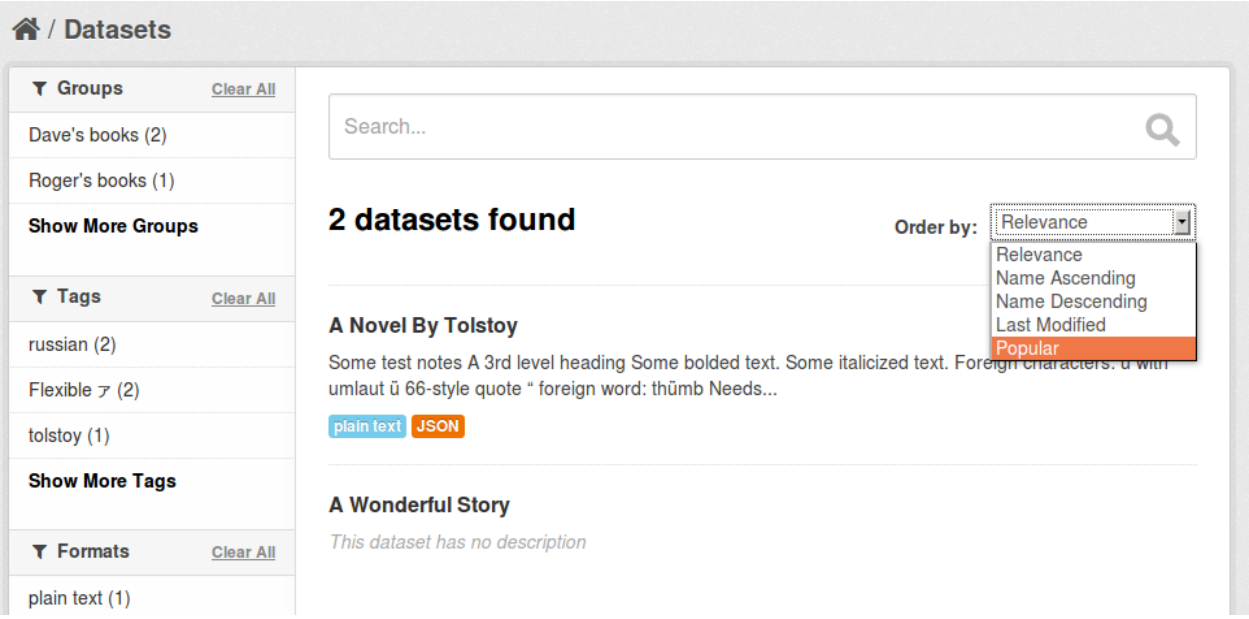
This can be used, for example, by custom templates to show the number of views next to datasets and resources. A dataset or resource's `recent` count is its number of views in the last 14 days, the `total` count is all of its tracked views (including recent ones).

You can also export tracking data for all datasets to a CSV file using the `ckan tracking export` command. For details, run `ckan tracking -h`.

Note: Repeatedly visiting the same page will not increase the page's view count! Page view counting is limited to one view per user per page per day.

3.17.3 Sorting Datasets by Popularity

Once you've enabled page view tracking on your CKAN site, you can view datasets most-popular-first by selecting `Popular` from the `Order by:` dropdown on the dataset search page:



The datasets are sorted by their number of recent views.

You can retrieve datasets most-popular-first from the *CKAN API* by passing `'sort': 'views_recent desc'` to the `package_search()` action. This could be used, for example, by a custom template to show a list of the most popular datasets on the site's front page.

Tip: You can also sort datasets by total views rather than recent views. Pass `'sort': 'views_total desc'` to the `package_search()` API, or use the URL `/dataset?q=&sort=views_total+desc` in the web interface.

3.17.4 Highlighting Popular Datasets and Resources

Once you've enabled page view tracking on your CKAN site, popular datasets and resources (those with more than 10 views) will be highlighted with a "popular" badge and a tooltip showing the number of views:

Tip: You can change the number of views that a dataset or resource needs to be considered popular by overriding

`ckanext/tracking/templates/snippets/popular.html` template. The default is 10.

3.18 Multilingual Extension

For translating CKAN's web interface see *Translating CKAN*. In addition to user interface internationalization, a CKAN administrator can also enter translations into CKAN's database for terms that may appear in the contents of datasets, groups or tags created by users. When a user is viewing the CKAN site, if the translation terms database contains a translation in the user's language for the name or description of a dataset or resource, the name of a tag or group, etc. then the translated term will be shown to the user in place of the original.

3.18.1 Setup and Configuration

By default term translations are disabled. To enable them, you have to specify the multilingual plugins using the `ckan.plugins` setting in your CKAN configuration file, for example:

```
# List the names of CKAN extensions to activate.
ckan.plugins = multilingual_dataset multilingual_group multilingual_tag
```

Of course, you won't see any terms getting translated until you load some term translations into the database. You can do this using the `term_translation_update` and `term_translation_update_many` actions of the CKAN API, See *API guide* for more details.

3.18.2 Loading Test Translations

If you want to quickly test the term translation feature without having to provide your own translations, you can load CKAN's test translations into the database by running this command from your shell:

```
ckan -c |ckan.ini| create-test-data translations
```

See *Command Line Interface (CLI)* for more details.

3.18.3 Testing The Multilingual Extension

If you have a source installation of CKAN you can test the multilingual extension by running the tests located in `ckanext/multilingual/tests`. You must first install the packages needed for running CKAN tests into your virtual environment, and then run this command from your shell:

```
pytest --ckan-ini=test-core.ini ckanext/multilingual/tests
```

See *Testing CKAN* for more information.

3.19 Stats Extension

CKAN's stats extension analyzes your CKAN database and displays several tables and graphs with statistics about your site, including:

- Total number of datasets
- Dataset revisions per week
- Top-rated datasets
- Most-edited Datasets
- Largest groups
- Top tags
- Users owning most datasets

See also:

CKAN's *built-in page view tracking feature*, which tracks visits to pages.

See also:

ckanext-googleanalytics
A CKAN extension that integrates Google Analytics into CKAN.

3.19.1 Enabling the Stats Extension

To enable the stats extensions add `stats` to the *ckan.plugins* option in your CKAN config file, for example:

```
ckan.plugins = stats
```

3.19.2 Viewing the Statistics

To view the statistics reported by the stats extension, visit the `/stats` page, for example: https://demo.ckan.org/stats

3.20 Configuration Options

The functionality and features of CKAN can be modified using many different configuration options. These are generally set in the *CKAN configuration file*, but some of them can also be set via Environment variables or at *runtime*.

Note: Looking for the available configuration options? Jump to *CKAN configuration file*.

3.20.1 Environment variables

Some of the CKAN configuration options can be defined as Environment variables on the server operating system.

These are generally low-level critical settings needed when setting up the application, like the database connection, the Solr server URL, etc. Sometimes it can be useful to define them as environment variables to automate and orchestrate deployments without having to first modify the *CKAN configuration file*.

These options are only read at startup time to update the `config` object used by CKAN, but they won't be accessed any more during the lifetime of the application.

CKAN environment variable names match the options in the configuration file, but they are always uppercase and prefixed with *CKAN_* (this prefix is added even if the corresponding option in the ini file does not have it), and replacing dots with underscores.

This is the list of currently supported environment variables, please refer to the entries in the *CKAN configuration file* section below for more details about each one:

```
CONFIG_FROM_ENV_VARS: dict[str, str] = {
    'sqlalchemy.url': 'CKAN_SQLALCHEMY_URL',
    'ckan.datastore.write_url': 'CKAN_DATASTORE_WRITE_URL',
    'ckan.datastore.read_url': 'CKAN_DATASTORE_READ_URL',
    'ckan.redis.url': 'CKAN_REDIS_URL',
    'solr_url': 'CKAN_SOLR_URL',
    'solr_user': 'CKAN_SOLR_USER',
    'solr_password': 'CKAN_SOLR_PASSWORD',
    'ckan.site_id': 'CKAN_SITE_ID',
    'ckan.site_url': 'CKAN_SITE_URL',
    'ckan.storage_path': 'CKAN_STORAGE_PATH',
    'ckan.datapusher.url': 'CKAN_DATAPUSHER_URL',
    'smtp.server': 'CKAN_SMTP_SERVER',
    'smtp.starttls': 'CKAN_SMTP_STARTTLS',
    'smtp.user': 'CKAN_SMTP_USER',
    'smtp.password': 'CKAN_SMTP_PASSWORD',
    'smtp.mail_from': 'CKAN_SMTP_MAIL_FROM',
    'ckan.max_resource_size': 'CKAN_MAX_UPLOAD_SIZE_MB'
}
```

3.20.2 Updating configuration options during runtime

CKAN configuration options are generally defined before starting the web application (either in the *CKAN configuration file* or via Environment variables).

A limited number of configuration options can also be edited during runtime. This can be done on the *administration interface* or using the `config_option_update()` API action. Only *sysadmins* can edit these runtime-editable configuration options. Changes made to these configuration options will be stored in the database and persisted when the server is restarted.

Extensions can add (or remove) configuration options to the ones that can be edited at runtime. For more details on how to do this check *Making configuration options runtime-editable*.

3.20.3 Config declaration

Tracking down all the possible config options in your CKAN site can be a challenging task. CKAN itself and its extensions change over time, deprecating features and providing new ones, which means that some new config options may be introduced, while other options no longer have any effect. In order to keep track of all valid config options, CKAN uses config declarations.

CKAN itself declares all the config options that are used throught the code base (You can see the core config declarations in the `ckan/config/config_declaration.yaml` file). This allows to validate the current configuration against the declaration, or check which config options in the CKAN config file are not declared (and might have no effect).

Declaring config options

Note: Starting from CKAN 2.11, CKAN will log a warning every time a non-declared configuration option is accessed. To prevent this, declare the configuration options offered by your extension using the methods below

Using a text file (JSON, YAML or TOML)

The recommended way of declaring config options is using the `config_declarations` *blanket*. It allows you to write less code and define your config options using JSON, YAML, or TOML (if the `toml` package is installed inside your virtual environment). That is how CKAN declares config options for all its built-in plugins, like `datastore` or `datatables_view`.

To use it, decorate the plugin with the `config_declarations` blanket:

```
import ckan.plugins as p
import ckan.plugins.toolkit as tk

@tk.blanket.config_declarations
class MyExt(p.SingletonPlugin):
    pass
```

Next, create a file *config_declaration.yaml* at the root directory of your extension: `ckanext/my_ext/config_declaration.yaml`. You can use the *.json* or *.toml* extension instead of *.yaml*.

Here is an example of the config declaration file. All the comments are added only for explanation and you don't need them in the real file:

```
# schema version of the config declaration. At the moment, the only valid value is `1`
version: 1

# an array of configuration blocks. Each block has an "annotation", that
# describes the block, and the list of options. These groups help to separate
# config options visually, but they have no extra meaning.
groups:

# short text that describes the group. It can be shown in the config file
# as following:
#   ## MyExt settings ##################
#   some.option = some.value
#   another.option = another.value
```

(continues on next page)

(continued from previous page)

```
- annotation: MyExt settings

  # an array of actual declarations
  options:

    # The only required item in the declaration is `key`. `key` defines the
    # name of the config option
    - key: my_ext.flag.do_something

      # default value, used when the option is missing from the config file.
      default: false

      # import path of the function that must be called in order to get the
      # default value. This can be used when the default value can be obtained from
      # an environment variable, database or any other external source.
      # IMPORTANT: use either `default` or `default_callable`, not both at the same time
      default_callable: ckanext.my_ext.utils:function_that_returns_default

      # Example of value that can be used for given option. If the config
      # option is missing from the config file, `placeholder` IS IGNORED. It
      # has only demonstration purpose. Good uses of `placeholder` are:
      # examples of secrets, examples of DB connection string.
      # IMPORTANT: do not use `default` and `placeholder` at the same
      # time. `placeholder` should be used INSTEAD OF the `default`
      # whenever you think it has a sense.
      placeholder: false

      # import path of the function that must be called in order to get the
      # placeholder value.  Basically, same as `default_callable`, but it
      # produces the value of `placeholder`.
      # IMPORTANT: use either `placeholder` or `placeholder_callable`, not both at the␣
↪same time
      placeholder_callable: ckanext.my_ext.utils:function_that_returns_placeholder

      # A dictionary with keyword-arguments that will be passed to
      # `default_callable` or `placeholder_callable`.  As mentioned above,
      # only one of these options may be used at the same time, so
      # `callable_args` can be used by any of these options without a conflict.
      callable_args:
        arg_1: 20
        arg_2: "hello"

      # an alternative example of a valid value for option. Used only in
      # CKAN documentation, thus has no value for extensions.
      example: some-valid-value

      # an explanation of the effect that option has. Don't hesistate to
      # put as much details here as possible
      description: |
          Nullam eu ante vel est convallis dignissim.  Fusce suscipit, wisi
          nec facilisis facilisis, est dui fermentum leo, quis tempor
          ligula erat quis odio.  Nunc porta vulputate tellus.  Nunc rutrum
```

(continues on next page)

(continued from previous page)

```
        turpis sed pede.  Sed bibendum.  Aliquam posuere.  Nunc aliquet,
        augue nec adipiscing interdum, lacus tellus malesuada massa, quis
        varius mi purus non odio.  Pellentesque condimentum, magna ut
        suscipit hendrerit, ipsum augue ornare nulla, non luctus diam
        neque sit amet urna.  Curabitur vulputate vestibulum lorem.
        Fusce sagittis, libero non molestie mollis, magna orci ultrices
        dolor, at vulputate neque nulla lacinia eros.  Sed id ligula quis
        est convallis tempor.  Curabitur lacinia pulvinar nibh.  Nam a
        sapien.

    # a space-separated list of validators, applied to the value of option.
    validators: not_missing boolean_validator

    # shortcut for the most common option types. It adds type validators to the option.
    # If both, `type` and `validators` are set, validators from `type` are added first,
    # then validators from `validators` are appended.
    # Valid types are: bool, int, list, dynamic (see below for more information on␣
↪dynamic
    # options)
    type: bool

    # boolean flag that marks config option as experimental. Such options are hidden␣
↪from
    # examples of configuration or any other auto-generated output. But they are␣
↪declared,
    # thus can be validated and do not produce undeclared-warning. Use it for options␣
↪that
    # are not stable and may be removed from your extension before the public release
    experimental: true

    # boolean flag that marks config option as ignored. Can be used for options that␣
↪are set
    # programmatically. This flag means that there is no sense in setting this option,␣
↪because
    # it will be overriden or won't be used at all.
    ignored: true

    # boolean flag that marks config option as hidden. Used for options that should␣
↪not be set
    # inside config file or anyhow used by others. Often this flag is used for options
    # that are added by Flask core or its extensions.
    internal: true

    # boolean flag that marks config option as required. Doesn't have a special effect␣
↪for now,
    # but may prevent application from startup in future, so use it only on options␣
↪that
    # are essential for your plugin and that have no sensible default value.
    required: true

    # boolean flag that marks config option as editable. Doesn't have a special effect␣
↪for now.
```

(continues on next page)

(continued from previous page)

```
    # It's recommended to enable this flag for options that are editable via AdminUI.
    editable: true

    # boolean flag that marks option as commented. Such options are added
    # as comments to the config file generated from template.
    commented: true

    # Deprecated name of the option. Can be used for options that were renamed.
    # When `key` is missing from config and `legacy_key` is available, the value of
    # `legacy_key` is used, printing a deprecation warning in the logs.
    legacy_key: my_ext.legacy.flag.do_something
```

The `IConfigDeclaration` interface

The *`IConfigDeclaration`* interface is available to plugins that want more control on how their own config options are declared.

New config options can only be declared inside the *`declare_config_options()`* method. This method accepts two arguments: a `Declaration` object that contains all the declarations, and a `Key` helper, which allows to declare more unusual config options.

A very basic config option may be declared in this way:

```
declaration.declare("ckanext.my_ext.option")
```

which just means that extension `my_ext` makes use of a config option named `ckanext.my_ext.option`. If we want to define the *default value* for this option we can write:

```
declaration.declare("ckanext.my_ext.option", True)
```

The second parameter to `declare()` specifies the default value of the declared option if it is not provided in the configuration file. If a default value is not specified, it's implicitly set to `None`.

You can assign validators to a declared config option:

```
option = declaration.declare("ckanext.my_ext.option", True)
option.set_validators("not_missing boolean_validator")
```

`set_validators` accepts a string with the names of validators that must be applied to the config option. These validators need to registered in CKAN core or in your own extension using the *`IValidators`* interface.

Note: Declared default values are also passed to validators. In addition, different validators can be applied to the same option multiple times. This means that validators must be idempotent and that the default value itself must be valid for the given set of validators.

If you need to declare a lot of options, you can declare all of them at once loading a dict:

```
declaration.load_dict(DICT_WITH_DECLARATIONS)
```

This allows to keep the configuration declaration in a separate file to make it easier to maintain if your plugin supports several config options.

Note: `declaration.load_dict()` takes only python dictionary as argument. If you store the declaration in an external file like a JSON, YAML file, you have to parse it into a Python dictionary yourself or use corresponding *blanket*. Read the following section for additional information.

Dynamic config options

There is a special option type, `dynamic`. This option type is used for a set of options that have common name-pattern. Because `dynamic` type defines multiple options, it has no default, validators and serves mostly documentation purposes. Let's use CKAN's `sqlalchemy.*` options as example. Every option whose name follows the pattern `sqlalchemy.SOMETHING` is passed to the SQLAlchemy engine created by CKAN. CKAN doesn't actually know which options are valid and it's up to you to provide valid values. Basically, we have a set of options with prefix `sqlalchemy.`. If use these options without declararing, it will trigger warnings about using undeclared options, which are harmless but can be annoying. Declaring them helps to make explicit which configuration options are actually being used. In order to declare such set of options, put some label surrounded with angle brackets instead of the dynamic part of option's name. In our case it can be `sqlalchemy.<OPTION>` or `sqlalchemy.<anything>`. Any word can be used as label, the only important part here are angle brackets:

```
- key: sqlalchemy.<OPTION>
  type: dynamic
  description: |
    Example::

     sqlalchemy.pool_pre_ping=True
     sqlalchemy.pool_size=10
     sqlalchemy.max_overflow=20

    Custom sqlalchemy config parameters used to establish the main
    database connection.
```

Use this feature sparsely, only when you really want to declare literally ANY value following the pattern. If you have finite set of possible options, consider declaring all of them, because it allows you to provide validators, defaults, and prevents you from accidental shadowing unrelated options.

Accessing config options

Using validators ensures that config values are normalized. Up until now you have probably seen code like this one:

```
is_enabled = toolkit.asbool(toolkit.config.get("ckanext.my_ext.enable", False))
```

Declaring this configuration option and assigning validators (*convert_int*, *boolean_validators*) and a default value means that we can use the `config.get(key)` instead of the expression above:

```
is_enabled = toolkit.config.get("ckanext.my_ext.enable")
```

This will ensure that:

1. If the value is not explicitly defined in the configuration file, the default one will be picked
2. This value is passed to the validators, and a valid value is returned

Note: An attempt to use `config.get()` with an undeclared config option will print a warning to the logs and return the option value or `None` as default.

Command line interface

The current configuration can be validated using the *config declaration CLI*:

```
ckan config validate
```

To get an example of the configuration for a given plugin, run `ckan config declaration <PLUGIN>`, eg:

```
ckan config declaration datastore

## Datastore settings ############################################################
ckan.datastore.write_url = postgresql://ckan_default:pass@localhost/datastore_default
ckan.datastore.read_url = postgresql://datastore_default:pass@localhost/datastore_default
ckan.datastore.sqlsearch.enabled = false
ckan.datastore.search.rows_default = 100
ckan.datastore.search.rows_max = 32000
# ckan.datastore.sqlalchemy.<OPTION> =

## PostgreSQL' full-text search parameters ######################################
ckan.datastore.default_fts_lang = english
ckan.datastore.default_fts_index_method = gist
```

To get an example of the declaration code itself in order to use it as a starting point in your own plugin, you can run `ckan config describe <PLUGIN>`, eg:

```
ckan config describe datapusher

# Output:
declaration.annotate('Datapusher settings')
declaration.declare(key.ckan.datapusher.formats, ...)
declaration.declare(key.ckan.datapusher.url)
declaration.declare(key.ckan.datapusher.callback_url_base)
declaration.declare(key.ckan.datapusher.assume_task_stale_after, 3600).set_validators(
↪'convert_int')
```

You can output the config declaration in different formats, which is useful if you want to keep them separately:

```
ckan config describe datapusher --format=dict # python dict
ckan config describe datapusher --format=json # JSON file
ckan config describe datapusher --format=yaml # YAML file
ckan config describe datapusher --format=toml # TOML file
```

3.20.4 CKAN configuration file

From CKAN 2.9, by default, the configuration file is located at `/etc/ckan/default/ckan.ini`. Previous releases the configuration file(s) were: `/etc/ckan/default/development.ini` or `/etc/ckan/default/production.ini`. This section documents all of the config file settings, for reference.

Note: After editing your config file, you need to restart your webserver for the changes to take effect.

Note: Unless otherwise noted, all configuration options should be set inside the `[app:main]` section of the config file (i.e. after the `[app:main]` line):

```
[DEFAULT]

...

[server:main]
use = egg:Paste#http
host = 0.0.0.0
port = 5000

# This setting will not work, because it's outside of [app:main].
ckan.site_logo = /images/masaq.png

[app:main]
# This setting will work.
ckan.plugins = stats text_view datatables_view
```

If the same option is set more than once in your config file, exeption will be raised and CKAN application will not start

Default settings

debug

Example:

```
debug = true
```

Default value: `False`

This enables the Flask-DebugToolbar in the web interface, makes Webassets serve unminified JS and CSS files, and enables CKAN templates' debugging features.

You will need to ensure the `Flask-DebugToolbar` python package is installed, by activating your ckan virtual environment and then running:

```
pip install -r /usr/lib/ckan/default/src/ckan/dev-requirements.txt
```

If you are running CKAN on Apache, you must change the WSGI configuration to run a single process of CKAN. Otherwise the execution will fail with: `AssertionError: The EvalException middleware is not usable in a multi-process environment`. Eg. change:

```
WSGIDaemonProcess ckan_default display-name=ckan_default processes=2 threads=15
to
WSGIDaemonProcess ckan_default display-name=ckan_default threads=15
```

> **Warning:** This option should be set to `False` for a public site. With debug mode enabled, a visitor to your site could execute malicious commands.

General settings

SECRET_KEY

Default value: none

This is the secret token that is used by security related tasks by CKAN and its extensions. `ckan generate config` generates a unique value for this each time it generates a config file. Alternatively you can generate one with the following command:

```
python -c "import secrets; print(secrets.token_urlsafe(20))"
```

When used in a cluster environment, the value must be the same on every machine.

ckan.legacy_route_mappings

Example:

```
ckan.legacy_route_mappings = {"home": "home.index", "about": "home.about", "search":
↪"dataset.search"}
```

Default value: `{}`

This can be used when using an extension that is still using old (Pylons-based) route names to maintain compatibility.

> **Warning:** This configuration will be removed when the migration to Flask is completed. Please update the extension code to use the new Flask-based route names.

config.mode

Example:

```
config.mode = strict
```

Default value: `strict`

> **Warning:** This configuration option has no effect starting from CKAN 2.11. The default behaviour going forward is the old `strict` mode, where CKAN will not start unless **all** config options are valid according to the validators defined in the configuration declaration. For every invalid config option, an error will be printed to the output stream.

Development settings

ckan.devserver.host

Example:

```
ckan.devserver.host = 0.0.0.0
```

Default value: `localhost`

Host name to use when running the development server.

ckan.devserver.port

Example:

```
ckan.devserver.port = 5005
```

Default value: `5000`

Port to use when running the development server.

ckan.devserver.threaded

Example:

```
ckan.devserver.threaded = true
```

Default value: `False`

Controls whether the development server should handle each request in a separate thread.

ckan.devserver.multiprocess

Example:

```
ckan.devserver.multiprocess = 8
```

Default value: `1`

If greater than 1 then the development server will handle each request in a new process, up to this maximum number of concurrent processes.

ckan.devserver.watch_patterns

Example:

```
ckan.devserver.watch_patterns = mytheme/**/*.yaml mytheme/**/*.json
```

Default value: none

A list of files the reloader should watch to restart the development server, in addition to the Python modules (for example configuration files)

ckan.devserver.ssl_cert

Example:

```
ckan.devserver.ssl_cert = path/to/host.cert
```

Default value: none

Path to a certificate file that will be used to enable SSL (ie to serve the local development server on https://localhost:5000). You can generate a self-signed certificate and key (see *ckan.devserver.ssl_key*) running the following commands:

```
openssl genrsa 2048 > host.key
chmod 400 host.key
openssl req -new -x509 -nodes -sha256 -days 3650 -key host.key > host.cert
```

After that you can run CKAN locally with SSL using this command:

```
ckan -c /path/to/ckan.ini run --ssl-cert=/path/to/host.cert --ssl-key=/path/to/host.key
```

Alternatively, setting this option to `adhoc` will automatically generate a new certificate file (on each server reload, which means that you'll get a browser warning about the certificate on each reload).

ckan.devserver.ssl_key

Example:

```
ckan.devserver.ssl_key = path/to/host.key
```

Default value: none

Path to a certificate file that will be used to enable SSL (ie to serve the local development server on https://localhost:5000). See *ckan.devserver.ssl_cert* for more details. This option also supports the `adhoc` value, with the same caveat.

Session settings

ckan.user.last_active_interval

Default value: `600`

The number of seconds between requests to record the last time a user was active on the site.

beaker.session.key

Default value: `ckan`

Name of the cookie key used to save the session under.

beaker.session.secret

Default value: none

This is the secret token that the beaker library uses to hash the cookie sent to the client. *ckan generate config* generates a unique value for this each time it generates a config file. When used in a cluster environment, the value must be the same on every machine. If not provided, the value of `SECRET_KEY` will be used.

beaker.session.auto

Default value: `False`

When set to True, the session will save itself anytime it is accessed during a request, negating the need to issue the save() method.

beaker.session.cookie_expires

Default value: `False`

Determines when the cookie used to track the client-side of the session will expire. When set to a boolean value, it will either expire at the end of the browsers session, or never expire. Setting to a datetime forces a hard ending time for the session (generally used for setting a session to a far off date). Setting to an integer will result in the cookie being set to expire in that many seconds. I.e. a value of 300 will result in the cookie being set to expire in 300 seconds. Defaults to never expiring.

beaker.session.cookie_domain

Default value: none

What domain the cookie should be set to. When using sub-domains, this should be set to the main domain the cookie should be valid for. For example, if a cookie should be valid under www.nowhere.com and files.nowhere.com then it should be set to .nowhere.com. Defaults to the current domain in its entirety.

beaker.session.save_accessed_time

Default value: `True`

Whether beaker should save the session's access time (true) or only modification time (false).

beaker.session.secure

Default value: `False`

Whether or not the session cookie should be marked as secure. When marked as secure, browsers are instructed to not send the cookie over anything other than an SSL connection.

beaker.session.timeout

Default value: none

Seconds until the session is considered invalid, after which it will be ignored and invalidated. This number is based on the time since the session was last accessed, not from when the session was created. Defaults to never expiring. Requires that save_accessed_time be true.

beaker.session.type

Default value: `cookie`

The type of session to use. The default is cookie, which uses a cookie to store the session id. Other options include file, which stores the session id in a file, and dbm, which stores the session id in a dbm file. The dbm option is not recommended as it is not thread-safe.

beaker.session.validate_key

Default value: none

This is the secret token that is used to sign the local encrypted session. *ckan generate config* generates a unique value for this each time it generates a config file. When used in a cluster environment, the value must be the same on every machine. If not provided, the value of `SECRET_KEY` will be used.

beaker.session.httponly

Default value: `True`

Whether or not the session cookie should be marked as http only. When marked as http only, browsers are instructed to not allow javascript access to the cookie.

beaker.session.samesite

Default value: `Lax`

Whether or not the session cookie should be marked as SameSite. When marked as SameSite, browsers are instructed to not send the cookie with cross-site requests. The value can be "Strict", "Lax" or "None".

Database settings

sqlalchemy.url

Example:

```
sqlalchemy.url = postgres://tester:pass@localhost/ckantest3
```

Default value: none

This defines the database that CKAN is to use. The format is:

```
sqlalchemy.url = postgres://USERNAME:PASSWORD@HOST/DBNAME
```

sqlalchemy.<OPTION>

Default value: none

Example:

```
sqlalchemy.pool_pre_ping=True
sqlalchemy.pool_size=10
sqlalchemy.max_overflow=20
```

Custom sqlalchemy config parameters used to establish the main database connection.

To get the list of all the available properties check the SQLAlchemy documentation

Site Settings

ckan.site_url

Example:

```
ckan.site_url = http://scotdata.ckan.net
```

Default value: none

Set this to the URL of your CKAN site. Many CKAN features that need an absolute URL to your site use this setting.

This setting should only contain the protocol (e.g. `http://`), host (e.g. `www.example.com`) and (optionally) the port (e.g. `:8080`). In particular, if you have mounted CKAN at a path other than `/` then the mount point must *not* be included in `ckan.site_url`. Instead, you need to set *ckan.root_path*.

Important: It is mandatory to complete this setting

Warning: This setting should not have a trailing / on the end.

apitoken_header_name

Example:

```
apitoken_header_name = X-CKAN-API-TOKEN
```

Default value: `Authorization`

This allows to customize the name of the HTTP header used to provide the CKAN API token. This is useful in some scenarios where using the default `Authorization` one causes problems.

ckan.cache_expires

Example:

```
ckan.cache_expires = 2592000
```

Default value: `0`

This sets `Cache-Control` header's max-age value.

ckan.cache_enabled

Example:

```
ckan.cache_enabled = true
```

Default value: `False`

This enables cache control headers on all requests. If the user is not logged in and there is no session data a `Cache-Control: public` header will be added. For all other requests the `Cache-control: private` header will be added.

ckan.mimetype_guess

Example:

```
ckan.mimetype_guess = file_contents
```

Default value: `file_ext`

There are three options for guessing the mimetype of uploaded or linked resources: file_ext, file_contents, None.

`file_ext` will guess the mimetype by the url first, then the file extension.

`file_contents` will guess the mimetype by the file itself, this tends to be inaccurate.

`None` will not store the mimetype for the resource.

ckan.static_max_age

Example:

```
ckan.static_max_age = 2592000
```

Default value: `3600`

Controls CKAN static files' cache max age, if we're serving and caching them.

ckan.valid_url_schemes

Example:

```
ckan.valid_url_schemes = http https ftp sftp
```

Default value: `http https ftp`

Controls what uri schemes are rendered as links.

ckan.requests.timeout

Example:

```
ckan.requests.timeout = 10
```

Default value: `5`

Defines how long (in seconds) requests calls should last before they will timeout.

ckan.hide_version

Example:

```
ckan.hide_version = True
```

Default value: `False`

If set to True, CKAN will not publicly expose its version number.

Authorization Settings

ckan.auth.anon_create_dataset

Example:

```
ckan.auth.anon_create_dataset = false
```

Default value: `False`

Allow users to create datasets without registering and logging in.

ckan.auth.create_unowned_dataset

Example:

```
ckan.auth.create_unowned_dataset = false
```

Default value: `False`

Allow the creation of datasets not owned by any organization.

ckan.auth.create_dataset_if_not_in_organization

Example:

```
ckan.auth.create_dataset_if_not_in_organization = false
```

Default value: `True`

Allow users who are not members of any organization to create datasets, default: true. `create_unowned_dataset` must also be True, otherwise setting `create_dataset_if_not_in_organization` to True is meaningless.

ckan.auth.user_create_groups

Example:

```
ckan.auth.user_create_groups = true
```

Default value: `True`

Allow users to create groups.

ckan.auth.user_create_organizations

Example:

```
ckan.auth.user_create_organizations = false
```

Default value: `True`

Allow users to create organizations.

ckan.auth.user_delete_groups

Example:

```
ckan.auth.user_delete_groups = false
```

Default value: `True`

Allow users to delete groups.

ckan.auth.user_delete_organizations

Example:

```
ckan.auth.user_delete_organizations = false
```

Default value: `True`

Allow users to delete organizations.

ckan.auth.create_user_via_api

Example:

```
ckan.auth.create_user_via_api = false
```

Default value: `False`

Allow new user accounts to be created via the API by anyone. When `False` only sysadmins are authorised.

ckan.auth.create_user_via_web

Example:

```
ckan.auth.create_user_via_web = true
```

Default value: `False`

Allow new user accounts to be created via the web UI. When `False` (default value), user accounts can only be created by:

- Being invited by an organization admin,
- Being created directly by a sysadmin in the `/user/register` endpoint, or
- Being created in the CLI using `ckan user add`

ckan.auth.roles_that_cascade_to_sub_groups

Example:

```
ckan.auth.roles_that_cascade_to_sub_groups = admin editor
```

Default value: `admin`

Makes role permissions apply to all the groups or organizations down the hierarchy from the groups or organizations that the role is applied to. e.g. a particular user has the 'admin' role for group 'Department of Health'. If you set the value of this option to 'admin' then the user will automatically have the same admin permissions for the child groups of 'Department of Health' such as 'Cancer Research' (and its children too and so on).

ckan.auth.public_user_details

Example:

```
ckan.auth.public_user_details = false
```

Default value: `True`

Restricts anonymous access to user information. If is set to `False` accessing users details when not logged in will raise a `Not Authorized` exception.

Note: This setting should be used when user registration is disabled (`ckan.auth.create_user_via_web = False`), otherwise users can just create an account to see other users details.

---

ckan.auth.public_activity_stream_detail

Example:

```
ckan.auth.public_activity_stream_detail = true
```

Default value: `False`

Restricts access to 'view this version' and 'changes' in the Activity Stream pages. These links provide users with the full edit history of datasets etc - what they showed in the past and the diffs between versions. If this option is set to `False` then only admins (e.g. whoever can edit the dataset) can see this detail. If set to `True`, anyone can see this detail (assuming they have permission to view the dataset etc).

ckan.auth.allow_dataset_collaborators

Example:

```
ckan.auth.allow_dataset_collaborators = true
```

Default value: `False`

Enables or disable collaborators in individual datasets. If `True`, in addition to the standard organization based permissions, users can be added as collaborators to individual datasets with different roles, regardless of the organization they belong to. For more information, check the documentation on *Dataset collaborators*.

> **Warning:** If this setting is turned off in a site where there already were collaborators created, you must reindex all datasets to update the permission labels, in order to prevent access to private datasets to the previous collaborators.

ckan.auth.allow_admin_collaborators

Example:

```
ckan.auth.allow_admin_collaborators = true
```

Default value: `False`

Allows dataset collaborators to have the "Admin" role, allowing them to add more collaborators or remove existing ones. By default, collaborators can only be managed by administrators of the organization the dataset belongs to. For more information, check the documentation on *Dataset collaborators*.

> **Warning:** If this setting is turned off in a site where admin collaborators have been already created, existing collaborators with role "admin" will no longer be able to add or remove collaborators, but they will still be able to edit and access the datasets that they are assigned to.

ckan.auth.allow_collaborators_to_change_owner_org

Example:

```
ckan.auth.allow_collaborators_to_change_owner_org = true
```

Default value: `False`

Allows dataset collaborators to change the owner organization of the datasets they are collaborators on. Defaults to False, meaning that collaborators with role admin or editor can edit the dataset metadata but not the organization field.

ckan.auth.create_default_api_keys

Example:

```
ckan.auth.create_default_api_keys = true
```

Default value: `False`

Determines if an API key should be automatically created for every user when creating a user account. If set to False (the default value), users can manually create an API token from their profile instead. See *Authentication and API tokens*: for more details.

ckan.auth.login_view

Default value: `user.login`

The name of the view to redirect to when the user needs to log in

ckan.auth.reveal_private_datasets

Default value: `False`

Determines whether unauthorised requests for private datasets should have the existence of the datasets revealed (True) or hidden (False). If True, then unauthenticated requests will be redirected to the login page, and redirected back to the dataset after logging in, while authenticated but unauthorised requests will receive HTTP 403 Forbidden. If False, all unauthorised requests will receive HTTP 404 Not Found. Default is False.

ckan.auth.enable_cookie_auth_in_api

Default value: `True`

When set to False, cookie-based authentication is entirely ignored in all API requests, and authentication must be always done using *API Tokens*. Note that this will break some existing JS modules from the frontend that perform API calls, so it should be used with caution.

ckan.auth.route_after_login

Default value: `dashboard.datasets`

Allows to customize the route that the user will get redirected to after a successful login.

CSRF Protection

WTF_CSRF_ENABLED

Default value: `True`

Set to False to disable all CSRF protection.

WTF_CSRF_CHECK_DEFAULT

Default value: `True`

When using the CSRF protection extension, this controls whether every view is protected by default.

WTF_CSRF_SECRET_KEY

Default value: none

Random data for generating secure tokens. If not provided, the value of `SECRET_KEY` will be used.

WTF_CSRF_METHODS

Default value: `POST PUT PATCH DELETE`

HTTP methods to protect from CSRF.

WTF_CSRF_FIELD_NAME

Default value: `_csrf_token`

Name of the form field and session key that holds the CSRF token.

WTF_CSRF_HEADERS

Default value: `X-CSRFToken X-CSRF-Token`

HTTP headers to search for CSRF token when it is not provided in the form.

WTF_CSRF_TIME_LIMIT

Default value: `3600`

Max age in seconds for CSRF tokens. This value is capped by the lifetime of the session.

WTF_CSRF_SSL_STRICT

Default value: `True`

Whether to enforce the same origin policy by checking that the referrer matches the host. Only applies to HTTPS requests. Default is True.

WTF_I18N_ENABLED

Default value: `True`

Set to False to disable Flask-Babel I18N support. Also set to False if you want to use WTForms's built-in messages directly, see more info here.

ckan.csrf_protection.ignore_extensions

Default value: `True`

Exempt plugins blueprints from CSRF protection.

> **Warning:** This feature will be deprecated in future versions.

Flask-Login Remember me cookie settings

REMEMBER_COOKIE_NAME

Default value: `remember_token`

The name of the cookie to store the "remember me" information in.

REMEMBER_COOKIE_DURATION

Default value: `31536000`

The amount of time before the cookie expires, as a datetime.timedelta object or integer seconds.

REMEMBER_COOKIE_DOMAIN

Default value: none

If the "Remember Me" cookie should cross domains, set the domain value here (i.e. .example.com would allow the cookie to be used on all subdomains of example.com).

REMEMBER_COOKIE_PATH

Default value: `/`

Limits the "Remember Me" cookie to a certain path.

REMEMBER_COOKIE_SECURE

Default value: `False`

Restricts the "Remember Me" cookie's scope to secure channels (typically HTTPS).

REMEMBER_COOKIE_HTTPONLY

Default value: `True`

Prevents the "Remember Me" cookie from being accessed by client-side scripts.

REMEMBER_COOKIE_REFRESH_EACH_REQUEST

Default value: `False`

If set to True the cookie is refreshed on every request, which bumps the lifetime. Works like Flask's SESSION_REFRESH_EACH_REQUEST.

REMEMBER_COOKIE_SAMESITE

Default value: `None`

Restricts the "Remember Me" cookie to first-party or same-site context.

API Token Settings

api_token.nbytes

Example:

```
api_token.nbytes = 20
```

Default value: `32`

Number of bytes used to generate unique id for API Token.

api_token.jwt.encode.secret

Example:

```
api_token.jwt.encode.secret = file:/path/to/private/key
```

Default value: none

A key suitable for the chosen algorithm(`api_token.jwt.algorithm`):

- for asymmetric algorithms(RS256): path to private key with `file:` prefix. I.e `file:/path/private/key`
- for symmetric algorithms(HS256): plain string, sufficiently long for security with `string:` prefix. I.e `string:123abc...`

Note: For symmetric algorithms this value must be identical to *api_token.jwt.decode.secret*. The algorithm used is controlled by the *api_token.jwt.algorithm* option.

Value must have prefix, which defines its type. Supported prefixes are:

- `string:` - Plain string, will be used as is.
- `file:` - Path to file. Content of the file will be used as key.

If not provided, `"string:" + SECRET_KEY` is used.

api_token.jwt.decode.secret

Example:

```
api_token.jwt.decode.secret = file:/path/to/public/key.pub
```

Default value: none

A key suitable for the chosen algorithm(`api_token.jwt.algorithm`):

- for asymmetric algorithms(RS256): path to public key with `file:` prefix. I.e `file:/path/public/key.pub`
- for symmetric algorithms(HS256): plain string, sufficiently long for security with `string:` prefix. I.e `string:123abc...`

Note: For symmetric algorithms this value must be identical to *api_token.jwt.encode.secret*. The algorithm used is defined by the *api_token.jwt.algorithm* option.

Value must have prefix, which defines it's type. Supported prefixes are:

- `string:` - Plain string, will be used as is.
- `file:` - Path to file. Content of the file will be used as key.

If not provided, `"string:" + SECRET_KEY` is used.

api_token.jwt.algorithm

Example:

```
api_token.jwt.algorithm = RS256
```

Default value: `HS256`

Algorithm to sign the token with, e.g. "ES256", "RS256"

Depending on the algorithm, additional restrictions may apply to *api_token.jwt.decode.secret* and *api_token.jwt.encode.secret*. For example, RS256 implies that *api_token.jwt.encode.secret* contains RSA private key and *api_token.jwt.decode.secret* contains public key. Whereas HS256(default value) requires both *api_token.jwt.decode.secret* and *api_token.jwt.encode.secret* to have exactly the same value.

Search Settings

ckan.site_id

Example:

```
ckan.site_id = my_ckan_instance
```

Default value: `default`

CKAN uses Solr to index and search packages. The search index is linked to the value of the `ckan.site_id`, so if you have more than one CKAN instance using the same *solr_url*, they will each have a separate search index as long as their `ckan.site_id` values are different. If you are only running a single CKAN instance then this can be ignored.

Note: If you change this value, you need to rebuild the search index.

solr_url

Example:

```
solr_url = http://solr.okfn.org:8983/solr/ckan-schema-2.0
```

Default value: none

This configures the Solr server used for search. The Solr schema found at that URL must be one of the ones in `ckan/config/solr` (generally the most recent one). A check of the schema version number occurs when CKAN starts.

Optionally, `solr_user` and `solr_password` can also be configured to specify HTTP Basic authentication details for all Solr requests.

Note: If you change this value, you need to rebuild the search index.

solr_user

Default value: none

User to use in HTTP Basic Authentication when connecting to Solr

solr_password

Default value: none

Password to use in HTTP Basic Authentication when connecting to Solr

ckan.search.remove_deleted_packages

Default value: `True`

By default, deleted datasets are removed from the search index so are no longer available in searches. To keep them in the search index, set this setting to `False`. This will enable the `include_deleted` parameter in the *ckan.logic.action.get.package_search()* API action.

ckan.search.solr_commit

Default value: `True`

Make ckan commit changes solr after every dataset update change. Turn this to false if on solr 4.0 and you have automatic (soft)commits enabled to improve dataset update/create speed (however there may be a slight delay before dataset gets seen in results).

ckan.search.solr_allowed_query_parsers

Example:

```
ckan.search.solr_allowed_query_parsers = ['bool', 'knn']
```

Default value: none

Local parameters are not allowed when passing queries to Solr. An exception to this is when passing local parameters for special query parsers, that need to be enabled explicitly using this config option. For instance, the example provided would allow sending queries like the following::

search_params["q"] = "{!bool must=test}..." search_params["q"] = "{!knn field=vector topK=10}..."

ckan.search.show_all_types

Example:

```
ckan.search.show_all_types = dataset
```

Default value: `dataset`

Controls whether a search page (e.g. `/dataset`) should also show custom dataset types. The default is `false` meaning that no search page for any type will show other types. `true` will show other types on the `/dataset` search page. Any other value (e.g. `dataset` or `document` will be treated as a dataset type and that type's search page will show datasets of all types.

ckan.search.default_include_private

Default value: `True`

Controls whether the default search page (`/dataset`) should include private datasets visible to the current user or only public datasets visible to everyone.

ckan.search.default_package_sort

Example:

```
ckan.search.default_package_sort = name asc
```

Default value: `score desc, metadata_modified desc`

Controls whether the default search page (`/dataset`) should different sorting parameter by default when the request does not specify sort.

search.facets.limit

Example:

```
search.facets.limit = 100
```

Default value: `50`

Sets the default number of searched facets returned in a query.

search.facets.default

Example:

```
search.facets.default = 10
```

Default value: `10`

Default number of facets shown in search results.

ckan.extra_resource_fields

Example:

```
ckan.extra_resource_fields = alt_url
```

Default value: none

List of the extra resource fields that would be used when searching.

ckan.search.rows_max

Example:

```
ckan.search.rows_max = 1000
```

Default value: `1000`

Maximum allowed value for rows returned. Specifically this limits:

- `package_search`'s `rows` parameter
- `group_show` and `organization_show`'s number of datasets returned when specifying `include_datasets=true`

ckan.group_and_organization_list_max

Example:

```
ckan.group_and_organization_list_max = 1000
```

Default value: `1000`

Maximum number of groups/organizations returned when listing them. Specifically this limits:

- `group_list`'s `limit` when `all_fields=false`
- `organization_list`'s `limit` when `all_fields=false`

ckan.group_and_organization_list_all_fields_max

Example:

```
ckan.group_and_organization_list_all_fields_max = 100
```

Default value: `25`

Maximum number of groups/organizations returned when listing them in detail. Specifically this limits:

- `group_list`'s `limit` when `all_fields=true`
- `organization_list`'s `limit` when `all_fields=true`

solr_timeout

Example:

```
solr_timeout = 120
```

Default value: `60`

The option defines the timeout in seconds until giving up on a request. Raising this value might help you if you encounter a timeout exception.

Redis Settings

ckan.redis.url

Example:

```
ckan.redis.url = redis://localhost:7000/1
```

Default value: `redis://localhost:6379/0`

URL to your Redis instance, including the database to be used.

CORS Settings

ckan.cors.origin_allow_all

Example:

```
ckan.cors.origin_allow_all = true
```

Default value: `False`

This setting must be present to enable CORS. If True, all origins will be allowed (the response header Access-Control-Allow-Origin is set to '*'). If False, only origins from the `ckan.cors.origin_whitelist` setting will be allowed.

ckan.cors.origin_whitelist

Example:

```
ckan.cors.origin_whitelist = http://www.myremotedomain1.com http://myremotedomain1.com
```

Default value: none

A space separated list of allowable origins. This setting is used when `ckan.cors.origin_allow_all = False`.

Plugins Settings

ckan.plugins

Example:

```
ckan.plugins = activity scheming_datasets datatables_view datastore xloader
```

Default value: none

Specify which CKAN plugins are to be enabled.

> **Warning:** If you specify a plugin but have not installed the code, CKAN will not start.

Format as a space-separated list of the plugin names. The plugin name is the key in the `[ckan.plugins]` section of the extension's `setup.py`. For more information on plugins and extensions, see *Extending guide*.

Note: The order of the plugin names in the configuration file influences the order that CKAN will load the plugins in. As long as each plugin class is implemented in a separate Python module (i.e. in a separate Python source code file), the plugins will be loaded in the order given in the configuration file.

When multiple plugins are implemented in the same Python module, CKAN will process the plugins in the order that they're given in the config file, but as soon as it reaches one plugin from a given Python module, CKAN will load all plugins from that Python module, in the order that the plugin classes are defined in the module.

For simplicity, we recommend implementing each plugin class in its own Python module.

Plugin loading order can be important, for example for plugins that add custom template files: templates found in template directories added earlier will override templates in template directories added later.

Todo: Fix CKAN's plugin loading order to simply load all plugins in the order they're given in the config file, regardless of which Python modules they're implemented in.

ckan.resource_proxy.timeout

Default value: `5`

Timeout in seconds to use on Resource Proxy requests.

Front-End Settings

ckan.site_title

Example:

```
ckan.site_title = Open Data Scotland
```

Default value: `CKAN`

This sets the name of the site, as displayed in the CKAN web interface.

ckan.site_description

Example:

```
ckan.site_description = The easy way to get, use and share data
```

Default value: none

This is for a description, or tag line for the site, as displayed in the header of the CKAN web interface.

ckan.site_intro_text

Example:

```
ckan.site_intro_text = Nice introductory paragraph about CKAN or the site in general.
```

Default value: none

This is for an introductory text used in the default template's index page.

ckan.site_logo

Example:

```
ckan.site_logo = /images/ckan_logo_fullname_long.png
```

Default value: `/base/images/ckan-logo.png`

This sets the logo used in the title bar.

ckan.site_about

Example:

```
ckan.site_about = A _community-driven_ catalogue of _open data_ for the Greenfield area.
```

Default value: none

Format tips:

- multiline strings can be used by indenting following lines
- the format is Markdown

Note: Whilst the default text is translated into many languages (switchable in the page footer), the text in this configuration option will not be translatable. For this reason, it's better to overload the snippet in `home/snippets/about_text.html`. For more information, see *Theming guide*.

ckan.theme

Example:

```
ckan.theme = my-extension/theme-asset
```

Default value: `css/main`

With this option, instead of using the default *css/main* asset with the theme, you can use your own.

ckan.favicon

Example:

```
ckan.favicon = http://okfn.org/wp-content/themes/okfn-master-wordpress-theme/images/
↪favicon.ico
```

Default value: `/base/images/ckan.ico`

This sets the site's *favicon*. This icon is usually displayed by the browser in the tab heading and bookmark.

ckan.datasets_per_page

Example:

```
ckan.datasets_per_page = 10
```

Default value: `20`

This controls the pagination of the dataset search results page. This is the maximum number of datasets viewed per page of results.

package_hide_extras

Example:

```
package_hide_extras = my_private_field other_field
```

Default value: none

This sets a space-separated list of extra field key values which will not be shown on the dataset read page.

> **Warning:** While this is useful to e.g. create internal notes, it is not a security measure. The keys will still be available via the API and in revision diffs.

ckan.recaptcha.publickey

Default value: none

The public key for your reCAPTCHA account, for example:

```
ckan.recaptcha.publickey = 6Lc...-KLc
```

To get a reCAPTCHA account, sign up at: http://www.google.com/recaptcha

ckan.recaptcha.privatekey

Default value: none

The private key for your reCAPTCHA account, for example:

```
ckan.recaptcha.privatekey = 6Lc...-jP
```

Setting both *ckan.recaptcha.publickey* and *ckan.recaptcha.privatekey* adds captcha to the user registration form. This has been effective at preventing bots registering users and creating spam packages.

ckan.featured_groups

Example:

```
ckan.featured_groups = group_one
```

Default value: none

Defines a list of group names or group ids. This setting is used to display a group and datasets on the home page in the default templates (1 group and 2 datasets are displayed).

ckan.featured_orgs

Example:

```
ckan.featured_orgs = org_one
```

Default value: none

Defines a list of organization names or ids. This setting is used to display an organization and datasets on the home page in the default templates (1 group and 2 datasets are displayed).

ckan.default_group_sort

Example:

```
ckan.default_group_sort = name
```

Default value: `title`

Defines if some other sorting is used in group_list and organization_list by default when the request does not specify sort.

ckan.gravatar_default

Example:

```
ckan.gravatar_default = disabled
```

Default value: `identicon`

This controls the default gravatar style. Gravatar is used by default when a user has not set a custom profile picture, but it can be turn completely off by setting this option to "disabled". In that case, a placeholder image will be shown instead, which can be customized overriding the `templates/user/snippets/placeholder.html` template.

ckan.debug_supress_header

Example:

```
ckan.debug_supress_header = false
```

Default value: `False`

This configs if the debug information showing the controller and action receiving the request being is shown in the header.

Note: This info only shows if debug is set to True.

ckan.site_custom_css

Default value: none

Custom CSS directives to include on all CKAN pages.

Resource Views Settings

ckan.views.default_views

Example:

```
ckan.views.default_views = image_view webpage_view datatables_view
```

Default value: `image_view datatables_view`

Defines the resource views that should be created by default when creating or updating a dataset. From this list only the views that are relevant to a particular resource format will be created. This is determined by each individual view.

If not present (or commented), the default value is used. If left empty, no default views are created.

Note: You must have the relevant view plugins loaded on the `ckan.plugins` setting to be able to create the default views, eg:: ckan.plugins = image_view webpage_view geo_view datatables_view … ckan.views.default_views = image_view webpage_view datatables_view

Theming Settings

ckan.template_title_delimiter

Example:

```
ckan.template_title_delimiter = |
```

Default value: -

This sets the delimiter between the site's subtitle (if there's one) and its title, in HTML's `<title>`.

extra_template_paths

Example:

```
extra_template_paths = /home/okfn/brazil_ckan_config/templates
```

Default value: none

Use this option to specify where CKAN should look for additional templates, before reverting to the `ckan/templates` folder. You can supply more than one folder, separating the paths with a comma (,).

For more information on theming, see *Theming guide*.

extra_public_paths

Example:

```
extra_public_paths = /home/okfn/brazil_ckan_config/public
```

Default value: none

To customise the display of CKAN you can supply replacements for static files such as HTML, CSS, script and PNG files. Use this option to specify where CKAN should look for additional files, before reverting to the `ckan/public` folder. You can supply more than one folder, separating the paths with a comma (,).

For more information on theming, see *Theming guide*.

ckan.base_public_folder

Example:

```
ckan.base_public_folder = public
```

Default value: `public`

This config option is used to configure the base folder for static files used by CKAN core. Starting CKAN 2.11 it only accepts: `public` as a value. (This variable is kept for backwards compatibility when updating Bootstrap versions.)

ckan.base_templates_folder

Example:

```
ckan.base_templates_folder = templates
```

Default value: `templates`

This config option is used to configure the base folder for templates used by CKAN core. Starting CKAN 2.11 it only accepts: `templates` as a value. (This variable is kept for backwards compatibility when updating Bootstrap versions.)

ckan.default.package_type

Default value: `dataset`

Default type of dataset that will be used in the UI links (eg. "New Dataset").

Use this option to change the dataset type that is used site-wide. Only existing dataset types can be used as a value for this option. Upon setting a custom value, the following happens:

- all new datasets have their `type` field set to the custom value(if no explicit value provided)
- all labels(e.g. "Create a Dataset", "My datasets", "Search datasets..") are adapted to the custom value
- all default links(e.g. `/dataset/new`, `/dataset/<name>/resource`) are adapted to the custom value

If labels require additional changes, register a chained helpers for *humanize_entity_type()*. For example, setting a dataset type `camel_photo` as default, will turn the "Datasets" link in the header into "Camel-photos". If "Camel Photos" is expected, the code below can be used:

```
@p.toolkit.chained_helper
def humanize_entity_type(next_helper: Callable[..., Any],
                         entity_type: str, object_type: str, purpose: str):
    if purpose == "main nav":
        return "Camel Photos"

    return next_helper(entity_type, object_type, purpose)
```

See *humanize_entity_type()* for additional details.

ckan.default.group_type

Default value: `group`

Default type of group that used in UI links(eg. "New Group" button, "Groups" link in header)

Same as *ckan.default.package_type*, but for groups.

ckan.default.organization_type

Default value: `organization`

Default type of group that used in UI links(eg. "New Organization" button, "Organizations" link in header)

Same as *ckan.default.package_type*, but for organizations.

Storage Settings

ckan.storage_path

Example:

```
ckan.storage_path = /var/lib/ckan/default
```

Default value: none

This defines the location of where CKAN will store all uploaded data.

ckan.max_resource_size

Example:

```
ckan.max_resource_size = 100
```

Default value: `10`

The maximum in megabytes a resources upload can be.

ckan.max_image_size

Example:

```
ckan.max_image_size = 10
```

Default value: `2`

The maximum in megabytes an image upload can be.

Uploader Settings

ckan.upload.user.types

Example:

```
ckan.upload.user.types = image text
```

Default value: `image`

File types allowed to upload as user's avatar. No restrictions applied when empty

ckan.upload.user.mimetypes

Example:

```
ckan.upload.user.mimetypes = image/png text/svg
```

Default value: `image/png image/gif image/jpeg`

File MIMETypes allowed to upload as user's avatar. No restrictions applied when empty

ckan.upload.group.types

Example:

```
ckan.upload.group.types = image text
```

Default value: `image`

File types allowed to upload as group image. No restrictions applied when empty

ckan.upload.group.mimetypes

Example:

```
ckan.upload.group.mimetypes = image/png text/svg
```

Default value: `image/png image/gif image/jpeg`

File MIMETypes allowed to upload as group image. No restrictions applied when empty

Webassets Settings

ckan.webassets.path

Example:

```
ckan.webassets.path = /var/lib/ckan/webassets
```

Default value: none

In order to increase performance, static assets (CSS and JS files) included via an `asset` tag inside templates are compiled only once, when the asset is used for the first time. All subsequent requests to the asset will use the existing file. CKAN stores the compiled webassets in the file system, in the path specified by this config option.

ckan.webassets.url

Example:

```
ckan.webassets.url = /serve/assets/from/here
```

Default value: `/webassets`

URL path for endpoint that serves webassets.

ckan.webassets.use_x_sendfile

Example:

```
ckan.webassets.use_x_sendfile = True
```

Default value: `False`

When serving static files, if this setting is `True`, the applicatin will set the `X-Sendfile` header instead of serving the files directly with Flask. This will increase performance when serving the assets, but it requires that the web server (eg Nginx) supports the `X-Sendfile` header. See *X-Sendfile* for more information.

User Settings

ckan.user_list_limit

Example:

```
ckan.user_list_limit = 50
```

Default value: `20`

This controls the number of users to show in the Users list. By default, it shows 20 users.

ckan.user_reset_landing_page

Example:

```
ckan.user_reset_landing_page = dataset
```

Default value: `home.index`

This controls the page where users will be sent after requesting a password reset. This is ordinarily the home page, but specific sites may prefer somewhere else.

Activity Streams Settings

ckan.activity_streams_enabled

Example:

```
ckan.activity_streams_enabled = false
```

Default value: `True`

Turns on and off the activity streams used to track changes on datasets, groups, users, etc. The activity feature has been moved to a separate activity plugin. To keep showing the activities in the UI and enable the activity related API actions you need to add the activity plugin to the ckan.plugins config option.

ckan.activity_streams_email_notifications

Example:

```
ckan.activity_streams_email_notifications = false
```

Default value: `False`

Turns on and off the activity streams' email notifications. You'd also need to setup a cron job to send the emails. For more information, visit *Email notifications*.

ckan.activity_list_limit

Example:

```
ckan.activity_list_limit = 31
```

Default value: `31`

This controls the number of activities to show in the Activity Stream.

ckan.activity_list_limit_max

Example:

```
ckan.activity_list_limit_max = 100
```

Default value: `100`

Maximum allowed value for Activity Stream `limit` parameter.

ckan.email_notifications_since

Example:

```
ckan.email_notifications_since = 2 days
```

Default value: `2 days`

Email notifications for events older than this time delta will not be sent. Accepted formats: '2 days', '14 days', '4:35:00' (hours, minutes, seconds), '7 days, 3:23:34', etc.

ckan.hide_activity_from_users

Example:

```
ckan.hide_activity_from_users = sysadmin
```

Default value: none

Hides activity from the specified users from activity stream. If unspecified, it'll use *ckan.site_id* to hide activity by the site user. The site user is a sysadmin user on every ckan user with a username that's equal to *ckan.site_id*. This user is used by ckan for performing actions from the command-line.

Feeds Settings

ckan.feeds.author_name

Example:

```
ckan.feeds.author_name = Michael Jackson
```

Default value: none

This controls the feed author's name. If unspecified, it'll use *ckan.site_id*.

ckan.feeds.author_link

Example:

```
ckan.feeds.author_link = http://okfn.org
```

Default value: none

This controls the feed author's link. If unspecified, it'll use *ckan.site_url*.

ckan.feeds.authority_name

Example:

```
ckan.feeds.authority_name = http://okfn.org
```

Default value: none

The domain name or email address of the default publisher of the feeds and elements. If unspecified, it'll use *ckan.site_url*.

ckan.feeds.date

Example:

```
ckan.feeds.date = 2012-03-22
```

Default value: none

A string representing the default date on which the authority_name is owned by the publisher of the feed.

ckan.feeds.limit

Default value: `20`

Number of items returned in the feeds

Internationalisation Settings

ckan.locale_default

Example:

```
ckan.locale_default = de
```

Default value: `en`

Use this to specify the locale (language of the text) displayed in the CKAN Web UI. This requires a suitable *mo* file installed for the locale in the ckan/i18n. For more information on internationalization, see *Translating CKAN*. If you don't specify a default locale, then it will default to the first locale offered, which is by default English (alter that with *ckan.locales_offered* and *ckan.locales_filtered_out*.

ckan.locales_offered

Example:

```
ckan.locales_offered = en de fr
```

Default value: none

By default, all locales found in the `ckan/i18n` directory will be offered to the user. To only offer a subset of these, list them under this option. The ordering of the locales is preserved when offered to the user.

ckan.locales_filtered_out

Example:

```
ckan.locales_filtered_out = pl ru
```

Default value: none

If you want to not offer particular locales to the user, then list them here to have them removed from the options.

ckan.locale_order

Example:

```
ckan.locale_order = fr de
```

Default value: none

If you want to specify the ordering of all or some of the locales as they are offered to the user, then specify them here in the required order. Any locales that are available but not specified in this option, will still be offered at the end of the list.

ckan.i18n_directory

Example:

```
ckan.i18n_directory = /opt/locales/i18n/
```

Default value: none

By default, the locales are searched for in the `ckan/i18n` directory. Use this option if you want to use another folder.

ckan.i18n.extra_directory

Example:

```
ckan.i18n.extra_directory = /opt/ckan/extra_translations/
```

Default value: none

If you wish to add extra translation strings and have them merged with the default ckan translations at runtime you can specify the location of the extra translations using this option.

ckan.i18n.extra_gettext_domain

Example:

```
ckan.i18n.extra_gettext_domain = mydomain
```

Default value: none

You can specify the name of the gettext domain of the extra translations. For example if your translations are stored as `i18n/<locale>/LC_MESSAGES/somedomain.mo` you would want to set this option to `somedomain`

ckan.i18n.extra_locales

Example:

```
ckan.i18n.extra_locales = fr es de
```

Default value: none

If you have set an extra i18n directory using `ckan.i18n.extra_directory`, you should specify the locales that have been translated in that directory in this option.

ckan.i18n.rtl_languages

Example:

```
ckan.i18n.rtl_languages = he ar fa_IR
```

Default value: `he ar fa_IR`

Allows to modify the right-to-left languages

ckan.i18n.rtl_theme

Example:

```
ckan.i18n.rtl_theme = my-extension/my-custom-rtl-asset
```

Default value: `css/main-rtl`

Allows to override the default rtl asset used for the languages defined in `ckan.i18n.rtl_languages`.

ckan.display_timezone

Example:

```
ckan.display_timezone = Europe/Zurich
```

Default value: `UTC`

By default, all datetimes are considered to be in the UTC timezone. Use this option to change the displayed dates on the frontend. Internally, the dates are always saved as UTC. This option only changes the way the dates are displayed.

The valid values for this options [can be found at pytz](http://pytz.sourceforge.net/#helpers) (`pytz.all_timezones`). You can specify the special value *server* to use the timezone settings of the server, that is running CKAN.

ckan.root_path

Example:

```
ckan.root_path = /my/custom/path/{{LANG}}/foo
```

Default value: none

This setting is used to construct URLs inside CKAN. It specifies two things:

- *At which path CKAN is mounted:* By default it is assumed that CKAN is mounted at `/`, i.e. at the root of your web server. If you have configured your web server to serve CKAN from a different mount point then you need to duplicate that setting here.
- *Where the locale is added to an URL:* By default, URLs are formatted as `/some/url` when using the default locale, or `/de/some/url` when using the `de` locale, for example. When `ckan.root_path` is set it must include the string `{{LANG}}`, which will be replaced by the locale.

Important: The setting must contain `{{LANG}}` exactly as written here. Do not add spaces between the brackets.

See also:

The host of your CKAN installation can be set via *ckan.site_url*.

The CKAN repoze config file `who.ini` file will also need to be edited by adding the path prefix to the options in the `[plugin:friendlyform]` section: `login_form_url`, `post_login_url` and `post_logout_url`. Do not change the login/logout_handler_path options.

ckan.resource_formats

Example:

```
ckan.resource_formats = /path/to/resource_formats
```

Default value: `/<CKAN_ROOT>/ckan/config/resource_formats.json`

The purpose of this file is to supply a thorough list of resource formats and to make sure the formats are normalized when saved to the database and presented.

The format of the file is a JSON object with following format:

```
["Format", "Description", "Mimetype", ["List of alternative representations"]]
```

Please look in ckan/config/resource_formats.json for full details and and as an example.

Form Settings

ckan.dataset.create_on_ui_requires_resources

Example:

```
ckan.dataset.create_on_ui_requires_resources = false
```

Default value: `True`

If False, there is no need to add any resources when creating a new dataset.

package_new_return_url

Default value: none

The URL to redirect the user to after they've submitted a new package form, example:

```
package_new_return_url = http://datadotgc.ca/new_dataset_complete?name=<NAME>
```

This is useful for integrating CKAN's new dataset form into a third-party interface, see *Form Integration*.

The `<NAME>` string is replaced with the name of the dataset created.

package_edit_return_url

Default value: none

The URL to redirect the user to after they've submitted an edit package form, example:

```
package_edit_return_url = http://datadotgc.ca/dataset/<NAME>
```

This is useful for integrating CKAN's edit dataset form into a third-party interface, see *Form Integration*.

The `<NAME>` string is replaced with the name of the dataset that was edited.

licenses_group_url

Example:

```
licenses_group_url = file:///path/to/my/local/json-list-of-licenses.json
```

Default value: none

A url pointing to a JSON file containing a list of license objects. This list determines the licenses offered by the system to users, for example when creating or editing a dataset.

This is entirely optional - by default, the system will use an internal cached version of the CKAN list of licenses available from the http://licenses.opendefinition.org/licenses/groups/ckan.json.

More details about the license objects - including the license format and some example license lists - can be found at the Open Licenses Service.

Email settings

smtp.server

Example:

```
smtp.server = smtp.example.com:587
```

Default value: `localhost`

The SMTP server to connect to when sending emails with optional port.

smtp.starttls

Example:

```
smtp.starttls = true
```

Default value: `False`

Whether or not to use STARTTLS when connecting to the SMTP server.

smtp.user

Example:

```
smtp.user = username@example.com
```

Default value: none

The username used to authenticate with the SMTP server.

smtp.password

Example:

```
smtp.password = yourpass
```

Default value: none

The password used to authenticate with the SMTP server.

smtp.mail_from

Example:

```
smtp.mail_from = ckan@example.com
```

Default value: none

The email address that emails sent by CKAN will come from. Note that, if left blank, the SMTP server may insert its own.

smtp.reply_to

Example:

```
smtp.reply_to = noreply.example.com
```

Default value: none

The email address that will be used if someone attempts to reply to a system email. If left blank, no `Reply-to` will be added to the email and the value of `smtp.mail_from` will be used.

email_to

Example:

```
email_to = errors@example.com
```

Default value: none

This controls where the error messages will be sent to.

error_email_from

Example:

```
error_email_from = ckan-errors@example.com
```

Default value: none

This controls from which email the error messages will come from.

Background Job Settings

ckan.jobs.timeout

Default value: `180`

The option defines the timeout in seconds until giving up on a job

Resource Proxy settings

ckan.resource_proxy.max_file_size

Example:

```
ckan.resource_proxy.max_file_size = 1048576
```

Default value: `1048576`

This sets the upper file size limit for in-line previews. Increasing the value allows CKAN to preview larger files (e.g. PDFs) in-line; however, a higher value might cause time-outs, or unresponsive browsers for CKAN users with lower bandwidth.

ckan.resource_proxy.chunk_size

Example:

```
ckan.resource_proxy.chunk_size = 8192
```

Default value: `4096`

This sets size of the chunk to read and write when proxying. Raising this value might save some CPU cycles. It makes no sense to lower it below the page size, which is default.

text_view settings

ckan.preview.text_formats

Example:

```
ckan.preview.text_formats = txt plain
```

Default value: `text/plain txt plain`

Space-delimited list of plain text based resource formats that will be rendered by the Text view plugin

ckan.preview.xml_formats

Example:

```
ckan.preview.xml_formats = xml rdf rss
```

Default value: `xml rdf rdf+xml owl+xml atom rss`

Space-delimited list of XML based resource formats that will be rendered by the Text view plugin

ckan.preview.json_formats

Example:

```
ckan.preview.json_formats = json
```

Default value: `json`

Space-delimited list of JSON based resource formats that will be rendered by the Text view plugin

ckan.preview.jsonp_formats

Default value: `jsonp`

Space-delimited list of JSONP based resource formats that will be rendered by the Text view plugin

image_view settings

ckan.preview.image_formats

Example:

```
ckan.preview.image_formats = png jpeg jpg gif
```

Default value: `png jpeg jpg gif`

Space-delimited list of image-based resource formats that will be rendered by the Image view plugin

datatables_view settings

ckan.datatables.page_length_choices

Example:

```
ckan.datatables.page_length_choices = 20 50 100 500 1000 5000
```

Default value: `20 50 100 500 1000`

Space-delimited list of the choices for the number of rows per page, with the lowest value being the default initial value.

Note: On larger screens, DataTables view will attempt to fill the table with as many rows that can fit using the lowest closest choice.

ckan.datatables.state_saving

Example:

```
ckan.datatables.state_saving = false
```

Default value: `True`

Enable or disable state saving. When enabled, DataTables view will store state information such as pagination position, page length, row selection/s, column visibility/ordering, filtering and sorting using the browser's localStorage. When the end user reloads the page, the table's state will be altered to match what they had previously set up.

This also enables/disables the "Reset" and "Share current view" buttons. "Reset" discards the saved state. "Share current view" base-64 encodes the state and passes it as a url parameter, acting like a "saved search" that can be used for embedding and sharing table searches.

ckan.datatables.state_duration

Example:

```
ckan.datatables.state_duration = 86400
```

Default value: `7200`

Duration (in seconds) for which the saved state information is considered valid. After this period has elapsed, the table's state will be returned to the default, and the state cleared from the browser's localStorage.

Note: The value `0` is a special value as it indicates that the state can be stored and retrieved indefinitely with no time limit.

ckan.datatables.data_dictionary_labels

Example:

```
ckan.datatables.data_dictionary_labels = false
```

Default value: `True`

Enable or disable data dictionary integration. When enabled, a column's data dictionary label will be used in the table header. A tooltip for each column with data dictionary information will also be integrated into the header.

ckan.datatables.ellipsis_length

Example:

```
ckan.datatables.ellipsis_length = 100
```

Default value: `100`

The maximum number of characters to show in a cell before it is truncated. An ellipsis (...) will be added at the truncation point and the full text of the cell will be available as a tooltip. This value can be overridden at the resource level when configuring a DataTables resource view.

Note: The value `0` is a special value as it indicates that the column's width will be determined by the column name, and cell content will word-wrap.

ckan.datatables.date_format

Example:

```
ckan.datatables.date_format = YYYY-MM-DD dd ww
```

Default value: `llll`

The moment.js date format to use to convert raw timestamps to a user-friendly date format using CKAN's current locale language code. This value can be overridden at the resource level when configuring a DataTables resource view.

Note: The value `NONE` is a special value as it indicates that no date formatting will be applied and the raw ISO-8601 timestamp will be displayed.

ckan.datatables.default_view

Example:

```
ckan.datatables.default_view = list
```

Default value: `table`

Indicates the default view mode of the DataTable (valid values: `table` or `list`). Table view is the typical grid layout, with horizontal scrolling. List view is a responsive table, automatically hiding columns as required to fit the browser

viewport. In addition, list view allows the user to view, copy and print the details of a specific row. This value can be overridden at the resource level when configuring a DataTables resource view.

ckan.datatables.null_label

Example:

```
ckan.datatables.null_label = N/A
```

Default value: none

The option defines the label used to display NoneType values for the front-end. This should be a string and can be translated via po files.

Datastore settings

ckan.datastore.write_url

Example:

```
ckan.datastore.write_url = postgresql://ckanuser:pass@localhost/datastore
```

Default value: `postgresql://ckan_default:pass@localhost/datastore_default`

The database connection to use for writing to the datastore (this can be ignored if you're not using the *DataStore extension*). Note that the database used should not be the same as the normal CKAN database. The format is the same as in *sqlalchemy.url*.

ckan.datastore.read_url

Example:

```
ckan.datastore.read_url = postgresql://readonlyuser:pass@localhost/datastore
```

Default value: `postgresql://datastore_default:pass@localhost/datastore_default`

The database connection to use for reading from the datastore (this can be ignored if you're not using the *DataStore extension*). The database used must be the same used in *ckan.datastore.write_url*, but the user should be one with read permissions only. The format is the same as in *sqlalchemy.url*.

ckan.datastore.sqlsearch.allowed_functions_file

Example:

```
ckan.datastore.sqlsearch.allowed_functions_file = /path/to/my_allowed_functions.txt
```

Default value: `/<CKAN_ROOT>/ckanext/datastore/allowed_functions.txt`

Allows to define the path to a text file listing the SQL functions that should be allowed to run on queries sent to the `datastore_search_sql()` function (if enabled, see *ckan.datastore.sqlsearch.enabled*). Function names should be listed one on each line, eg:

```
abbrev
abs
abstime
...
```

ckan.datastore.sqlsearch.enabled

Example:

```
ckan.datastore.sqlsearch.enabled = true
```

Default value: `False`

This option allows you to enable the *datastore_search_sql()* action function, and corresponding API endpoint.

This action function has protections from abuse including:

- parsing of the query to prevent unsafe functions from being called, see *ckan.datastore.sqlsearch.allowed_functions_file*
- parsing of the query to prevent multiple statements
- prevention of data modification by using a read-only database role
- use of `explain` to resolve tables accessed in the query to check against user permissions
- use of a statement timeout to prevent queries from running indefinitely

These protections offer some safety but are not designed to prevent all types of abuse. Depending on the sensitivity of private data in your datastore and the likelihood of abuse of your site you may choose to disable this action function or restrict its use with a *IAuthFunctions* plugin.

ckan.datastore.search.rows_default

Example:

```
ckan.datastore.search.rows_default = 1000
```

Default value: `100`

Default number of rows returned by `datastore_search`, unless the client specifies a different `limit` (up to `ckan.datastore.search.rows_max`).

NB this setting does not affect `datastore_search_sql`.

ckan.datastore.search.rows_max

Example:

```
ckan.datastore.search.rows_max = 1000000
```

Default value: `32000`

Maximum allowed value for the number of rows returned by the datastore.

Specifically this limits:

- `datastore_search`'s `limit` parameter.
- `datastore_search_sql` queries have this limit inserted.

ckan.datastore.sqlalchemy.<OPTION>

Default value: none

Custom sqlalchemy config parameters used to establish the DataStore database connection.

To get the list of all the available properties check the SQLAlchemy documentation

ckan.datastore.default_fts_lang

Example:

```
ckan.datastore.default_fts_lang = english
```

Default value: `english`

The default language used when creating full-text search indexes and querying them. It can be overwritten by the user by passing the "lang" parameter to "datastore_search" and "datastore_create".

ckan.datastore.default_fts_index_method

Example:

```
ckan.datastore.default_fts_index_method = gist
```

Default value: `gist`

The default method used when creating full-text search indexes. Currently it can be "gin" or "gist". Refer to PostgreSQL's documentation to understand the characteristics of each one and pick the best for your instance.

Datapusher settings

ckan.datapusher.formats

Example:

```
ckan.datapusher.formats = csv xls
```

Default value: `csv xls xlsx tsv application/csv application/vnd.ms-excel application/vnd.openxmlformats-officedocument.spreadsheetml.sheet ods application/vnd.oasis.opendocument.spreadsheet`

File formats that will be pushed to the DataStore by the DataPusher. When adding or editing a resource which links to a file in one of these formats, the DataPusher will automatically try to import its contents to the DataStore.

ckan.datapusher.url

Example:

```
ckan.datapusher.url = http://127.0.0.1:8800/
```

Default value: none

DataPusher endpoint to use when enabling the `datapusher` extension. If you installed CKAN via *Installing CKAN from package*, the DataPusher was installed for you running on port 8800. If you want to manually install the DataPusher, follow the installation instructions.

ckan.datapusher.api_token

Default value: none

Starting from CKAN 2.10, DataPusher requires a valid API token to operate (see *Authentication and API tokens*), and will fail to start if this option is not set.

ckan.datapusher.callback_url_base

Example:

```
ckan.datapusher.callback_url_base = http://ckan:5000/
```

Default value: none

Alternative callback URL for DataPusher when performing a request to CKAN. This is useful on scenarios where the host where DataPusher is running can not access the public CKAN site URL.

ckan.datapusher.assume_task_stale_after

Example:

```
ckan.datapusher.assume_task_stale_after = 86400
```

Default value: `3600`

In case a DataPusher task gets stuck and fails to recover, this is the minimum amount of time (in seconds) after a resource is submitted to DataPusher that the resource can be submitted again.

CHAPTER

FOUR

API GUIDE

This section documents CKAN APIs, for developers who want to write code that interacts with CKAN sites and their data.

CKAN's **Action API** is a powerful, RPC-style API that exposes all of CKAN's core features to API clients. All of a CKAN website's core functionality (everything you can do with the web interface and more) can be used by external code that calls the CKAN API. For example, using the CKAN API your app can:

- Get JSON-formatted lists of a site's datasets, groups or other CKAN objects:

  http://demo.ckan.org/api/3/action/package_list

  http://demo.ckan.org/api/3/action/group_list

  http://demo.ckan.org/api/3/action/tag_list

- Get a full JSON representation of a dataset, resource or other object:

  http://demo.ckan.org/api/3/action/package_show?id=adur_district_spending

  http://demo.ckan.org/api/3/action/tag_show?id=gold

  http://demo.ckan.org/api/3/action/group_show?id=data-explorer

- Search for packages or resources matching a query:

  http://demo.ckan.org/api/3/action/package_search?q=spending

  http://demo.ckan.org/api/3/action/resource_search?query=name:District%20Names

- Create, update and delete datasets, resources and other objects
- Get an activity stream of recently changed datasets on a site:

  http://demo.ckan.org/api/3/action/recently_changed_packages_activity_list

Note: CKAN's FileStore and DataStore have their own APIs, see:

- *FileStore and file uploads*
- *DataStore extension*

Note: For documentation of CKAN's legacy API's, see *Legacy APIs*.

4.1 Legacy APIs

Warning: The legacy APIs documented in this section are provided for backwards-compatibility, but support for new CKAN features will not be added to these APIs. These endpoints will be removed in the future.

Note: The REST API was deprecated in CKAN v2.0 and removed starting from CKAN v2.8.

4.1.1 Search API

Search resources are available at published locations. They are represented with a variety of data formats. Each resource location supports a number of methods.

The data formats of the requests and the responses are defined below.

Search Resources

Here are the published resources of the Search API.

Search Resource	Location
Dataset Search	`/search/dataset`
Resource Search	`/search/resource`
Revision Search	`/search/revision`
Tag Counts	`/tag_counts`

See below for more information about dataset and revision search parameters.

Search Methods

Here are the methods of the Search API.

Resource	Method	Request	Response
Dataset Search	POST	Dataset-Search-Params	Dataset-Search-Response
Resource Search	POST	Resource-Search-Params	Resource-Search-Response
Revision Search	POST	Revision-Search-Params	Revision-List
Tag Counts	GET		Tag-Count-List

It is also possible to supply the search parameters in the URL of a GET request, for example `/api/search/dataset?q=geodata&allfields=1`.

Search Formats

Here are the data formats for the Search API.

Name	Format
Dataset-Search-Params Resource-Search-Params Revision-Search-Params	{ Param-Key: Param-Value, Param-Key: Param-Value, … } See below for full details of search parameters across the various domain objects.
Dataset-Search-Response	{ count: Count-int, results: [Dataset, Dataset, …] }
Resource-Search-Response	{ count: Count-int, results: [Resource, Resource, …] }
Revision-List	[Revision-Id, Revision-Id, Revision-Id, …] NB: Ordered with youngest revision first. NB: Limited to 50 results at a time.
Tag-Count-List	[[Name-String, Integer], [Name-String, Integer], …]

Dataset Parameters

Param-Key	Param-Value	Examples	Notes
q	Search-String	q=geodata q=government+sweden q=%22drug%20abuse%22 q=tags:"river pollution"	Criteria to search the dataset fields for. URL-encoded search text. (You can also concatenate words with a '+' symbol in a URL.) Search results must contain all the specified words. You can also search within specific fields.
qjson	JSON encoded options	['q':'geodata']	All search parameters can be json-encoded and supplied to this parameter as a more flexible alternative in GET requests.
title, tags, notes, groups, author, maintainer, update_frequency, or any 'extra' field name e.g. department	Search-String	title=uk&tags=health department=environment tags=health&tags=pollution tags=river%20pollution	Search in a particular a field.
order_by	field-name (default=rank)	order_by=name	Specify either rank or the field to sort the results by
offset, limit	result-int (defaults: offset=0, limit=20)	offset=40&limit=20	Pagination options. Offset is the number of the first result and limit is the number of results to return.
all_fields	0 (default) or 1	all_fields=1	Each matching search result is given as either a dataset name (0) or the full dataset record (1).

Note: filter_by_openness and filter_by_downloadable were dropped from CKAN version 1.5 onwards.

Note: Only public datasets can be accessed via the legacy search API, regardless of the provided authorization. If you need to access private datasets via the API you will need to use the *package_search* method of the *API guide*.

Resource Parameters

Param-Key	Param-Value	Example	Notes
url, format, description	Search-String	url=statistics.org format=xls description=Research+Instit	Criteria to search the dataset fields for. URL-encoded search text. This search string must be found somewhere within the field to match. Case insensitive.
qjson	JSON encoded options	['url':'www.statistics.org']	All search parameters can be json-encoded and supplied to this parameter as a more flexible alternative in GET requests.
hash	Search-String	hash=b0d7c260-35d4-42ab-9e3d-c1f4db9bc2f0	Searches for an match of the hash field. An exact match or match up to the length of the hash given.
all_fields	0 (default) or 1	all_fields=1	Each matching search result is given as either an ID (0) or the full resource record
offset, limit	result-int (defaults: offset=0, limit=20)	offset=40&limit=20	Pagination options. Offset is the number of the first result and limit is the number of results to return.

Note: Powerful searching from the command-line can be achieved with curl and the qjson parameter. In this case you need to remember to escapt the curly braces and use url encoding (e.g. spaces become `%20`). For example:

```
curl 'http://thedatahub.org/api/search/dataset?qjson=\{"author":"The%20Stationery
↪%20Office%20Limited"\}'
```

Revision Parameters

Param-Key	Param-Value	Example	Notes
since_time	Date-Time	since_time=2010-05-05T19:42:45.854533	The time can be less precisely stated (e.g 2010-05-05).
since_id	Uuid	since_id=6c9f32ef-1f93-4b2f-891b-fd01924ebe08	The stated id will not be included in the results.

4.1.2 Util API

The Util API provides various utility APIs – e.g. auto-completion APIs used by front-end javascript.

All Util APIs are read-only. The response format is JSON. Javascript calls may want to use the JSONP formatting.

dataset autocomplete

There an autocomplete API for package names which matches on name or title.

This URL:

```
/api/2/util/dataset/autocomplete?incomplete=a%20novel
```

Returns:

```
{"ResultSet": {"Result": [{"match_field": "title", "match_displayed": "A Novel By␣
↪Tolstoy (annakarenina)", "name": "annakarenina", "title": "A Novel By Tolstoy"}]}}
```

tag autocomplete

There is also an autocomplete API for tags which looks like this:

This URL:

```
/api/2/util/tag/autocomplete?incomplete=ru
```

Returns:

```
{"ResultSet": {"Result": [{"Name": "russian"}]}}
```

resource format autocomplete

Similarly, there is an autocomplete API for the resource format field which is available at:

```
/api/2/util/resource/format_autocomplete?incomplete=cs
```

This returns:

```
{"ResultSet": {"Result": [{"Format": "csv"}]}}
```

munge package name

For taking an readable identifier and munging it to ensure it is a valid dataset id. Symbols and whitespeace are converted into dashes. Example:

```
/api/util/dataset/munge_name?name=police%20spending%20figures%202009
```

Returns:

```
"police-spending-figures-2009"
```

munge title to package name

For taking a title of a package and munging it to a readable and valid dataset id. Symbols and whitespeace are converted into dashes, with multiple dashes collapsed. Ensures that long titles with a year at the end preserves the year should it need to be shortened. Example:

```
/api/util/dataset/munge_title_to_name?title=police:%20spending%20figures%202009
```

Returns:

```
"police-spending-figures-2009"
```

munge tag

For taking a readable word/phrase and munging it to a valid tag (name). Symbols and whitespeace are converted into dashes. Example:

```
/api/util/tag/munge?tag=water%20quality
```

Returns:

```
"water-quality"
```

4.1.3 Status Codes

Standard HTTP status codes are used to signal method outcomes.

Code	Name
200	OK
201	OK and new object created (referred to in the Location header)
301	Moved Permanently
400	Bad Request
403	Not Authorized
404	Not Found
409	Conflict (e.g. name already exists)
500	Service Error

Note: On early CKAN versions, datasets were called "packages" and this name has stuck in some places, specially internally and on API calls. Package has exactly the same meaning as "dataset".

4.2 Making an API request

To call the CKAN API, post a JSON dictionary in an HTTP POST request to one of CKAN APIs URLs. The parameters for the API function should be given in the JSON dictionary. CKAN will also return its response in a JSON dictionary.

One way to post a JSON dictionary to a URL is using the command-line client Curl. For example, to get a list of the names of all the datasets in the `data-explorer` group on demo.ckan.org, install curl and then call the `group_list` API function by running this command in a terminal:

```
curl https://demo.ckan.org/api/3/action/group_list
```

The response from CKAN will look like this:

```
{
    "help": "...",
    "result": [
        "data-explorer",
        "department-of-ricky",
        "geo-examples",
        "geothermal-data",
        "reykjavik",
        "skeenawild-conservation-trust"
    ],
    "success": true
}
```

The response is a JSON dictionary with three keys:

1. `"success"`: `true` or `false`.

   The API aims to always return `200 OK` as the status code of its HTTP response, whether there were errors with the request or not, so it's important to always check the value of the `"success"` key in the response dictionary and (if success is `false`) check the value of the `"error"` key.

Note: If there are major formatting problems with a request to the API, CKAN may still return an HTTP response with a `409`, `400` or `500` status code (in increasing order of severity). In future CKAN versions we intend to remove these responses, and instead send a `200 OK` response and use the `"success"` and `"error"` items.

2. `"result"`: the returned result from the function you called. The type and value of the result depend on which function you called. In the case of the `group_list` function it's a list of strings, the names of all the datasets that belong to the group.

   If there was an error responding to your request, the dictionary will contain an `"error"` key with details of the error instead of the `"result"` key. A response dictionary containing an error will look like this:

   ```
   {
       "help": "Creates a package",
       "success": false,
       "error": {
           "message": "Access denied",
           "__type": "Authorization Error"
           }
    }
   ```

3. `"help"`: the documentation string for the function you called.

The same HTTP request can be made using Python's standard `urllib2` module, with this Python code:

```
#!/usr/bin/env python
import urllib2
import urllib
import json
import pprint

# Make the HTTP request.
response = urllib2.urlopen('http://demo.ckan.org/api/3/action/group_list',
        data_string)
assert response.code == 200

# Use the json module to load CKAN's response into a dictionary.
response_dict = json.loads(response.read())

# Check the contents of the response.
assert response_dict['success'] is True
result = response_dict['result']
pprint.pprint(result)
```

4.3 Example: Importing datasets with the CKAN API

You can add datasets using CKAN's web interface, but when importing many datasets it's usually more efficient to automate the process in some way. In this example, we'll show you how to use the CKAN API to write a Python script to import datasets into CKAN.

Todo: Make this script more interesting (eg. read data from a CSV file), and all put the script in a .py file somewhere with tests and import it here.

```
#!/usr/bin/env python
import urllib2
import urllib
import json
import pprint

# Put the details of the dataset we're going to create into a dict.
dataset_dict = {
    'name': 'my_dataset_name',
    'notes': 'A long description of my dataset',
    'owner_org': 'org_id_or_name'
}

# Use the json module to dump the dictionary to a string for posting.
data_string = urllib.quote(json.dumps(dataset_dict))

# We'll use the package_create function to create a new dataset.
request = urllib2.Request(
    'http://www.my_ckan_site.com/api/action/package_create')
```

(continues on next page)

(continued from previous page)

```
# Creating a dataset requires an authorization header.
# Replace *** with your API key, from your user account on the CKAN site
# that you're creating the dataset on.
request.add_header('Authorization', '***')

# Make the HTTP request.
response = urllib2.urlopen(request, data_string)
assert response.code == 200

# Use the json module to load CKAN's response into a dictionary.
response_dict = json.loads(response.read())
assert response_dict['success'] is True

# package_create returns the created package as its result.
created_package = response_dict['result']
pprint.pprint(created_package)
```

For more examples, see *API Examples*.

4.4 API versions

The CKAN APIs are versioned. If you make a request to an API URL without a version number, CKAN will choose the latest version of the API:

```
http://demo.ckan.org/api/action/package_list
```

Alternatively, you can specify the desired API version number in the URL that you request:

```
http://demo.ckan.org/api/3/action/package_list
```

Version 3 is currently the only version of the Action API.

We recommend that you specify the API version number in your requests, because this ensures that your API client will work across different sites running different version of CKAN (and will keep working on the same sites, when those sites upgrade to new versions of CKAN). Because the latest version of the API may change when a site is upgraded to a new version of CKAN, or may differ on different sites running different versions of CKAN, the result of an API request that doesn't specify the API version number cannot be relied on.

4.5 Authentication and API tokens

Warning: Starting from CKAN 2.9, API tokens are the preferred way of authenticating API calls. The old legacy API keys will still work but they will be removed in future versions so it is recommended to switch to use API tokens. Read below for more details.

Some API functions require authorization. The API uses the same authorization functions and configuration as the web interface, so if a user is authorized to do something in the web interface they'll be authorized to do it via the API as well.

When calling an API function that requires authorization, you must authenticate yourself by providing an authentication key with your HTTP request. Starting from CKAN 2.9 the recommended mechanism to use are API tokens. These are encrypted keys that can be generated manually from the UI (User Profile > Manage > API tokens) or via the *api_token_create()* function. A user can create as many tokens as needed for different uses, and revoke one or multiple tokens at any time. In addition, enabling the `expire_api_token` core plugin allows to define the expiration timestamp for a token.

Site maintainers can use *API Token Settings* to configure the token generation.

Legacy API keys (UUIDs that look like *ec5c0860-9e48-41f3-8850-4a7128b18df8*) are still supported, but its use is discouraged as they are not as secure as tokens and are limited to one per user. Support for legacy API keys will be removed in future CKAN versions.

To provide your API token in an HTTP request, include it in an `Authorization` header. (The name of the HTTP header can be configured with the :ref:`apitoken_header_name` option in your CKAN configuration file.)

For example, to ask whether or not you're currently following the user `markw` on demo.ckan.org using curl, run this command:

```
curl -H "Authorization: XXX"  https://demo.ckan.org/api/3/action/am_following_user?
↪id=markw
```

(Replacing `XXX` with your API token.)

Or, to get the list of activities from your user dashboard on demo.ckan.org, run this Python code:

```
request = urllib2.Request('https://demo.ckan.org/api/3/action/dashboard_activity_list')
request.add_header('Authorization', 'XXX')
response_dict = json.loads(urllib2.urlopen(request, '{}').read())
```

4.6 GET-able API functions

Functions defined in *ckan.logic.action.get* can also be called with an HTTP GET request. For example, to get the list of datasets (packages) from demo.ckan.org, open this URL in your browser:

http://demo.ckan.org/api/3/action/package_list

Or, to search for datasets (packages) matching the search query `spending`, on demo.ckan.org, open this URL in your browser:

http://demo.ckan.org/api/3/action/package_search?q=spending

Tip: Browser plugins like JSONView for Firefox or Chrome will format and color CKAN's JSON response nicely in your browser.

The search query is given as a URL parameter `?q=spending`. Multiple URL parameters can be appended, separated by & characters, for example to get only the first 10 matching datasets open this URL:

http://demo.ckan.org/api/3/action/package_search?q=spending&rows=10

When an action requires a list of strings as the value of a parameter, the value can be sent by giving the parameter multiple times in the URL:

http://demo.ckan.org/api/3/action/term_translation_show?terms=russian&terms=romantic%20novel

4.7 JSONP support

To cater for scripts from other sites that wish to access the API, the data can be returned in JSONP format, where the JSON data is 'padded' with a function call. The function is named in the 'callback' parameter. For example:

http://demo.ckan.org/api/3/action/package_show?id=adur_district_spending&callback=myfunction

Note: This only works for GET requests

4.8 API Examples

4.8.1 Tags (not in a vocabulary)

A list of all tags:

- browser: http://demo.ckan.org/api/3/action/tag_list
- curl: `curl http://demo.ckan.org/api/3/action/tag_list`
- ckanapi: `ckanapi -r http://demo.ckan.org action tag_list`

Top 10 tags used by datasets:

- browser: http://demo.ckan.org/api/action/package_search?facet.field={[}%22tags%22{]}&facet.limit=10&rows=0
- curl: `curl 'http://demo.ckan.org/api/action/package_search?facet.field=\["tags"\]&facet.limit=10&rows=0'`
- ckanapi: `ckanapi -r http://demo.ckan.org action package_search facet.field='["tags"]' facet.limit=10 rows=0`

All datasets that have tag 'economy':

- browser: http://demo.ckan.org/api/3/action/package_search?fq=tags:economy
- curl: `curl 'http://demo.ckan.org/api/3/action/package_search?fq=tags:economy'`
- ckanapi: `ckanapi -r http://demo.ckan.org action package_search fq='tags:economy'`

4.8.2 Tag Vocabularies

Top 10 tags and vocabulary tags used by datasets:

- browser: http://demo.ckan.org/api/action/package_search?facet.field={[}%22tags%22{]}&facet.limit=10&rows=0
- curl: `curl 'http://demo.ckan.org/api/action/package_search?facet.field=\["tags"\]&facet.limit=10&rows=0'`
- ckanapi: `ckanapi -r http://demo.ckan.org action package_search facet.field='["tags"]' facet.limit=10 rows=0`

e.g. Facet: *vocab_Topics* means there is a vocabulary called Topics, and its top tags are listed under it.

A list of datasets using tag 'education' from vocabulary 'Topics':

- browser: https://data.hdx.rwlabs.org/api/3/action/package_search?fq=vocab_Topics:education

- curl: `curl 'https://data.hdx.rwlabs.org/api/3/action/package_search?fq=vocab_Topics:education'`
- ckanapi: `ckanapi -r https://data.hdx.rwlabs.org action package_search fq='vocab_Topics:education'`

4.8.3 Uploading a new version of a resource file

You can use the `upload` parameter of the *`resource_patch()`* function to upload a new version of a resource file. This requires a `multipart/form-data` request, with curl you can do this using the `@file.csv`:

```
curl -X POST  -H "Content-Type: multipart/form-data"  -H "Authorization: XXXX"  -F "id=
↪<resource_id>" -F "upload=@updated_file.csv" https://demo.ckan.org/api/3/action/
↪resource_patch
```

4.9 Action API reference

Note: If you call one of the action functions listed below and the function raises an exception, the API will return a JSON dictionary with keys `"success": false` and an `"error"` key indicating the exception that was raised.

For example *`member_list()`* (which returns a list of the members of a group) raises `NotFound` if the group doesn't exist. If you called it over the API, you'd get back a JSON dict like this:

```
{
    "success": false
    "error": {
        "__type": "Not Found Error",
        "message": "Not found"
    },
    "help": "...",
}
```

4.9.1 ckan.logic.action.get

API functions for searching for and getting data from CKAN.

`ckan.logic.action.get.`**`package_list`**(*context: Context*, *data_dict: dict[str, Any]*) → List[str]

Return a list of the names of the site's datasets (packages).

Parameters

- **`limit`** (*int*) – if given, the list of datasets will be broken into pages of at most `limit` datasets per page and only one page will be returned at a time (optional)
- **`offset`** (*int*) – when `limit` is given, the offset to start returning packages from

Return type

list of strings

`ckan.logic.action.get.`**`current_package_list_with_resources`**(*context: Context*, *data_dict: dict[str, Any]*) → List[dict[str, Any]]

Return a list of the site's datasets (packages) and their resources.

The list is sorted most-recently-modified first.

Parameters

- **limit** (*int*) – if given, the list of datasets will be broken into pages of at most `limit` datasets per page and only one page will be returned at a time (optional)
- **offset** (*int*) – when `limit` is given, the offset to start returning packages from
- **page** (*int*) – when `limit` is given, which page to return, Deprecated: use `offset`

Return type
list of dictionaries

`ckan.logic.action.get.`**`member_list`**(*context: Context*, *data_dict: dict[str, Any]*) → List[Tuple[Any, ...]]

Return the members of a group.

The user must have permission to 'get' the group.

Parameters

- **id** (*string*) – the id or name of the group
- **object_type** (*string*) – restrict the members returned to those of a given type, e.g. `'user'` or `'package'` (optional, default: `None`)
- **capacity** (*string*) – restrict the members returned to those with a given capacity, e.g. `'member'`, `'editor'`, `'admin'`, `'public'`, `'private'` (optional, default: `None`)

Return type
list of (id, type, capacity) tuples

Raises
`ckan.logic.NotFound`: if the group doesn't exist

`ckan.logic.action.get.`**`package_collaborator_list`**(*context: Context*, *data_dict: dict[str, Any]*) → List[dict[str, Any]]

Return the list of all collaborators for a given package.

Currently you must be an Admin on the package owner organization to manage collaborators.

Note: This action requires the collaborators feature to be enabled with the *ckan.auth.allow_dataset_collaborators* configuration option.

Parameters

- **id** (*string*) – the id or name of the package
- **capacity** (*string*) – (optional) If provided, only users with this capacity are returned

Returns
a list of collaborators, each a dict including the package and user id, the capacity and the last modified date

Return type
list of dictionaries

`ckan.logic.action.get.`**`package_collaborator_list_for_user`**(*context: Context*, *data_dict: dict[str, Any]*) → List[dict[str, Any]]

Return a list of all package the user is a collaborator in

Note: This action requires the collaborators feature to be enabled with the *ckan.auth.allow_dataset_collaborators* configuration option.

Parameters

- **id** (*string*) – the id or name of the user
- **capacity** (*string*) – (optional) If provided, only packages where the user has this capacity are returned

Returns
a list of packages, each a dict including the package id, the capacity and the last modified date

Return type
list of dictionaries

`ckan.logic.action.get.group_list`(*context: Context*, *data_dict: dict[str, Any]*) → List[dict[str, Any]]

Return a list of the names of the site's groups.

Parameters

- **type** (*string*) – the type of group to list (optional, default: `'group'`), See docs for *IGroupForm*
- **order_by** (*string*) – the field to sort the list by, must be `'name'` or `'packages'` (optional, default: `'name'`) Deprecated use sort.
- **sort** (*string*) – sorting of the search results. Optional. Default: "title asc" string of field name and sort-order. The allowed fields are 'name', 'package_count' and 'title'
- **limit** (*int*) – the maximum number of groups returned (optional) Default: `1000` when all_fields=false unless set in site's configuration `ckan.group_and_organization_list_max` Default: `25` when all_fields=true unless set in site's configuration `ckan.group_and_organization_list_all_fields_max`
- **offset** (*int*) – when `limit` is given, the offset to start returning groups from
- **groups** (*list of strings*) – a list of names of the groups to return, if given only groups whose names are in this list will be returned (optional)
- **all_fields** (*bool*) – return group dictionaries instead of just names. Only core fields are returned - get some more using the include_* options. Returning a list of packages is too expensive, so the *packages* property for each group is deprecated, but there is a count of the packages in the *package_count* property. (optional, default: `False`)
- **include_dataset_count** (*bool*) – if all_fields, include the full package_count (optional, default: `True`)
- **include_extras** (*bool*) – if all_fields, include the group extra fields (optional, default: `False`)
- **include_tags** (*bool*) – if all_fields, include the group tags (optional, default: `False`)
- **include_groups** (*bool*) – if all_fields, include the groups the groups are in (optional, default: `False`).
- **include_users** (*bool*) – if all_fields, include the group users (optional, default: `False`).

Return type
list of strings

`ckan.logic.action.get.organization_list(context: Context, data_dict: dict[str, Any]) → List[dict[str, Any]]`

Return a list of the names of the site's organizations.

Parameters

- **type** (*string*) – the type of organization to list (optional, default: `'organization'`), See docs for *IGroupForm*
- **order_by** (*string*) – the field to sort the list by, must be `'name'` or `'packages'` (optional, default: `'name'`) Deprecated use sort.
- **sort** (*string*) – sorting of the search results. Optional. Default: "title asc" string of field name and sort-order. The allowed fields are 'name', 'package_count' and 'title'
- **limit** (*int*) – the maximum number of organizations returned (optional) Default: `1000` when all_fields=false unless set in site's configuration `ckan.group_and_organization_list_max` Default: `25` when all_fields=true unless set in site's configuration `ckan.group_and_organization_list_all_fields_max`
- **offset** (*int*) – when `limit` is given, the offset to start returning organizations from
- **organizations** (*list of strings*) – a list of names of the groups to return, if given only groups whose names are in this list will be returned (optional)
- **all_fields** (*bool*) – return group dictionaries instead of just names. Only core fields are returned - get some more using the include_* options. Returning a list of packages is too expensive, so the *packages* property for each group is deprecated, but there is a count of the packages in the *package_count* property. (optional, default: `False`)
- **include_dataset_count** (*bool*) – if all_fields, include the full package_count (optional, default: `True`)
- **include_extras** (*bool*) – if all_fields, include the organization extra fields (optional, default: `False`)
- **include_tags** (*bool*) – if all_fields, include the organization tags (optional, default: `False`)
- **include_groups** (*bool*) – if all_fields, include the organizations the organizations are in (optional, default: `False`)
- **include_users** (*bool*) – if all_fields, include the organization users (optional, default: `False`).

Return type

list of strings

`ckan.logic.action.get.group_list_authz(context: Context, data_dict: dict[str, Any]) → List[dict[str, Any]]`

Return the list of groups that the user is authorized to edit.

Parameters

- **available_only** (*bool*) – remove the existing groups in the package (optional, default: `False`)
- **am_member** (*bool*) – if `True` return only the groups the logged-in user is a member of, otherwise return all groups that the user is authorized to edit (for example, sysadmin users are authorized to edit all groups) (optional, default: `False`)

Returns

list of dictized groups that the user is authorized to edit

Return type
list of dicts

`ckan.logic.action.get.organization_list_for_user(context: Context, data_dict: dict[str, Any]) → List[dict[str, Any]]`

Return the organizations that the user has a given permission for.

Specifically it returns the list of organizations that the currently authorized user has a given permission (for example: "manage_group") against.

By default this returns the list of organizations that the currently authorized user is member of, in any capacity.

When a user becomes a member of an organization in CKAN they're given a "capacity" (sometimes called a "role"), for example "member", "editor" or "admin".

Each of these roles has certain permissions associated with it. For example the admin role has the "admin" permission (which means they have permission to do anything). The editor role has permissions like "create_dataset", "update_dataset" and "delete_dataset". The member role has the "read" permission.

This function returns the list of organizations that the authorized user has a given permission for. For example the list of organizations that the user is an admin of, or the list of organizations that the user can create datasets in. This takes account of when permissions cascade down an organization hierarchy.

Parameters

- **id** (*string*) – the name or id of the user to get the organization list for (optional, defaults to the currently authorized user (logged in or via API key))
- **permission** (*string*) – the permission the user has against the returned organizations, for example `"read"` or `"create_dataset"` (optional, default: `"manage_group"`)
- **include_dataset_count** (*bool*) – include the package_count in each org (optional, default: `False`)

Returns
list of organizations that the user has the given permission for

Return type
list of dicts

`ckan.logic.action.get.license_list(context: Context, data_dict: dict[str, Any]) → List[dict[str, Any]]`

Return the list of licenses available for datasets on the site.

Return type
list of dictionaries

`ckan.logic.action.get.tag_list(context: Context, data_dict: dict[str, Any]) → List[dict[str, Any]] | List[str]`

Return a list of the site's tags.

By default only free tags (tags that don't belong to a vocabulary) are returned. If the `vocabulary_id` argument is given then only tags belonging to that vocabulary will be returned instead.

Parameters

- **query** (*string*) – a tag name query to search for, if given only tags whose names contain this string will be returned (optional)
- **vocabulary_id** (*string*) – the id or name of a vocabulary, if give only tags that belong to this vocabulary will be returned (optional)
- **all_fields** (*bool*) – return full tag dictionaries instead of just names (optional, default: `False`)

Return type
list of dictionaries

`ckan.logic.action.get.user_list(context: Context, data_dict: DataDict) → ActionResult.UserList`

Return a list of the site's user accounts.

Parameters

- **q** (*string*) – filter the users returned to those whose names contain a string (optional)
- **email** (*string*) – filter the users returned to those whose email match a string (optional) (you must be a sysadmin to use this filter)
- **order_by** (*string*) – which field to sort the list by (optional, default: `'display_name'`). Users can be sorted by `'id'`, `'name'`, `'fullname'`, `'display_name'`, `'created'`, `'about'`, `'sysadmin'` or `'number_created_packages'`.
- **all_fields** (*bool*) – return full user dictionaries instead of just names. (optional, default: `True`)
- **include_site_user** (*bool*) – add site_user to the result (optional, default: `False`)

Return type
list of user dictionaries. User properties include: `number_created_packages` which excludes datasets which are private or draft state.

`ckan.logic.action.get.package_relationships_list(context: Context, data_dict: dict[str, Any]) → List[dict[str, Any]]`

Return a dataset (package)'s relationships.

Parameters

- **id** (*string*) – the id or name of the first package
- **id2** (*string*) – the id or name of the second package
- **rel** – relationship as string see *package_relationship_create()* for the relationship types (optional)

Return type
list of dictionaries

`ckan.logic.action.get.package_show(context: Context, data_dict: dict[str, Any]) → dict[str, Any]`

Return the metadata of a dataset (package) and its resources.

Parameters

- **id** (*string*) – the id or name of the dataset
- **use_default_schema** (*bool*) – use default package schema instead of a custom schema defined with an IDatasetForm plugin (default: `False`)
- **include_plugin_data** – Include the internal plugin data object (sysadmin only, optional, default:`False`)

Type
include_plugin_data: bool

Return type
dictionary

`ckan.logic.action.get.resource_show(context: Context, data_dict: dict[str, Any]) → dict[str, Any]`

Return the metadata of a resource.

Parameters
id (*string*) – the id of the resource

Return type
dictionary

`ckan.logic.action.get.`**`resource_view_show`**(*context: Context*, *data_dict: dict[str, Any]*) → dict[str, Any]

Return the metadata of a resource_view.

Parameters
id (*string*) – the id of the resource_view

Return type
dictionary

`ckan.logic.action.get.`**`resource_view_list`**(*context: Context*, *data_dict: dict[str, Any]*) → List[dict[str, Any]]

Return the list of resource views for a particular resource.

Parameters
id (*string*) – the id of the resource

Return type
list of dictionaries.

`ckan.logic.action.get.`**`group_show`**(*context: Context*, *data_dict: dict[str, Any]*) → dict[str, Any]

Return the details of a group.

Parameters

- **id** (*string*) – the id or name of the group
- **include_datasets** (*bool*) – include a truncated list of the group's datasets (optional, default: `False`)
- **include_dataset_count** (*bool*) – include the full package_count (optional, default: `True`)
- **include_extras** (*bool*) – include the group's extra fields (optional, default: `True`)
- **include_users** (*bool*) – include the group's users (optional, default: `True` if `ckan.auth.public_user_details` is `True` otherwise `False`)
- **include_groups** (*bool*) – include the group's sub groups (optional, default: `True`)
- **include_tags** (*bool*) – include the group's tags (optional, default: `True`)
- **include_followers** (*bool*) – include the group's number of followers (optional, default: `True`)

Return type
dictionary

Note: Only its first 1000 datasets are returned

`ckan.logic.action.get.`**`organization_show`**(*context: Context*, *data_dict: dict[str, Any]*) → dict[str, Any]

Return the details of a organization.

Parameters

- **id** (*string*) – the id or name of the organization

- **include_datasets** (*bool*) – include a truncated list of the org's datasets (optional, default: `False`)
- **include_dataset_count** (*bool*) – include the full package_count (optional, default: `True`)
- **include_extras** (*bool*) – include the organization's extra fields (optional, default: `True`)
- **include_users** (*bool*) – include the organization's users (optional, default: `True` if `ckan.auth.public_user_details` is `True` otherwise `False`)
- **include_groups** (*bool*) – include the organization's sub groups (optional, default: `True`)
- **include_tags** (*bool*) – include the organization's tags (optional, default: `True`)
- **include_followers** (*bool*) – include the organization's number of followers (optional, default: `True`)

Return type
dictionary

Note: Only its first 10 datasets are returned

`ckan.logic.action.get.`**`group_package_show`**(*context: Context*, *data_dict: dict[str, Any]*) → List[dict[str, Any]]

Return the datasets (packages) of a group.

Parameters

- **id** (*string*) – the id or name of the group
- **limit** (*int*) – the maximum number of datasets to return (optional)

Return type
list of dictionaries

`ckan.logic.action.get.`**`tag_show`**(*context: Context*, *data_dict: dict[str, Any]*) → dict[str, Any]

Return the details of a tag and all its datasets.

Parameters

- **id** (*string*) – the name or id of the tag
- **vocabulary_id** (*string*) – the id or name of the tag vocabulary that the tag is in - if it is not specified it will assume it is a free tag. (optional)
- **include_datasets** (*bool*) – include a list of the tag's datasets. (Up to a limit of 1000 - for more flexibility, use package_search - see *package_search()* for an example.) (optional, default: `False`)

Returns
the details of the tag, including a list of all of the tag's datasets and their details

Return type
dictionary

`ckan.logic.action.get.`**`user_show`**(*context: Context*, *data_dict: dict[str, Any]*) → dict[str, Any]

Return a user account.

Either the `id` should be passed or the user should be logged in.

Parameters

- **id** (*string*) – the id or name of the user (optional)
- **include_datasets** (*bool*) – Include a list of datasets the user has created. If it is the same user or a sysadmin requesting, it includes datasets that are draft or private. (optional, default:`False`, limit:50)
- **include_num_followers** (*bool*) – Include the number of followers the user has (optional, default:`False`)
- **include_password_hash** (*bool*) – Include the stored password hash (sysadmin only, optional, default:`False`)
- **include_plugin_extras** (*bool*) – Include the internal plugin extras object (sysadmin only, optional, default:`False`)

Returns
the details of the user. Includes email_hash and number_created_packages (which excludes draft or private datasets unless it is the same user or sysadmin making the request). Excludes the password (hash) and reset_key. If it is the same user or a sysadmin requesting, the email and apikey are included.

Return type
dictionary

`ckan.logic.action.get.`**`package_autocomplete`**(*context: Context*, *data_dict: dict[str, Any]*) → List[dict[str, Any]]

Return a list of datasets (packages) that match a string.

Datasets with names or titles that contain the query string will be returned.

Parameters

- **q** (*string*) – the string to search for
- **limit** (*int*) – the maximum number of resource formats to return (optional, default: `10`)

Return type
list of dictionaries

`ckan.logic.action.get.`**`format_autocomplete`**(*context: Context*, *data_dict: dict[str, Any]*) → List[str]

Return a list of resource formats whose names contain a string.

Parameters

- **q** (*string*) – the string to search for
- **limit** (*int*) – the maximum number of resource formats to return (optional, default: `5`)

Return type
list of strings

`ckan.logic.action.get.`**`user_autocomplete`**(*context: Context*, *data_dict: dict[str, Any]*) → List[dict[str, Any]]

Return a list of user names that contain a string.

Parameters

- **q** (*string*) – the string to search for
- **limit** (*int*) – the maximum number of user names to return (optional, default: `20`)

Return type
a list of user dictionaries each with keys `'name'`, `'fullname'`, and `'id'`

`ckan.logic.action.get.group_autocomplete(context: Context, data_dict: dict[str, Any]) → List[dict[str, Any]]`

Return a list of group names that contain a string.

Parameters

- **q** (*string*) – the string to search for
- **limit** (*int*) – the maximum number of groups to return (optional, default: 20)

Return type
a list of group dictionaries each with keys `'name'`, `'title'`, and `'id'`

`ckan.logic.action.get.organization_autocomplete(context: Context, data_dict: dict[str, Any]) → List[dict[str, Any]]`

Return a list of organization names that contain a string.

Parameters

- **q** (*string*) – the string to search for
- **limit** (*int*) – the maximum number of organizations to return (optional, default: `20`)

Return type
a list of organization dictionaries each with keys `'name'`, `'title'`, and `'id'`

`ckan.logic.action.get.package_search(context: Context, data_dict: dict[str, Any]) → dict[str, Any]`

Searches for packages satisfying a given search criteria.

This action accepts solr search query parameters (details below), and returns a dictionary of results, including dictized datasets that match the search criteria, a search count and also facet information.

Solr Parameters:

For more in depth treatment of each paramter, please read the Solr Documentation.

This action accepts a *subset* of solr's search query parameters:

Parameters

- **q** (*string*) – the solr query. Optional. Default: `"*:*"`
- **fq** (*string*) – any filter queries to apply. Note: `+site_id:{ckan_site_id}` is added to this string prior to the query being executed.
- **fq_list** (*list of strings*) – additional filter queries to apply.
- **sort** (*string*) – sorting of the search results. Optional. Default: `'score desc, metadata_modified desc'`. As per the solr documentation, this is a comma-separated string of field names and sort-orderings.
- **rows** (*int*) – the maximum number of matching rows (datasets) to return. (optional, default: `10`, upper limit: `1000` unless set in site's configuration `ckan.search.rows_max`)
- **start** (*int*) – the offset in the complete result for where the set of returned datasets should begin.
- **facet** (*string*) – whether to enable faceted results. Default: `True`.
- **facet.mincount** (*int*) – the minimum counts for facet fields should be included in the results.
- **facet.limit** (*int*) – the maximum number of values the facet fields return. A negative value means unlimited. This can be set instance-wide with the *search.facets.limit* config option. Default is 50.

- **facet.field** (*list of strings*) – the fields to facet upon. Default empty. If empty, then the returned facet information is empty.
- **include_drafts** (*bool*) – if `True`, draft datasets will be included in the results. A user will only be returned their own draft datasets, and a sysadmin will be returned all draft datasets. Optional, the default is `False`.
- **include_deleted** (*bool*) – if `True`, deleted datasets will be included in the results (site configuration "ckan.search.remove_deleted_packages" must be set to False). Optional, the default is `False`.
- **include_private** (*bool*) – if `True`, private datasets will be included in the results. Only private datasets from the user's organizations will be returned and sysadmins will be returned all private datasets. Optional, the default is `False`.
- **use_default_schema** (*bool*) – use default package schema instead of a custom schema defined with an IDatasetForm plugin (default: `False`)

The following advanced Solr parameters are supported as well. Note that some of these are only available on particular Solr versions. See Solr's dismax and edismax documentation for further details on them:

`qf`, `wt`, `bf`, `boost`, `tie`, `defType`, `mm`

Examples:

`q=flood` datasets containing the word *flood*, *floods* or *flooding* `fq=tags:economy` datasets with the tag *economy* `facet.field=["tags"] facet.limit=10 rows=0` top 10 tags

Results:

The result of this action is a dict with the following keys:

Return type
A dictionary with the following keys

Parameters

- **count** (*int*) – the number of results found. Note, this is the total number of results found, not the total number of results returned (which is affected by limit and row parameters used in the input).
- **results** (*list of dictized datasets.*) – ordered list of datasets matching the query, where the ordering defined by the sort parameter used in the query.
- **facets** (*DEPRECATED dict*) – DEPRECATED. Aggregated information about facet counts.
- **search_facets** (*nested dict of dicts.*) – aggregated information about facet counts. The outer dict is keyed by the facet field name (as used in the search query). Each entry of the outer dict is itself a dict, with a "title" key, and an "items" key. The "items" key's value is a list of dicts, each with "count", "display_name" and "name" entries. The display_name is a form of the name that can be used in titles.

An example result:

```
{'count': 2,
 'results': [ { <snip> }, { <snip> }],
 'search_facets': {u'tags': {'items': [{'count': 1,
                                        'display_name': u'tolstoy',
                                        'name': u'tolstoy'},
                                       {'count': 2,
                                        'display_name': u'russian',
```

(continues on next page)

(continued from previous page)

```
                                        'name': u'russian'}
                                    ]
                            }
                    }
}
```

Limitations:

The full solr query language is not exposed, including.

fl

The parameter that controls which fields are returned in the solr query. fl can be None or a list of result fields, such as ['id', 'extras_custom_field']. if fl = None, datasets are returned as a list of full dictionary.

`ckan.logic.action.get.`**`resource_search`**(*context: Context*, *data_dict: dict[str, Any]*) → dict[str, Any]

Searches for resources in public Datasets satisfying the search criteria.

It returns a dictionary with 2 fields: `count` and `results`. The `count` field contains the total number of Resources found without the limit or query parameters having an effect. The `results` field is a list of dictized Resource objects.

The 'query' parameter is a required field. It is a string of the form `{field}:{term}` or a list of strings, each of the same form. Within each string, `{field}` is a field or extra field on the Resource domain object.

If `{field}` is `"hash"`, then an attempt is made to match the *{term}* as a *prefix* of the `Resource.hash` field.

If `{field}` is an extra field, then an attempt is made to match against the extra fields stored against the Resource.

Note: The search is limited to search against extra fields declared in the config setting `ckan.extra_resource_fields`.

Note: Due to a Resource's extra fields being stored as a json blob, the match is made against the json string representation. As such, false positives may occur:

If the search criteria is:

```
query = "field1:term1"
```

Then a json blob with the string representation of:

```
{"field1": "foo", "field2": "term1"}
```

will match the search criteria! This is a known short-coming of this approach.

All matches are made ignoring case; and apart from the `"hash"` field, a term matches if it is a substring of the field's value.

Finally, when specifying more than one search criteria, the criteria are AND-ed together.

The `order` parameter is used to control the ordering of the results. Currently only ordering one field is available, and in ascending order only.

The context may contain a flag, *search_query*, which if True will make this action behave as if being used by the internal search api. ie - the results will not be dictized, and SearchErrors are thrown for bad search queries (rather than ValidationErrors).

Parameters

- **query** (string or list of strings of the form `{field}:{term1}`) – The search criteria. See above for description.

- **order_by** (*string*) – A field on the Resource model that orders the results.
- **offset** (*int*) – Apply an offset to the query.
- **limit** (*int*) – Apply a limit to the query.

Returns
A dictionary with a `count` field, and a `results` field.

Return type
dict

`ckan.logic.action.get.`**`tag_search`**(*context: Context*, *data_dict: dict[str, Any]*) → dict[str, Any]

Return a list of tags whose names contain a given string.

By default only free tags (tags that don't belong to any vocabulary) are searched. If the `vocabulary_id` argument is given then only tags belonging to that vocabulary will be searched instead.

Parameters

- **query** (*string or list of strings*) – the string(s) to search for
- **vocabulary_id** (*string*) – the id or name of the tag vocabulary to search in (optional)
- **fields** (*dictionary*) – deprecated
- **limit** (*int*) – the maximum number of tags to return
- **offset** (*int*) – when `limit` is given, the offset to start returning tags from

Returns

A dictionary with the following keys:

`'count'`
The number of tags in the result.

`'results'`
The list of tags whose names contain the given string, a list of dictionaries.

Return type
dictionary

`ckan.logic.action.get.`**`tag_autocomplete`**(*context: Context*, *data_dict: dict[str, Any]*) → List[str]

Return a list of tag names that contain a given string.

By default only free tags (tags that don't belong to any vocabulary) are searched. If the `vocabulary_id` argument is given then only tags belonging to that vocabulary will be searched instead.

Parameters

- **query** (*string*) – the string to search for
- **vocabulary_id** (*string*) – the id or name of the tag vocabulary to search in (optional)
- **fields** (*dictionary*) – deprecated
- **limit** (*int*) – the maximum number of tags to return
- **offset** (*int*) – when `limit` is given, the offset to start returning tags from

Return type
list of strings

`ckan.logic.action.get.task_status_show(context: Context, data_dict: dict[str, Any]) → dict[str, Any]`

Return a task status.

Either the `id` parameter *or* the `entity_id`, `task_type` *and* `key` parameters must be given.

Parameters

- **id** (*string*) – the id of the task status (optional)
- **entity_id** (*string*) – the entity_id of the task status (optional)
- **task_type** (*string*) – the task_type of the task status (optional)
- **key** (*string*) – the key of the task status (optional)

Return type
dictionary

`ckan.logic.action.get.term_translation_show(context: Context, data_dict: dict[str, Any]) → List[dict[str, Any]]`

Return the translations for the given term(s) and language(s).

Parameters

- **terms** (*list of strings*) – the terms to search for translations of, e.g. `'Russian'`, `'romantic novel'`
- **lang_codes** (*list of language code strings*) – the language codes of the languages to search for translations into, e.g. `'en'`, `'de'` (optional, default is to search for translations into any language)

Return type
a list of term translation dictionaries each with keys `'term'` (the term searched for, in the source language), `'term_translation'` (the translation of the term into the target language) and `'lang_code'` (the language code of the target language)

`ckan.logic.action.get.get_site_user(context: Context, data_dict: dict[str, Any]) → dict[str, Any]`

Return the ckan site user

Parameters
defer_commit (*bool*) – by default (or if set to false) get_site_user will commit and clean up the current transaction. If set to true, caller is responsible for commiting transaction after get_site_user is called. Leaving open connections can cause cli commands to hang! (optional, default: `False`)

`ckan.logic.action.get.status_show(context: Context, data_dict: dict[str, Any]) → dict[str, Any]`

Return a dictionary with information about the site's configuration.

Return type
dictionary

`ckan.logic.action.get.vocabulary_list(context: Context, data_dict: dict[str, Any]) → List[dict[str, Any]]`

Return a list of all the site's tag vocabularies.

Return type
list of dictionaries

`ckan.logic.action.get.vocabulary_show(context: Context, data_dict: dict[str, Any]) → dict[str, Any]`

Return a single tag vocabulary.

Parameters
id (*string*) – the id or name of the vocabulary

Returns
the vocabulary.

Return type
dictionary

`ckan.logic.action.get.user_follower_count(context: Context, data_dict: dict[str, Any]) → int`

Return the number of followers of a user.

Parameters
id (*string*) – the id or name of the user

Return type
int

`ckan.logic.action.get.dataset_follower_count(context: Context, data_dict: dict[str, Any]) → int`

Return the number of followers of a dataset.

Parameters
id (*string*) – the id or name of the dataset

Return type
int

`ckan.logic.action.get.group_follower_count(context: Context, data_dict: dict[str, Any]) → int`

Return the number of followers of a group.

Parameters
id (*string*) – the id or name of the group

Return type
int

`ckan.logic.action.get.organization_follower_count(context: Context, data_dict: dict[str, Any]) → int`

Return the number of followers of an organization.

Parameters
id (*string*) – the id or name of the organization

Return type
int

`ckan.logic.action.get.user_follower_list(context: Context, data_dict: dict[str, Any]) → List[dict[str, Any]]`

Return the list of users that are following the given user.

Parameters
id (*string*) – the id or name of the user

Return type
list of dictionaries

`ckan.logic.action.get.dataset_follower_list(context: Context, data_dict: dict[str, Any]) → List[dict[str, Any]]`

Return the list of users that are following the given dataset.

Parameters
id (*string*) – the id or name of the dataset

Return type
list of dictionaries

`ckan.logic.action.get.`**`group_follower_list`**(*context: Context*, *data_dict: dict[str, Any]*) → List[dict[str, Any]]

Return the list of users that are following the given group.

Parameters
id (*string*) – the id or name of the group

Return type
list of dictionaries

`ckan.logic.action.get.`**`organization_follower_list`**(*context: Context*, *data_dict: dict[str, Any]*) → List[dict[str, Any]]

Return the list of users that are following the given organization.

Parameters
id (*string*) – the id or name of the organization

Return type
list of dictionaries

`ckan.logic.action.get.`**`am_following_user`**(*context: Context*, *data_dict: dict[str, Any]*) → bool

Return `True` if you're following the given user, `False` if not.

Parameters
id (*string*) – the id or name of the user

Return type
bool

`ckan.logic.action.get.`**`am_following_dataset`**(*context: Context*, *data_dict: dict[str, Any]*) → bool

Return `True` if you're following the given dataset, `False` if not.

Parameters
id (*string*) – the id or name of the dataset

Return type
bool

`ckan.logic.action.get.`**`am_following_group`**(*context: Context*, *data_dict: dict[str, Any]*) → bool

Return `True` if you're following the given group, `False` if not.

Parameters
id (*string*) – the id or name of the group

Return type
bool

`ckan.logic.action.get.`**`followee_count`**(*context: Context*, *data_dict: dict[str, Any]*) → int

Return the number of objects that are followed by the given user.

Counts all objects, of any type, that the given user is following (e.g. followed users, followed datasets, followed groups).

Parameters
id (*string*) – the id of the user

Return type
int

`ckan.logic.action.get.`**`user_followee_count`**(*context: Context*, *data_dict: dict[str, Any]*) → int

Return the number of users that are followed by the given user.

Parameters
id (*string*) – the id of the user

Return type
int

`ckan.logic.action.get.dataset_followee_count(context: Context, data_dict: dict[str, Any]) → int`

Return the number of datasets that are followed by the given user.

Parameters
id (*string*) – the id of the user

Return type
int

`ckan.logic.action.get.group_followee_count(context: Context, data_dict: dict[str, Any]) → int`

Return the number of groups that are followed by the given user.

Parameters
id (*string*) – the id of the user

Return type
int

`ckan.logic.action.get.organization_followee_count(context: Context, data_dict: dict[str, Any]) → int`

Return the number of organizations that are followed by the given user.

Parameters
id (*string*) – the id of the user

Return type
int

`ckan.logic.action.get.followee_list(context: Context, data_dict: dict[str, Any]) → List[dict[str, Any]]`

Return the list of objects that are followed by the given user.

Returns all objects, of any type, that the given user is following (e.g. followed users, followed datasets, followed groups..).

Parameters

- **id** (*string*) – the id of the user
- **q** (*string*) – a query string to limit results by, only objects whose display name begins with the given string (case-insensitive) wil be returned (optional)

Return type
list of dictionaries, each with keys `'type'` (e.g. `'user'`, `'dataset'` or `'group'`), `'display_name'` (e.g. a user's display name, or a package's title) and `'dict'` (e.g. a dict representing the followed user, package or group, the same as the dict that would be returned by *user_show()*, *package_show()* or *group_show()*)

`ckan.logic.action.get.user_followee_list(context: Context, data_dict: dict[str, Any]) → List[dict[str, Any]]`

Return the list of users that are followed by the given user.

Parameters
id (*string*) – the id of the user

Return type
list of dictionaries

`ckan.logic.action.get.`**`dataset_followee_list`**(*context: Context*, *data_dict: dict[str, Any]*) → List[dict[str, Any]]

Return the list of datasets that are followed by the given user.

Parameters
id (*string*) – the id or name of the user

Return type
list of dictionaries

`ckan.logic.action.get.`**`group_followee_list`**(*context: Context*, *data_dict: dict[str, Any]*) → List[dict[str, Any]]

Return the list of groups that are followed by the given user.

Parameters
id (*string*) – the id or name of the user

Return type
list of dictionaries

`ckan.logic.action.get.`**`organization_followee_list`**(*context: Context*, *data_dict: dict[str, Any]*) → List[dict[str, Any]]

Return the list of organizations that are followed by the given user.

Parameters
id (*string*) – the id or name of the user

Return type
list of dictionaries

`ckan.logic.action.get.`**`member_roles_list`**(*context: Context*, *data_dict: dict[str, Any]*) → List[dict[str, Any]]

Return the possible roles for members of groups and organizations.

Parameters
group_type (*string*) – the group type, either `"group"` or `"organization"` (optional, default `"organization"`)

Returns
a list of dictionaries each with two keys: `"text"` (the display name of the role, e.g. `"Admin"`) and `"value"` (the internal name of the role, e.g. `"admin"`)

Return type
list of dictionaries

`ckan.logic.action.get.`**`help_show`**(*context: Context*, *data_dict: dict[str, Any]*) → str | None

Return the help string for a particular API action.

Parameters
name (*string*) – Action function name (eg *user_create*, *package_search*)

Returns
The help string for the action function, or None if the function does not have a docstring.

Return type
string

Raises
`ckan.logic.NotFound`: if the action function doesn't exist

`ckan.logic.action.get.`**`config_option_show`**(*context: Context*, *data_dict: dict[str, Any]*) → Any

Show the current value of a particular configuration option.

Only returns runtime-editable config options (the ones returned by `config_option_list()`), which can be updated with the `config_option_update()` action.

Parameters
key (*string*) – The configuration option key

Returns
The value of the config option from either the system_info table or ini file.

Return type
string

Raises
`ckan.logic.ValidationError`: if config option is not in the schema (whitelisted as editable).

`ckan.logic.action.get.`**`config_option_list`**(*context: Context*, *data_dict: dict[str, Any]*) → List[str]

Return a list of runtime-editable config options keys that can be
updated with `config_option_update()`.

Returns
A list of config option keys.

Return type
list

`ckan.logic.action.get.`**`job_list`**(*context: Context*, *data_dict: dict[str, Any]*) → List[dict[str, Any]]

List enqueued background jobs.

Parameters
queues (*list*) – Queues to list jobs from. If not given then the jobs from all queues are listed.

Returns
The currently enqueued background jobs.

Return type
list

New in version 2.7.

`ckan.logic.action.get.`**`job_show`**(*context: Context*, *data_dict: dict[str, Any]*) → dict[str, Any]

Show details for a background job.

Parameters
id (*string*) – The ID of the background job.

Returns
Details about the background job.

Return type
dict

New in version 2.7.

`ckan.logic.action.get.`**`api_token_list`**(*context: Context*, *data_dict: dict[str, Any]*) → List[dict[str, Any]]

Return list of all available API Tokens for current user.

Parameters
user_id (*string*) – The user ID or name

Returns
collection of all API Tokens

Return type
list

New in version 2.9.

4.9.2 ckan.logic.action.create

API functions for adding data to CKAN.

`ckan.logic.action.create.package_create`(*context: Context*, *data_dict: dict[str, Any]*) → dict[str, Any] | str

Create a new dataset (package).

You must be authorized to create new datasets. If you specify any groups for the new dataset, you must also be authorized to edit these groups.

Plugins may change the parameters of this function depending on the value of the `type` parameter, see the *IDatasetForm* plugin interface.

Parameters

- **name** (*string*) – the name of the new dataset, must be between 2 and 100 characters long and contain only lowercase alphanumeric characters, `-` and `_`, e.g. `'warandpeace'`
- **title** (*string*) – the title of the dataset (optional, default: same as `name`)
- **private** (*bool*) – If `True` creates a private dataset
- **author** (*string*) – the name of the dataset's author (optional)
- **author_email** (*string*) – the email address of the dataset's author (optional)
- **maintainer** (*string*) – the name of the dataset's maintainer (optional)
- **maintainer_email** (*string*) – the email address of the dataset's maintainer (optional)
- **license_id** (*license id string*) – the id of the dataset's license, see *license_list()* for available values (optional)
- **notes** (*string*) – a description of the dataset (optional)
- **url** (*string*) – a URL for the dataset's source (optional)
- **version** (*string, no longer than 100 characters*) – (optional)
- **state** (*string*) – the current state of the dataset, e.g. `'active'` or `'deleted'`, only active datasets show up in search results and other lists of datasets, this parameter will be ignored if you are not authorized to change the state of the dataset (optional, default: `'active'`)
- **type** (*string*) – the type of the dataset (optional), *IDatasetForm* plugins associate themselves with different dataset types and provide custom dataset handling behaviour for these types
- **resources** (*list of resource dictionaries*) – the dataset's resources, see *resource_create()* for the format of resource dictionaries (optional)
- **tags** (*list of tag dictionaries*) – the dataset's tags, see *tag_create()* for the format of tag dictionaries (optional)
- **extras** (*list of dataset extra dictionaries*) – the dataset's extras (optional), extras are arbitrary (key: value) metadata items that can be added to datasets, each extra dictionary should have keys `'key'` (a string), `'value'` (a string)

- **plugin_data** (*dict*) – private package data belonging to plugins. Only sysadmin users may set this value. It should be a dict that can be dumped into JSON, and plugins should namespace their data with the plugin name to avoid collisions with other plugins, eg:

```
{
    "name": "test-dataset",
    "plugin_data": {
        "plugin1": {"key1": "value1"},
        "plugin2": {"key2": "value2"}
    }
}
```

- **relationships_as_object** (*list of relationship dictionaries*) – see *package_relationship_create()* for the format of relationship dictionaries (optional)
- **relationships_as_subject** (*list of relationship dictionaries*) – see *package_relationship_create()* for the format of relationship dictionaries (optional)
- **groups** (*list of dictionaries*) – the groups to which the dataset belongs (optional), each group dictionary should have one or more of the following keys which identify an existing group: `'id'` (the id of the group, string), or `'name'` (the name of the group, string), to see which groups exist call *group_list()*
- **owner_org** (*string*) – the id of the dataset's owning organization, see *organization_list()* or *organization_list_for_user()* for available values. This parameter can be made optional if the config option *ckan.auth.create_unowned_dataset* is set to `True`.

Returns
the newly created dataset (unless 'return_id_only' is set to True in the context, in which case just the dataset id will be returned)

Return type
dictionary

`ckan.logic.action.create.`**`resource_create`**(*context: Context*, *data_dict: dict[str, Any]*) → dict[str, Any]

Appends a new resource to a datasets list of resources.

Parameters

- **package_id** (*string*) – id of package that the resource should be added to.
- **url** (*string*) – url of resource
- **description** (*string*) – (optional)
- **format** (*string*) – (optional)
- **hash** (*string*) – (optional)
- **name** (*string*) – (optional)
- **resource_type** (*string*) – (optional)
- **mimetype** (*string*) – (optional)
- **mimetype_inner** (*string*) – (optional)
- **cache_url** (*string*) – (optional)
- **size** (*int*) – (optional)

- **created** (*iso date string*) – (optional)
- **last_modified** (*iso date string*) – (optional)
- **cache_last_updated** (*iso date string*) – (optional)
- **upload** (*FieldStorage (optional) needs multipart/form-data*) – (optional)

Returns
the newly created resource

Return type
dictionary

`ckan.logic.action.create.`**`resource_view_create`**(*context: Context*, *data_dict: dict[str, Any]*) → dict[str, Any]

Creates a new resource view.

Parameters

- **resource_id** (*string*) – id of the resource
- **title** (*string*) – the title of the view
- **description** (*string*) – a description of the view (optional)
- **view_type** (*string*) – type of view
- **config** (*JSON string*) – options necessary to recreate a view state (optional)

Returns
the newly created resource view

Return type
dictionary

`ckan.logic.action.create.`**`resource_create_default_resource_views`**(*context: Context*, *data_dict: dict[str, Any]*) → List[dict[str, Any]]

Creates the default views (if necessary) on the provided resource

The function will get the plugins for the default views defined in the configuration, and if some were found the *can_view* method of each one of them will be called to determine if a resource view should be created. Resource views extensions get the resource dict and the parent dataset dict.

If the latter is not provided, *package_show* is called to get it.

By default only view plugins that don't require the resource data to be in the DataStore are called. See `ckan.logic.action.create.package_create_default_resource_views.`` ()` for details on the `create_datastore_views` parameter.

Parameters

- **resource** (*dict*) – full resource dict
- **package** (*dict*) – full dataset dict (optional, if not provided *package_show()* will be called).
- **create_datastore_views** (*bool*) – whether to create views that rely on data being on the DataStore (optional, defaults to False)

Returns
a list of resource views created (empty if none were created)

Return type
list of dictionaries

`ckan.logic.action.create.`**`package_create_default_resource_views`**(*context: Context*, *data_dict: dict[str, Any]*) → List[dict[str, Any]]

Creates the default views on all resources of the provided dataset

By default only view plugins that don't require the resource data to be in the DataStore are called. Passing *create_datastore_views* as True will only create views that require data to be in the DataStore. The first case happens when the function is called from *package_create* or *package_update*, the second when it's called from the DataPusher when data was uploaded to the DataStore.

Parameters

- **package** (*dict*) – full dataset dict (ie the one obtained calling *package_show()*).
- **create_datastore_views** (*bool*) – whether to create views that rely on data being on the DataStore (optional, defaults to False)

Returns
a list of resource views created (empty if none were created)

Return type
list of dictionaries

`ckan.logic.action.create.`**`package_relationship_create`**(*context: Context*, *data_dict: dict[str, Any]*) → dict[str, Any]

Create a relationship between two datasets (packages).

You must be authorized to edit both the subject and the object datasets.

Parameters

- **subject** (*string*) – the id or name of the dataset that is the subject of the relationship
- **object** – the id or name of the dataset that is the object of the relationship
- **type** (*string*) – the type of the relationship, one of `'depends_on'`, `'dependency_of'`, `'derives_from'`, `'has_derivation'`, `'links_to'`, `'linked_from'`, `'child_of'` or `'parent_of'`
- **comment** (*string*) – a comment about the relationship (optional)

Returns
the newly created package relationship

Return type
dictionary

`ckan.logic.action.create.`**`member_create`**(*context: Context*, *data_dict: dict[str, Any]*) → dict[str, Any]

Make an object (e.g. a user, dataset or group) a member of a group.

If the object is already a member of the group then the capacity of the membership will be updated.

You must be authorized to edit the group.

Parameters

- **id** (*string*) – the id or name of the group to add the object to
- **object** (*string*) – the id or name of the object to add
- **object_type** (*string*) – the type of the object being added, e.g. `'package'` or `'user'`

- **capacity** (*string*) – the capacity of the membership

Returns
the newly created (or updated) membership

Return type
dictionary

`ckan.logic.action.create.`**`package_collaborator_create`**(*context: Context*, *data_dict: dict[str, Any]*) → dict[str, Any]

Make a user a collaborator in a dataset.

If the user is already a collaborator in the dataset then their capacity will be updated.

Currently you must be an Admin on the dataset owner organization to manage collaborators.

Note: This action requires the collaborators feature to be enabled with the *ckan.auth.allow_dataset_collaborators* configuration option.

Parameters

- **id** (*string*) – the id or name of the dataset
- **user_id** (*string*) – the id or name of the user to add or edit
- **capacity** (*string*) – the capacity or role of the membership. Must be one of "editor" or "member". Additionally if *ckan.auth.allow_admin_collaborators* is set to True, "admin" is also allowed.

Returns
the newly created (or updated) collaborator

Return type
dictionary

`ckan.logic.action.create.`**`group_create`**(*context: Context*, *data_dict: dict[str, Any]*) → str | dict[str, Any]

Create a new group.

You must be authorized to create groups.

Plugins may change the parameters of this function depending on the value of the `type` parameter, see the *`IGroupForm`* plugin interface.

Parameters

- **name** (*string*) – the name of the group, a string between 2 and 100 characters long, containing only lowercase alphanumeric characters, - and _
- **id** (*string*) – the id of the group (optional)
- **title** (*string*) – the title of the group (optional)
- **description** (*string*) – the description of the group (optional)
- **image_url** (*string*) – the URL to an image to be displayed on the group's page (optional)
- **type** (*string*) – the type of the group (optional, default: `'group'`), *`IGroupForm`* plugins associate themselves with different group types and provide custom group handling behaviour for these types Cannot be 'organization'
- **state** (*string*) – the current state of the group, e.g. `'active'` or `'deleted'`, only active groups show up in search results and other lists of groups, this parameter will be ignored if you are not authorized to change the state of the group (optional, default: `'active'`)
- **approval_status** (*string*) – (optional)

- **extras** (*list of dataset extra dictionaries*) – the group's extras (optional), extras are arbitrary (key: value) metadata items that can be added to groups, each extra dictionary should have keys `'key'` (a string), `'value'` (a string), and optionally `'deleted'`
- **packages** (*list of dictionaries*) – the datasets (packages) that belong to the group, a list of dictionaries each with keys `'name'` (string, the id or name of the dataset) and optionally `'title'` (string, the title of the dataset)
- **groups** (*list of dictionaries*) – the groups that belong to the group, a list of dictionaries each with key `'name'` (string, the id or name of the group) and optionally `'capacity'` (string, the capacity in which the group is a member of the group)
- **users** (*list of dictionaries*) – the users that belong to the group, a list of dictionaries each with key `'name'` (string, the id or name of the user) and optionally `'capacity'` (string, the capacity in which the user is a member of the group)

Returns
the newly created group (unless 'return_id_only' is set to True in the context, in which case just the group id will be returned)

Return type
dictionary

`ckan.logic.action.create.`**`organization_create`**(*context: Context*, *data_dict: dict[str, Any]*) → str | dict[str, Any]

Create a new organization.

You must be authorized to create organizations.

Plugins may change the parameters of this function depending on the value of the `type` parameter, see the *IGroupForm* plugin interface.

Parameters

- **name** (*string*) – the name of the organization, a string between 2 and 100 characters long, containing only lowercase alphanumeric characters, `-` and `_`
- **id** (*string*) – the id of the organization (optional)
- **title** (*string*) – the title of the organization (optional)
- **description** (*string*) – the description of the organization (optional)
- **image_url** (*string*) – the URL to an image to be displayed on the organization's page (optional)
- **state** (*string*) – the current state of the organization, e.g. `'active'` or `'deleted'`, only active organizations show up in search results and other lists of organizations, this parameter will be ignored if you are not authorized to change the state of the organization (optional, default: `'active'`)
- **approval_status** (*string*) – (optional)
- **extras** (*list of dataset extra dictionaries*) – the organization's extras (optional), extras are arbitrary (key: value) metadata items that can be added to organizations, each extra dictionary should have keys `'key'` (a string), `'value'` (a string), and optionally `'deleted'`
- **packages** (*list of dictionaries*) – the datasets (packages) that belong to the organization, a list of dictionaries each with keys `'name'` (string, the id or name of the dataset) and optionally `'title'` (string, the title of the dataset)

- **users** (*list of dictionaries*) – the users that belong to the organization, a list of dictionaries each with key `'name'` (string, the id or name of the user) and optionally `'capacity'` (string, the capacity in which the user is a member of the organization)

Returns
the newly created organization (unless 'return_id_only' is set to True in the context, in which case just the organization id will be returned)

Return type
dictionary

`ckan.logic.action.create.`**`user_create`**(*context: Context*, *data_dict: dict[str, Any]*) → dict[str, Any]

Create a new user.

You must be authorized to create users.

Parameters

- **name** (*string*) – the name of the new user, a string between 2 and 100 characters in length, containing only lowercase alphanumeric characters, - and _
- **email** (*string*) – the email address for the new user
- **password** (*string*) – the password of the new user, a string of at least 4 characters
- **id** (*string*) – the id of the new user (optional)
- **fullname** (*string*) – the full name of the new user (optional)
- **about** (*string*) – a description of the new user (optional)
- **image_url** (*string*) – the URL to an image to be displayed on the group's page (optional)
- **plugin_extras** (*dict*) – private extra user data belonging to plugins. Only sysadmin users may set this value. It should be a dict that can be dumped into JSON, and plugins should namespace their extras with the plugin name to avoid collisions with other plugins, eg:

```
{
    "name": "test_user",
    "email": "test@example.com",
    "plugin_extras": {
        "my_plugin": {
            "private_extra": 1
        },
        "another_plugin": {
            "another_extra": True
        }
    }
}
```

- **with_apitoken** (*bool*) – whether to create an API token for the user. (Optional)

Returns
the newly created user

Return type
dictionary

`ckan.logic.action.create.`**`user_invite`**(*context: Context*, *data_dict: dict[str, Any]*) → dict[str, Any]

Invite a new user.

You must be authorized to create group members.

Parameters

- **email** (*string*) – the email of the user to be invited to the group
- **group_id** (*string*) – the id or name of the group
- **role** (*string*) – role of the user in the group. One of `member`, `editor`, or `admin`

Returns
the newly created user

Return type
dictionary

`ckan.logic.action.create.`**`vocabulary_create`**(*context: Context*, *data_dict: dict[str, Any]*) → dict[str, Any]

Create a new tag vocabulary.

You must be a sysadmin to create vocabularies.

Parameters

- **name** (*string*) – the name of the new vocabulary, e.g. `'Genre'`
- **tags** (*list of tag dictionaries*) – the new tags to add to the new vocabulary, for the format of tag dictionaries see *tag_create()*

Returns
the newly-created vocabulary

Return type
dictionary

`ckan.logic.action.create.`**`tag_create`**(*context: Context*, *data_dict: dict[str, Any]*) → dict[str, Any]

Create a new vocabulary tag.

You must be a sysadmin to create vocabulary tags.

You can only use this function to create tags that belong to a vocabulary, not to create free tags. (To create a new free tag simply add the tag to a package, e.g. using the *package_update()* function.)

Parameters

- **name** (*string*) – the name for the new tag, a string between 2 and 100 characters long containing only alphanumeric characters, spaces and the characters `-`, `_` and `.`, e.g. `'Jazz'`
- **vocabulary_id** (*string*) – the id of the vocabulary that the new tag should be added to, e.g. the id of vocabulary `'Genre'`

Returns
the newly-created tag

Return type
dictionary

`ckan.logic.action.create.`**`follow_user`**(*context: Context*, *data_dict: dict[str, Any]*) → dict[str, Any]

Start following another user.

You must provide your API key in the Authorization header.

Parameters
id (*string*) – the id or name of the user to follow, e.g. `'joeuser'`

Returns
a representation of the 'follower' relationship between yourself and the other user

Return type
dictionary

`ckan.logic.action.create.`**`follow_dataset`**(*context: Context*, *data_dict: dict[str, Any]*) → dict[str, Any]

Start following a dataset.

You must provide your API key in the Authorization header.

Parameters
id (*string*) – the id or name of the dataset to follow, e.g. `'warandpeace'`

Returns
a representation of the 'follower' relationship between yourself and the dataset

Return type
dictionary

`ckan.logic.action.create.`**`group_member_create`**(*context: Context*, *data_dict: dict[str, Any]*) → dict[str, Any]

Make a user a member of a group.

You must be authorized to edit the group.

Parameters

- **id** (*string*) – the id or name of the group
- **username** (*string*) – name or id of the user to be made member of the group
- **role** (*string*) – role of the user in the group. One of `member`, `editor`, or `admin`

Returns
the newly created (or updated) membership

Return type
dictionary

`ckan.logic.action.create.`**`organization_member_create`**(*context: Context*, *data_dict: dict[str, Any]*) → dict[str, Any]

Make a user a member of an organization.

You must be authorized to edit the organization.

Parameters

- **id** (*string*) – the id or name of the organization
- **username** (*string*) – name or id of the user to be made member of the organization
- **role** (*string*) – role of the user in the organization. One of `member`, `editor`, or `admin`

Returns
the newly created (or updated) membership

Return type
dictionary

`ckan.logic.action.create.`**`follow_group`**(*context: Context*, *data_dict: dict[str, Any]*) → dict[str, Any]

Start following a group.

You must provide your API key in the Authorization header.

Parameters
id (*string*) – the id or name of the group to follow, e.g. `'roger'`

Returns
a representation of the 'follower' relationship between yourself and the group

Return type
dictionary

`ckan.logic.action.create.`**`api_token_create`**(*context: Context*, *data_dict: dict[str, Any]*) → dict[str, Any]

Create new API Token for current user.

Apart from the *user* and *name* field that are required by default implementation, there may be additional fields registered by extensions.

Parameters

- **`user`** (*string*) – name or id of the user who owns new API Token
- **`name`** (*string*) – distinctive name for API Token

Returns
Returns a dict with the key "token" containing the encoded token value. Extensions can privide additional fields via *add_extra* method of `IApiToken`

Return type
dictionary

4.9.3 ckan.logic.action.update

API functions for updating existing data in CKAN.

`ckan.logic.action.update.`**`resource_update`**(*context: Context*, *data_dict: dict[str, Any]*) → dict[str, Any]

Update a resource.

To update a resource you must be authorized to update the dataset that the resource belongs to.

Note: Update methods may delete parameters not explicitly provided in the data_dict. If you want to edit only a specific attribute use *resource_patch* instead.

For further parameters see `resource_create()`.

Parameters
`id` (*string*) – the id of the resource to update

Returns
the updated resource

Return type
string

`ckan.logic.action.update.`**`resource_view_update`**(*context: Context*, *data_dict: dict[str, Any]*) → dict[str, Any]

Update a resource view.

To update a resource_view you must be authorized to update the resource that the resource_view belongs to.

For further parameters see `resource_view_create()`.

Parameters
`id` (*string*) – the id of the resource_view to update

Returns
the updated resource_view

Return type
string

`ckan.logic.action.update.`**`resource_view_reorder`**(*context: Context*, *data_dict: dict[str, Any]*) → dict[str, Any]

Reorder resource views.

Parameters

- **id** (*string*) – the id of the resource
- **order** (*list of strings*) – the list of id of the resource to update the order of the views

Returns
the updated order of the view

Return type
dictionary

`ckan.logic.action.update.`**`package_update`**(*context: Context*, *data_dict: dict[str, Any]*) → str | dict[str, Any]

Update a dataset (package).

You must be authorized to edit the dataset and the groups that it belongs to.

Note: Update methods may delete parameters not explicitly provided in the data_dict. If you want to edit only a specific attribute use *package_patch* instead.

It is recommended to call *ckan.logic.action.get.package_show()*, make the desired changes to the result, and then call `package_update()` with it.

Plugins may change the parameters of this function depending on the value of the dataset's `type` attribute, see the *IDatasetForm* plugin interface.

For further parameters see *package_create()*.

Parameters
id (*string*) – the name or id of the dataset to update

Returns
the updated dataset (if `'return_id_only'` is `False` in the context, which is the default. Otherwise returns just the dataset id)

Return type
dictionary

`ckan.logic.action.update.`**`package_revise`**(*context: Context*, *data_dict: dict[str, Any]*) → dict[str, Any]

Revise a dataset (package) selectively with match, filter and update parameters.

You must be authorized to edit the dataset and the groups that it belongs to.

Parameters

- **match** (*dict*) – a dict containing "id" or "name" values of the dataset to update, all values provided must match the current dataset values or a ValidationError will be raised. e.g. `{"name": "my-data", "resources": {["name": "big.csv"]}}` would abort if the my-data dataset's first resource name is not "big.csv".

- **filter** (*comma-separated string patterns or list of string patterns*) – a list of string patterns of fields to remove from the current dataset. e.g. `"-resources__1"` would remove the second resource, `"+title, +resources, -*"` would remove all fields at the dataset level except title and all resources (default: `[]`)
- **update** (*dict*) – a dict with values to update/create after filtering e.g. `{"resources": [{"description": "file here"}]}` would update the description for the first resource
- **include** (*comma-separated-string patterns or list of string patterns*) – a list of string pattern of fields to include in response e.g. `"-*"` to return nothing (default: `[]` all fields returned)

`match` and `update` parameters may also be passed as "flattened keys", using either the item numeric index or its unique id (with a minimum of 5 characters), e.g. `update__resource__1f9ab__description="guidebook"` would set the description of the resource with id starting with "1f9ab" to "guidebook", and `update__resource__-1__description="guidebook"` would do the same on the last resource in the dataset.

The `extend` suffix can also be used on the update parameter to add a new item to a list, e.g. `update__resources__extend=[{"name": "new resource", "url": "https://example.com"}]` will add a new resource to the dataset with the provided `name` and `url`.

Usage examples:

- Change description in dataset, checking for old description:

```
match={"notes": "old notes", "name": "xyz"}
update={"notes": "new notes"}
```

- Identical to above, but using flattened keys:

```
match__name="xyz"
match__notes="old notes"
update__notes="new notes"
```

- Replace all fields at dataset level only, keep resources (note: dataset id and type fields can't be deleted)

```
match={"id": "1234abc-1420-cbad-1922"}
filter=["+resources", "-*"]
update={"name": "fresh-start", "title": "Fresh Start"}
```

- Add a new resource (__extend on flattened key):

```
match={"id": "abc0123-1420-cbad-1922"}
update__resources__extend=[{"name": "new resource", "url": "http://example.com"}
↪]
```

- Update a resource by its index:

```
match={"name": "my-data"}
update__resources__0={"name": "new name, first resource"}
```

- Update a resource by its id (provide at least 5 characters):

```
match={"name": "their-data"}
update__resources__19cfad={"description": "right one for sure"}
```

- Replace all fields of a resource:

```
match={"id": "34a12bc-1420-cbad-1922"}
filter=["+resources__1492a__id", "-resources__1492a__*"]
update__resources__1492a={"name": "edits here", "url": "http://example.com"}
```

Returns
a dict containing 'package':the updated dataset with fields filtered by include parameter

Return type
dictionary

`ckan.logic.action.update.`**`package_resource_reorder`**(*context: Context*, *data_dict: dict[str, Any]*) → dict[str, Any]

Reorder resources against datasets. If only partial resource ids are supplied then these are assumed to be first and the other resources will stay in their original order

Parameters

- **id** (*string*) – the id or name of the package to update
- **order** (*list*) – a list of resource ids in the order needed

`ckan.logic.action.update.`**`package_relationship_update`**(*context: Context*, *data_dict: dict[str, Any]*) → dict[str, Any]

Update a relationship between two datasets (packages).

The subject, object and type parameters are required to identify the relationship. Only the comment can be updated.

You must be authorized to edit both the subject and the object datasets.

Parameters

- **subject** (*string*) – the name or id of the dataset that is the subject of the relationship
- **object** (*string*) – the name or id of the dataset that is the object of the relationship
- **type** (*string*) – the type of the relationship, one of `'depends_on'`, `'dependency_of'`, `'derives_from'`, `'has_derivation'`, `'links_to'`, `'linked_from'`, `'child_of'` or `'parent_of'`
- **comment** (*string*) – a comment about the relationship (optional)

Returns
the updated relationship

Return type
dictionary

`ckan.logic.action.update.`**`group_update`**(*context: Context*, *data_dict: dict[str, Any]*) → dict[str, Any]

Update a group.

You must be authorized to edit the group.

Note: Update methods may delete parameters not explicitly provided in the data_dict. If you want to edit only a specific attribute use *group_patch* instead.

Plugins may change the parameters of this function depending on the value of the group's `type` attribute, see the *`IGroupForm`* plugin interface.

For further parameters see *`group_create()`*.

Parameters
id (*string*) – the name or id of the group to update

Returns
the updated group

Return type
dictionary

`ckan.logic.action.update.organization_update(context: Context, data_dict: dict[str, Any]) → dict[str, Any]`

Update a organization.

You must be authorized to edit the organization.

Note: Update methods may delete parameters not explicitly provided in the data_dict. If you want to edit only a specific attribute use *organization_patch* instead.

For further parameters see `organization_create()`.

Parameters

- **id** (*string*) – the name or id of the organization to update
- **packages** – ignored. use `package_owner_org_update()` to change package ownership

Returns
the updated organization

Return type
dictionary

`ckan.logic.action.update.user_update(context: Context, data_dict: dict[str, Any]) → dict[str, Any]`

Update a user account.

Normal users can only update their own user accounts. Sysadmins can update any user account. Can not modify exisiting user's name.

Note: Update methods may delete parameters not explicitly provided in the data_dict. If you want to edit only a specific attribute use *user_patch* instead.

For further parameters see `user_create()`.

Parameters
id (*string*) – the name or id of the user to update

Returns
the updated user account

Return type
dictionary

`ckan.logic.action.update.task_status_update(context: Context, data_dict: dict[str, Any]) → dict[str, Any]`

Update a task status.

Parameters

- **id** (*string*) – the id of the task status to update

- **entity_id** (*string*) –
- **entity_type** (*string*) –
- **task_type** (*string*) –
- **key** (*string*) –
- **value** – (optional)
- **state** – (optional)
- **last_updated** – (optional)
- **error** – (optional)

Returns
the updated task status

Return type
dictionary

`ckan.logic.action.update.task_status_update_many`(*context: Context*, *data_dict: dict[str, Any]*) → dict[str, Any]

Update many task statuses at once.

Parameters
data (*list of dictionaries*) – the task_status dictionaries to update, for the format of task status dictionaries see *task_status_update()*

Returns
the updated task statuses

Return type
list of dictionaries

`ckan.logic.action.update.term_translation_update`(*context: Context*, *data_dict: dict[str, Any]*) → dict[str, Any]

Create or update a term translation.

You must be a sysadmin to create or update term translations.

Parameters

- **term** (*string*) – the term to be translated, in the original language, e.g. `'romantic novel'`
- **term_translation** (*string*) – the translation of the term, e.g. `'Liebesroman'`
- **lang_code** (*string*) – the language code of the translation, e.g. `'de'`

Returns
the newly created or updated term translation

Return type
dictionary

`ckan.logic.action.update.term_translation_update_many`(*context: Context*, *data_dict: dict[str, Any]*) → dict[str, Any]

Create or update many term translations at once.

Parameters
data (*list of dictionaries*) – the term translation dictionaries to create or update, for the format of term translation dictionaries see *term_translation_update()*

Returns
a dictionary with key `'success'` whose value is a string stating how many term translations were updated

Return type
string

`ckan.logic.action.update.vocabulary_update(context: Context, data_dict: dict[str, Any]) → dict[str, Any]`

Update a tag vocabulary.

You must be a sysadmin to update vocabularies.

For further parameters see *vocabulary_create()*.

Parameters
id (*string*) – the id of the vocabulary to update

Returns
the updated vocabulary

Return type
dictionary

`ckan.logic.action.update.package_owner_org_update(context: Context, data_dict: dict[str, Any]) → None`

Update the owning organization of a dataset

Parameters

- **id** (*string*) – the name or id of the dataset to update
- **organization_id** (*string*) – the name or id of the owning organization

`ckan.logic.action.update.bulk_update_private(context: Context, data_dict: dict[str, Any]) → None`

Make a list of datasets private

Parameters

- **datasets** (*list of strings*) – list of ids of the datasets to update
- **org_id** (*string*) – id of the owning organization

`ckan.logic.action.update.bulk_update_public(context: Context, data_dict: dict[str, Any]) → None`

Make a list of datasets public

Parameters

- **datasets** (*list of strings*) – list of ids of the datasets to update
- **org_id** (*string*) – id of the owning organization

`ckan.logic.action.update.bulk_update_delete(context: Context, data_dict: dict[str, Any]) → None`

Make a list of datasets deleted

Parameters

- **datasets** (*list of strings*) – list of ids of the datasets to update
- **org_id** (*string*) – id of the owning organization

`ckan.logic.action.update.config_option_update(context: Context, data_dict: dict[str, Any]) → dict[str, Any]`

New in version 2.4.

Allows to modify some CKAN runtime-editable config options

It takes arbitrary key, value pairs and checks the keys against the config options update schema. If some of the provided keys are not present in the schema a `ValidationError` is raised. The values are then validated against the schema, and if validation is passed, for each key, value config option:

- It is stored on the `system_info` database table
- The Pylons `config` object is updated.
- The `app_globals` (`g`) object is updated (this only happens for options explicitly defined in the `app_globals` module.

The following lists a `key` parameter, but this should be replaced by whichever config options want to be updated, eg:

```
get_action('config_option_update)({}, {
    'ckan.site_title': 'My Open Data site',
})
```

Parameters
key (*string*) – a configuration option key (eg `ckan.site_title`). It must be present on the `update_configuration_schema`

Returns
a dictionary with the options set

Return type
dictionary

Note: You can see all available runtime-editable configuration options calling the *`config_option_list()`* action

---

Note: Extensions can modify which configuration options are runtime-editable. For details, check *Making configuration options runtime-editable*.

Warning: You should only add config options that you are comfortable they can be edited during runtime, such as ones you've added in your own extension, or have reviewed the use of in core CKAN.

4.9.4 ckan.logic.action.patch

New in version 2.3. API functions for partial updates of existing data in CKAN

`ckan.logic.action.patch.`**`package_patch`**(*context: Context*, *data_dict: dict[str, Any]*) → str | dict[str, Any]

Patch a dataset (package).

Parameters
id (*string*) – the id or name of the dataset

The difference between the update and patch methods is that the patch will perform an update of the provided parameters, while leaving all other parameters unchanged, whereas the update methods deletes all parameters not explicitly provided in the data_dict.

You are able to partially update and/or create resources with package_patch. If you are updating existing resources be sure to provide the resource id. Existing resources excluded from the package_patch data_dict will be removed.

Resources in the package data_dict without an id will be treated as new resources and will be added. New resources added with the patch method do not create the default views.

You must be authorized to edit the dataset and the groups that it belongs to.

`ckan.logic.action.patch.`**`resource_patch`**(*context: Context*, *data_dict: dict[str, Any]*) → dict[str, Any]

Patch a resource

Parameters
id (*string*) – the id of the resource

The difference between the update and patch methods is that the patch will perform an update of the provided parameters, while leaving all other parameters unchanged, whereas the update methods deletes all parameters not explicitly provided in the data_dict

`ckan.logic.action.patch.`**`group_patch`**(*context: Context*, *data_dict: dict[str, Any]*) → dict[str, Any]

Patch a group

Parameters
id (*string*) – the id or name of the group

The difference between the update and patch methods is that the patch will perform an update of the provided parameters, while leaving all other parameters unchanged, whereas the update methods deletes all parameters not explicitly provided in the data_dict

`ckan.logic.action.patch.`**`organization_patch`**(*context: Context*, *data_dict: dict[str, Any]*) → dict[str, Any]

Patch an organization

Parameters
id (*string*) – the id or name of the organization

The difference between the update and patch methods is that the patch will perform an update of the provided parameters, while leaving all other parameters unchanged, whereas the update methods deletes all parameters not explicitly provided in the data_dict

`ckan.logic.action.patch.`**`user_patch`**(*context: Context*, *data_dict: dict[str, Any]*) → dict[str, Any]

Patch a user

Parameters
id (*string*) – the id or name of the user

The difference between the update and patch methods is that the patch will perform an update of the provided parameters, while leaving all other parameters unchanged, whereas the update methods deletes all parameters not explicitly provided in the data_dict

4.9.5 ckan.logic.action.delete

API functions for deleting data from CKAN.

`ckan.logic.action.delete.`**`user_delete`**(*context: Context*, *data_dict: dict[str, Any]*) → None

Delete a user.

Only sysadmins can delete users.

Parameters
id (*string*) – the id or usernamename of the user to delete

`ckan.logic.action.delete.`**`package_delete`**(*context: Context*, *data_dict: dict[str, Any]*) → None

Delete a dataset (package).

This makes the dataset disappear from all web & API views, apart from the trash.

You must be authorized to delete the dataset.

Parameters
id (*string*) – the id or name of the dataset to delete

`ckan.logic.action.delete.`**`dataset_purge`**(*context: Context*, *data_dict: dict[str, Any]*) → None

Purge a dataset.

> **Warning:** Purging a dataset cannot be undone!

Purging a database completely removes the dataset from the CKAN database, whereas deleting a dataset simply marks the dataset as deleted (it will no longer show up in the front-end, but is still in the db).

You must be authorized to purge the dataset.

Parameters
id (*string*) – the name or id of the dataset to be purged

`ckan.logic.action.delete.`**`resource_delete`**(*context: Context*, *data_dict: dict[str, Any]*) → None

Delete a resource from a dataset.

You must be a sysadmin or the owner of the resource to delete it.

Parameters
id (*string*) – the id of the resource

`ckan.logic.action.delete.`**`resource_view_delete`**(*context: Context*, *data_dict: dict[str, Any]*) → None

Delete a resource_view.

Parameters
id (*string*) – the id of the resource_view

`ckan.logic.action.delete.`**`resource_view_clear`**(*context: Context*, *data_dict: dict[str, Any]*) → None

Delete all resource views, or all of a particular type.

Parameters
view_types (*list*) – specific types to delete (optional)

`ckan.logic.action.delete.`**`package_relationship_delete`**(*context: Context*, *data_dict: dict[str, Any]*) → None

Delete a dataset (package) relationship.

You must be authorised to delete dataset relationships, and to edit both the subject and the object datasets.

Parameters

- **subject** (*string*) – the id or name of the dataset that is the subject of the relationship
- **object** (*string*) – the id or name of the dataset that is the object of the relationship
- **type** (*string*) – the type of the relationship

`ckan.logic.action.delete.`**`member_delete`**(*context: Context*, *data_dict: dict[str, Any]*) → None

Remove an object (e.g. a user, dataset or group) from a group.

You must be authorized to edit a group to remove objects from it.

Parameters

- **id** (*string*) – the id of the group
- **object** (*string*) – the id or name of the object to be removed
- **object_type** (*string*) – the type of the object to be removed, e.g. `package` or `user`

`ckan.logic.action.delete.`**`package_collaborator_delete`**(*context: Context*, *data_dict: dict[str, Any]*) → None

Remove a collaborator from a dataset.

Currently you must be an Admin on the dataset owner organization to manage collaborators.

Note: This action requires the collaborators feature to be enabled with the *ckan.auth.allow_dataset_collaborators* configuration option.

Parameters

- **id** (*string*) – the id or name of the dataset
- **user_id** (*string*) – the id or name of the user to remove

`ckan.logic.action.delete.`**`group_delete`**(*context: Context*, *data_dict: dict[str, Any]*) → None

Delete a group.

You must be authorized to delete the group.

Parameters

id (*string*) – the name or id of the group

`ckan.logic.action.delete.`**`organization_delete`**(*context: Context*, *data_dict: dict[str, Any]*) → None

Delete an organization.

You must be authorized to delete the organization and no datasets should belong to the organization unless 'ckan.auth.create_unowned_dataset=True'

Parameters

id (*string*) – the name or id of the organization

`ckan.logic.action.delete.`**`group_purge`**(*context: Context*, *data_dict: dict[str, Any]*) → None

Purge a group.

> **Warning:** Purging a group cannot be undone!

Purging a group completely removes the group from the CKAN database, whereas deleting a group simply marks the group as deleted (it will no longer show up in the frontend, but is still in the db).

Datasets in the organization will remain, just not in the purged group.

You must be authorized to purge the group.

Parameters

id (*string*) – the name or id of the group to be purged

`ckan.logic.action.delete.`**`organization_purge`**(*context: Context*, *data_dict: dict[str, Any]*) → None

Purge an organization.

> **Warning:** Purging an organization cannot be undone!

Purging an organization completely removes the organization from the CKAN database, whereas deleting an organization simply marks the organization as deleted (it will no longer show up in the frontend, but is still in the db).

Datasets owned by the organization will remain, just not in an organization any more.

You must be authorized to purge the organization.

Parameters
id (*string*) – the name or id of the organization to be purged

`ckan.logic.action.delete.task_status_delete(context: Context, data_dict: dict[str, Any]) → None`

Delete a task status.

You must be a sysadmin to delete task statuses.

Parameters
id (*string*) – the id of the task status to delete

`ckan.logic.action.delete.vocabulary_delete(context: Context, data_dict: dict[str, Any]) → None`

Delete a tag vocabulary.

You must be a sysadmin to delete vocabularies.

Parameters
id (*string*) – the id of the vocabulary

`ckan.logic.action.delete.tag_delete(context: Context, data_dict: dict[str, Any]) → None`

Delete a tag.

You must be a sysadmin to delete tags.

Parameters

- **id** (*string*) – the id or name of the tag
- **vocabulary_id** (*string*) – the id or name of the vocabulary that the tag belongs to (optional, default: None)

`ckan.logic.action.delete.unfollow_user(context: Context, data_dict: dict[str, Any]) → None`

Stop following a user.

Parameters
id (*string*) – the id or name of the user to stop following

`ckan.logic.action.delete.unfollow_dataset(context: Context, data_dict: dict[str, Any]) → None`

Stop following a dataset.

Parameters
id (*string*) – the id or name of the dataset to stop following

`ckan.logic.action.delete.group_member_delete(context: Context, data_dict: dict[str, Any]) → None`

Remove a user from a group.

You must be authorized to edit the group.

Parameters

- **id** (*string*) – the id or name of the group
- **username** (*string*) – name or id of the user to be removed

`ckan.logic.action.delete.organization_member_delete(context: Context, data_dict: dict[str, Any]) → None`

Remove a user from an organization.

You must be authorized to edit the organization.

Parameters

- **id** (*string*) – the id or name of the organization
- **username** (*string*) – name or id of the user to be removed

`ckan.logic.action.delete.unfollow_group(context: Context, data_dict: dict[str, Any]) → None`

Stop following a group.

Parameters

id (*string*) – the id or name of the group to stop following

`ckan.logic.action.delete.job_clear(context: Context, data_dict: dict[str, Any]) → list[str]`

Clear background job queues.

Does not affect jobs that are already being processed.

Parameters

queues (*list*) – The queues to clear. If not given then ALL queues are cleared.

Returns

The cleared queues.

Return type

list

New in version 2.7.

`ckan.logic.action.delete.job_cancel(context: Context, data_dict: dict[str, Any]) → None`

Cancel a queued background job.

Removes the job from the queue and deletes it.

Parameters

id (*string*) – The ID of the background job.

New in version 2.7.

`ckan.logic.action.delete.api_token_revoke(context: Context, data_dict: dict[str, Any]) → None`

Delete API Token.

Parameters

- **token** (*string*) – Token to remove(required if *jti* not specified).
- **jti** (*string*) – Id of the token to remove(overrides *token* if specified).

New in version 3.0.

CHAPTER

FIVE

EXTENDING GUIDE

The following sections will teach you how to customize and extend CKAN's features by developing your own CKAN extensions.

See also:

Some **core extensions** come packaged with CKAN. Core extensions don't need to be installed before you can use them as they're installed when you install CKAN, they can simply be enabled by following the setup instructions in each extension's documentation (some core extensions are already enabled by default). For example, the *datastore extension*, *multilingual extension*, and *stats extension* are all core extensions, and the *data viewer* also uses core extensions to enable data previews for different file formats.

See also:

External extensions are CKAN extensions that don't come packaged with CKAN, but must be downloaded and installed separately. Find external extensions at https://extensions.ckan.org/. Again, follow each extension's own documentation to install, setup, and use the extension.

5.1 Writing extensions tutorial

This tutorial will walk you through the process of creating a simple CKAN extension, and introduce the core concepts that CKAN extension developers need to know along the way. As an example, we'll use the `example_iauthfunctions` extension that's packaged with CKAN. This is a simple CKAN extension that customizes some of CKAN's authorization rules.

5.1.1 Installing CKAN

Before you can start developing a CKAN extension, you'll need a working source install of CKAN on your system. If you don't have a CKAN source install already, follow the instructions in *Installing CKAN from source* before continuing.

Note: If you are developing extension without actual source installation of CKAN(i.e. if you have installed CKAN as package via *pip install ckan*), you can install all main and dev dependencies with the following commands:

```
pip install -r https://raw.githubusercontent.com/ckan/ckan/ckan-2.10.4/requirements.txt
pip install -r https://raw.githubusercontent.com/ckan/ckan/ckan-2.10.4/dev-requirements.
↪txt
```

5.1.2 Creating a new extension

Extensions

A CKAN *extension* is a Python package that modifies or extends CKAN. Each extension contains one or more *plugins* that must be added to your CKAN config file to activate the extension's features.

You can use `cookiecutter` command to create an "empty" extension from a template. Or the CLI command `ckan generate extension`. For whichever method you choose, the first step is to activate your CKAN virtual environment:

```
. /usr/lib/ckan/default/bin/activate
```

`cookiecutter`

When you run `cookiecutter`, your new extension's directory will be created in the current working directory by default (you can override this with the `-o` option), so change to the directory that you want your extension to be created in. Usually you'll want to track your extension code using a version control system such as `git`, so you wouldn't want to create your extension in the `ckan` source directory because that directory already contains the CKAN git repo. Let's use the parent directory instead:

```
cd /usr/lib/ckan/default/src
```

Now run `cookiecutter` to create your extension:

```
cookiecutter ckan/contrib/cookiecutter/ckan_extension/
```

CLI Command

Using the `ckan generate extension` place the extension's directory in the `ckan` source code's parent directory (this can be changed the using the `-o` option). Run the command to create the extension:

```
ckan generate extension
```

The commands will present a few prompts. The information you give will end up in your extension's `setup.py` file (where you can edit them later if you want).

Note: The first prompt is for the name of your next extension. CKAN extension names *have* to begin with `ckanext-`. This tutorial uses the project name `ckanext-iauthfunctions`.

Once the command has completed, your new CKAN extension's project directory will have been created and will contain a few directories and files to get you started:

```
ckanext-iauthfunctions/
    ckanext/
        __init__.py
        iauthfunctions/
            __init__.py
    ckanext_iauthfunctions.egg-info/
    setup.py
```

`ckanext_iauthfunctions.egg_info` is a directory containing automatically generated metadata about your project. It's used by Python's packaging and distribution tools. In general, you don't need to edit or look at anything in this directory, and you should not add it to version control.

`setup.py` is the setup script for your project. As you'll see later, you use this script to install your project into a virtual environment. It contains several settings that you'll update as you develop your project.

`ckanext/iauthfunctions` is the Python package directory where we'll add the source code files for our extension.

5.1.3 Creating a plugin class

Plugins

Each CKAN extension contains one or more plugins that provide the extension's features.

`cookiecutter` should have created the following file file `ckanext-iauthfunctions/ckanext/iauthfunctions/plugin.py`. Edit it to match the following:

```
# encoding: utf-8

import ckan.plugins as plugins


class ExampleIAuthFunctionsPlugin(plugins.SingletonPlugin):
    pass
```

Our plugin is a normal Python class, named `ExampleIAuthFunctionsPlugin` in this example, that inherits from CKAN's *SingletonPlugin* class.

Note: Every CKAN plugin class should inherit from *SingletonPlugin*.

5.1.4 Adding the plugin to `setup.py`

Now let's add our class to the `entry_points` in `setup.py`. This identifies the plugin class to CKAN once the extension is installed in CKAN's virtualenv, and associates a plugin name with the class. Edit `ckanext-iauthfunctions/setup.py` and add a line to the `entry_points` section like this:

```
entry_points='''
    [ckan.plugins]
    example_iauthfunctions=ckanext.iauthfunctions.plugin:ExampleIAuthFunctionsPlugin
''',
```

5.1.5 Installing the extension

When you *install CKAN*, you create a Python virtual environment in a directory on your system (/usr/lib/ckan/default by default) and install the CKAN Python package and the other packages that CKAN depends on into this virtual environment. Before we can use our plugin, we must install our extension into our CKAN virtual environment.

Make sure your virtualenv is activated, change to the extension's directory, and run `python setup.py develop`:

```
. /usr/lib/ckan/default/bin/activate
cd /usr/lib/ckan/default/src/ckanext-iauthfunctions
python setup.py develop
```

5.1.6 Enabling the plugin

An extension's plugins must be added to the *ckan.plugins* setting in your CKAN config file so that CKAN will call the plugins' methods. The name that you gave to your plugin class in the *left-hand-side of the assignment in the setup.py file* (`example_iauthfunctions` in this example) is the name you'll use for your plugin in CKAN's config file:

```
ckan.plugins = stats text_view datatables_view example_iauthfunctions
```

You should now be able to start CKAN in the development web server and have it start up without any problems:

```
$ ckan -c /etc/ckan/default/ckan.ini run
Starting server in PID 13961.
serving on 0.0.0.0:5000 view at http://127.0.0.1:5000
```

If your plugin is in the *ckan.plugins* setting and CKAN starts without crashing, then your plugin is installed and CKAN can find it. Of course, your plugin doesn't *do* anything yet.

5.1.7 Troubleshooting

`PluginNotFoundException`

If CKAN crashes with a `PluginNotFoundException` like this:

```
ckan.plugins.core.PluginNotFoundException: example_iauthfunctions
```

then:

- Check that the name you've used for your plugin in your CKAN config file is the same as the name you've used in your extension's `setup.py` file
- Check that you've run `python setup.py develop` in your extension's directory, with your CKAN virtual environment activated. Every time you add a new plugin to your extension's `setup.py` file, you need to run `python setup.py develop` again before you can use the new plugin.

ImportError

If you get an `ImportError` from CKAN relating to your plugin, it's probably because the path to your plugin class in your `setup.py` file is wrong.

5.1.8 Implementing the `IAuthFunctions` plugin interface

Plugin interfaces

CKAN provides a number of *plugin interfaces* that plugins must implement to hook into CKAN and modify or extend it. Each plugin interface defines a number of methods that a plugin that implements the interface must provide. CKAN will call your plugin's implementations of these methods, to allow your plugin to do its stuff.

To modify CKAN's authorization behavior, we'll implement the *`IAuthFunctions`* plugin interface. This interface defines just one method, that takes no parameters and returns a dictionary:

`get_auth_functions`()	Return the authorization functions provided by this plugin.

Action functions and authorization functions

At this point, it's necessary to take a short diversion to explain how authorization works in CKAN.

Every action that can be carried out using the CKAN web interface or API is implemented by an *action function* in one of the four files `ckan/logic/action/{create,delete,get,update}.py`.

For example, when creating a dataset either using the web interface or using the *`package_create()`* API call, *`ckan.logic.action.create.package_create()`* is called. There's also *`ckan.logic.action.get.package_show()`*, *`ckan.logic.action.update.package_update()`*, and *`ckan.logic.action.delete.package_delete()`*.

For a full list of the action functions available in CKAN, see the *Action API reference*.

Each action function has a corresponding authorization function in one of the four files `ckan/logic/auth/{create,delete,get,update}.py`, CKAN calls this authorization function to decide whether the user is authorized to carry out the requested action. For example, when creating a new package using the web interface or API, `ckan.logic.auth.create.package_create()` is called.

The *`IAuthFunctions`* plugin interface allows CKAN plugins to hook into this authorization system to add their own authorization functions or override the default authorization functions. In this way, plugins have complete control to customize CKAN's auth.

Whenever a user tries to create a new group via the web interface or the API, CKAN calls the `group_create()` authorization function to decide whether to allow the action. Let's override this function and simply prevent anyone from creating new groups(Note: this is default behavior. In order to go further, you need to change `ckan.auth.user_create_groups` to *True* in configuration file). Edit your `plugin.py` file so that it looks like this:

```
# encoding: utf-8
from __future__ import annotations

from typing import Any, Optional
from ckan.types import AuthResult, Context
```

(continues on next page)

(continued from previous page)

```
import ckan.plugins as plugins


def group_create(
        context: Context,
        data_dict: Optional[dict[str, Any]] = None) -> AuthResult:
    return {'success': False, 'msg': 'No one is allowed to create groups'}


class ExampleIAuthFunctionsPlugin(plugins.SingletonPlugin):
    plugins.implements(plugins.IAuthFunctions)

    def get_auth_functions(self):
        return {'group_create': group_create}
```

Our `ExampleIAuthFunctionsPlugin` class now calls `implements()` to tell CKAN that it implements the *`IAuthFunctions`* interface, and provides an implementation of the interface's *`get_auth_functions()`* method that overrides the default `group_create()` function with a custom one.

See also:

Starting from CKAN 2.10, you can also use the `ckan.plugins.toolkit.blanket` decorators to implement common interfaces in your plugins. See the `blanket` method in the *Plugins toolkit reference*.

Our custom function simply returns `{'success': False}` to refuse to let anyone create a new group.

If you now restart CKAN and reload the `/group` page, as long as you're not a sysadmin user you should see the `Add Group` button disappear. The CKAN web interface automatically hides buttons that the user is not authorized to use. Visiting `/group/new` directly will redirect you to the login page. If you try to call *`group_create()`* via the API, you'll receive an `Authorization Error` from CKAN:

```
$ http 127.0.0.1:5000/api/3/action/group_create Authorization:*** name=my_group
HTTP/1.0 403 Forbidden
Access-Control-Allow-Headers: Authorization, Content-Type
Access-Control-Allow-Methods: POST, PUT, GET, DELETE, OPTIONS
Access-Control-Allow-Origin: *
Cache-Control: no-cache
Content-Length: 2866
Content-Type: application/json;charset=utf-8
Date: Wed, 12 Jun 2013 13:38:01 GMT
Pragma: no-cache
Server: PasteWSGIServer/0.5 Python/2.7.4

{
    "error": {
        "__type": "Authorization Error",
        "message": "Access denied"
    },
    "help": "Create a new group...",
    "success": false
}
```

If you're logged in as a sysadmin user however, you'll still be able to create new groups. Sysadmin users can always carry out any action, they bypass the authorization functions.

5.1.9 Using the plugins toolkit

Let's make our custom authorization function a little smarter, and allow only users who are members of a particular group named `curators` to create new groups.

First run CKAN, login and then create a new group called `curators`. Then edit `plugin.py` so that it looks like this:

Note: This version of `plugin.py` will crash if the user is not logged in or if the site doesn't have a group called `curators`. You'll want to create a `curators` group in your CKAN before editing your plugin to look like this. See *Exception handling* below.

```
# encoding: utf-8
from __future__ import annotations

from ckan.types import (
    AuthFunction, AuthResult, Context, ContextValidator, DataDict)
from typing import Optional, cast
import ckan.plugins as plugins
import ckan.plugins.toolkit as toolkit


def group_create(
        context: Context, data_dict: Optional[DataDict] = None) -> AuthResult:
    # Get the user name of the logged-in user.
    user_name: str = context['user']

    # Get a list of the members of the 'curators' group.
    members = toolkit.get_action('member_list')(
        {},
        {'id': 'curators', 'object_type': 'user'})

    # 'members' is a list of (user_id, object_type, capacity) tuples, we're
    # only interested in the user_ids.
    member_ids = [member_tuple[0] for member_tuple in members]

    # We have the logged-in user's user name, get their user id.
    convert_user_name_or_id_to_id = cast(
        ContextValidator,
        toolkit.get_converter('convert_user_name_or_id_to_id'))
    user_id = convert_user_name_or_id_to_id(user_name, context)

    # Finally, we can test whether the user is a member of the curators group.
    if user_id in member_ids:
        return {'success': True}
    else:
        return {'success': False,
                'msg': 'Only curators are allowed to create groups'}


class ExampleIAuthFunctionsPlugin(plugins.SingletonPlugin):
    plugins.implements(plugins.IAuthFunctions)
```

(continues on next page)

(continued from previous page)

```
    def get_auth_functions(self) -> dict[str, AuthFunction]:
        return {'group_create': group_create}
```

context

The `context` parameter of our `group_create()` function is a dictionary that CKAN passes to all authorization and action functions containing some computed variables. Our function gets the name of the logged-in user from `context`:

```
    user_name: str = context['user']
```

data_dict

The `data_dict` parameter of our `group_create()` function is another dictionary that CKAN passes to all authorization and action functions. `data_dict` contains any data posted by the user to CKAN, eg. any fields they've completed in a web form they're submitting or any `JSON` fields they've posted to the API. If we inspect the contents of the `data_dict` passed to our `group_create()` authorization function, we'll see that it contains the details of the group the user wants to create:

```
{'description': u'A really cool group',
 'image_url': u'',
 'name': u'my_group',
 'title': u'My Group',
 'type': 'group',
 'users': [{'capacity': 'admin', 'name': u'seanh'}]}
```

The plugins toolkit

CKAN's *plugins toolkit* is a Python module containing core CKAN functions, classes and exceptions for use by CKAN extensions.

The toolkit's `get_action()` function returns a CKAN action function. The action functions available to extensions are the same functions that CKAN uses internally to carry out actions when users make requests to the web interface or API. Our code uses `get_action()` to get the *`member_list()`* action function, which it uses to get a list of the members of the `curators` group:

```
    members = toolkit.get_action('member_list')(
        {},
        {'id': 'curators', 'object_type': 'user'})
```

Calling *`member_list()`* in this way is equivalent to posting the same data dict to the `/api/3/action/member_list` API endpoint. For other action functions available from `get_action()`, see *Action API reference*.

The toolkit's `get_validator()` function returns validator and converter functions from *`ckan.logic.converters`* for plugins to use. This is the same set of converter functions that CKAN's action functions use to convert user-provided data. Our code uses `get_validator()` to get the *`convert_user_name_or_id_to_id()`* converter function, which it uses to convert the name of the logged-in user to their user `id`:

```
    convert_user_name_or_id_to_id = cast(
        ContextValidator,
        toolkit.get_converter('convert_user_name_or_id_to_id'))
    user_id = convert_user_name_or_id_to_id(user_name, context)
```

Finally, we can test whether the logged-in user is a member of the `curators` group, and allow or refuse the action:

```
    if user_id in member_ids:
        return {'success': True}
    else:
        return {'success': False,
                'msg': 'Only curators are allowed to create groups'}
```

5.1.10 Exception handling

There are two bugs in our `plugin.py` file that need to be fixed using exception handling. First, the class will crash if the site does not have a group named `curators`.

Tip: If you've already created a `curators` group and want to test what happens when the site has no `curators` group, you can use CKAN's command line interface to *clean and reinitialize your database*.

Try visiting the `/group` page in CKAN with our `example_iauthfunctions` plugin activated in your CKAN config file and with no `curators` group in your site. If you have `debug = false` in your CKAN config file, you'll see something like this in your browser:

```
Error 500

Server Error

An internal server error occurred
```

If you have `debug = true` in your CKAN config file, then you'll see a traceback page with details about the crash.

You'll also get a `500 Server Error` if you try to create a group using the `group_create` API action.

To handle the situation where the site has no `curators` group without crashing, we'll have to handle the exception that CKAN's *`member_list()`* function raises when it's asked to list the members of a group that doesn't exist. Replace the `member_list` line in your `plugin.py` file with these lines:

```
    try:
        members = toolkit.get_action('member_list')(
            {},
            {'id': 'curators', 'object_type': 'user'})
    except toolkit.ObjectNotFound:
        # The curators group doesn't exist.
        return {'success': False,
                'msg': "The curators groups doesn't exist, so only sysadmins "
                       "are authorized to create groups."}
```

With these `try` and `except` clauses added, we should be able to load the `/group` page and add groups, even if there isn't already a group called `curators`.

Second, `plugin.py` will crash if a user who is not logged-in tries to create a group. If you logout of CKAN, and then visit `/group/new` you'll see another `500 Server Error`. You'll also get this error if you post to the *group_create()* API action without *providing an API key*.

When the user isn't logged in, `context['user']` contains the user's IP address instead of a user name:

```
{'model': <module 'ckan.model' from ...>,
 'user': u'127.0.0.1'}
```

When we pass this IP address as the user name to *convert_user_name_or_id_to_id()*, the converter function will raise an exception because no user with that user name exists. We need to handle that exception as well, replace the `convert_user_name_or_id_to_id` line in your `plugin.py` file with these lines:

```
    convert_user_name_or_id_to_id = cast(
        ContextValidator,
        toolkit.get_converter('convert_user_name_or_id_to_id'))
    try:
        user_id = convert_user_name_or_id_to_id(user_name, context)
    except toolkit.Invalid:
        # The user doesn't exist (e.g. they're not logged-in).
        return {'success': False,
                'msg': 'You must be logged-in as a member of the curators '
                       'group to create new groups.'}
```

5.1.11 We're done!

Here's our final, working `plugin.py` module in full:

```
# encoding: utf-8

from ckan.types import AuthResult, Context, ContextValidator, DataDict
from typing import Optional, cast
import ckan.plugins as plugins
import ckan.plugins.toolkit as toolkit


def group_create(
        context: Context, data_dict: Optional[DataDict] = None) -> AuthResult:
    # Get the user name of the logged-in user.
    user_name = context['user']

    # Get a list of the members of the 'curators' group.
    try:
        members = toolkit.get_action('member_list')(
            {},
            {'id': 'curators', 'object_type': 'user'})
    except toolkit.ObjectNotFound:
        # The curators group doesn't exist.
        return {'success': False,
```

(continues on next page)

(continued from previous page)

```
                'msg': "The curators groups doesn't exist, so only sysadmins "
                       "are authorized to create groups."}

    # 'members' is a list of (user_id, object_type, capacity) tuples, we're
    # only interested in the user_ids.
    member_ids = [member_tuple[0] for member_tuple in members]

    # We have the logged-in user's user name, get their user id.
    convert_user_name_or_id_to_id = cast(
        ContextValidator,
        toolkit.get_converter('convert_user_name_or_id_to_id'))
    try:
        user_id = convert_user_name_or_id_to_id(user_name, context)
    except toolkit.Invalid:
        # The user doesn't exist (e.g. they're not logged-in).
        return {'success': False,
                'msg': 'You must be logged-in as a member of the curators '
                       'group to create new groups.'}

    # Finally, we can test whether the user is a member of the curators group.
    if user_id in member_ids:
        return {'success': True}
    else:
        return {'success': False,
                'msg': 'Only curators are allowed to create groups'}


class ExampleIAuthFunctionsPlugin(plugins.SingletonPlugin):
    plugins.implements(plugins.IAuthFunctions)

    def get_auth_functions(self):
        return {'group_create': group_create}
```

In working through this tutorial, you've covered all the key concepts needed for writing CKAN extensions, including:

- Creating an extension
- Creating a plugin within your extension
- Adding your plugin to your extension's `setup.py` file, and installing your extension
- Making your plugin implement one of CKAN's *plugin interfaces*
- Using the *plugins toolkit*
- Handling exceptions

5.1.12 Troubleshooting

AttributeError

If you get an AttributeError like this one:

```
AttributeError: 'ExampleIAuthFunctionsPlugin' object has no attribute 'get_auth_functions
↪'
```

it means that your plugin class does not implement one of the plugin interface's methods. A plugin must implement every method of every plugin interface that it implements.

Todo: Can you user inherit=True to avoid having to implement them all?

Other AttributeErrors can happen if your method returns the wrong type of value, check the documentation for each plugin interface method to see what your method should return.

TypeError

If you get a TypeError like this one:

```
TypeError: get_auth_functions() takes exactly 3 arguments (1 given)
```

it means that one of your plugin methods has the wrong number of parameters. A plugin has to implement each method in a plugin interface with the same parameters as in the interface.

5.2 Using custom config settings in extensions

Extensions can define their own custom config settings that users can add to their CKAN config files to configure the behavior of the extension.

Continuing with the *IAuthFunctions* example from *Writing extensions tutorial*, let's make an alternative version of the extension that allows users to create new groups if a new config setting ckan.iauthfunctions.users_can_create_groups is True:

```
# encoding: utf-8

from typing import Optional
from ckan.types import AuthResult, Context, DataDict

import ckan.plugins as plugins
import ckan.plugins.toolkit as toolkit
from ckan.config.declaration import Declaration, Key


def group_create(
        context: Context,
        data_dict: Optional[DataDict] = None) -> AuthResult:

    # Get the value of the ckan.iauthfunctions.users_can_create_groups
    # setting from the CKAN config file as a string, or False if the setting
```

(continues on next page)

(continued from previous page)

```
    # isn't in the config file.
    users_can_create_groups = toolkit.config.get(
        'ckan.iauthfunctions.users_can_create_groups')

    if users_can_create_groups:
        return {'success': True}
    else:
        return {'success': False,
                'msg': 'Only sysadmins can create groups'}


class ExampleIAuthFunctionsPlugin(plugins.SingletonPlugin):
    plugins.implements(plugins.IAuthFunctions)
    plugins.implements(plugins.IConfigDeclaration)

    def get_auth_functions(self):
        return {'group_create': group_create}

    # IConfigDeclaration

    def declare_config_options(self, declaration: Declaration, key: Key):
        declaration.declare_bool(
            key.ckan.iauthfunctions.users_can_create_groups)
```

The `group_create` authorization function in this plugin uses `config` to read the setting from the config file, then calls `ckan.plugins.toolkit.asbool()` to convert the value from a string (all config settings values are strings, when read from the file) to a boolean.

Note: There are also `asint()` and `aslist()` functions in the plugins toolkit.

With this plugin enabled, you should find that users can create new groups if you have `ckan.iauthfunctions.users_can_create_groups = True` in the `[app:main]` section of your CKAN config file. Otherwise, only sysadmin users will be allowed to create groups.

Note: Names of config settings provided by extensions should include the name of the extension, to avoid conflicting with core config settings or with config settings from other extensions. See *Avoid name clashes*.

Note: The users still need to be logged-in to create groups. In general creating, updating or deleting content in CKAN requires the user to be logged-in to a registered user account, no matter what the relevant authorization function says.

5.3 Making configuration options runtime-editable

Extensions can allow certain configuration options to be edited during *runtime*, as opposed to having to edit the configuration file and restart the server.

Warning: Only configuration options which are not critical, sensitive or could cause the CKAN instance to break should be made runtime-editable. You should only add config options that you are comfortable they can be edited during runtime, such as ones you've added in your own extension, or have reviewed the use of in core CKAN.

Note: Only sysadmin users are allowed to modify runtime-editable configuration options.

In this tutorial we will show how to make changes to our extension to make two configuration options runtime-editable: *ckan.datasets_per_page* and a custom one named `ckanext.example_iconfigurer.test_conf`. You can see the changes in the `example_iconfigurer` extension that's packaged with CKAN. If you haven't done yet, you should check the *Writing extensions tutorial* first.

This tutorial assumes that we have CKAN running on the paster development server at http://localhost:5000, and that we are using the *API key* of a sysadmin user.

First of all, let's call the *`config_option_list()`* API action to see what configuration options are editable during runtime (the `| python -m json.tool` bit at the end is added to format the output nicely):

```
curl -H "Authorization: XXX" http://localhost:5000/api/action/config_option_list |␣
↪python -m json.tool
{
    "help": "http://localhost:5000/api/3/action/help_show?name=config_option_list",
    "result": [
        "ckan.site_custom_css",
        "ckan.theme",
        "ckan.site_title",
        "ckan.site_about",
        "ckan.site_url",
        "ckan.site_logo",
        "ckan.site_description",
        "ckan.site_intro_text",
        "ckan.hola"
    ],
    "success": true
}
```

We can see that the two options that we want to make runtime-editable are not on the list. Trying to update one of them with the *`config_option_update()`* action would return an error.

To include them, we need to add them to the schema that CKAN will use to decide which configuration options can be edited safely at runtime. This is done with the *`update_config_schema()`* method of the *`IConfigurer`* interface.

Let's have a look at how our extension should look like:

```
# encoding: utf-8

import ckan.plugins as plugins
import ckan.plugins.toolkit as toolkit
```

(continues on next page)

(continued from previous page)

```
from ckan.types import Schema


class ExampleIConfigurerPlugin(plugins.SingletonPlugin):

    plugins.implements(plugins.IConfigurer)

    # IConfigurer

    def update_config_schema(self, schema: Schema):

        ignore_missing = toolkit.get_validator('ignore_missing')
        unicode_safe = toolkit.get_validator('unicode_safe')
        is_positive_integer = toolkit.get_validator('is_positive_integer')

        schema.update({
            # This is an existing CKAN core configuration option, we are just
            # making it available to be editable at runtime
            'ckan.datasets_per_page': [ignore_missing, is_positive_integer],

            # This is a custom configuration option
            'ckanext.example_iconfigurer.test_conf': [ignore_missing,
                                                      unicode_safe],
        })

        return schema
```

The `update_config_schema` method will receive the default schema for runtime-editable configuration options used by CKAN core. We can then add keys to it to make new options runtime-editable (or remove them if we don't want them to be runtime-editable). The schema is a dictionary mapping configuration option keys to lists of validator and converter functions to be applied to those keys. To get validator functions defined in CKAN core we use the `get_validator()` function.

Note: Make sure that the first validator applied to each key is the `ignore_missing` one, otherwise this key will need to be always set when updating the configuration.

Restart the web server and do another request to the *`config_option_list()`* API action:

```
curl -H "Authorization: XXX" http://localhost:5000/api/action/config_option_list |␣
↪python -m json.tool
{
    "help": "http://localhost:5000/api/3/action/help_show?name=config_option_list",
    "result": [
        "ckan.datasets_per_page",
        "ckanext.example_iconfigurer.test_conf",
        "ckan.site_custom_css",
        "ckan.theme",
        "ckan.site_title",
        "ckan.site_about",
        "ckan.site_url",
        "ckan.site_logo",
```

(continues on next page)

(continued from previous page)

```
        "ckan.site_description",
        "ckan.site_intro_text",
        "ckan.hola"
    ],
    "success": true
}
```

Our two new configuration options are available to be edited at runtime. We can test it calling the *config_option_update()* action:

```
curl -X POST -H "Authorization: XXX" http://localhost:5000/api/action/config_option_
↪update -d "{\"ckan.datasets_per_page\": 5}" | python -m json.tool
{
    "help": "http://localhost:5001/api/3/action/help_show?name=config_option_update",
    "result": {
        "ckan.datasets_per_page": 5
    },
    "success": true
}
```

The configuration has now been updated. If you visit the main search page at http://localhost:5000/dataset only 5 datasets should appear in the results as opposed to the usual 20.

At this point both our configuration options can be updated via the API, but we also want to make them available on the *administration interface* so non-technical users don't need to use the API to change them.

To do so, we will extend the CKAN core template as described in the *Customizing CKAN's templates* documentation.

First add the *update_config()* method to your plugin and register the extension `templates` folder:

```
# encoding: utf-8

import ckan.plugins as plugins
import ckan.plugins.toolkit as toolkit
from ckan.types import Schema
from ckan.common import CKANConfig


class ExampleIConfigurerPlugin(plugins.SingletonPlugin):

    plugins.implements(plugins.IConfigurer)

    # IConfigurer

    def update_config(self, config: CKANConfig):
        # Add extension templates directory
        toolkit.add_template_directory(config, 'templates')

    def update_config_schema(self, schema: Schema):

        ignore_missing = toolkit.get_validator('ignore_missing')
        unicode_safe = toolkit.get_validator('unicode_safe')
        is_positive_integer = toolkit.get_validator('is_positive_integer')
```

(continues on next page)

(continued from previous page)

```
        schema.update({
            # This is an existing CKAN core configuration option, we are just
            # making it available to be editable at runtime
            'ckan.datasets_per_page': [ignore_missing, is_positive_integer],

            # This is a custom configuration option
            'ckanext.example_iconfigurer.test_conf': [ignore_missing,
                                                      unicode_safe],
        })

        return schema
```

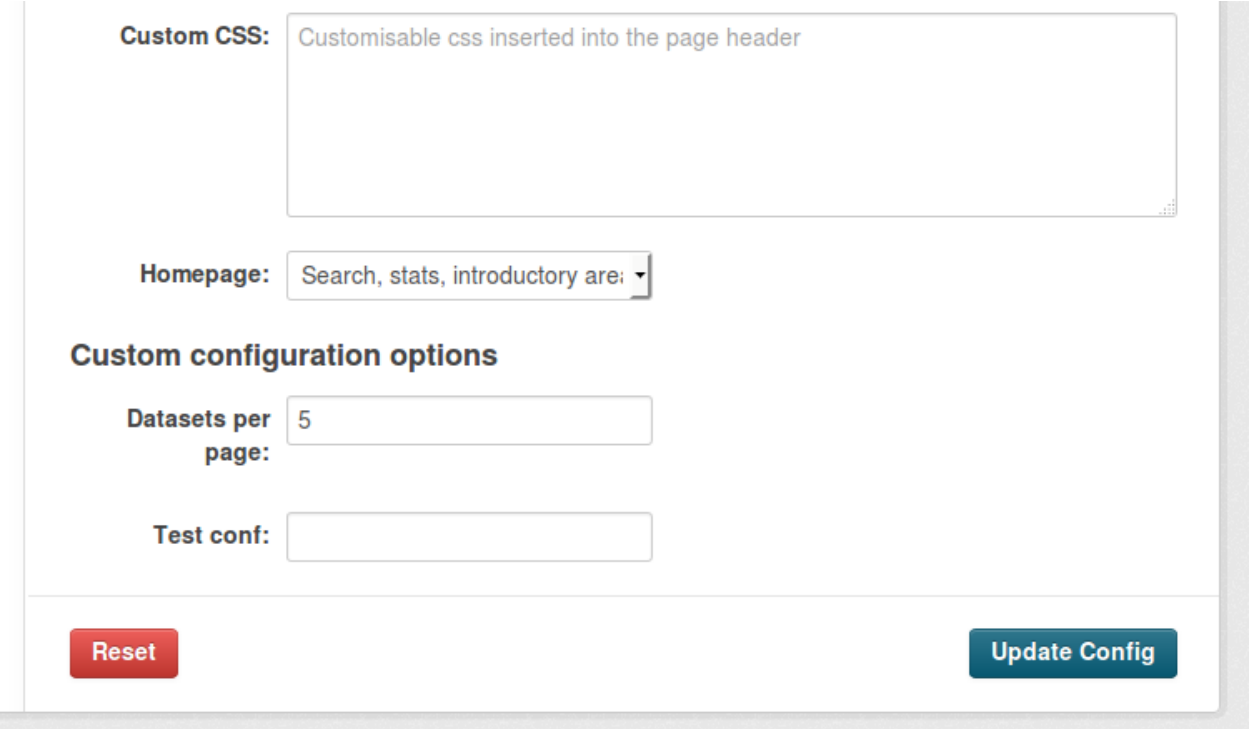
Now create a new file `config.html` file under `ckanext/yourextension/templates/admin/` with the following contents:

```
{% ckan_extends %}

{% import 'macros/form.html' as form %}

{% block admin_form %}

  {{ super() }}

  <h3>Custom configuration options </h3>

  {{ form.input('ckan.datasets_per_page', id='field-ckan.datasets_per_page', label=_(
↪'Datasets per page'), value=data['ckan.datasets_per_page'], error=errors['ckan.
↪datasets_per_page']) }}

  {{ form.input('ckanext.example_iconfigurer.test_conf', id='field-ckanext.example_
↪iconfigurer.test_conf', label=_('Test conf'), value=data['ckanext.example_iconfigurer.
↪test_conf'], error=errors['ckanext.example_iconfigurer.test_conf']) }}


{% endblock %}


{% block admin_form_help %}

  {{ super() }}

  <p><strong>Datasets per page:</strong> Number of datasets displayed in dataset␣
↪listings (eg search page).</p>

  <p><strong>Test conf:</strong> An example configuration option, set from an extension.
↪</p>

{% endblock %}
```

This template is extending the default core one. The first block adds two new fields for our configuration options below the existing ones. The second adds a helper text for them on the left hand column.

Restart the server and navigate to http://localhost:5000/ckan-admin/config. You should see the newfields at the bottom

of the form:



Updating the values on the form should update the configuration as before.

5.4 Testing extensions

CKAN extensions can have their own tests that are run using `pytest` in much the same way as running CKAN's own tests (see *Testing CKAN*).

Continuing with our *example_iauthfunctions extension*, first we need a CKAN config file to be used when running our tests. Create the file `ckanext-iauthfunctions/test.ini` with the following contents:

```
[app:main]
use = config:../ckan/test-core.ini
```

The `use` line declares that this config file inherits the settings from the config file used to run CKAN's own tests (`../ckan` should be the path to your CKAN source directory, relative to your `test.ini` file).

The `test.ini` file is a CKAN config file just like your /etc/ckan/default/ckan.ini file, and it can contain any *CKAN config file settings* that you want CKAN to use when running your tests, for example:

```
[app:main]
use = config:../ckan/test-core.ini
ckan.site_title = My Test CKAN Site
ckan.site_description = A test site for testing my CKAN extension
```

Next, make the directory that will contain our test modules:

```
mkdir ckanext-iauthfunctions/ckanext/iauthfunctions/tests/
```

Finally, create the file `ckanext-iauthfunctions/ckanext/iauthfunctions/tests/test_iauthfunctions.py` with the following contents:

```
# encoding: utf-8
'''Tests for the ckanext.example_iauthfunctions extension.

'''
import pytest

import ckan.logic as logic
import ckan.tests.factories as factories
import ckan.tests.helpers as helpers
from ckan.plugins.toolkit import NotAuthorized, ObjectNotFound


@pytest.mark.ckan_config('ckan.plugins',
                         'example_iauthfunctions_v6_parent_auth_functions')
@pytest.mark.usefixtures('clean_db', 'with_plugins')
class TestAuthV6(object):
    def test_resource_delete_editor(self):
        '''Normally organization admins can delete resources
        Our plugin prevents this by blocking delete organization.

        Ensure the delete button is not displayed (as only resource delete
        is checked for showing this)

        '''
        user = factories.User()
        owner_org = factories.Organization(users=[{
            'name': user['id'],
            'capacity': 'admin'
        }])
        dataset = factories.Dataset(owner_org=owner_org['id'])
        resource = factories.Resource(package_id=dataset['id'])
        with pytest.raises(logic.NotAuthorized) as e:
            logic.check_access('resource_delete', {'user': user['name']},
                               {'id': resource['id']})

        assert e.value.message == 'User %s not authorized to delete resource %s' % (
            user['name'], resource['id'])

    def test_resource_delete_sysadmin(self):
        '''Normally organization admins can delete resources
        Our plugin prevents this by blocking delete organization.

        Ensure the delete button is not displayed (as only resource delete
        is checked for showing this)

        '''
        user = factories.Sysadmin()
        owner_org = factories.Organization(users=[{
            'name': user['id'],
            'capacity': 'admin'
```

(continues on next page)

(continued from previous page)

```
        }])
        dataset = factories.Dataset(owner_org=owner_org['id'])
        resource = factories.Resource(package_id=dataset['id'])
        assert logic.check_access('resource_delete', {'user': user['name']},
                                  {'id': resource['id']})


@pytest.mark.ckan_config('ckan.plugins',
                         'example_iauthfunctions_v5_custom_config_setting')
@pytest.mark.ckan_config('ckan.iauthfunctions.users_can_create_groups', False)
@pytest.mark.usefixtures('clean_db', 'with_plugins')
class TestAuthV5(object):

    def test_sysadmin_can_create_group_when_config_is_false(self):
        sysadmin = factories.Sysadmin()
        context = {'ignore_auth': False, 'user': sysadmin['name']}
        helpers.call_action('group_create', context, name='test-group')

    def test_user_cannot_create_group_when_config_is_false(self):
        user = factories.User()
        context = {'ignore_auth': False, 'user': user['name']}
        with pytest.raises(NotAuthorized):
            helpers.call_action('group_create', context, name='test-group')

    def test_visitor_cannot_create_group_when_config_is_false(self):
        context = {'ignore_auth': False, 'user': None}
        with pytest.raises(NotAuthorized):
            helpers.call_action('group_create', context, name='test-group')


@pytest.mark.ckan_config('ckan.plugins',
                         'example_iauthfunctions_v5_custom_config_setting')
@pytest.mark.ckan_config('ckan.iauthfunctions.users_can_create_groups', True)
@pytest.mark.usefixtures('clean_db', 'with_plugins')
class TestAuthV5WithUserCreateGroup(object):

    def test_sysadmin_can_create_group_when_config_is_true(self):
        sysadmin = factories.Sysadmin()
        context = {'ignore_auth': False, 'user': sysadmin['name']}
        helpers.call_action('group_create', context, name='test-group')

    def test_user_can_create_group_when_config_is_true(self):
        user = factories.User()
        context = {'ignore_auth': False, 'user': user['name']}
        helpers.call_action('group_create', context, name='test-group')

    def test_visitor_cannot_create_group_when_config_is_true(self):
        context = {'ignore_auth': False, 'user': None}
        with pytest.raises(NotAuthorized):
            helpers.call_action('group_create', context, name='test-group')
```

(continues on next page)

(continued from previous page)

```
@pytest.fixture
def curators_group():
    '''This is a helper method for test methods to call when they want
    the 'curators' group to be created.
    '''
    sysadmin = factories.Sysadmin()

    # Create a user who will *not* be a member of the curators group.
    noncurator = factories.User()

    # Create a user who will be a member of the curators group.
    curator = factories.User()

    # Create the curators group, with the 'curator' user as a member.
    users = [{'name': curator['name'], 'capacity': 'member'}]
    context = {'ignore_auth': False, 'user': sysadmin['name']}
    group = helpers.call_action('group_create',
                                context,
                                name='curators',
                                users=users)

    return (noncurator, curator, group)


@pytest.mark.ckan_config('ckan.plugins', 'example_iauthfunctions_v4')
@pytest.mark.usefixtures('clean_db', 'with_plugins')
def test_group_create_with_no_curators_group():
    '''Test that group_create doesn't crash when there's no curators group.
    '''
    sysadmin = factories.Sysadmin()

    # Make sure there's no curators group.
    assert 'curators' not in helpers.call_action('group_list', {})

    # Make our sysadmin user create a group. CKAN should not crash.
    context = {'ignore_auth': False, 'user': sysadmin['name']}
    helpers.call_action('group_create', context, name='test-group')


@pytest.mark.ckan_config('ckan.plugins', 'example_iauthfunctions_v4')
@pytest.mark.usefixtures('clean_db', 'with_plugins')
def test_group_create_with_visitor(curators_group):
    '''A visitor (not logged in) should not be able to create a group.

    Note: this also tests that the group_create auth function doesn't
    crash when the user isn't logged in.
    '''
    context = {'ignore_auth': False, 'user': None}
    with pytest.raises(NotAuthorized):
        helpers.call_action('group_create',
                            context,
                            name='this_group_should_not_be_created')
```

(continues on next page)

(continued from previous page)

```
@pytest.mark.ckan_config('ckan.plugins', 'example_iauthfunctions_v4')
@pytest.mark.usefixtures('clean_db', 'with_plugins')
def test_group_create_with_non_curator(curators_group):
    '''A user who isn't a member of the curators group should not be able
    to create a group.
    '''
    noncurator, _, _ = curators_group
    context = {'ignore_auth': False, 'user': noncurator['name']}
    with pytest.raises(NotAuthorized):
        helpers.call_action('group_create',
                            context,
                            name='this_group_should_not_be_created')


@pytest.mark.ckan_config('ckan.plugins', 'example_iauthfunctions_v4')
@pytest.mark.usefixtures('clean_db', 'with_plugins')
def test_group_create_with_curator(curators_group):
    '''A member of the curators group should be able to create a group.
    '''
    _, curator, _ = curators_group
    name = 'my-new-group'
    context = {'ignore_auth': False, 'user': curator['name']}
    result = helpers.call_action('group_create', context, name=name)

    assert result['name'] == name
```

To run these extension tests, `cd` into the `ckanext-iauthfunctions` directory and run this command:

```
pytest --ckan-ini=test.ini ckanext/iauthfunctions/tests
```

Some notes on how these tests work:

- Pytest has lots of useful functions for testing, see the pytest documentation.
- We're calling `ckan.tests.call_action()` This is a convenience function that CKAN provides for its own tests.
- The CKAN core *Testing coding standards* can usefully be applied to writing tests for plugins.
- CKAN core provides:
  - *`ckan.tests.factories`* for creating test data
  - *`ckan.tests.helpers`* a collection of helper functions for use in tests
  - *`ckan.tests.pytest_ckan.fixtures`* for setting up the test environment

  which are also useful for testing extensions.
- You might also find it useful to read the Flask testing documentation (or Pylons testing documentation for plugins using legacy pylons controllers).
- Avoid importing the plugin modules directly into your test modules (e.g from example_iauthfunctions import plugin_v5_custom_config_setting). This causes the plugin to be registered and loaded before the entire test run,

so the plugin will be loaded for all tests. This can cause conflicts and test failures.

5.4.1 Using the test client

It is possible to make requests to the CKAN application from within your tests in order to test the actual responses returned by CKAN. To do so you need to import the `app` fixture:

```
def test_some_ckan_page(app):

  pass
```

The `app` fixture extends Flask's Test client, and can be used to perform GET and POST requests. A Werkzeug's `TestResponse` object (reference) will be returned:

```
from ckan.plugins.toolkit import url_for

def test_dataset_new_page(app):

  url = url_for("group.index")
  response = app.get(url)

  assert "Search groups" in response.body
```

By default, requests are not authenticated. If you want to make the request impersonating a user in particular, you can pass *an API Token* in the `headers` parameter:

```
from ckan.plugins.toolkit import url_for

def test_group_new_page(app):

    user = factories.UserWithToken()

    url = url_for("group.new")
    response = app.get(
      url,
      headers={"Authorization": user["token"]}
    )

    assert "Create a Group" in response.body

def test_submit_group_form_page(app):

    user = factories.UserWithToken()

    url = url_for("group.new")
    data = {
      "name": "test-group",
      "title": "Test Group",
      "description": "Some test group",
      "save": ""
    }
    response = app.post(
      url,
```

(continues on next page)

(continued from previous page)

```
    headers={"Authorization": user["token"]},
    data=data,
  )

  assert data["title"] in response.body
  assert call_action("group_show", id=data["name"])
```

Todo: Link to CKAN guidelines for *how* to write tests, once those guidelines have been written. Also add any more extension-specific testing details here.

5.5 Best practices for writing extensions

5.5.1 Follow CKAN's coding standards

See *Contributing guide*.

5.5.2 Use the plugins toolkit instead of importing CKAN

Try to limit your extension to interacting with CKAN only through CKAN's *plugin interfaces* and *plugins toolkit*. It's a good idea to keep your extension code separate from CKAN as much as possible, so that internal changes in CKAN from one release to the next don't break your extension.

5.5.3 Don't edit CKAN's database tables

An extension can create its own tables in the CKAN database, but it should *not* write to core CKAN tables directly, add columns to core tables, or use foreign keys against core tables.

5.5.4 Don't automatically modify the database structure

If your extension uses custom database tables then it needs to modify the database structure, for example to add the tables after its installation or to migrate them after an update. These modifications should not be performed automatically when the extension is loaded, since this can lead to dead-locks and other problems.

5.5.5 Use migrations when introducing new models

Any new model provided by extension must use migration script for creating and updating relevant tables. As well as core tables, extensions should provide *revisioned workflow* for reproducing correct state of DB. There are few convenient tools available in CKAN for this purpose:

- New migration script can be created via CLI interface:

  ```
  ckan generate migration -p PLUGIN_NAME -m 'MIGRATION MESSAGE'
  ```

  One should take care and use actual plugin's name, not extension name instead of *PLUGIN_NAME*. This may become important when an extension provides multiple plugins, which contain migration scripts. If those scripts should be applied independently(i.e., there is no sense in particular migrations, unless specific plugin is enabled),

`-p/--plugin` option gives you enough control. Otherwise, if extenson named *ckanext-ext* contains just single plugin *ext*, command for new migration will look like *ckan generate migration -p ext*.

Migration scripts are created under *EXTENSION_ROOT/ckanext/EXTENSION_NAME/migration/PLUGIN_NAME/versions*. Once created, migration script contains empty *upgrade* and *downgrade* function, which need to be updated according to desired changes. More details abailable in Alembic documentation.

- Apply migration script with:

```
ckan db upgrade -p PLUGIN_NAME
```

This command will check current state of DB and apply only required migrations, so it's idempotent.

- Revert changes introduced by plugin's migration scripts with:

```
ckan db downgrade -p PLUGIN_NAME
```

5.5.6 Declare models using shared metadata

New in version 2.10.

Use the `BaseModel` class from the plugins toolkit to implement SQLAlchemy declarative models in your extension. It is attached to the core metadata object, so it helps SQLAlchemy to resolve cascade relationships and control orphan removals. In addition, the `clean_db` test fixture will also handle these tables when cleaning the database.

Example:

```
from ckan.plugins import toolkit

class ExtModel(toolkit.BaseModel):
    __tablename__ = "ext_model"
    id = Column(String(50), primary_key=True)
    ...
```

In previous versions of CKAN, you can link to the `ckan.model.meta.metadata` object directly in your own class:

```
import ckan.model as model
from sqlalchemy.ext.declarative import declarative_base

Base = declarative_base(metadata=model.meta.metadata)

class ExtModel(Base):
    __tablename__ = "ext_model"
    id = Column(String(50), primary_key=True)
    ...
```

5.5.7 Implement each plugin class in a separate Python module

This keeps CKAN's plugin loading order simple, see *ckan.plugins*.

5.5.8 Avoid name clashes

Many of the names you pick for your identifiers and files must be unique in relation to the names used by core CKAN and other extensions. To avoid conflicts you should prefix any public name that your extension introduces with the name of your extension. For example:

- The names of *configuration settings* introduced by your extension should have the form `my_extension.my_config_setting`.
- The names of *templates and template snippets* introduced by your extension should begin with the name of your extension:

  ```
  snippets/my_extension_useful_snippet.html
  ```

  If you have add a lot of templates you can also put them into a separate folder named after your extension instead.
- The names of *template helper functions* introduced by your extension should begin with the name of your extension. For example:

  ```
  def get_helpers(self):
      "Register the most_popular_groups() function above as a template
      helper function.

      '''
      # Template helper function names should begin with the name of the
      # extension they belong to, to avoid clashing with functions from
      # other extensions.
      return {'example_theme_most_popular_groups': most_popular_groups}
  ```

- The names of *JavaScript modules* introduced by your extension should begin with the name of your extension. For example `assets/example_theme_popover.js`:

  ```
  // Enable JavaScript's strict mode. Strict mode catches some common
  // programming errors and throws exceptions, prevents some unsafe actions from
  // being taken, and disables some confusing and bad JavaScript features.
  "use strict";

  ckan.module('example_theme_popover', function ($) {
    return {
      initialize: function () {
        console.log("I've been initialized for element: ", this.el);
      }
    };
  });
  ```

- The names of *API action functions* introduced by your extension should begin with the name of your extension. For example `my_extension_foobarize_everything`.
- The names of *background job queues* introduced by your extension should begin with the name of your extension. For example `my_extension:super-special-job-queue`.

In some situations, a resource may even be shared between multiple CKAN *instances*, which requires an even higher degree of uniqueness for the corresponding names. In that case, you should also prefix your identifiers with the CKAN site ID, which is available via

```
try:
    # CKAN 2.7 and later
    from ckan.common import config
except ImportError:
    # CKAN 2.6 and earlier
    from pylons import config

site_id = config[u'ckan.site_id']
```

Currently this only affects the *Redis database*:

- All *keys in the Redis database* created by your extension should be prefixed with both the CKAN site ID and your extension's name.

5.5.9 Internationalize user-visible strings

All user-visible strings should be internationalized, see *String internationalization*.

5.5.10 Add third party libraries to requirements.txt

If your extension requires third party libraries, rather than adding them to `setup.py`, they should be added to `requirements.txt`, which can be installed with:

```
pip install -r requirements.txt
```

To prevent accidental breakage of your extension through backwards-incompatible behaviour of newer versions of your dependencies, their versions should be pinned, such as:

```
requests==2.7.0
```

On the flip side, be mindful that this could also create version conflicts with requirements of considerably newer or older extensions.

5.5.11 Implementing CSRF protection

CKAN 2.10 introduces CSRF protection for all the frontend forms. Extensions are currently excluded from the CSRF protection to give time to update them, but CSRF protection will be enforced in the future.

To add CSRF protection to your extensions add the following helper call to your form templates:

```
<form class="dataset-form form-horizontal" method="post" enctype="multipart/form-data">
  {{ h.csrf_input() }}
```

If your extension needs to support older CKAN versions, use the following:

```
<form class="dataset-form form-horizontal" method="post" enctype="multipart/form-data">
  {{ h.csrf_input() if 'csrf_input' in h }}
```

Forms that are submitted via JavaScript modules also need to submit the CSRF token, here's an example of how to append it to an existing form:

```
// Get the csrf value from the page meta tag
var csrf_value = $('meta[name=_csrf_token]').attr('content')
// Create the hidden input
var hidden_csrf_input = $('<input name="_csrf_token" type="hidden" value="'+csrf_value+'
↪">')
// Insert the hidden input at the beginning of the form
hidden_csrf_input.prependTo(form)
```

API calls performed from JavaScript modules from the UI (which use cookie-based authentication) should also include the token, in this case in the `X-CSRFToken` header. CKAN Modules using the builtin client) to perform API calls will have the header added automatically. If you are performing API calls directly from a UI module you will need to add the header yourself.

5.6 Customizing dataset and resource metadata fields using IDatasetForm

Storing additional metadata for a dataset beyond the default metadata in CKAN is a common use case. CKAN provides a simple way to do this by allowing you to store arbitrary key/value pairs against a dataset when creating or updating the dataset. These appear under the "Additional Information" section on the web interface and in 'extras' field of the JSON when accessed via the API.

Default extras can only take strings for their keys and values, no validation is applied to the inputs and you cannot make them mandatory or restrict the possible values to a defined list. By using CKAN's IDatasetForm plugin interface, a CKAN plugin can add custom, first-class metadata fields to CKAN datasets, and can do custom validation of these fields.

See also:

In this tutorial we are assuming that you have read the *Writing extensions tutorial*.

You may also want to check the [ckanext-scheming](https://github.com/ckan/ckanext-scheming) extension, as it will allow metadata schema configuration using a YAML or JSON schema description, replete with custom validation and template snippets for editing and display.

5.6.1 CKAN schemas and validation

When a dataset is created, updated or viewed, the parameters passed to CKAN (e.g. via the web form when creating or updating a dataset, or posted to an API end point) are validated against a schema. For each parameter, the schema will contain a corresponding list of functions that will be run against the value of the parameter. Generally these functions are used to validate the value (and raise an error if the value fails validation) or convert the value to a different value.

For example, the schemas can allow optional values by using the `ignore_missing()` validator or check that a dataset exists using `package_id_exists()`. A list of available validators can be found at the *Validator functions reference*. You can also define your own *Custom validators*.

We will be customizing these schemas to add our additional fields. The `IDatasetForm` interface allows us to override the schemas for creation, updating and displaying of datasets.

create_package_schema()	Return the schema for validating new dataset dicts.
update_package_schema()	Return the schema for validating updated dataset dicts.
show_package_schema()	Return a schema to validate datasets before they're shown to the user.
is_fallback()	Return `True` if this plugin is the fallback plugin.
package_types()	Return an iterable of dataset (package) types that this plugin handles.

CKAN allows you to have multiple IDatasetForm plugins, each handling different dataset types. So you could customize the CKAN web front end, for different types of datasets. In this tutorial we will be defining our plugin as the fallback plugin. This plugin is used if no other IDatasetForm plugin is found that handles that dataset type.

The IDatasetForm also has other additional functions that allow you to provide a custom template to be rendered for the CKAN frontend, but we will not be using them for this tutorial.

Adding custom fields to datasets

Create a new plugin named `ckanext-extrafields` and create a class named `ExampleIDatasetFormPlugins` inside `ckanext-extrafields/ckanext/extrafields/plugin.py` that implements the `IDatasetForm` interface and inherits from `SingletonPlugin` and `DefaultDatasetForm`.

```
# encoding: utf-8
from __future__ import annotations

from ckan.types import Schema
import ckan.plugins as p
import ckan.plugins.toolkit as tk


class ExampleIDatasetFormPlugin(tk.DefaultDatasetForm, p.SingletonPlugin):
    p.implements(p.IDatasetForm)
```

Updating the CKAN schema

The *create_package_schema()* function is used whenever a new dataset is created, we'll want update the default schema and insert our custom field here. We will fetch the default schema defined in `default_create_package_schema()` by running *create_package_schema()*'s super function and update it.

```
    def create_package_schema(self) -> Schema:
        # let's grab the default schema in our plugin
        schema: Schema = super(
            ExampleIDatasetFormPlugin, self).create_package_schema()
        # our custom field
        schema.update({
            'custom_text': [tk.get_validator('ignore_missing'),
                            tk.get_converter('convert_to_extras')]
        })
        return schema
```

The CKAN schema is a dictionary where the key is the name of the field and the value is a list of validators and converters. Here we have a validator to tell CKAN to not raise a validation error if the value is missing and a converter

to convert the value to and save as an extra. We will want to change the *update_package_schema()* function with the same update code.

```
    def update_package_schema(self) -> Schema:
        schema: Schema = super(
            ExampleIDatasetFormPlugin, self).update_package_schema()
        # our custom field
        schema.update({
            'custom_text': [tk.get_validator('ignore_missing'),
                            tk.get_converter('convert_to_extras')]
        })
        return schema
```

The *show_package_schema()* is used when the *package_show()* action is called, we want the default_show_package_schema to be updated to include our custom field. This time, instead of converting to an extras field, we want our field to be converted *from* an extras field. So we want to use the *convert_from_extras()* converter.

```
    def show_package_schema(self) -> Schema:
        schema: Schema = super(
            ExampleIDatasetFormPlugin, self).show_package_schema()
        schema.update({
            'custom_text': [tk.get_converter('convert_from_extras'),
                            tk.get_validator('ignore_missing')]
        })
        return schema
```

Dataset types

The *package_types()* function defines a list of dataset types that this plugin handles. Each dataset has a field containing its type. Plugins can register to handle specific types of dataset and ignore others. Since our plugin is not for any specific type of dataset and we want our plugin to be the default handler, we update the plugin code to contain the following:

```
        schema: Schema = super(
            ExampleIDatasetFormPlugin, self).show_package_schema()
        schema.update({
            'custom_text': [tk.get_converter('convert_from_extras'),
                            tk.get_validator('ignore_missing')]
        })
        return schema

    def is_fallback(self):
        # Return True to register this plugin as the default handler for
        # package types not handled by any other IDatasetForm plugin.
        return True

    def package_types(self) -> list[str]:
        # This plugin doesn't handle any special package types, it just
        # registers itself as the default (above).
        return []
```

Updating templates

In order for our new field to be visible on the CKAN front-end, we need to update the templates. Add an additional line to make the plugin implement the IConfigurer interface

```
class ExampleIDatasetFormPlugin(tk.DefaultDatasetForm, p.SingletonPlugin):
    p.implements(p.IDatasetForm)
    p.implements(p.IConfigurer)
```

This interface allows to implement a function `update_config()` that allows us to update the CKAN config, in our case we want to add an additional location for CKAN to look for templates. Add the following code to your plugin.

```
    def update_config(self, config: CKANConfig):
        # Add this plugin's templates dir to CKAN's extra_template_paths, so
        # that CKAN will use this plugin's custom templates.
        tk.add_template_directory(config, 'templates')
```

You will also need to add a directory under your extension directory to store the templates. Create a directory called `ckanext-extrafields/ckanext/extrafields/templates/` and the subdirectories `ckanext-extrafields/ckanext/extrafields/templates/package/snippets/`.

We need to override a few templates in order to get our custom field rendered. A common option when using a custom schema is to remove the default custom field handling that allows arbitrary key/value pairs. Create a template file in our templates directory called `package/snippets/package_metadata_fields.html` containing

```
{% ckan_extends %}

{# You could remove 'free extras' from the package form like this, but we keep them for␣
↪this example's tests.
  {% block custom_fields %}
  {% endblock %}
#}
```

This overrides the custom_fields block with an empty block so the default CKAN custom fields form does not render.

New in version 2.3: Starting from CKAN 2.3 you can combine free extras with custom fields handled with `convert_to_extras` and `convert_from_extras`. On prior versions you'll always need to remove the free extras handling.

Next add a template in our template directory called `package/snippets/package_basic_fields.html` containing

```
{% ckan_extends %}

{% block package_basic_fields_custom %}
  {{ form.input('custom_text', label=_('Custom Text'), id='field-custom_text',␣
↪placeholder=_('custom text'), value=data.custom_text, error=errors.custom_text,␣
↪classes=['control-medium']) }}
{% endblock %}
```

This adds our custom_text field to the editing form. Finally we want to display our custom_text field on the dataset page. Add another file called `package/snippets/additional_info.html` containing

```
{% ckan_extends %}

{% block extras %}
  {% if pkg_dict.custom_text %}
    <tr>
      <th scope="row" class="dataset-label">{{ _("Custom Text") }}</th>
      <td class="dataset-details">{{ pkg_dict.custom_text }}</td>
    </tr>
  {% endif %}
{% endblock %}
```

This template overrides the default extras rendering on the dataset page and replaces it to just display our custom field.

You're done! Make sure you have your plugin installed and setup as in the *extension/tutorial*. Then run a development server and you should now have an additional field called "Custom Text" when displaying and adding/editing a dataset.

Cleaning up the code

Before we continue further, we can clean up the `create_package_schema()` and `update_package_schema()`. There is a bit of duplication that we could remove. Replace the two functions with:

```
    def _modify_package_schema(self, schema: Schema) -> Schema:
        schema.update({
            'custom_text': [tk.get_validator('ignore_missing'),
                            tk.get_converter('convert_to_extras')]
        })
        return schema

    def create_package_schema(self):
        schema: Schema = super(
            ExampleIDatasetFormPlugin, self).create_package_schema()
        schema = self._modify_package_schema(schema)
        return schema

    def update_package_schema(self):
        schema: Schema = super(
            ExampleIDatasetFormPlugin, self).update_package_schema()
        schema = self._modify_package_schema(schema)
        return schema
```

5.6.2 Custom validators

You may define custom validators in your extensions and you can share validators between extensions by registering them with the `IValidators` interface.

Any of the following objects may be used as validators as part of a custom dataset, group or organization schema. CKAN's validation code will check for and attempt to use them in this order:

1. a function taking a single parameter: `validator(value)`
2. a function taking four parameters: `validator(key, flattened_data, errors, context)`

3. a function taking two parameters `validator(value, context)`

Note: Object constructors(including str, int, etc.) and some built-in functions cannot be used as validators. In order to use them, create a thin wrapper which passes values into these callables and converts expected exceptions into `ckan.plugins.toolkit.Invalid`.

Example:

```
def int_validator(value):
    try:
        return int(value)
    except ValueError:
        raise Invalid(f"Invalid literal for integer: {value}")
```

validator(value)

The simplest form of validator is a callable taking a single parameter. For example:

```
from ckan.plugins.toolkit import Invalid

def starts_with_b(value):
    if not value.startswith('b'):
        raise Invalid("Doesn't start with b")
    return value
```

The `starts_with_b` validator causes a validation error for values not starting with 'b'. On a web form this validation error would appear next to the field to which the validator was applied.

`return value` must be used by validators when accepting data or the value will be converted to None. This form is useful for converting data as well, because the return value will replace the field value passed:

```
def embiggen(value):
    return value.upper()
```

The `embiggen` validator will convert values passed to all-uppercase.

validator(value, context)

Validators that need access to the database or information about the user may be written as a callable taking two parameters. `context['session']` is the sqlalchemy session object and `context['user']` is the username of the logged-in user:

```
from ckan.plugins.toolkit import Invalid

def fred_only(value, context):
    if value and context['user'] != 'fred':
        raise Invalid('only fred may set this value')
    return value
```

Otherwise this is the same as the single-parameter form above.

validator(key, flattened_data, errors, context)

Validators that need to access or update multiple fields may be written as a callable taking four parameters.

All fields and errors in a `flattened` form are passed to the validator. The validator must fetch values from `flattened_data` and may replace values in `flattened_data`. The return value from this function is ignored.

`key` is the flattened key for the field to which this validator was applied. For example `('notes',)` for the dataset notes field or `('resources', 0, 'url')` for the url of the first resource of the dataset. These flattened keys are the same in both the `flattened_data` and `errors` dicts passed.

`errors` contains lists of validation errors for each field.

`context` is the same value passed to the two-parameter form above.

Note that this form can be tricky to use because some of the values in `flattened_data` will have had validators applied but other fields won't. You may add this type of validator to the special schema fields `'__before'` or `'__after'` to have them run before or after all the other validation takes place to avoid the problem of working with partially-validated data.

The validator has to be registered. Example:

```
from ckan import plugins


class ExampleIValidatorsPlugin(plugins.SingletonPlugin):
    plugins.implements(plugins.IValidators)

    def get_validators(self) -> dict[str, Validator]:
        return {
            u'equals_fortytwo': equals_fortytwo,
            u'negate': negate,
            u'unicode_only': unicode_please,
        }
```

5.6.3 Tag vocabularies

If you need to add a custom field where the input options are restricted to a provided list of options, you can use tag vocabularies *Tag Vocabularies*. We will need to create our vocabulary first. By calling `vocabulary_create()`. Add a function to your plugin.py above your plugin class.

```
def create_country_codes():
    user = tk.get_action('get_site_user')({'ignore_auth': True}, {})
    context: Context = {'user': user['name']}
    try:
        data = {'id': 'country_codes'}
        tk.get_action('vocabulary_show')(context, data)
    except tk.ObjectNotFound:
        data = {'name': 'country_codes'}
        vocab = tk.get_action('vocabulary_create')(context, data)
        for tag in (u'uk', u'ie', u'de', u'fr', u'es'):
            data: dict[str, Any] = {'name': tag, 'vocabulary_id': vocab['id']}
            tk.get_action('tag_create')(context, data)
```

This code block is taken from the `example_idatsetform plugin`. `create_country_codes` tries to fetch the vocabulary country_codes using *vocabulary_show()*. If it is not found it will create it and iterate over the list of countries 'uk', 'ie', 'de', 'fr', 'es'. For each of these a vocabulary tag is created using *tag_create()*, belonging to the vocabulary `country_code`.

Although we have only defined five tags here, additional tags can be created at any point by a sysadmin user by calling *tag_create()* using the API or action functions. Add a second function below `create_country_codes`

```
def country_codes():
    create_country_codes()
    try:
        tag_list = tk.get_action('tag_list')
        country_codes = tag_list({}, {'vocabulary_id': 'country_codes'})
        return country_codes
    except tk.ObjectNotFound:
        return None
```

country_codes will call `create_country_codes` so that the `country_codes` vocabulary is created if it does not exist. Then it calls *tag_list()* to return all of our vocabulary tags together. Now we have a way of retrieving our tag vocabularies and creating them if they do not exist. We just need our plugin to call this code.

Adding tags to the schema

Update `_modify_package_schema()` and *show_package_schema()*

```
    def _modify_package_schema(self, schema: Schema):
        schema.update({
            'custom_text': [tk.get_validator('ignore_missing'),
                            tk.get_converter('convert_to_extras')]
        })
        schema.update({
            'country_code': [
                tk.get_validator('ignore_missing'),
                cast(ValidatorFactory,
                     tk.get_converter('convert_to_tags'))('country_codes'),
            ]
        })
        return schema

    def show_package_schema(self) -> Schema:
        schema: Any = super(
            ExampleIDatasetFormPlugin, self).show_package_schema()
        schema.update({
            'custom_text': [tk.get_converter('convert_from_extras'),
                            tk.get_validator('ignore_missing')]
        })

        schema['tags']['__extras'].append(tk.get_converter('free_tags_only'))
        schema.update({
            'country_code': [
                cast(ValidatorFactory,
                     tk.get_converter('convert_from_tags'))('country_codes'),
```

(continues on next page)

(continued from previous page)

```
                tk.get_validator('ignore_missing')]
        })
        return schema
```

We are adding our tag to our plugin's schema. A converter is required to convert the field in to our tag in a similar way to how we converted our field to extras earlier. In *show_package_schema()* we convert from the tag back again but we have an additional line with another converter containing *free_tags_only()*. We include this line so that vocab tags are not shown mixed with normal free tags.

Adding tags to templates

Add an additional plugin.implements line to to your plugin to implement the *ITemplateHelpers*, we will need to add a *get_helpers()* function defined for this interface.

```
    p.implements(p.ITemplateHelpers)

    def get_helpers(self):
        return {'country_codes': country_codes}
```

Our intention here is to tie our country_code fetching/creation to when they are used in the templates. Add the code below to `package/snippets/package_metadata_fields.html`

```
#}

{% block package_metadata_fields %}

  <div class="control-group">
    <label class="form-label" for="field-country_code">{{ _("Country Code") }}</label>
    <div class="controls">
      <select id="field-country_code" name="country_code" data-module="autocomplete">
        {% for country_code in h.country_codes()  %}
          <option value="{{ country_code }}" {% if country_code in data.get('country_code
↪', []) %}selected="selected"{% endif %}>{{ country_code }}</option>
        {% endfor %}
      </select>
    </div>
  </div>

  {{ super() }}

{% endblock %}
```

This adds our country code to our template, here we are using the additional helper country_codes that we defined in our get_helpers function in our plugin.

5.6.4 Adding custom fields to resources

In order to customize the fields in a resource the schema for resources needs to be modified in a similar way to the datasets. The resource schema is nested in the dataset dict as package['resources']. We modify this dict in a similar way to the dataset schema. Change `_modify_package_schema` to the following.

```
    def _modify_package_schema(self, schema: Schema):
        # Add our custom country_code metadata field to the schema.

        schema.update({
                'country_code': [
                    tk.get_validator('ignore_missing'),
                    cast(
                        ValidatorFactory,
                        tk.get_converter('convert_to_tags'))('country_codes')]
                })
        # Add our custom_test metadata field to the schema, this one will use
        # convert_to_extras instead of convert_to_tags.
        schema.update({
                'custom_text': [tk.get_validator('ignore_missing'),
                    tk.get_converter('convert_to_extras')]
                })
        # Add our custom_resource_text metadata field to the schema
        cast(Schema, schema['resources']).update({
                'custom_resource_text' : [ tk.get_validator('ignore_missing') ]
                })
        return schema
```

Update *show_package_schema()* similarly

```
    def show_package_schema(self) -> Schema:
        schema: Schema = super(
            ExampleIDatasetFormPlugin, self).show_package_schema()

        # Don't show vocab tags mixed in with normal 'free' tags
        # (e.g. on dataset pages, or on the search page)
        _extras = cast("list[Validator]",
                       cast(Schema, schema['tags'])['__extras'])
        _extras.append(tk.get_converter('free_tags_only'))

        # Add our custom country_code metadata field to the schema.
        schema.update({
            'country_code': [
                cast(
                    ValidatorFactory,
                    tk.get_converter('convert_from_tags'))('country_codes'),
                tk.get_validator('ignore_missing')]
            })

        # Add our custom_text field to the dataset schema.
        schema.update({
            'custom_text': [tk.get_converter('convert_from_extras'),
                tk.get_validator('ignore_missing')]
            })
```

(continues on next page)

(continued from previous page)

```
        cast(Schema, schema['resources']).update({
                'custom_resource_text' : [ tk.get_validator('ignore_missing') ]
            })
        return schema
```

Add the code below to `package/snippets/resource_form.html`

```
{% ckan_extends %}

{% block basic_fields_url %}
{{ super() }}

  {{ form.input('custom_resource_text', label=_('Custom Text'), id='field-custom_
↪resource_text', placeholder=_('custom resource text'), value=data.custom_resource_text,
↪ error=errors.custom_resource_text, classes=['control-medium']) }}
{% endblock %}
```

This adds our custom_resource_text to the editing form of the resources.

Save and reload your development server CKAN will take any additional keys from the resource schema and save them the its extras field. The templates will automatically check this field and display them in the resource_read page.

5.6.5 Sorting by custom fields on the dataset search page

Now that we've added our custom field, we can customize the CKAN web front end search page to sort datasets by our custom field. Add a new file called `ckanext-extrafields/ckanext/extrafields/templates/package/search.html` containing:

```
{% ckan_extends %}

{% block form %}
  {% set facets = {
    'fields': fields_grouped,
    'search': search_facets,
    'titles': facet_titles,
    'translated_fields': translated_fields,
    'remove_field': remove_field }
  %}
  {% set sorting = [
    (_('Relevance'), 'score desc, metadata_modified desc'),
    (_('Name Ascending'), 'title_string asc'),
    (_('Name Descending'), 'title_string desc'),
    (_('Last Modified'), 'metadata_modified desc'),
    (_('Custom Field Ascending'), 'custom_text asc'),
    (_('Custom Field Descending'), 'custom_text desc')
    ]
  %}
  {% snippet 'snippets/search_form.html', type='dataset', query=q, sorting=sorting,␣
↪sorting_selected=sort_by_selected, count=page.item_count, facets=facets, show_
↪empty=request.args, error=query_error %}
{% endblock %}
```

This overrides the search ordering drop down code block, the code is the same as the default dataset search block but we are adding two additional lines that define the display name of that search ordering (e.g. Custom Field Ascending) and the SOLR sort ordering (e.g. custom_text asc). If you reload your development server you should be able to see these two additional sorting options on the dataset search page.

The SOLR sort ordering can define arbitrary functions for custom sorting, but this is beyond the scope of this tutorial for further details see http://wiki.apache.org/solr/CommonQueryParameters#sort and http://wiki.apache.org/solr/FunctionQuery

You can find the complete source for this tutorial at https://github.com/ckan/ckan/tree/master/ckanext/example_idatasetform

5.7 Plugin interfaces reference

`ckan.plugins` contains a few core classes and functions for plugins to use:

ckan.plugins

class ckan.plugins.**SingletonPlugin**(**args*, ***kwargs*)

Base class for plugins which are singletons (ie most of them)

One singleton instance of this class will be created when the plugin is loaded. Subsequent calls to the class constructor will always return the same singleton instance.

class ckan.plugins.**Plugin**(**args*, ***kwargs*)

Base class for plugins which require multiple instances.

Unless you need multiple instances of your plugin object you should probably use SingletonPlugin.

ckan.plugins.**implements**(*interface*, *inherit=None*, *namespace=None*, *service=False*)

Can be used in the class definition of *Plugin* subclasses to declare the extension points that are implemented by this interface class.

ckan.plugins.interfaces

A collection of interfaces that CKAN plugins can implement to customize and extend CKAN.

class ckan.plugins.interfaces.**Interface**

Base class for custom interfaces.

Marker base class for extension point interfaces. This class is not intended to be instantiated. Instead, the declaration of subclasses of Interface are recorded, and these classes are used to define extension points.

classmethod provided_by(*instance:* SingletonPlugin) → bool

Check that the object is an instance of the class that implements the interface.

classmethod implemented_by(*other: Type['SingletonPlugin']*) → bool

Check whether the class implements the current interface.

class ckan.plugins.interfaces.**IMiddleware**

Hook into the CKAN middleware stack

Note that methods on this interface will be called two times, one for the Pylons stack and one for the Flask stack (eventually there will be only the Flask stack).

make_middleware(*app: CKANApp*, *config: CKANConfig*) → CKANApp

Return an app configured with this middleware

When called on the Flask stack, this method will get the actual Flask app so plugins wanting to install Flask extensions can do it like this:

```
import ckan.plugins as p
from flask_mail import Mail

class MyPlugin(p.SingletonPlugin):

    p.implements(p.IMiddleware)

    def make_middleware(app, config):

        mail = Mail(app)

        return app
```

make_error_log_middleware(*app: CKANFlask*, *config: CKANConfig*) → CKANFlask

Return an app configured with this error log middleware

Note that both on the Flask and Pylons middleware stacks, this method will receive a wrapped WSGI app, not the actual Flask or Pylons app.

class ckan.plugins.interfaces.**IAuthFunctions**

Override CKAN's authorization functions, or add new auth functions.

get_auth_functions() → dict[str, AuthFunction]

Return the authorization functions provided by this plugin.

Return a dictionary mapping authorization function names (strings) to functions. For example:

```
{'user_create': my_custom_user_create_function,
 'group_create': my_custom_group_create}
```

When a user tries to carry out an action via the CKAN API or web interface and CKAN or a CKAN plugin calls `check_access('some_action')` as a result, an authorization function named `'some_action'` will be searched for in the authorization functions registered by plugins and in CKAN's core authorization functions (found in `ckan/logic/auth/`).

For example when action function `'package_create'` is called, a `'package_create'` authorization function is searched for.

If an extension registers an authorization function with the same name as one of CKAN's default authorization functions (as with `'user_create'` and `'group_create'` above), the extension's function will override the default one.

Each authorization function should take two parameters `context` and `data_dict`, and should return a dictionary `{'success': True}` to authorize the action or `{'success': False}` to deny it, for example:

```
def user_create(context, data_dict=None):
    if (some condition):
        return {'success': True}
    else:
        return {'success': False, 'msg': 'Not allowed to register'}
```

The context object will contain a `model` that can be used to query the database, a `user` containing the name of the user doing the request (or their IP if it is an anonymous web request) and an `auth_user_obj` containing the actual model.User object (or None if it is an anonymous request).

See `ckan/logic/auth/` for more examples.

Note that by default, all auth functions provided by extensions are assumed to require a validated user or API key, otherwise a `ckan.logic.NotAuthorized`: exception will be raised. This check will be performed *before* calling the actual auth function. If you want to allow anonymous access to one of your actions, its auth function must be decorated with the `auth_allow_anonymous_access` decorator, available in the plugins toolkit.

For example:

```
import ckan.plugins as p

@p.toolkit.auth_allow_anonymous_access
def my_search_action(context, data_dict):
    # Note that you can still return {'success': False} if for some
    # reason access is denied.

def my_create_action(context, data_dict):
    # Unless there is a logged in user or a valid API key provided
    # NotAuthorized will be raised before reaching this function.
```

By decorating a registered auth function with the `ckan.plugins.toolkit.chained_auth_function` decorator you can create a chain of auth checks that are completed when auth is requested. This chain starts with the last chained auth function to be registered and ends with the original auth function (or a non-chained plugin override version). Chained auth functions must accept an extra parameter, specifically the next auth function in the chain, for example:

```
auth_function(next_auth, context, data_dict).
```

The chained auth function may call the next_auth function, optionally passing different values, handling exceptions, returning different values and/or raising different exceptions to the caller.

class `ckan.plugins.interfaces.`**`IDomainObjectModification`**

Receives notification of new, changed and deleted datasets.

`notify`(*entity: Any*, *operation: str*) → None

Send a notification on entity modification.

Parameters

- **`entity`** – instance of module.Package.
- **`operation`** – 'new', 'changed' or 'deleted'.

`notify_after_commit`(*entity: Any*, *operation: Any*) → None

** DEPRECATED **

Supposed to send a notification after entity modification, but it doesn't work.

Parameters

- **`entity`** – instance of module.Package.
- **`operation`** – 'new', 'changed' or 'deleted'.

class ckan.plugins.interfaces.**IFeed**

For extending the default Atom feeds

get_feed_class() → PFeedFactory

Allows plugins to provide a custom class to generate feed items.

Returns
feed class

Return type
type

The feed item generator's constructor is called as follows:

```
feed_class(
    feed_title,        # Mandatory
    feed_link,         # Mandatory
    feed_description,  # Mandatory
    language,          # Optional, always set to 'en'
    author_name,       # Optional
    author_link,       # Optional
    feed_guid,         # Optional
    feed_url,          # Optional
    previous_page,     # Optional, url of previous page of feed
    next_page,         # Optional, url of next page of feed
    first_page,        # Optional, url of first page of feed
    last_page,         # Optional, url of last page of feed
)
```

get_item_additional_fields(*dataset_dict: dict[str, Any]*) → dict[str, Any]

Allows plugins to set additional fields on a feed item.

Parameters
dataset_dict (*dictionary*) – the dataset metadata

Returns
the fields to set

Return type
dictionary

class ckan.plugins.interfaces.**IGroupController**

Hook into the Group view. These methods will usually be called just before committing or returning the respective object i.e. when all validation, synchronization and authorization setup are complete.

read(*entity: model.Group*) → None

Called after IGroupController.before_view inside group_read.

create(*entity: model.Group*) → None

Called after group has been created inside group_create.

edit(*entity: model.Group*) → None

Called after group has been updated inside group_update.

delete(*entity: model.Group*) → None

Called before commit inside group_delete.

before_view(*data_dict: dict[str, Any]*) → dict[str, Any]

Extensions will receive this before the group gets displayed. The dictionary passed will be the one that gets sent to the template.

class ckan.plugins.interfaces.**IOrganizationController**

Hook into the Organization view. These methods will usually be called just before committing or returning the respective object i.e. when all validation, synchronization and authorization setup are complete.

read(*entity: model.Group*) → None

Called after IOrganizationController.before_view inside organization_read.

create(*entity: model.Group*) → None

Called after organization had been created inside organization_create.

edit(*entity: model.Group*) → None

Called after organization had been updated inside organization_update.

delete(*entity: model.Group*) → None

Called before commit inside organization_delete.

before_view(*data_dict: dict[str, Any]*) → dict[str, Any]

Extensions will receive this before the organization gets displayed. The dictionary passed will be the one that gets sent to the template.

class ckan.plugins.interfaces.**IPackageController**

Hook into the dataset view.

read(*entity: model.Package*) → None

Called after IPackageController.before_dataset_view inside package_show.

create(*entity: model.Package*) → None

Called after the dataset had been created inside package_create.

edit(*entity: model.Package*) → None

Called after the dataset had been updated inside package_update.

delete(*entity: model.Package*) → None

Called before commit inside package_delete.

after_dataset_create(*context: Context*, *pkg_dict: dict[str, Any]*) → None

Extensions will receive the validated data dict after the dataset has been created (Note that the create method will return a dataset domain object, which may not include all fields). Also the newly created dataset id will be added to the dict.

after_dataset_update(*context: Context*, *pkg_dict: dict[str, Any]*) → None

Extensions will receive the validated data dict after the dataset has been updated.

Note that bulk dataset update actions (*bulk_update_private*, *bulk_update_public*) will bypass this callback. See `ckan.plugins.toolkit.chained_action` to wrap those actions if required.

after_dataset_delete(*context: Context*, *pkg_dict: dict[str, Any]*) → None

Extensions will receive the data dict (typically containing just the dataset id) after the dataset has been deleted.

Note that the *bulk_update_delete* action will bypass this callback. See `ckan.plugins.toolkit.chained_action` to wrap that action if required.

after_dataset_show(*context: Context*, *pkg_dict: dict[str, Any]*) → None

Extensions will receive the validated data dict after the dataset is ready for display.

before_dataset_search(*search_params: dict[str, Any]*) → dict[str, Any]

Extensions will receive a dictionary with the query parameters, and should return a modified (or not) version of it.

search_params will include an *extras* dictionary with all values from fields starting with *ext_*, so extensions can receive user input from specific fields.

after_dataset_search(*search_results: dict[str, Any]*, *search_params: dict[str, Any]*) → dict[str, Any]

Extensions will receive the search results, as well as the search parameters, and should return a modified (or not) object with the same structure:

```
{'count': '', 'results': '', 'search_facets': ''}
```

Note that count and facets may need to be adjusted if the extension changed the results for some reason.

search_params will include an *extras* dictionary with all values from fields starting with *ext_*, so extensions can receive user input from specific fields.

before_dataset_index(*pkg_dict: dict[str, Any]*) → dict[str, Any]

Extensions will receive what will be given to Solr for indexing. This is essentially a flattened dict (except for multi-valued fields such as tags) of all the terms sent to the indexer. The extension can modify this by returning an altered version.

before_dataset_view(*pkg_dict: dict[str, Any]*) → dict[str, Any]

Extensions will receive this before the dataset gets displayed. The dictionary passed will be the one that gets sent to the template.

class ckan.plugins.interfaces.**IPluginObserver**

Hook into the plugin loading mechanism itself

before_load(*plugin:* SingletonPlugin) → None

Called before a plugin is loaded. This method is passed the plugin class.

after_load(*service: Any*) → None

Called after a plugin has been loaded. This method is passed the instantiated service object.

before_unload(*plugin:* SingletonPlugin) → None

Called before a plugin is loaded. This method is passed the plugin class.

after_unload(*service: Any*) → None

Called after a plugin has been unloaded. This method is passed the instantiated service object.

class ckan.plugins.interfaces.**IConfigurable**

Hook called during the startup of CKAN

See also *IConfigurer*.

configure(*config: CKANConfig*) → None

Called during CKAN's initialization.

This function allows plugins to initialize themselves during CKAN's initialization. It is called after most of the environment (e.g. the database) is already set up.

Note that this function is not only called during the initialization of the main CKAN process but also during the execution of paster commands and background jobs, since these run in separate processes and are therefore initialized independently.

Parameters
config (ckan.common.CKANConfig) – dict-like configuration object

class ckan.plugins.interfaces.**IConfigDeclaration**

Register additional configuration options.

While it's not necessary, declared config options can be printed out using CLI or additionally verified in code. This makes the task of adding new configuration, removing obsolete config options, checking the sanity of config options much simpler for extension consumers.

declare_config_options(*declaration: Declaration*, *key: Key*)

Register extra config options.

Example:

```
from ckan.config.declaration import Declaration, Key

def declare_config_options(
    self, declaration: Declaration, key: Key):

    declaration.annotate("MyExt config section")
    group = key.ckanext.my_ext.feature
    declaration.declare(group.enabled, "no").set_description(
        "Enables feature"
    )
    declaration.declare(group.mode, "simple").set_description(
        "Execution mode"
    )
```

Run `ckan config declaration my_ext --include-docs` and get the following config suggestion:

```
## MyExt config section #####################
# Enables feature
ckanext.my_ext.feature.enabled = no
# Execution mode
ckanext.my_ext.feature.mode = simple
```

See *declare configuration* guide for details.

Parameters

- **declaration** (ckan.config.declaration.Declaration) – object containing all the config declarations
- **key** (ckan.config.declaration.Key) – object for generic option access.

class ckan.plugins.interfaces.**IConfigurer**

Configure the CKAN environment via the `config` object

See also `IConfigurable`.

update_config(*config: CKANConfig*) → None

Called by load_environment at the earliest point that config is available to plugins. The config should be updated in place.

Parameters
config – config object

update_config_schema(*schema: Schema*) → Schema

Return a schema with the runtime-editable config options.

CKAN will use the returned schema to decide which configuration options can be edited during runtime (using `ckan.logic.action.update.config_option_update()`) and to validate them before storing them.

Defaults to `ckan.logic.schema.default_update_configuration_schema()`, which will be passed to all extensions implementing this method, which can add or remove runtime-editable config options to it.

Parameters
schema (*dictionary*) – a dictionary mapping runtime-editable configuration option keys to lists of validator and converter functions to be applied to those keys

Returns
a dictionary mapping runtime-editable configuration option keys to lists of validator and converter functions to be applied to those keys

Return type
dictionary

class ckan.plugins.interfaces.**IActions**

Allow adding of actions to the logic layer.

get_actions() → dict[str, Callable[[Context, dict[str, Any]], Any]]

Should return a dict, the keys being the name of the logic function and the values being the functions themselves.

By decorating a function with the `ckan.logic.side_effect_free` decorator, the associated action will be made available to a GET request (as well as the usual POST request) through the Action API.

By decorating a function with `ckan.plugins.toolkit.chained_action`, the action will 'intercept' calls to an existing action function. This allows a plugin to modify the behaviour of an existing action function. Chained actions must be defined as `action_function(original_action, context, data_dict)`, where the function's name matches the original action function it intercepts, the first parameter is the action function it intercepts (in the next plugin or in core ckan). The chained action may call the original_action function, optionally passing different values, handling exceptions, returning different values and/or raising different exceptions to the caller. When multiple plugins chain to an action, the first plugin declaring is called first, and if it chooses to call the original_action, then the chained action in the next plugin to be declared next is called, and so on.

class ckan.plugins.interfaces.**IResourceUrlChange**

Receives notification of changed URL on a resource.

notify(*resource: model.Resource*) → None

Called when a resource url has changed.

:param resource, instance of model.Resource

class ckan.plugins.interfaces.**IDatasetForm**

Customize CKAN's dataset (package) schemas and forms.

By implementing this interface plugins can customise CKAN's dataset schema, for example to add new custom fields to datasets.

Multiple IDatasetForm plugins can be used at once, each plugin associating itself with different dataset types using the `package_types()` and `is_fallback()` methods below, and then providing different schemas and templates for different types of dataset. When a dataset view action is invoked, the `type` field of the dataset will determine which IDatasetForm plugin (if any) gets delegated to.

When implementing IDatasetForm, you can inherit from `ckan.plugins.toolkit.DefaultDatasetForm`, which provides default implementations for each of the methods defined in this interface.

See `ckanext/example_idatasetform` for an example plugin.

package_types() → Sequence[str]

Return an iterable of dataset (package) types that this plugin handles.

If a request involving a dataset of one of the returned types is made, then this plugin instance will be delegated to.

There cannot be two IDatasetForm plugins that return the same dataset type, if this happens then CKAN will raise an exception at startup.

Return type
iterable of strings

is_fallback() → bool

Return `True` if this plugin is the fallback plugin.

When no IDatasetForm plugin's `package_types()` match the `type` of the dataset being processed, the fallback plugin is delegated to instead.

There cannot be more than one IDatasetForm plugin whose `is_fallback()` method returns `True`, if this happens CKAN will raise an exception at startup.

If no IDatasetForm plugin's `is_fallback()` method returns `True`, CKAN will use `DefaultDatasetForm` as the fallback.

Return type
bool

create_package_schema() → Schema

Return the schema for validating new dataset dicts.

CKAN will use the returned schema to validate and convert data coming from users (via the dataset form or API) when creating new datasets, before entering that data into the database.

If it inherits from `ckan.plugins.toolkit.DefaultDatasetForm`, a plugin can call `DefaultDatasetForm`'s `create_package_schema()` method to get the default schema and then modify and return it.

CKAN's `convert_to_tags()` or `convert_to_extras()` functions can be used to convert custom fields into dataset tags or extras for storing in the database.

See `ckanext/example_idatasetform` for examples.

Returns
a dictionary mapping dataset dict keys to lists of validator and converter functions to be applied to those keys

Return type
dictionary

update_package_schema() → Schema

Return the schema for validating updated dataset dicts.

CKAN will use the returned schema to validate and convert data coming from users (via the dataset form or API) when updating datasets, before entering that data into the database.

If it inherits from `ckan.plugins.toolkit.DefaultDatasetForm`, a plugin can call `DefaultDatasetForm`'s `update_package_schema()` method to get the default schema and then modify and return it.

CKAN's `convert_to_tags()` or `convert_to_extras()` functions can be used to convert custom fields into dataset tags or extras for storing in the database.

See `ckanext/example_idatasetform` for examples.

Returns
a dictionary mapping dataset dict keys to lists of validator and converter functions to be applied to those keys

Return type
dictionary

show_package_schema() → Schema

Return a schema to validate datasets before they're shown to the user.

CKAN will use the returned schema to validate and convert data coming from the database before it is returned to the user via the API or passed to a template for rendering.

If it inherits from `ckan.plugins.toolkit.DefaultDatasetForm`, a plugin can call `DefaultDatasetForm`'s `show_package_schema()` method to get the default schema and then modify and return it.

If you have used `convert_to_tags()` or `convert_to_extras()` in your `create_package_schema()` and `update_package_schema()` then you should use `convert_from_tags()` or `convert_from_extras()` in your `show_package_schema()` to convert the tags or extras in the database back into your custom dataset fields.

See `ckanext/example_idatasetform` for examples.

Returns
a dictionary mapping dataset dict keys to lists of validator and converter functions to be applied to those keys

Return type
dictionary

setup_template_variables(*context: Context*, *data_dict: dict[str, Any]*) → None

Add variables to the template context for use in dataset templates.

This function is called before a dataset template is rendered. If you have custom dataset templates that require some additional variables, you can add them to the template context `ckan.plugins.toolkit.c` here and they will be available in your templates. See `ckanext/example_idatasetform` for an example.

new_template(*package_type: str*) → str

Return the path to the template for the new dataset page.

The path should be relative to the plugin's templates dir, e.g. `'package/new.html'`.

Return type
string

read_template(*package_type: str*) → str

Return the path to the template for the dataset read page.

The path should be relative to the plugin's templates dir, e.g. `'package/read.html'`.

If the user requests the dataset in a format other than HTML, then CKAN will try to render a template file with the same path as returned by this function, but a different filename extension, e.g. `'package/read.rdf'`. If your extension (or another one) does not provide this version of the template file, the user will get a 404 error.

Return type
string

edit_template(*package_type: str*) → str

Return the path to the template for the dataset edit page.

The path should be relative to the plugin's templates dir, e.g. `'package/edit.html'`.

Return type
string

search_template(*package_type: str*) → str

Return the path to the template for use in the dataset search page.

This template is used to render each dataset that is listed in the search results on the dataset search page.

The path should be relative to the plugin's templates dir, e.g. `'package/search.html'`.

Return type
string

history_template(*package_type: str*) → str

> **Warning:** This template is removed. The function exists for compatibility. It now returns None.

resource_template(*package_type: str*) → str

Return the path to the template for the resource read page.

The path should be relative to the plugin's templates dir, e.g. `'package/resource_read.html'`.

Return type
string

package_form(*package_type: str*) → str

Return the path to the template for the dataset form.

The path should be relative to the plugin's templates dir, e.g. `'package/form.html'`.

Return type
string

resource_form(*package_type: str*) → str

Return the path to the template for the resource form.

The path should be relative to the plugin's templates dir, e.g. `'package/snippets/resource_form.html'`

Return type
string

validate(*context: Context*, *data_dict: DataDict*, *schema: Schema*, *action: str*) → tuple[dict[str, Any], dict[str, Any]] | None

Customize validation of datasets.

When this method is implemented it is used to perform all validation for these datasets. The default implementation calls and returns the result from `ckan.plugins.toolkit.navl_validate`.

This is an adavanced interface. Most changes to validation should be accomplished by customizing the schemas returned from `show_package_schema()`, `create_package_schema()` and `update_package_schema()`. If you need to have a different schema depending on the user or value of any field stored in the dataset, or if you wish to use a different method for validation, then this method may be used.

Parameters

- **context** (*dictionary*) – extra information about the request
- **data_dict** (*dictionary*) – the dataset to be validated
- **schema** (*dictionary*) – a schema, typically from `show_package_schema()`, `create_package_schema()` or `update_package_schema()`
- **action** (*string*) – `'package_show'`, `'package_create'` or `'package_update'`

Returns
(data_dict, errors) where data_dict is the possibly-modified dataset and errors is a dictionary with keys matching data_dict and lists-of-string-error-messages as values

Return type
(dictionary, dictionary)

prepare_dataset_blueprint(*package_type: str*, *blueprint: Blueprint*) → Blueprint

Update or replace dataset blueprint for given package type.

Internally CKAN registers blueprint for every custom dataset type. Before default routes added to this blueprint and it registered inside application this method is called. It can be used either for registration of the view function under new path or under existing path(like */new*), in which case this new function will be used instead of default one.

Note, this blueprint has prefix */{package_type}*.

Return type
flask.Blueprint

prepare_resource_blueprint(*package_type: str*, *blueprint: Blueprint*) → Blueprint

Update or replace resource blueprint for given package type.

Internally CKAN registers separate resource blueprint for every custom dataset type. Before default routes added to this blueprint and it registered inside application this method is called. It can be used either for registration of the view function under new path or under existing path(like */new*), in which case this new function will be used instead of default one.

Note, this blueprint has prefix */{package_type}/<id>/resource*.

Return type
flask.Blueprint

class ckan.plugins.interfaces.**IValidators**

Add extra validators to be returned by `ckan.plugins.toolkit.get_validator()`.

get_validators() → dict[str, Validator]

Return the validator functions provided by this plugin.

Return a dictionary mapping validator names (strings) to validator functions. For example:

```
{'valid_shoe_size': shoe_size_validator,
 'valid_hair_color': hair_color_validator}
```

These validator functions would then be available when a plugin calls `ckan.plugins.toolkit.get_validator()`.

class ckan.plugins.interfaces.**IResourceView**

Add custom view renderings for different resource types.

info() → dict[str, Any]

Returns a dictionary with configuration options for the view.

The available keys are:

Parameters

- **name** – name of the view type. This should match the name of the actual plugin (eg `image_view` or `datatables_view`).
- **title** – title of the view type. Will be displayed on the frontend. This should be translatable (ie wrapped with `toolkit._('Title')`).
- **default_title** – default title that will be used if the view is created automatically (optional, defaults to 'View').
- **default_description** – default description that will be used if the view is created automatically (optional, defaults to '').
- **icon** – icon for the view type. Should be one of the Font Awesome types without the *fa fa-* prefix eg. *compass* (optional, defaults to 'picture').
- **always_available** – the view type should be always available when creating new views regardless of the format of the resource (optional, defaults to False).
- **iframed** – the view template should be iframed before rendering. You generally want this option to be True unless the view styles and JavaScript don't clash with the main site theme (optional, defaults to True).
- **preview_enabled** – the preview button should appear on the edit view form. Some view types have their previews integrated with the form (optional, defaults to False).
- **full_page_edit** – the edit form should take the full page width of the page (optional, defaults to False).
- **schema** – schema to validate extra configuration fields for the view (optional). Schemas are defined as a dictionary, with the keys being the field name and the values a list of validator functions that will get applied to the field. For instance:

```
{
    'offset': [ignore_empty, natural_number_validator],
    'limit': [ignore_empty, natural_number_validator],
}
```

Example configuration object:

```
{'name': 'image_view',
 'title': toolkit._('Image'),
 'schema': {
    'image_url': [ignore_empty, unicode]
 },
 'icon': 'image',
 'always_available': True,
 'iframed': False,
 }
```

Returns

a dictionary with the view type configuration

Return type
dict

`can_view`(*data_dict: dict[str, Any]*) → bool

Returns whether the plugin can render a particular resource.

The `data_dict` contains the following keys:

Parameters

- **resource** – dict of the resource fields
- **package** – dict of the full parent dataset

Returns
True if the plugin can render a particular resource, False otherwise

Return type
bool

`setup_template_variables`(*context: Context*, *data_dict: dict[str, Any]*) → dict[str, Any]

Adds variables to be passed to the template being rendered.

This should return a new dict instead of updating the input `data_dict`.

The `data_dict` contains the following keys:

Parameters

- **resource_view** – dict of the resource view being rendered
- **resource** – dict of the parent resource fields
- **package** – dict of the full parent dataset

Returns
a dictionary with the extra variables to pass

Return type
dict

`view_template`(*context: Context*, *data_dict: dict[str, Any]*) → str

Returns a string representing the location of the template to be rendered when the view is displayed

The path will be relative to the template directory you registered using the `add_template_directory()` on the *update_config* method, for instance `views/my_view.html`.

Parameters

- **resource_view** – dict of the resource view being rendered
- **resource** – dict of the parent resource fields
- **package** – dict of the full parent dataset

Returns
the location of the view template.

Return type
string

`form_template`(*context: Context*, *data_dict: dict[str, Any]*) → str

Returns a string representing the location of the template to be rendered when the edit view form is displayed

The path will be relative to the template directory you registered using the `add_template_directory()` on the *update_config* method, for instance `views/my_view_form.html`.

Parameters

- **resource_view** – dict of the resource view being rendered
- **resource** – dict of the parent resource fields
- **package** – dict of the full parent dataset

Returns
the location of the edit view form template.

Return type
string

class ckan.plugins.interfaces.**IResourceController**

Hook into the resource view.

before_resource_create(*context: Context*, *resource: dict[str, Any]*) → None

Extensions will receive this before a resource is created.

Parameters

- **context** (*dictionary*) – The context object of the current request, this includes for example access to the `model` and the `user`.
- **resource** (*dictionary*) – An object representing the resource to be added to the dataset (the one that is about to be created).

after_resource_create(*context: Context*, *resource: dict[str, Any]*) → None

Extensions will receive this after a resource is created.

Parameters

- **context** (*dictionary*) – The context object of the current request, this includes for example access to the `model` and the `user`.
- **resource** (*dictionary*) – An object representing the latest resource added to the dataset (the one that was just created). A key in the resource dictionary worth mentioning is `url_type` which is set to `upload` when the resource file is uploaded instead of linked.

before_resource_update(*context: Context*, *current: dict[str, Any]*, *resource: dict[str, Any]*) → None

Extensions will receive this before a resource is updated.

Parameters

- **context** (*dictionary*) – The context object of the current request, this includes for example access to the `model` and the `user`.
- **current** (*dictionary*) – The current resource which is about to be updated
- **resource** (*dictionary*) – An object representing the updated resource which will replace the `current` one.

after_resource_update(*context: Context*, *resource: dict[str, Any]*) → None

Extensions will receive this after a resource is updated.

Parameters

- **context** (*dictionary*) – The context object of the current request, this includes for example access to the `model` and the `user`.
- **resource** (*dictionary*) – An object representing the updated resource in the dataset (the one that was just updated). As with `after_resource_create`, a noteworthy key in the

resource dictionary `url_type` which is set to `upload` when the resource file is uploaded instead of linked.

Note that the datastore will bypass this callback when updating the `datastore_active` flag on a resource that has been added to the datastore.

before_resource_delete(*context: Context*, *resource: dict[str, Any]*, *resources: list[dict[str, Any]]*) → None

Extensions will receive this before a resource is deleted.

Parameters

- **context** (*dictionary*) – The context object of the current request, this includes for example access to the `model` and the `user`.
- **resource** (*dictionary*) – An object representing the resource that is about to be deleted. This is a dictionary with one key: `id` which holds the id `string` of the resource that should be deleted.
- **resources** (*list*) – The list of resources from which the resource will be deleted (including the resource to be deleted if it existed in the dataset).

after_resource_delete(*context: Context*, *resources: list[dict[str, Any]]*) → None

Extensions will receive this after a resource is deleted.

Parameters

- **context** (*dictionary*) – The context object of the current request, this includes for example access to the `model` and the `user`.
- **resources** – A list of objects representing the remaining resources after a resource has been removed.

before_resource_show(*resource_dict: dict[str, Any]*) → dict[str, Any]

Extensions will receive the validated data dict before the resource is ready for display.

Be aware that this method is not only called for UI display, but also in other methods, like when a resource is deleted, because package_show is used to get access to the resources in a dataset.

class ckan.plugins.interfaces.**IGroupForm**

Allows customisation of the group form and its underlying schema.

The behaviour of the plugin is determined by 5 method hooks:

- group_form(self)
- form_to_db_schema(self)
- db_to_form_schema(self)
- setup_template_variables(self, context, data_dict)

Furthermore, there can be many implementations of this plugin registered at once. With each instance associating itself with 0 or more group type strings. When a group form action is invoked, the group type determines which of the registered plugins to delegate to. Each implementation must implement these methods which are used to determine the group-type -> plugin mapping:

- is_fallback(self)
- group_types(self)
- group_controller(self)

Implementations might want to consider mixing in ckan.lib.plugins.DefaultGroupForm which provides default behaviours for the 5 method hooks.

is_fallback() → bool

Returns true if this provides the fallback behaviour, when no other plugin instance matches a group's type.

There must be exactly one fallback view defined, any attempt to register more than one will throw an exception at startup. If there's no fallback registered at startup the ckan.lib.plugins.DefaultGroupForm used as the fallback.

group_types() → Iterable[str]

Returns an iterable of group type strings.

If a request involving a group of one of those types is made, then this plugin instance will be delegated to.

There must only be one plugin registered to each group type. Any attempts to register more than one plugin instance to a given group type will raise an exception at startup.

group_controller() → str

Returns the name of the group view

The group view is the view, that is used to handle requests of the group type(s) of this plugin.

If this method is not provided, the default group view is used (*group*).

new_template(*group_type: str*) → str

Returns a string representing the location of the template to be rendered for the 'new' page. Uses the default_group_type configuration option to determine which plugin to use the template from.

index_template(*group_type: str*) → str

Returns a string representing the location of the template to be rendered for the index page. Uses the default_group_type configuration option to determine which plugin to use the template from.

read_template(*group_type: str*) → str

Returns a string representing the location of the template to be rendered for the read page

history_template(*group_type: str*) → str

Returns a string representing the location of the template to be rendered for the history page

edit_template(*group_type: str*) → str

Returns a string representing the location of the template to be rendered for the edit page

group_form(*group_type: str*) → str

Returns a string representing the location of the template to be rendered. e.g. `group/new_group_form.html`.

form_to_db_schema() → Schema

Returns the schema for mapping group data from a form to a format suitable for the database.

db_to_form_schema() → Schema

Returns the schema for mapping group data from the database into a format suitable for the form (optional)

setup_template_variables(*context: Context*, *data_dict: dict[str, Any]*) → None

Add variables to c just prior to the template being rendered.

validate(*context: Context*, *data_dict: DataDict*, *schema: Schema*, *action: str*) → tuple[dict[str, Any], dict[str, Any]] | None

Customize validation of groups.

When this method is implemented it is used to perform all validation for these groups. The default implementation calls and returns the result from `ckan.plugins.toolkit.navl_validate`.

This is an adavanced interface. Most changes to validation should be accomplished by customizing the schemas returned from `form_to_db_schema()` and `db_to_form_schema()` If you need to have a different schema depending on the user or value of any field stored in the group, or if you wish to use a different method for validation, then this method may be used.

Parameters

- **context** (*dictionary*) – extra information about the request
- **data_dict** (*dictionary*) – the group to be validated
- **schema** (*dictionary*) – a schema, typically from `form_to_db_schema()`, or `db_to_form_schema()`
- **action** (*string*) – `'group_show'`, `'group_create'`, `'group_update'`, `'organization_show'`, `'organization_create'` or `'organization_update'`

Returns
(data_dict, errors) where data_dict is the possibly-modified group and errors is a dictionary with keys matching data_dict and lists-of-string-error-messages as values

Return type
(dictionary, dictionary)

prepare_group_blueprint(*group_type: str*, *blueprint: Blueprint*) → Blueprint

Update or replace group blueprint for given group type.

Internally CKAN registers separate blueprint for every custom group type. Before default routes added to this blueprint and it registered inside application this method is called. It can be used either for registration of the view function under new path or under existing path(like */new*), in which case this new function will be used instead of default one.

Note, this blueprint has prefix */{group_type}*.

Return type
flask.Blueprint

class ckan.plugins.interfaces.**ITagController**

Hook into the Tag view. These will usually be called just before committing or returning the respective object, i.e. when all validation, synchronization and authorization setup are complete.

before_view(*tag_dict: dict[str, Any]*) → dict[str, Any]

Extensions will receive this before the tag gets displayed. The dictionary passed will be the one that gets sent to the template.

class ckan.plugins.interfaces.**ITemplateHelpers**

Add custom template helper functions.

By implementing this plugin interface plugins can provide their own template helper functions, which custom templates can then access via the `h` variable.

See `ckanext/example_itemplatehelpers` for an example plugin.

get_helpers() → dict[str, Callable[[...], Any]]

Return a dict mapping names to helper functions.

The keys of the dict should be the names with which the helper functions will be made available to templates, and the values should be the functions themselves. For example, a dict like: `{'example_helper': example_helper}` allows templates to access the `example_helper` function via `h.example_helper()`.

Function names should start with the name of the extension providing the function, to prevent name clashes between extensions.

By decorating a registered helper function with the `ckan.plugins.toolkit.chained_helper` decorator you can create a chain of helpers that are called in a sequence. This chain starts with the first chained helper to be registered and ends with the original helper (or a non-chained plugin override version). Chained helpers must accept an extra parameter, specifically the next helper in the chain, for example:

```
helper(next_helper, *args, **kwargs).
```

The chained helper function may call the next_helper function, optionally passing different values, handling exceptions, returning different values and/or raising different exceptions to the caller.

class `ckan.plugins.interfaces.IFacets`

Customize the search facets shown on search pages.

By implementing this interface plugins can customize the search facets that are displayed for filtering search results on the dataset search page, organization pages and group pages.

The `facets_dict` passed to each of the functions below is an `OrderedDict` in which the keys are CKAN's internal names for the facets and the values are the titles that will be shown for the facets in the web interface. The order of the keys in the dict determine the order that facets appear in on the page. For example:

```
{'groups': _('Groups'),
 'tags': _('Tags'),
 'res_format': _('Formats'),
 'license': _('License')}
```

To preserve ordering, make sure to add new facets to the existing dict rather than updating it, ie do this:

```
facets_dict['groups'] = p.toolkit._('Publisher')
facets_dict['secondary_publisher'] = p.toolkit._('Secondary Publisher')
```

rather than this:

```
facets_dict.update({
   'groups': p.toolkit._('Publisher'),
   'secondary_publisher': p.toolkit._('Secondary Publisher'),
})
```

Dataset searches can be faceted on any field in the dataset schema that it makes sense to facet on. This means any dataset field that is in CKAN's Solr search index, basically any field that you see returned by *package_show()*.

If there are multiple `IFacets` plugins active at once, each plugin will be called (in the order that they're listed in the CKAN config file) and they will each be able to modify the facets dict in turn.

dataset_facets(*facets_dict: OrderedDict[str, Any]*, *package_type: str*) → OrderedDict[str, Any]

Modify and return the `facets_dict` for the dataset search page.

The `package_type` is the type of dataset that these facets apply to. Plugins can provide different search facets for different types of dataset. See *IDatasetForm*.

Parameters

- **facets_dict** (*OrderedDict*) – the search facets as currently specified
- **package_type** (*string*) – the dataset type that these facets apply to

Returns
the updated `facets_dict`

Return type
OrderedDict

group_facets(*facets_dict: OrderedDict[str, Any]*, *group_type: str*, *package_type: str | None*) → OrderedDict[str, Any]

Modify and return the `facets_dict` for a group's page.

The `package_type` is the type of dataset that these facets apply to. Plugins can provide different search facets for different types of dataset. See *IDatasetForm*.

The `group_type` is the type of group that these facets apply to. Plugins can provide different search facets for different types of group. See *IGroupForm*.

Parameters

- **facets_dict** (*OrderedDict*) – the search facets as currently specified
- **group_type** (*string*) – the group type that these facets apply to
- **package_type** (*string*) – the dataset type that these facets apply to

Returns
the updated `facets_dict`

Return type
OrderedDict

organization_facets(*facets_dict: OrderedDict[str, Any]*, *organization_type: str*, *package_type: str | None*) → OrderedDict[str, Any]

Modify and return the `facets_dict` for an organization's page.

The `package_type` is the type of dataset that these facets apply to. Plugins can provide different search facets for different types of dataset. See *IDatasetForm*.

The `organization_type` is the type of organization that these facets apply to. Plugins can provide different search facets for different types of organization. See *IGroupForm*.

Parameters

- **facets_dict** (*OrderedDict*) – the search facets as currently specified
- **organization_type** (*string*) – the organization type that these facets apply to
- **package_type** (*string*) – the dataset type that these facets apply to

Returns
the updated `facets_dict`

Return type
OrderedDict

class ckan.plugins.interfaces.**IAuthenticator**

Allows custom authentication methods to be integrated into CKAN.

All interface methods except for the `abort()` one support returning a Flask response object. This can be used for instance to issue redirects or set cookies in the response. If a response object is returned there will be no further processing of the current request and that response will be returned. This can be used by plugins to:

- Issue a redirect:

```
def identify(self):

    return toolkit.redirect_to('myplugin.custom_endpoint')
```

- Set or clear cookies (or headers):

```
from Flask import make_response

def identify(self)::

    response = make_response(toolkit.render('my_page.html'))
    response.set_cookie(cookie_name, expires=0)

    return response
```

identify() → Response | None

Called to identify the user.

If the user is identified then it should set:

- g.user: The name of the user
- g.userobj: The actual user object

Alternatively, plugins can return a response object in order to prevent the default CKAN authorization flow. See the *IAuthenticator* documentation for more details.

login() → Response | None

Called before the login starts (that is before asking the user for user name and a password in the default authentication).

Plugins can return a response object to prevent the default CKAN authorization flow. See the *IAuthenticator* documentation for more details.

logout() → Response | None

Called before the logout starts (that is before clicking the logout button in the default authentication).

Plugins can return a response object to prevent the default CKAN authorization flow. See the *IAuthenticator* documentation for more details.

abort(*status_code: int*, *detail: str*, *headers: dict[str, Any] | None*, *comment: str | None*) → tuple[int, str, dict[str, Any] | None, str | None]

Called on abort. This allows aborts due to authorization issues to be overridden

authenticate(*identity: Mapping[str, Any]*) → User | None

Called before the authentication starts (that is after clicking the login button)

Plugins should return a user object if the authentication was successful, or `None` otherwise.

class ckan.plugins.interfaces.**ITranslation**

Allows extensions to provide their own translation strings.

i18n_directory() → str

Change the directory of the .mo translation files

i18n_locales() → list[str]

Change the list of locales that this plugin handles

i18n_domain() → str

Change the gettext domain handled by this plugin

class ckan.plugins.interfaces.**IUploader**

Extensions implementing this interface can provide custom uploaders to upload resources and group images.

get_uploader(*upload_to: str*, *old_filename: str | None*) → PUploader | None

Return an uploader object to upload general files that must implement the following methods:

`__init__(upload_to, old_filename=None)`

Set up the uploader.

Parameters

- **upload_to** (*string*) – name of the subdirectory within the storage directory to upload the file
- **old_filename** (*string*) – name of an existing image asset, so the extension can replace it if necessary

`update_data_dict(data_dict, url_field, file_field, clear_field)`

Allow the data_dict to be manipulated before it reaches any validators.

Parameters

- **data_dict** (*dictionary*) – data_dict to be updated
- **url_field** (*string*) – name of the field where the upload is going to be
- **file_field** (*string*) – name of the key where the FieldStorage is kept (i.e the field where the file data actually is).
- **clear_field** (*string*) – name of a boolean field which requests the upload to be deleted.

`upload(max_size)`

Perform the actual upload.

Parameters
max_size (*int*) – upload size can be limited by this value in MBs.

get_resource_uploader(*resource: dict[str, Any]*) → PResourceUploader | None

Return an uploader object used to upload resource files that must implement the following methods:

`__init__(resource)`

Set up the resource uploader.

Parameters
resource (*dictionary*) – resource dict

Optionally, this method can set the following two attributes on the class instance so they are set in the resource object:

- filesize (int): Uploaded file filesize.
- mimetype (str): Uploaded file mimetype.

`upload(id, max_size)`

Perform the actual upload.

Parameters

- **id** (*string*) – resource id, can be used to create filepath
- **max_size** (*int*) – upload size can be limited by this value in MBs.

`get_path(id)`

Required by the `resource_download` action to determine the path to the file.

Parameters
id (*string*) – resource id

class ckan.plugins.interfaces.**IBlueprint**

Register an extension as a Flask Blueprint.

get_blueprint() → list[Blueprint] | Blueprint

Return either a single Flask Blueprint object or a list of Flask Blueprint objects to be registered by the app.

class ckan.plugins.interfaces.**IPermissionLabels**

Extensions implementing this interface can override the permission labels applied to datasets to precisely control which datasets are visible to each user.

Implementations might want to consider mixing in `ckan.lib.plugins.DefaultPermissionLabels` which provides default behaviours for these methods.

See `ckanext/example_ipermissionlabels` for an example plugin.

get_dataset_labels(*dataset_obj: model.Package*) → list[str]

Return a list of unicode strings to be stored in the search index as the permission lables for a dataset dict.

Parameters
dataset_obj (*Package model object*) – dataset details

Returns
permission labels

Return type
list of unicode strings

get_user_dataset_labels(*user_obj: 'model.User' | None*) → list[str]

Return the permission labels that give a user permission to view a dataset. If any of the labels returned from this method match any of the labels returned from *get_dataset_labels()* then this user is permitted to view that dataset.

Parameters
user_obj (*User model object or None*) – user details

Returns
permission labels

Return type
list of unicode strings

class ckan.plugins.interfaces.**IForkObserver**

Observe forks of the CKAN process.

before_fork() → None

Called shortly before the CKAN process is forked.

class ckan.plugins.interfaces.**IApiToken**

Extend functionality of API Tokens.

This interface is unstable and new methods may be introduced in future. Always use *inherit=True* when implementing it.

Example:

```
p.implements(p.IApiToken, inherit=True)
```

create_api_token_schema(*schema: Schema*) → Schema

Return the schema for validating new API tokens.

Parameters

schema (*dict*) – a dictionary mapping api_token dict keys to lists of validator and converter functions to be applied to those keys

Returns

a dictionary mapping api_token dict keys to lists of validator and converter functions to be applied to those keys

Return type

dict

decode_api_token(*encoded: str*, ***kwargs: Any*) → dict[str, Any] | None

Make an attempt to decode API Token provided in request.

Decode token if it possible and return dictionary with mandatory *jti* key(token id for DB lookup) and optional additional items, which will be used further in *preprocess_api_token*.

Parameters

- **encoded** (*str*) – API Token provided in request
- **kwargs** (*dict*) – any additional parameters that can be added in future or by plugins. Current implementation won't pass any additional fields, but plugins may use this feature, passing JWT *aud* or *iss* claims, for example

Returns

dictionary with all the decoded fields or None

Return type

dict | None

encode_api_token(*data: dict[str, Any]*, ***kwargs: Any*) → str | None

Make an attempt to encode API Token.

Encode token if it possible and return string, that will be shown to user.

Parameters

- **data** (*dict*) – dictionary, containing all postprocessed data
- **kwargs** (*dict*) – any additional parameters that can be added in future or by plugins. Current implementation won't pass any additional fields, but plugins may use this feature, passing JWT *aud* or *iss* claims, for example

Returns

token as encodes string or None

Return type

str | None

preprocess_api_token(*data: Mapping[str, Any]*) → Mapping[str, Any]

Handle additional info from API Token.

Allows decoding or extracting any kind of additional information from API Token, before it used for fetching current user from database.

Parameters

data (*dict*) – dictionary with all fields that were previously created in *postprocess_api_token* (potentially modified by some other plugin already.)

Returns
dictionary that will be passed into other plugins and, finally, used for fetching User instance

Return type
dict

postprocess_api_token(*data: dict[str, Any]*, *jti: str*, *data_dict: dict[str, Any]*) → dict[str, Any]

Encode additional information into API Token.

Allows passing any kind of additional information into API Token or performing side effects, before it shown to user.

Parameters

- **data** (*dict*) – dictionary representing newly generated API Token. May be already modified by some plugin.
- **jti** (*str*) – Id of the token
- **data_dict** (*dict*) – data used for token creation.

Returns
dictionary with fields that will be encoded into final API Token

Return type
dict

add_extra_fields(*data_dict: dict[str, Any]*) → dict[str, Any]

Provide additional information alongside with API Token.

Any extra information that is not itself a part of a token, but can extend its functionality(for example, refresh token) is registered here.

Parameters
data_dict (*dict*) – dictionary that will bre returned from *api_token_create* API call.

Returns
dictionary with token and optional set of extra fields.

Return type
dict

class ckan.plugins.interfaces.**IClick**

Allow extensions to define click commands.

get_commands() → list[click.Command]

Return a list of command functions objects to be registered by the click.add_command.

Example:

```
p.implements(p.IClick)
# IClick
def get_commands(self):
    """Call me via: `ckan hello`"""
    import click
    @click.command()
    def hello():
        click.echo('Hello, World!')
    return [hello]
```

Returns
command functions objects

Return type
list of function objects

class ckan.plugins.interfaces.**ISignal**

Subscribe to CKAN signals.

get_signal_subscriptions() → Dict[Signal, Iterable[Any | Dict[str, Any]]]

Return a mapping of signals to their listeners.

Note that keys are not strings, they are instances of `blinker.Signal`. When using signals provided by CKAN core, it is better to use the references from the *plugins toolkit* for better future compatibility. Values should be a list of listener functions:

```
def get_signal_subscriptions(self):
    import ckan.plugins.toolkit as tk

    # or, even better, but requires additional dependency:
    # pip install ckantoolkit
    import ckantoolkit as tk

    return {
        tk.signals.request_started: [request_listener],
        tk.signals.register_blueprint: [
            first_blueprint_listener,
            second_blueprint_listener
        ]
    }
```

Listeners are callables that accept one mandatory argument (`sender`) and an arbitrary number of named arguments (text). The best signature for a listener is `def(sender, **kwargs)`.

The `sender` argument will be different depending on the signal and will be generally used to conditionally executing code on the listener. For example, the `register_blueprint` signal is sent every time a custom dataset/group/organization blueprint is registered (using *ckan.plugins.interfaces.IDatasetForm* or *ckan.plugins.interfaces.IGroupForm*). Depending on the kind of blueprint, `sender` may be 'dataset', 'group', 'organization' or 'resource'. If you want to do some work only for 'dataset' blueprints, you may end up with something similar to:

```
import ckan.plugins.toolkit as tk

def dataset_blueprint_listener(sender, **kwargs):
    if sender != 'dataset':
        return
    # Otherwise, do something..

class ExamplePlugin(plugins.SingletonPlugin)
    plugins.implements(plugins.ISignal)

    def get_signal_subscriptions(self):

        return {
            tk.signals.register_blueprint: [
```

(continues on next page)

(continued from previous page)

```
                dataset_blueprint_listener,
            ]
        }
```

Because this is a really common use case, there is additional form of listener registration supported. Instead of just callables, one can use dictionaries of form `{'receiver': CALLABLE, 'sender': DESIRED_SENDER}`. The following code snippet has the same effect than the previous one:

```
import ckan.plugins.toolkit as tk

def dataset_blueprint_listener(sender, **kwargs):
    # do something..

class ExamplePlugin(plugins.SingletonPlugin)
    plugins.implements(plugins.ISignal)

    def get_signal_subscriptions(self):

        return {
            tk.signals.register_blueprint: [{
                'receiver': dataset_blueprint_listener,
                'sender': 'dataset'
            }]
        }
```

The two forms of registration can be mixed when multiple listeners are registered, callables and dictionaries with `receiver/sender` keys:

```
import ckan.plugins.toolkit as tk

def log_registration(sender, **kwargs):
    log.info("Log something")

class ExamplePlugin(plugins.SingletonPlugin)
    plugins.implements(plugins.ISignal)

    def get_signal_subscriptions(self):
        return {
            tk.signals.request_started: [
                log_registration,
                {'receiver': log_registration, 'sender': 'dataset'}
            ]
        }
```

Even though it is possible to change mutable arguments inside the listener, or return something from it, the main purpose of signals is the triggering of side effects, like logging, starting background jobs, calls to external services, etc.

Any mutation or attempt to change CKAN behavior through signals should be considered unsafe and may lead to hard to track bugs in the future. So never modify the arguments of signal listener and treat them as constants.

Always check for the presence of the desired value inside the received context (named arguments). Arguments passed to signals may change over time, and some arguments may disappear.

Returns
mapping of subscriptions to signals

Return type
dict

5.8 Plugins toolkit reference

As well as using the variables made available to them by implementing plugin interfaces, plugins will likely want to be able to use parts of the CKAN core library. To allow this, CKAN provides a stable set of classes and functions that plugins can use safe in the knowledge that this interface will remain stable, backward-compatible and with clear deprecation guidelines when new versions of CKAN are released. This interface is available in CKAN's *plugins toolkit*.

class ckan.plugins.toolkit.**BaseModel**

Base class for SQLAlchemy declarative models.

Models extending `BaseModel` class are attached to the SQLAlchemy's metadata object automatically:

```
from ckan.plugins import toolkit

class ExtModel(toolkit.BaseModel):
    __tablename__ = "ext_model"
    id = Column(String(50), primary_key=True)
    ...
```

class ckan.plugins.toolkit.**CkanVersionException**

Exception raised by *requires_ckan_version()* if the required CKAN version is not available.

class ckan.plugins.toolkit.**DefaultDatasetForm**

The default implementatinon of *IDatasetForm*.

This class serves two purposes:

1. It provides a base class for plugin classes that implement *IDatasetForm* to inherit from, so they can inherit the default behavior and just modify the bits they need to.
2. It is used as the default fallback plugin when no registered *IDatasetForm* plugin handles the given dataset type and no other plugin has registered itself as the fallback plugin.

Note: *DefaultDatasetForm* doesn't call `implements()`, because we don't want it being registered.

class ckan.plugins.toolkit.**DefaultGroupForm**

Provides a default implementation of the pluggable Group controller behaviour.

This class has 2 purposes:

- it provides a base class for IGroupForm implementations to use if only a subset of the method hooks need to be customised.
- it provides the fallback behaviour if no plugin is setup to provide the fallback behaviour.

Note: this isn't a plugin implementation. This is deliberate, as we don't want this being registered.

class ckan.plugins.toolkit.**DefaultOrganizationForm**

Provides a default implementation of the pluggable Group controller behaviour.

This class has 2 purposes:

- it provides a base class for IGroupForm implementations to use if only a subset of the method hooks need to be customised.
- it provides the fallback behaviour if no plugin is setup to provide the fallback behaviour.

Note: this isn't a plugin implementation. This is deliberate, as we don't want this being registered.

class ckan.plugins.toolkit.**HelperError**

Raised if an attempt to access an undefined helper is made.

Normally, this would be a subclass of AttributeError, but Jinja2 will catch and ignore them. We want this to be an explicit failure re #2908.

class ckan.plugins.toolkit.**Invalid**

Exception raised by some validator, converter and dictization functions when the given value is invalid.

class ckan.plugins.toolkit.**NotAuthorized**

Exception raised when the user is not authorized to call the action.

For example *package_create()* raises *NotAuthorized* if the user is not authorized to create packages.

class ckan.plugins.toolkit.**ObjectNotFound**

Exception raised by logic functions when a given object is not found.

For example *package_show()* raises *ObjectNotFound* if no package with the given `id` exists.

class ckan.plugins.toolkit.**StopOnError**

error to stop validations for a particualar key

class ckan.plugins.toolkit.**UnknownValidator**

Exception raised when a requested validator function cannot be found.

class ckan.plugins.toolkit.**ValidationError**

Exception raised by action functions when validating their given `data_dict` fails.

ckan.plugins.toolkit.**_**(**args: 'Any', **kwargs: 'Any'*) → 'str'

Translates a string to the current locale.

The `_()` function is a reference to the `ugettext()` function. Everywhere in your code where you want strings to be internationalized (made available for translation into different languages), wrap them in the `_()` function, eg.:

```
msg = toolkit._("Hello")
```

Returns the localized unicode string.

ckan.plugins.toolkit.**abort**(*status_code: 'int', detail: 'str' = '', headers: 'Optional[dict[str, Any]]' = None, comment: 'Optional[str]' = None*) → 'NoReturn'

Abort the current request immediately by returning an HTTP exception.

This is a wrapper for `flask.abort()` that adds some CKAN custom behavior, including allowing *IAuthenticator* plugins to alter the abort response, and showing flash messages in the web interface.

`ckan.plugins.toolkit.add_public_directory(config_: 'CKANConfig', relative_path: 'str')`

Add a path to the *extra_public_paths* config setting.

The path is relative to the file calling this function.

Webassets addition: append directory to webassets load paths in order to correctly rewrite relative css paths and resolve public urls.

`ckan.plugins.toolkit.add_resource(path: 'str', name: 'str')`

Add a WebAssets library to CKAN.

WebAssets libraries are directories containing static resource files (e.g. CSS, JavaScript or image files) that can be compiled into WebAsset Bundles.

See *Theming guide* for more details.

`ckan.plugins.toolkit.add_template_directory(config_: 'CKANConfig', relative_path: 'str')`

Add a path to the *extra_template_paths* config setting.

The path is relative to the file calling this function.

`ckan.plugins.toolkit.asbool(obj: 'Any') → 'bool'`

Convert a string (e.g. 1, true, True) into a boolean.

Example:

```
assert asbool("yes") is True
```

`ckan.plugins.toolkit.asint(obj: 'Any') → 'int'`

Convert a string into an int.

Example:

```
assert asint("111") == 111
```

`ckan.plugins.toolkit.aslist(obj: 'Any', sep: 'Optional[str]' = None, strip: 'bool' = True) → 'Any'`

Convert a space-separated string into a list.

Example:

```
assert aslist("a b c") == ["a", "b", "c"]
```

`ckan.plugins.toolkit.auth_allow_anonymous_access(action: 'Decorated') → 'Decorated'`

Flag an auth function as not requiring a logged in user

This means that check_access won't automatically raise a NotAuthorized exception if an authenticated user is not provided in the context. (The auth function can still return False if for some reason access is not granted).

`ckan.plugins.toolkit.auth_disallow_anonymous_access(action: 'Decorated') → 'Decorated'`

Flag an auth function as requiring a logged in user

This means that check_access will automatically raise a NotAuthorized exception if an authenticated user is not provided in the context, without calling the actual auth function.

`ckan.plugins.toolkit.auth_sysadmins_check(action: 'Decorated') → 'Decorated'`

A decorator that prevents sysadmins from being automatically authorized to call an action function.

Normally sysadmins are allowed to call any action function (for example when they're using the *Action API* or the web interface), if the user is a sysadmin the action function's authorization function will not even be called.

If an action function is decorated with this decorator, then its authorization function will always be called, even if the user is a sysadmin.

`ckan.plugins.toolkit.base`

The base functionality for web-views.

Provides functions for rendering templates, aborting the request, etc.

`ckan.plugins.toolkit.blanket`

Quick implementations of simple plugin interfaces.

Blankets allow to reduce boilerplate code in plugins by simplifying the way common interfaces are registered.

For instance, this is how template helpers are generally added using the *ITemplateHelpers* interface:

```
from ckan import plugins as p
from ckanext.myext import helpers


class MyPlugin(p.SingletonPlugin):

    p.implements(ITemplateHelpers)

    def get_helpers(self):

        return {
            'my_ext_custom_helper_1': helpers.my_ext_custom_helper_1,
            'my_ext_custom_helper_2': helpers.my_ext_custom_helper_2,
        }
```

The same pattern is used for *IActions*, *IAuthFunctions*, etc.

With Blankets, assuming that you have created your module in the expected path with the expected name (see below), you can automate the registration of your helpers using the corresponding blanket decorator from the plugins toolkit:

```
@p.toolkit.blanket.helpers
class MyPlugin(p.SingletonPlugin):
    pass
```

The following table lists the available blanket decorators, the interface they implement and the default source where the blanket will automatically look for items to import:

Decorator	Interfaces	Default source
toolkit.blanket.helpers	*ITemplateHelpers*	ckanext.myext.helpers
toolkit.blanket.auth_functions	*IAuthFunctions*	ckanext.myext.logic.auth
toolkit.blanket.actions	*IActions*	ckanext.myext.logic.action
toolkit.blanket.validators	*IValidators*	ckanext.myext.logic.validators
toolkit.blanket.blueprints	*IBlueprint*	ckanext.myext.logic.views
toolkit.blanket.cli	*IClick*	ckanext.myext.cli
toolkit.blanket.config_declarati	*IConfigDeclarati*	ckanext/myext/config_declaration.{json,yaml,toml}

Note: By default, all local module members, whose `__name__`/`name` doesn't start with an underscore are exported. If the module has `__all__` list, only members listed inside this list will be exported.

If your extension uses a different naming convention for your modules, it is still possible to use blankets by passing the relevant module as a parameter to the decorator:

```
import ckanext.myext.custom_actions as custom_module

@p.toolkit.blanket.actions(custom_module)
class MyPlugin(p.SingletonPlugin):
    pass
```

Note: The `config_declarations` blanket is an exception. Instead of a module object it accepts path to the JSON, YAML or TOML file with the config declarations.

You can also pass a function that produces the artifacts required by the interface:

```
def all_actions():
    return {'ext_action': ext_action}

@p.toolkit.blanket.actions(all_actions)
class MyPlugin(p.SingletonPlugin):
    pass
```

Or just a dict with the items required by the interface:

```
all_actions = {'ext_action': ext_action}

@p.toolkit.blanket.actions(all_actions)
class MyPlugin(p.SingletonPlugin):
    pass
```

`ckan.plugins.toolkit.c`

The Pylons template context object.

[Deprecated]: Use `toolkit.g` instead.

This object is used to pass request-specific information to different parts of the code in a thread-safe way (so that variables from different requests being executed at the same time don't get confused with each other).

Any attributes assigned to `c` are available throughout the template and application code, and are local to the current request.

`ckan.plugins.toolkit.chained_action`(*func: 'ChainedAction'*) → 'ChainedAction'

Decorator function allowing action function to be chained.

This allows a plugin to modify the behaviour of an existing action function. A Chained action function must be defined as `action_function(original_action, context, data_dict)` where the first parameter will be set to the action function in the next plugin or in core ckan. The chained action may call the original_action function, optionally passing different values, handling exceptions, returning different values and/or raising different exceptions to the caller.

Usage:

```
from ckan.plugins.toolkit import chained_action

@chained_action
@side_effect_free
def package_search(original_action, context, data_dict):
    return original_action(context, data_dict)
```

Parameters
func (*callable*) – chained action function

Returns
chained action function

Return type
callable

`ckan.plugins.toolkit.chained_auth_function`(*func: 'ChainedAuthFunction'*) → 'ChainedAuthFunction'

Decorator function allowing authentication functions to be chained.

This chain starts with the last chained auth function to be registered and ends with the original auth function (or a non-chained plugin override version). Chained auth functions must accept an extra parameter, specifically the next auth function in the chain, for example:

```
auth_function(next_auth, context, data_dict).
```

The chained auth function may call the next_auth function, optionally passing different values, handling exceptions, returning different values and/or raising different exceptions to the caller.

Usage:

```
from ckan.plugins.toolkit import chained_auth_function

@chained_auth_function
@auth_allow_anonymous_access
def user_show(next_auth, context, data_dict=None):
    return next_auth(context, data_dict)
```

Parameters
func (*callable*) – chained authentication function

Returns
chained authentication function

Return type
callable

`ckan.plugins.toolkit.chained_helper`(*func: 'Helper'*) → 'Helper'

Decorator function allowing helper functions to be chained.

This chain starts with the first chained helper to be registered and ends with the original helper (or a non-chained plugin override version). Chained helpers must accept an extra parameter, specifically the next helper in the chain, for example:

```
helper(next_helper, *args, **kwargs).
```

The chained helper function may call the next_helper function, optionally passing different values, handling exceptions, returning different values and/or raising different exceptions to the caller.

Usage:

```
from ckan.plugins.toolkit import chained_helper

@chained_helper
def ckan_version(next_func, **kw):

    return next_func(**kw)
```

Parameters
func (*callable*) – chained helper function

Returns
chained helper function

Return type
callable

`ckan.plugins.toolkit.check_access`(*action: 'str', context: 'Context', data_dict: 'Optional[dict[str, Any]]' = None*) → 'Literal[True]'

Calls the authorization function for the provided action

This is the only function that should be called to determine whether a user (or an anonymous request) is allowed to perform a particular action.

The function accepts a context object, which should contain a 'user' key with the name of the user performing the action, and optionally a dictionary with extra data to be passed to the authorization function.

For example:

```
check_access('package_update', context, data_dict)
```

If not already there, the function will add an *auth_user_obj* key to the context object with the actual User object (in case it exists in the database). This check is only performed once per context object.

Raise `NotAuthorized` if the user is not authorized to call the named action function.

If the user *is* authorized to call the action, return `True`.

Parameters

- **action** (*string*) – the name of the action function, eg. `'package_create'`
- **context** (*dict*) –
- **data_dict** (*dict*) –

Raises

`NotAuthorized` if the user is not authorized to call the named action

`ckan.plugins.toolkit.`**check_ckan_version**(*min_version: 'Optional[str]' = None*, *max_version: 'Optional[str]' = None*)

Return `True` if the CKAN version is greater than or equal to `min_version` and less than or equal to `max_version`, return `False` otherwise.

If no `min_version` is given, just check whether the CKAN version is less than or equal to `max_version`.

If no `max_version` is given, just check whether the CKAN version is greater than or equal to `min_version`.

Parameters

- **min_version** (*string*) – the minimum acceptable CKAN version, eg. `'2.1'`
- **max_version** (*string*) – the maximum acceptable CKAN version, eg. `'2.3'`

`ckan.plugins.toolkit.`**ckan**

`ckan` package itself.

`ckan.plugins.toolkit.`**config**

The CKAN configuration object.

It stores the configuration values defined in the *CKAN configuration file*, eg:

```
title = toolkit.config.get("ckan.site_title")
```

`ckan.plugins.toolkit.`**current_user**

`ckan.plugins.toolkit.`**enqueue_job**(*fn: 'Callable[..., Any]'*, *args: 'Optional[Union[tuple[Any], list[Any], None]]' = None*, *kwargs: 'Optional[dict[str, Any]]' = None*, *title: 'Optional[str]' = None*, *queue: 'str' = 'default'*, *rq_kwargs: 'Optional[dict[str, Any]]' = None*) → 'Job'

Enqueue a job to be run in the background.

Parameters

- **fn** (*function*) – Function to be executed in the background
- **args** (*list*) – List of arguments to be passed to the function. Pass an empty list if there are no arguments (default).
- **kwargs** (*dict*) – Dict of keyword arguments to be passed to the function. Pass an empty dict if there are no keyword arguments (default).

- **title** (*string*) – Optional human-readable title of the job.
- **queue** (*string*) – Name of the queue. If not given then the default queue is used.
- **rq_kwargs** (*dict*) – Dict of keyword arguments that will get passed to the RQ `enqueue_call` invocation (eg `timeout`, `depends_on`, `ttl` etc).

Returns
The enqueued job.

Return type
`rq.job.Job`

`ckan.plugins.toolkit.error_shout(exception: 'Any')` → 'None'

Report CLI error with a styled message.

`ckan.plugins.toolkit.fresh_context(context: 'Context')` → 'Context'

Copy just the minimum fields into a new context for cases in which we reuse the context and we want a clean version with minimum fields

`ckan.plugins.toolkit.g`

The Flask global object.

This object is used to pass request-specific information to different parts of the code in a thread-safe way (so that variables from different requests being executed at the same time don't get confused with each other).

Any attributes assigned to *g* are available throughout the template and application code, and are local to the current request.

It is a bad pattern to pass variables to the templates using the `g` object. Pass them explicitly from the view functions as `extra_vars`, eg:

```
return toolkit.render(
    'myext/package/read.html',
    extra_vars={
        u'some_var': some_value,
        u'some_other_var': some_other_value,
    }
)
```

`ckan.plugins.toolkit.get_action(action: 'str')` → 'Action'

Return the named `ckan.logic.action` function.

For example `get_action('package_create')` will normally return the *ckan.logic.action.create.package_create()* function.

For documentation of the available action functions, see *Action API reference*.

You should always use `get_action()` instead of importing an action function directly, because *IActions* plugins can override action functions, causing `get_action()` to return a plugin-provided function instead of the default one.

Usage:

```
import ckan.plugins.toolkit as toolkit

# Call the package_create action function:
toolkit.get_action('package_create')(context, data_dict)
```

As the context parameter passed to an action function is commonly:

```
context = {'model': ckan.model, 'session': ckan.model.Session,
           'user': user}
```

an action function returned by `get_action()` will automatically add these parameters to the context if they are not defined. This is especially useful for plugins as they should not really be importing parts of ckan eg `ckan.model` and as such do not have access to `model` or `model.Session`.

If a `context` of `None` is passed to the action function then the default context dict will be created.

Note: Many action functions modify the context dict. It can therefore not be reused for multiple calls of the same or different action functions.

Parameters
action (*string*) – name of the action function to return, eg. `'package_create'`

Returns
the named action function

Return type
callable

`ckan.plugins.toolkit.get_converter(validator: 'str')` → 'Union[Validator, ValidatorFactory]'

Return a validator function by name.

Parameters
validator (*string*) – the name of the validator function to return, eg. `'package_name_exists'`

Raises
UnknownValidator if the named validator is not found

Returns
the named validator function

Return type
`types.FunctionType`

`ckan.plugins.toolkit.get_endpoint()` → 'Union[tuple[str, str], tuple[None, None]]'

Returns tuple in format: (blueprint, view).

`ckan.plugins.toolkit.get_or_bust(data_dict: 'dict[str, Any]', keys: 'Union[str, Iterable[str]]')` → 'Union[Any, tuple[Any, ...]]'

Return the value(s) from the given data_dict for the given key(s).

Usage:

```
single_value = get_or_bust(data_dict, 'a_key')
value_1, value_2 = get_or_bust(data_dict, ['key1', 'key2'])
```

Parameters

- **data_dict** (*dictionary*) – the dictionary to return the values from
- **keys** (*either a string or a list*) – the key(s) for the value(s) to return

Returns
a single value from the dict if a single key was given, or a tuple of values if a list of keys was given

Raises
`ckan.logic.ValidationError` if one of the given keys is not in the given dictionary

`ckan.plugins.toolkit.get_validator(validator: 'str')` → 'Union[Validator, ValidatorFactory]'

Return a validator function by name.

Parameters
validator (*string*) – the name of the validator function to return, eg. `'package_name_exists'`

Raises
`UnknownValidator` if the named validator is not found

Returns
the named validator function

Return type
`types.FunctionType`

`ckan.plugins.toolkit.h`

Collection of CKAN native and extension-provided helpers.

`class ckan.plugins.toolkit.literal`

Represents an HTML literal.

`ckan.plugins.toolkit.login_user(user, remember=False, duration=None, force=False, fresh=True)`

Logs a user in. You should pass the actual user object to this. If the user's *is_active* property is `False`, they will not be logged in unless *force* is `True`.

This will return `True` if the log in attempt succeeds, and `False` if it fails (i.e. because the user is inactive).

Parameters

- **user** (*object*) – The user object to log in.
- **remember** (*bool*) – Whether to remember the user after their session expires. Defaults to `False`.
- **duration** (`datetime.timedelta`) – The amount of time before the remember cookie expires. If `None` the value set in the settings is used. Defaults to `None`.
- **force** (*bool*) – If the user is inactive, setting this to `True` will log them in regardless. Defaults to `False`.
- **fresh** (*bool*) – setting this to `False` will log in the user with a session marked as not "fresh". Defaults to `True`.

`ckan.plugins.toolkit.logout_user()`

Logs a user out. (You do not need to pass the actual user.) This will also clean up the remember me cookie if it exists.

`ckan.plugins.toolkit.mail_recipient(recipient_name: 'str', recipient_email: 'str', subject: 'str', body: 'str', body_html: 'Optional[str]' = None, headers: 'Optional[dict[str, Any]]' = None, attachments: 'Optional[Iterable[Attachment]]' = None)` → 'None'

Sends an email to a an email address.

Note: You need to set up the *Email settings* to able to send emails.

Parameters

- **recipient_name** – the name of the recipient
- **recipient_email** – the email address of the recipient
- **subject** (*string*) – the email subject
- **body** (*string*) – the email body, in plain text
- **body_html** (*string*) – the email body, in html format (optional)

Headers
extra headers to add to email, in the form {'Header name': 'Header value'}

Type
dict

Attachments
a list of tuples containing file attachments to add to the email. Tuples should contain the file name and a file-like object pointing to the file contents:

```
[
    ('some_report.csv', file_object),
]
```

Optionally, you can add a third element to the tuple containing the media type. If not provided, it will be guessed using the `mimetypes` module:

```
[
    ('some_report.csv', file_object, 'text/csv'),
]
```

Type
list

`ckan.plugins.toolkit.mail_user`(*recipient: 'model.User', subject: 'str', body: 'str', body_html: 'Optional[str]' = None, headers: 'Optional[dict[str, Any]]' = None, attachments: 'Optional[Iterable[Attachment]]' = None*) → 'None'

Sends an email to a CKAN user.

You need to set up the *Email settings* to able to send emails.

Parameters
recipient (*a model.User object*) – a CKAN user object

For further parameters see `mail_recipient()`.

`ckan.plugins.toolkit.missing`

`ckan.plugins.toolkit.navl_validate`(*data: 'dict[str, Any]', schema: 'dict[str, Any]', context: 'Optional[Context]' = None*) → 'tuple[dict[str, Any], dict[str, Any]]'

Validate an unflattened nested dict against a schema.

`ckan.plugins.toolkit.redirect_to`(**args: 'Any', **kw: 'Any'*) → 'Response'

Issue a redirect: return an HTTP response with a `302 Moved` header.

This is a wrapper for `flask.redirect()` that maintains the user's selected language when redirecting.

The arguments to this function identify the route to redirect to, they're the same arguments as *`ckan.plugins.toolkit.url_for()`* accepts, for example:

```
import ckan.plugins.toolkit as toolkit

# Redirect to /dataset/my_dataset.
return toolkit.redirect_to('dataset.read',
                    id='my_dataset')
```

Or, using a named route:

```
return toolkit.redirect_to('dataset.read', id='changed')
```

If given a single string as argument, this redirects without url parsing

return toolkit.redirect_to('http://example.com') return toolkit.redirect_to('/dataset') return toolkit.redirect_to('/some/other/path')

`ckan.plugins.toolkit.render`(*template_name: 'str', extra_vars: 'Optional[dict[str, Any]]' = None*) → 'str'

Render a template and return the output.

This is CKAN's main template rendering function.

Params template_name
relative path to template inside registered tpl_dir

Params extra_vars
additional variables available in template

`ckan.plugins.toolkit.render_snippet`(*template: 'str', data: 'Optional[dict[str, Any]]' = None*)

Render a template snippet and return the output.

See *Theming guide*.

`ckan.plugins.toolkit.request`

Flask request object.

A new request object is created for each HTTP request. It has methods and attributes for getting things like the request headers, query-string variables, request body variables, cookies, the request URL, etc.

`ckan.plugins.toolkit.requires_ckan_version`(*min_version: 'str', max_version: 'Optional[str]' = None*)

Raise *`CkanVersionException`* if the CKAN version is not greater than or equal to `min_version` and less then or equal to `max_version`.

If no `max_version` is given, just check whether the CKAN version is greater than or equal to `min_version`.

Plugins can call this function if they require a certain CKAN version, other versions of CKAN will crash if a user tries to use the plugin with them.

Parameters

- **`min_version`** (*`string`*) – the minimum acceptable CKAN version, eg. `'2.1'`
- **`max_version`** (*`string`*) – the maximum acceptable CKAN version, eg. `'2.3'`

`ckan.plugins.toolkit.side_effect_free`(*action: 'Decorated'*) → 'Decorated'

A decorator that marks the given action function as side-effect-free.

Action functions decorated with this decorator can be called with an HTTP GET request to the *Action API*. Action functions that don't have this decorator must be called with a POST request.

If your CKAN extension defines its own action functions using the `IActions` plugin interface, you can use this decorator to make your actions available with GET requests instead of just with POST requests.

Example:

```
import ckan.plugins.toolkit as toolkit

@toolkit.side_effect_free
def my_custom_action_function(context, data_dict):
    ...
```

(Then implement `IActions` to register your action function with CKAN.)

`ckan.plugins.toolkit.signals`

Contains `ckan` and `ckanext` namespaces for signals as well as a bunch of predefined core-level signals.

Check *Signals* for extra detais.

`ckan.plugins.toolkit.ungettext`(**args: 'Any', **kwargs: 'Any'*) → 'str'

Translates a string with plural forms to the current locale.

Mark a string for translation that has pural forms in the format `ungettext(singular, plural, n)`. Returns the localized unicode string of the pluralized value.

Mark a string to be localized as follows:

```
msg = toolkit.ungettext("Mouse", "Mice", len(mouses))
```

`ckan.plugins.toolkit.url_for`(**args: 'Any', **kw: 'Any'*) → 'str'

Return the URL for an endpoint given some parameters.

This is a wrapper for `flask.url_for()` and `routes.url_for()` that adds some extra features that CKAN needs.

To build a URL for a Flask view, pass the name of the blueprint and the view function separated by a period `.`, plus any URL parameters:

```
url_for('api.action', ver=3, logic_function='status_show')
# Returns /api/3/action/status_show
```

For a fully qualified URL pass the `_external=True` parameter. This takes the `ckan.site_url` and `ckan.root_path` settings into account:

```
url_for('api.action', ver=3, logic_function='status_show',
        _external=True)
# Returns http://example.com/api/3/action/status_show
```

URLs built by Pylons use the Routes syntax:

```
url_for(controller='my_ctrl', action='my_action', id='my_dataset')
# Returns '/dataset/my_dataset'
```

Or, using a named route:

```
url_for('dataset.read', id='changed')
# Returns '/dataset/changed'
```

Use `qualified=True` for a fully qualified URL when targeting a Pylons endpoint.

For backwards compatibility, an effort is made to support the Pylons syntax when building a Flask URL, but this support might be dropped in the future, so calls should be updated.

5.9 Validator functions reference

Validators in CKAN are user-defined functions that serves two purposes:

- Ensuring that the input satisfies certain requirements
- Converting the input to an expected form

Validators can be defined as a function that accepts one, two or four arguments. But this is an implementation detail and validators should not be called directly. Instead, the `ckan.plugins.toolkit.navl_validate()` function must be used whenever input requires validation.

```
import ckan.plugins.toolkit as tk
from ckanext.my_ext.validators import is_valid

data, errors = tk.navl_validate(
    {"input": "value"},
    {"input": [is_valid]},
)
```

And in order to be more flexible and allow overrides, don't import validator functions directly. Instead, register them via the *`IValidators`* interface and use the `ckan.plugins.tookit.get_validator()` function:

```
import ckan.plugins as p
import ckan.plugins.toolkit as tk

def is_valid(value):
    return value

class MyPlugin(p.SingletonPlugin)
    p.implements(p.IValidators)

    def get_validators(self):
        return {"is_valid": is_valid}

...
# somewhere in code
data, errors = tk.navl_validate(
    {"input": "value"},
    {"input": [tk.get_validator("is_valid")]},
)
```

As you should have already noticed, `navl_validate` requires two parameters and additionally accepts an optional one. That's their purpose:

1. Data that requires validation. Must be a *dict* object, with keys being the names of the fields.

2. The validation schema. It's a mapping of field names to the lists of validators for that particular field.
3. Optional context. Contains any extra details that can change validation workflow in special cases. For the simplicity sake, we are not going to use context in this section, and in general is best not to rely on context variables inside validators.

Let's imagine an input that contains two fields `first` and `second`. The `first` field must be an integer and must be provided, while the `second` field is an optional string. If we have following four validators:

- `is_integer`
- `is_string`
- `is_required`
- `is_optional`

we can validate data in the following way:

```
input = {"first": "123"}
schema = {
    "first": [is_required, is_integer],
    "second": [is_optional, is_string],
}

data, errors = tk.navl_validate(input, schema)
```

If the input is valid, `data` contains validated input and `errors` is an empty dictionary. Otherwise, `errors` contains all the validation errors for the provided input.

5.9.1 Built-in validators

`ckan.lib.navl.validators.`**`keep_extras`**(*key: tuple[Any, ...], data: dict[tuple[Any, ...], Any], errors: dict[tuple[Any, ...], list[str]], context: Context*) → None

Convert dictionary into simple fields.

```
data, errors = tk.navl_validate(
    {"input": {"hello": 1, "world": 2}},
    {"input": [keep_extras]}
)
assert data == {"hello": 1, "world": 2}
```

`ckan.lib.navl.validators.`**`not_missing`**(*key: tuple[Any, ...], data: dict[tuple[Any, ...], Any], errors: dict[tuple[Any, ...], list[str]], context: Context*) → None

Ensure value is not missing from the input, but may be empty.

```
data, errors = tk.navl_validate(
    {},
    {"hello": [not_missing]}
)
assert errors == {"hello": [error_message]}
```

`ckan.lib.navl.validators.`**`not_empty`**(*key: tuple[Any, ...], data: dict[tuple[Any, ...], Any], errors: dict[tuple[Any, ...], list[str]], context: Context*) → None

Ensure value is available in the input and is not empty.

```
data, errors = tk.navl_validate(
    {"hello": None},
    {"hello": [not_empty]}
)
assert errors == {"hello": [error_message]}
```

ckan.lib.navl.validators.**if_empty_same_as**(*other_key: str*) → Callable[[...], Any]

Copy value from other field when current field is missing or empty.

```
data, errors = tk.navl_validate(
    {"hello": 1},
    {"hello": [], "world": [if_empty_same_as("hello")]}
)
assert data == {"hello": 1, "world": 1}
```

ckan.lib.navl.validators.**both_not_empty**(*other_key: str*) → Callable[[Any], Any] | Callable[[Any, Context], Any] | Callable[[tuple[Any, ...], dict[tuple[Any, ...], Any], dict[tuple[Any, ...], list[str]], Context], None]

Ensure that both, current value and other field has value.

```
data, errors = tk.navl_validate(
    {"hello": 1},
    {"hello": [], "world": [both_not_empty("hello")]}
)
assert errors == {"world": [error_message]}

data, errors = tk.navl_validate(
    {"world": 1},
    {"hello": [], "world": [both_not_empty("hello")]}
)
assert errors == {"world": [error_message]}

data, errors = tk.navl_validate(
    {"hello": 1, "world": 2},
    {"hello": [], "world": [both_not_empty("hello")]}
)
assert not errors
```

ckan.lib.navl.validators.**empty**(*key: tuple[Any, ...], data: dict[tuple[Any, ...], Any], errors: dict[tuple[Any, ...], list[str]], context: Context*) → None

Ensure that value is not present in the input.

```
data, errors = tk.navl_validate(
    {"hello": 1},
    {"hello": [empty]}
)
assert errors == {"hello": [error_message]}
```

ckan.lib.navl.validators.**ignore**(*key: tuple[Any, ...], data: dict[tuple[Any, ...], Any], errors: dict[tuple[Any, ...], list[str]], context: Context*) → NoReturn

Remove the value from the input and skip the rest of validators.

```
data, errors = tk.navl_validate(
    {"hello": 1},
    {"hello": [ignore]}
)
assert data == {}
```

ckan.lib.navl.validators.**default**(*default_value: Any*) → Callable[[Any], Any] | Callable[[Any, Context], Any] | Callable[[tuple[Any, ...], dict[tuple[Any, ...], Any], dict[tuple[Any, ...], list[str]], Context], None]

Convert missing or empty value to the default one.

```
data, errors = tk.navl_validate(
    {},
    {"hello": [default("not empty")]}
)
assert data == {"hello": "not empty"}
```

ckan.lib.navl.validators.**configured_default**(*config_name: str*, *default_value_if_not_configured: Any*) → Callable[[Any], Any] | Callable[[Any, Context], Any] | Callable[[tuple[Any, ...], dict[tuple[Any, ...], Any], dict[tuple[Any, ...], list[str]], Context], None]

When key is missing or value is an empty string or None, replace it with a default value from config, or if that isn't set from the default_value_if_not_configured.

ckan.lib.navl.validators.**ignore_missing**(*key: tuple[Any, ...]*, *data: dict[tuple[Any, ...], Any]*, *errors: dict[tuple[Any, ...], list[str]]*, *context: Context*) → None

If the key is missing from the data, ignore the rest of the key's schema.

By putting ignore_missing at the start of the schema list for a key, you can allow users to post a dict without the key and the dict will pass validation. But if they post a dict that does contain the key, then any validators after ignore_missing in the key's schema list will be applied.

Raises
ckan.lib.navl.dictization_functions.StopOnError – if `data[key]` is `ckan.lib.navl.dictization_functions.missing` or `None`

Returns
`None`

ckan.lib.navl.validators.**ignore_empty**(*key: tuple[Any, ...]*, *data: dict[tuple[Any, ...], Any]*, *errors: dict[tuple[Any, ...], list[str]]*, *context: Context*) → None

Skip the rest of validators if the value is empty or missing.

```
data, errors = tk.navl_validate(
    {"hello": ""},
    {"hello": [ignore_empty, isodate]}
)
assert data == {}
assert not errors
```

ckan.lib.navl.validators.**convert_int**(*value: Any*) → int

Ensure that the value is a valid integer.

```
data, errors = tk.navl_validate(
    {"hello": "world"},
    {"hello": [convert_int]}
)
assert errors == {"hello": [error_message]}
```

ckan.lib.navl.validators.**unicode_only**(*value: Any*) → str

Accept only unicode values

```
data, errors = tk.navl_validate(
    {"hello": 1},
    {"hello": [unicode_only]}
)
assert errors == {"hello": [error_message]}
```

ckan.lib.navl.validators.**unicode_safe**(*value: Any*) → str

Make sure value passed is treated as unicode, but don't raise an error if it's not, just make a reasonable attempt to convert other types passed.

This validator is a safer alternative to the old ckan idiom of using the unicode() function as a validator. It tries not to pollute values with Python repr garbage e.g. when passed a list of strings (uses json format instead). It also converts binary strings assuming either UTF-8 or CP1252 encodings (not ASCII, with occasional decoding errors)

ckan.lib.navl.validators.**limit_to_configured_maximum**(*config_option: str*, *default_limit: int*) → Callable[[Any], Any] | Callable[[Any, Context], Any] | Callable[[tuple[Any, ...], dict[tuple[Any, ...], Any], dict[tuple[Any, ...], list[str]], Context], None]

If the value is over a limit, it changes it to the limit. The limit is defined by a configuration option, or if that is not set, a given int default_limit.

ckan.logic.validators.**owner_org_validator**(*key: tuple[Any, ...]*, *data: dict[tuple[Any, ...], Any]*, *errors: dict[tuple[Any, ...], list[str]]*, *context: Context*) → Any

Validate organization for the dataset.

Depending on the settings and user's permissions, this validator checks whether organization is optional and ensures that specified organization can be set as an owner of dataset.

ckan.logic.validators.**package_id_not_changed**(*value: Any*, *context: Context*) → Any

Ensures that package's ID is not changed during the update.

ckan.logic.validators.**int_validator**(*value: Any*, *context: Context*) → Any

Return an integer for value, which may be a string in base 10 or a numeric type (e.g. int, long, float, Decimal, Fraction). Return None for None or empty/all-whitespace string values.

Raises

ckan.lib.navl.dictization_functions.Invalid for other inputs or non-whole values

ckan.logic.validators.**natural_number_validator**(*value: Any*, *context: Context*) → Any

Ensures that the value is non-negative integer.

ckan.logic.validators.**is_positive_integer**(*value: Any*, *context: Context*) → Any

Ensures that the value is an integer that is greater than zero.

ckan.logic.validators.**datetime_from_timestamp_validator**(*value: Any*, *context: Context*) → Any

`ckan.logic.validators.boolean_validator(value: Any, context: Context) → Any`

Return a boolean for value. Return value when value is a python bool type. Return True for strings 'true', 'yes', 't', 'y', and '1'. Return False in all other cases, including when value is an empty string or None

`ckan.logic.validators.isodate(value: Any, context: Context) → Any`

Convert the value into `datetime.datetime` object.

`ckan.logic.validators.package_id_exists(value: str, context: Context) → Any`

Ensures that the value is an existing package's ID or name.

`ckan.logic.validators.package_id_does_not_exist(value: str, context: Context) → Any`

Ensures that the value is not used as a package's ID or name.

`ckan.logic.validators.resource_id_does_not_exist(key: tuple[Any, ...], data: dict[tuple[Any, ...], Any], errors: dict[tuple[Any, ...], list[str]], context: Context) → Any`

`ckan.logic.validators.package_name_exists(value: str, context: Context) → Any`

Ensures that the value is an existing package's name.

`ckan.logic.validators.package_id_or_name_exists(package_id_or_name: str, context: Context) → Any`

Return the given package_id_or_name if such a package exists.

Raises
ckan.lib.navl.dictization_functions.Invalid if there is no package with the given id or name

`ckan.logic.validators.resource_id_exists(value: Any, context: Context) → Any`

Ensures that the value is not used as a resource's ID or name.

`ckan.logic.validators.resource_id_validator(value: Any) → Any`

`ckan.logic.validators.user_id_exists(user_id: str, context: Context) → Any`

Raises Invalid if the given user_id does not exist in the model given in the context, otherwise returns the given user_id.

`ckan.logic.validators.user_id_or_name_exists(user_id_or_name: str, context: Context) → Any`

Return the given user_id_or_name if such a user exists.

Raises
ckan.lib.navl.dictization_functions.Invalid if no user can be found with the given id or user name

`ckan.logic.validators.group_id_exists(group_id: str, context: Context) → Any`

Raises Invalid if the given group_id does not exist in the model given in the context, otherwise returns the given group_id.

`ckan.logic.validators.group_id_or_name_exists(reference: str, context: Context) → Any`

Raises Invalid if a group identified by the name or id cannot be found.

`ckan.logic.validators.name_validator(value: Any, context: Context) → Any`

Return the given value if it's a valid name, otherwise raise Invalid.

If it's a valid name, the given value will be returned unmodified.

This function applies general validation rules for names of packages, groups, users, etc.

Most schemas also have their own custom name validator function to apply custom validation rules after this function, for example a `package_name_validator()` to check that no package with the given name already exists.

Raises
ckan.lib.navl.dictization_functions.Invalid – if `value` is not a valid name

`ckan.logic.validators.`**`package_name_validator`**(*key: tuple[Any, ...], data: dict[tuple[Any, ...], Any], errors: dict[tuple[Any, ...], list[str]], context: Context*) → Any

Ensures that value can be used as a package's name

`ckan.logic.validators.`**`package_version_validator`**(*value: Any, context: Context*) → Any

Ensures that value can be used as a package's version

`ckan.logic.validators.`**`duplicate_extras_key`**(*key: tuple[Any, ...], data: dict[tuple[Any, ...], Any], errors: dict[tuple[Any, ...], list[str]], context: Context*) → Any

Ensures that there are no duplicated extras.

`ckan.logic.validators.`**`group_name_validator`**(*key: tuple[Any, ...], data: dict[tuple[Any, ...], Any], errors: dict[tuple[Any, ...], list[str]], context: Context*) → Any

Ensures that value can be used as a group's name

`ckan.logic.validators.`**`tag_length_validator`**(*value: Any, context: Context*) → Any

Ensures that tag length is in the acceptable range.

`ckan.logic.validators.`**`tag_name_validator`**(*value: Any, context: Context*) → Any

Ensures that tag does not contain wrong characters

`ckan.logic.validators.`**`tag_not_uppercase`**(*value: Any, context: Context*) → Any

Ensures that tag is lower-cased.

`ckan.logic.validators.`**`tag_string_convert`**(*key: tuple[Any, ...], data: dict[tuple[Any, ...], Any], errors: dict[tuple[Any, ...], list[str]], context: Context*) → Any

Takes a list of tags that is a comma-separated string (in data[key]) and parses tag names. These are added to the data dict, enumerated. They are also validated.

`ckan.logic.validators.`**`ignore_not_package_admin`**(*key: tuple[Any, ...], data: dict[tuple[Any, ...], Any], errors: dict[tuple[Any, ...], list[str]], context: Context*) → Any

Ignore if the user is not allowed to administer the package specified.

`ckan.logic.validators.`**`ignore_not_sysadmin`**(*key: tuple[Any, ...], data: dict[tuple[Any, ...], Any], errors: dict[tuple[Any, ...], list[str]], context: Context*) → Any

Ignore the field if user not sysadmin or ignore_auth in context.

`ckan.logic.validators.`**`ignore_not_group_admin`**(*key: tuple[Any, ...], data: dict[tuple[Any, ...], Any], errors: dict[tuple[Any, ...], list[str]], context: Context*) → Any

Ignore if the user is not allowed to administer for the group specified.

`ckan.logic.validators.`**`user_name_validator`**(*key: tuple[Any, ...], data: dict[tuple[Any, ...], Any], errors: dict[tuple[Any, ...], list[str]], context: Context*) → Any

Validate a new user name.

Append an error message to `errors[key]` if a user named `data[key]` already exists. Otherwise, do nothing.

Raises
ckan.lib.navl.dictization_functions.Invalid – if `data[key]` is not a string

Return type
None

`ckan.logic.validators.`**`user_both_passwords_entered`**(*key: tuple[Any, ...], data: dict[tuple[Any, ...], Any], errors: dict[tuple[Any, ...], list[str]], context: Context*) → Any

Ensures that both password and password confirmation is not empty

`ckan.logic.validators.`**`user_password_validator`**(*key: tuple[Any, ...], data: dict[tuple[Any, ...], Any], errors: dict[tuple[Any, ...], list[str]], context: Context*) → Any

Ensures that password is safe enough.

`ckan.logic.validators.`**`user_passwords_match`**(*key: tuple[Any, ...], data: dict[tuple[Any, ...], Any], errors: dict[tuple[Any, ...], list[str]], context: Context*) → Any

Ensures that password and password confirmation match.

`ckan.logic.validators.`**`user_password_not_empty`**(*key: tuple[Any, ...], data: dict[tuple[Any, ...], Any], errors: dict[tuple[Any, ...], list[str]], context: Context*) → Any

Only check if password is present if the user is created via action API. If not, user_both_passwords_entered will handle the validation

`ckan.logic.validators.`**`user_about_validator`**(*value: Any, context: Context*) → Any

Ensures that user's `about` field does not contains links.

`ckan.logic.validators.`**`vocabulary_name_validator`**(*name: str, context: Context*) → Any

Ensures that the value can be used as a tag vocabulary name.

`ckan.logic.validators.`**`vocabulary_id_not_changed`**(*value: Any, context: Context*) → Any

Ensures that vocabulary ID is not changed during the update.

`ckan.logic.validators.`**`vocabulary_id_exists`**(*value: Any, context: Context*) → Any

Ensures that value contains existing vocabulary's ID or name.

`ckan.logic.validators.`**`tag_in_vocabulary_validator`**(*value: Any, context: Context*) → Any

Ensures that the tag belongs to the vocabulary.

`ckan.logic.validators.`**`tag_not_in_vocabulary`**(*key: tuple[Any, ...], tag_dict: dict[tuple[Any, ...], Any], errors: dict[tuple[Any, ...], list[str]], context: Context*) → Any

Ensures that the tag does not belong to the vocabulary.

`ckan.logic.validators.`**`url_validator`**(*key: tuple[Any, ...], data: dict[tuple[Any, ...], Any], errors: dict[tuple[Any, ...], list[str]], context: Context*) → Any

Checks that the provided value (if it is present) is a valid URL

`ckan.logic.validators.`**`user_name_exists`**(*user_name: str, context: Context*) → Any

Ensures that user's name exists.

`ckan.logic.validators.`**`role_exists`**(*role: str, context: Context*) → Any

Ensures that value is an existing CKAN Role name.

`ckan.logic.validators.`**`datasets_with_no_organization_cannot_be_private`**(*key: tuple[Any, ...], data: dict[tuple[Any, ...], Any], errors: dict[tuple[Any, ...], list[str]], context: Context*) → Any

`ckan.logic.validators.list_of_strings(key: tuple[Any, ...], data: dict[tuple[Any, ...], Any], errors: dict[tuple[Any, ...], list[str]], context: Context) → Any`

Ensures that value is a list of strings.

`ckan.logic.validators.if_empty_guess_format(key: tuple[Any, ...], data: dict[tuple[Any, ...], Any], errors: dict[tuple[Any, ...], list[str]], context: Context) → Any`

Make an attempt to guess resource's format using URL.

`ckan.logic.validators.clean_format(format: str)`

Normalize resource's format.

`ckan.logic.validators.no_loops_in_hierarchy(key: tuple[Any, ...], data: dict[tuple[Any, ...], Any], errors: dict[tuple[Any, ...], list[str]], context: Context) → Any`

Checks that the parent groups specified in the data would not cause a loop in the group hierarchy, and therefore cause the recursion up/down the hierarchy to get into an infinite loop.

`ckan.logic.validators.filter_fields_and_values_should_have_same_length(key: tuple[Any, ...], data: dict[tuple[Any, ...], Any], errors: dict[tuple[Any, ...], list[str]], context: Context) → Any`

`ckan.logic.validators.filter_fields_and_values_exist_and_are_valid(key: tuple[Any, ...], data: dict[tuple[Any, ...], Any], errors: dict[tuple[Any, ...], list[str]], context: Context) → Any`

`ckan.logic.validators.extra_key_not_in_root_schema(key: tuple[Any, ...], data: dict[tuple[Any, ...], Any], errors: dict[tuple[Any, ...], list[str]], context: Context) → Any`

Ensures that extras are not duplicating base fields

`ckan.logic.validators.empty_if_not_sysadmin(key: tuple[Any, ...], data: dict[tuple[Any, ...], Any], errors: dict[tuple[Any, ...], list[str]], context: Context) → Any`

Only sysadmins may pass this value

`ckan.logic.validators.strip_value(value: str)`

Trims the Whitespace

`ckan.logic.validators.email_validator(value: Any, context: Context) → Any`

Validate email input

`ckan.logic.validators.collect_prefix_validate(prefix: str, *validator_names: str) → Callable[[Any], Any] | Callable[[Any, Context], Any] | Callable[[tuple[Any, ...], dict[tuple[Any, ...], Any], dict[tuple[Any, ...], list[str]], Context], None]`

Return a validator that will collect top-level keys starting with prefix then apply validator_names to each one. Results are moved to a dict under the prefix name, with prefix removed from keys

`ckan.logic.validators.dict_only(value: Any) → dict[Any, Any]`

Ensures that the value is a dictionary

`ckan.logic.validators.email_is_unique(key: tuple[Any, ...], data: dict[tuple[Any, ...], Any], errors: dict[tuple[Any, ...], list[str]], context: Context) → Any`

Validate email is unique

`ckan.logic.validators.one_of`(*list_of_value: Container[Any]*) → Callable[[Any], Any] | Callable[[Any, Context], Any] | Callable[[tuple[Any, ...], dict[tuple[Any, ...], Any], dict[tuple[Any, ...], list[str]], Context], None]

Checks if the provided value is present in a list or is an empty string

`ckan.logic.validators.json_object`(*value: Any*) → Any

Make sure value can be serialized as a JSON object

`ckan.logic.validators.extras_valid_json`(*extras: Any*, *context: Context*) → Any

Ensures that every item in the value dictionary is JSON-serializable.

`ckan.logic.converters.convert_to_extras`(*key: tuple[Any, ...]*, *data: dict[tuple[Any, ...], Any]*, *errors: dict[tuple[Any, ...], list[str]]*, *context: Context*) → Any

Convert given field into an extra field.

`ckan.logic.converters.convert_from_extras`(*key: tuple[Any, ...]*, *data: dict[tuple[Any, ...], Any]*, *errors: dict[tuple[Any, ...], list[str]]*, *context: Context*) → Any

Restore field using object's extras.

`ckan.logic.converters.extras_unicode_convert`(*extras: dict[tuple[Any, ...], Any]*, *context: Context*)

Convert every value of the dictionary into string.

`ckan.logic.converters.free_tags_only`(*key: tuple[Any, ...]*, *data: dict[tuple[Any, ...], Any]*, *errors: dict[tuple[Any, ...], list[str]]*, *context: Context*) → Any

Ensure that none of the tags belong to a vocabulary.

`ckan.logic.converters.convert_to_tags`(*vocab: Any*) → Callable[[tuple[Any, ...], dict[tuple[Any, ...], Any], dict[tuple[Any, ...], list[str]], Context], None]

Convert list of tag names into a list of tag dictionaries

`ckan.logic.converters.convert_from_tags`(*vocab: Any*) → Callable[[tuple[Any, ...], dict[tuple[Any, ...], Any], dict[tuple[Any, ...], list[str]], Context], None]

`ckan.logic.converters.convert_user_name_or_id_to_id`(*user_name_or_id: Any*, *context: Context*) → Any

Return the user id for the given user name or id.

The point of this function is to convert user names to ids. If you have something that may be a user name or a user id you can pass it into this function and get the user id out either way.

Also validates that a user with the given name or id exists.

Returns
the id of the user with the given user name or id

Return type
string

Raises
ckan.lib.navl.dictization_functions.Invalid if no user can be found with the given id or user name

`ckan.logic.converters.convert_package_name_or_id_to_id`(*package_name_or_id: Any*, *context: Context*) → Any

Return the package id for the given package name or id.

The point of this function is to convert package names to ids. If you have something that may be a package name or id you can pass it into this function and get the id out either way.

Also validates that a package with the given name or id exists.

Returns
the id of the package with the given name or id

Return type
string

Raises
ckan.lib.navl.dictization_functions.Invalid if there is no package with the given name or id

`ckan.logic.converters.`**`convert_group_name_or_id_to_id`**(*group_name_or_id: Any*, *context: Context*) → Any

Return the group id for the given group name or id.

The point of this function is to convert group names to ids. If you have something that may be a group name or id you can pass it into this function and get the id out either way.

Also validates that a group with the given name or id exists.

Returns
the id of the group with the given name or id

Return type
string

Raises
ckan.lib.navl.dictization_functions.Invalid if there is no group with the given name or id

`ckan.logic.converters.`**`convert_to_json_if_string`**(*value: Any*, *context: Context*) → Any

Parse string value as a JSON object.

`ckan.logic.converters.`**`as_list`**(*value: Any*)

Convert whitespace separated string into a list of strings.

`ckan.logic.converters.`**`convert_to_list_if_string`**(*value: Any*) → Any

Transform string into one-item list

`ckan.logic.converters.`**`json_or_string`**(*value: Any*) → Any

parse string values as json, return string if that fails

`ckan.logic.converters.`**`json_list_or_string`**(*value: Any*) → Any

parse string values as json or comma-separated lists, return string as a one-element list if that fails

`ckan.logic.converters.`**`remove_whitespace`**(*value: Any*, *context: Context*) → Any

Trim whitespaces from the value.

5.10 Internationalizing strings in extensions

See also:

In order to internationalize your extension you must *mark its strings for internationalization*. See also *Translating CKAN*.

This tutorial assumes that you have read the *Writing extensions tutorial*.

We will create a simple extension to demonstrate the translation of strings inside extensions. After running:

```
ckan -c |ckan.ini| create -t ckanext ckanext-itranslation
```

Change the `plugin.py` file to:

```
# encoding: utf-8

from ckan.common import CKANConfig
from ckan import plugins
from ckan.plugins import toolkit


class ExampleITranslationPlugin(plugins.SingletonPlugin):
    plugins.implements(plugins.IConfigurer)

    def update_config(self, config: CKANConfig):
        toolkit.add_template_directory(config, 'templates')
```

Add a template file `ckanext-itranslation/templates/home/index.html` containing:

```
{% ckan_extends %}

{% block primary_content %}
{% trans %}This is an untranslated string{% endtrans %}
{% endblock %}
```

This template provides a sample string that we will internationalize in this tutorial.

Note: While this tutorial only covers Python/Jinja templates it is also possible (since CKAN 2.7) to *translate strings in an extension's JavaScript modules*.

5.10.1 Extract strings

Tip: If you have generated a new extension whilst following this tutorial the default template will have generated these files for you and you can simply run the `extract_messages` command immediately.

Check your `setup.py` file in your extension for the following lines

```
setup(
    entry_points='''
        [ckan.plugins]
```

(continues on next page)

(continued from previous page)

```
        itranslation=ckanext.itranslation.plugin:ExampleITranslationPlugin
        [babel.extractors]
        ckan = ckan.lib.extract:extract_ckan
    '''

    message_extractors={
        'ckanext': [
            ('**.py', 'python', None),
            ('**.js', 'javascript', None),
            ('**/templates/**.html', 'ckan', None),
        ],
    }
```

These lines will already be present in our example, but if you are adding internationalization to an older extension, you may need to add them. If you have your templates in a directory differing from the default location (`ckanext/yourplugin/i18n`), you may need to change the `message_extractors` stanza. You can read more about message extractors in the babel documentation.

Add a directory to store your translations:

```
mkdir ckanext-itranslations/ckanext/itranslations/i18n
```

Next you will need a babel config file. Add a `setup.cfg` file containing the following (make sure you replace `itranslations` with the name of your extension):

```
[extract_messages]
keywords = translate isPlural
add_comments = TRANSLATORS:
output_file = ckanext/itranslation/i18n/ckanext-itranslation.pot
width = 80

[init_catalog]
domain = ckanext-itranslation
input_file = ckanext/itranslation/i18n/ckanext-itranslation.pot
output_dir = ckanext/itranslation/i18n

[update_catalog]
domain = ckanext-itranslation
input_file = ckanext/itranslation/i18n/ckanext-itranslation.pot
output_dir = ckanext/itranslation/i18n

[compile_catalog]
domain = ckanext-itranslation
directory = ckanext/itranslation/i18n
statistics = true
```

This file tells babel where the translation files are stored. You can then run the `extract_messages` command to extract the strings from your extension:

```
python setup.py extract_messages
```

This will create a template PO file named `ckanext/itranslations/i18n/ckanext-itranslation.pot`.

At this point, you can either upload and manage your translations using Transifex or manually edit your translations.

5.10.2 Manually create translations

We will create translation files for the `fr` locale. Create the translation PO files for the locale that you are translating for by running init_catalog:

```
python setup.py init_catalog -l fr
```

This will generate a file called `i18n/fr/LC_MESSAGES/ckanext-itranslation.po`. This file should contain the untranslated string on our template. You can manually add a translation for it by editing the `msgstr` section:

```
msgid "This is an untranslated string"
msgstr "This is a itranslated string"
```

5.10.3 Translations with Transifex

Once you have created your translations, you can manage them using Transifex. This is out side of the scope of this tutorial, but the Transifex documentation provides tutorials on how to upload translations and how to manage them using the command line client.

5.10.4 Compiling the catalog

Once the translation files (`po`) have been updated, either manually or via Transifex, compile them by running:

```
python setup.py compile_catalog
```

This will generate a `mo` file containing your translations that can be used by CKAN.

5.10.5 The ITranslation interface

Once you have created the translated strings, you will need to inform CKAN that your extension is translated by implementing the `ITranslation` interface in your extension. Edit your `plugin.py` to contain the following.

```
# encoding: utf-8

from ckan.common import CKANConfig
from ckan import plugins
from ckan.plugins import toolkit
from ckan.lib.plugins import DefaultTranslation


class ExampleITranslationPlugin(plugins.SingletonPlugin, DefaultTranslation):
    plugins.implements(plugins.ITranslation)
    plugins.implements(plugins.IConfigurer)

    def update_config(self, config: CKANConfig):
        toolkit.add_template_directory(config, 'templates')
```

You're done! To test your translated extension, make sure you add the extension to your /etc/ckan/default/ckan.ini, run a `ckan run` command and browse to http://localhost:5000. You should find that switching to the `fr` locale in the web interface will change the home page string to `this is an itranslated string`.

Advanced ITranslation usage

If you are translating a CKAN extension that already exists, or you have structured your extension differently from the default layout. You may have to tell CKAN where to locate your translated files, you can do this by not having your plugin inherit from the `DefaultTranslation` class and instead implement the `ITranslation` interface yourself.

i18n_directory()	Change the directory of the .mo translation files
i18n_locales()	Change the list of locales that this plugin handles
i18n_domain()	Change the gettext domain handled by this plugin

5.11 Migration from Pylons to Flask

On CKAN 2.6, work started to migrate from the Pylons web framework to a more modern alternative, Flask. This will be a gradual process spanning multiple CKAN versions, where both the Pylons app and the Flask app will live side by side with their own controllers or blueprints which handle the incoming requests. The idea is that any other lower level code, like templates, logic actions and authorization are shared between them as much as possible. You can learn more about the approach followed and the work already done on this page in the CKAN wiki:

https://github.com/ckan/ckan/wiki/Migration-from-Pylons-to-Flask

This page lists changes and deprecations that both core and extensions developers should be aware of going forward, as well as common exceptions and how to fix them.

5.11.1 Always import methods and objects from the plugins toolkit if available

This is a *good practice in general* when writing extensions but in the context of the Flask migration it becomes specially important with these methods and objects:

```
from ckan.plugins.toolkit import url_for, redirect_to, request, config

url_for()
redirect_to()
request
config
```

The reason is that these are actually wrappers provided by CKAN that will proxy the call to the relevant Pylons or Flask underlying object or method depending on who is handling the request. For instance in the `config` case, if you use `pylons.config` directly from your extension changes in configuration will only be applied to the Pylons application, and the Flask application will be misconfigured.

Note: `config` was added to the plugins toolkit on CKAN 2.6. If your extension needs to target CKAN versions lower and greater than CKAN 2.6 you can use *ckantoolkit <https://github.com/ckan/ckantoolkit>*, a separate package that provides wrappers for cross-version CKAN compatibility:

```
from ckantoolkit import config
```

5.12 Signals

CKAN provides built-in signal support, powered by blinker.

The same library is used by Flask and anything written in the Flask documentation also applies to CKAN. Probably, the most important point:

> Flask comes with a couple of signals and other extensions might provide more. Also keep in mind that signals are intended to notify subscribers and should not encourage subscribers to modify data. You will notice that there are signals that appear to do the same thing like some of the builtin decorators do (eg: request_started is very similar to before_request()). However, there are differences in how they work. The core before_request() handler, for example, is executed in a specific order and is able to abort the request early by returning a response. In contrast all signal handlers are executed in undefined order and do not modify any data.

`ckan.lib.signals` provides two namespaces for signals: `ckan` and `ckanext`. All core signals reside in `ckan`, while signals from extensions (`datastore`, `datapusher`, third-party extensions) are registered under `ckanext`. This is a recommended pattern and nothing prevents developers from creating and using their own namespaces.

Signal subscribers **MUST** always be defined as callable accepting one mandatory argument *sender* and arbitary number of keyword arguments:

```
def subscriber(sender, **kwargs):
    ...
```

CKAN core doesn't make any guarantees as for the concrete named arguments that will be passed to subscriber. For particular CKAN version one can use signlal-listing below as a reference, but in future versions signature may change. In additon, any event can be fired by a third-party plugin, so it is always safer to check whether a particular argument is available inisde the provided *kwargs*.

Even though it is possible to register subscribers using decorators:

```
@p.toolkit.signals.before_action.connect
def action_subscriber(sender, **kwargs):
    pass
```

the recommended approach is to use the *ckan.plugins.interfaces.ISignal* interface, in order to give CKAN more control over the subscriptions available depending on the enabled plugins:

```
class ExampleISignalPlugin(p.SingletonPlugin):
    p.implements(p.ISignal)

    def get_signal_subscriptions(self):
        return {
            p.toolkit.signals.before_action: [
                # when subscribing to every signal of type
                action_subscriber,

                # when subscribing to signals from particular sender
                {u'receiver': action_subscriber, u'sender': 'sender_name'}
            ]
        }
```

> **Warning:** Arguments passed to subscribers should never be modified. Use subscribers only to trigger side effects and not to change existing CKAN behavior. If one needs to alter CKAN behavior use *ckan.plugins.interfaces* instead.

There are a number of built-in signals in CKAN (check the list at the bottom of the page). All of them are created inside one of the available namespaces: `ckan` and `ckanext`. For simplicity sake, all built in signals have aliases inside `ckan.lib.signals` (or `ckan.plugins.toolkit.signals`, or `ckantoolkit.signals`), but you can always get signals directly from corresponding the namespace (you shouldn't use this directly unless you are familiar with the `blinker` library):

```
from ckan.lib.signals import (
    ckan as ckan_namespace,
    register_blueprint, request_started
)
assert register_blueprint is ckan_namespace.signal('register_blueprint')
assert request_started is ckan_namespace.signal('request_started')
```

This information may be quite handy, if you want to define custom signals inside your extension. Just use `ckanext` namespace and call its method `signal` in order to create a new signal (or get an existing one). In order to avoid name collisions and unexpected behavior, always use your plugin's name as prefix for the signal.:

```
# ckanext-custom/ckanext/custom/signals.py
import ckan.plugins.toolkit as tk

# create signal and use it somewhere inside your extension
custom_something_happened = tk.signals.ckanext.signal('custom_something_happened)

# after this, you can notify subscribers using following code:
custom_signal_happened.send(SENDER, ARG1=VALUE1, ARG2=VALUE2, ...)
```

From now on, everyone who is using your extension can subscribe to your signal from another extension:

```
# ckanext-ext/ckanext/ext/plugin.py
import ckan.plugins as p
from ckanext.custom.signals import custom_something_happened
from ckanext.ext import listeners # here you'll define listeners

class ExtPlugin(p.SingletonPlugin):
    p.implements(p.ISignal)

    def get_signal_subscriptions(self):
        return {
            custom_something_happened: [
                listeners.custom_listener
            ]
        }
```

There is a small problem in snippet above. If `ckanext-custom` is not installed, you'll get `ImportError`. This is perfectly fine if you are sure that you are using `ckanext-custom`, but may be a problem for some general-use plugin. To avoid this, import signals from the `ckanext` namespace instead:

```
# ckanext-ext/ckanext/ext/plugin.py
import ckan.plugins as p
```

(continues on next page)

(continued from previous page)

```
from ckanext.ext import listeners

class ExtPlugin(p.SingletonPlugin):
    p.implements(p.ISignal)

    def get_signal_subscriptions(self):
        custom_something_happened = p.toolkit.signals.ckanext.signal(
            'custom_something_happened'
        )

        return {
            custom_something_happened: [
                listeners.custom_listener
            ]
        }
```

All signals are singletons inside their namespace. If `ckanext-custom` is installed, you'll get its existing signal, otherwise you'll create a new signal that is never sent. So your subscription will work only when `ckanext-custom` is available and do nothing otherwise.

`ckan.lib.signals` contains a few core signals for plugins to subscribe:

`ckan.lib.signals.`**`request_started (app)`**

This signal is sent when the request context is set up, before any request processing happens.

`ckan.lib.signals.`**`request_finished (app, response)`**

This signal is sent right before the response is sent to the client.

`ckan.lib.signals.`**`register_blueprint (blueprint_type, blueprint)`**

This signal is sent when a blueprint for dataset/resource/group/organization is going to be registered inside the application.

`ckan.lib.signals.`**`resource_download (resource_id)`**

This signal is sent just before a file from an uploaded resource is sent to the user.

`ckan.lib.signals.`**`user_logged_in (app, user)`**

Sent when a user is logged in.

`ckan.lib.signals.`**`user_logged_out (app, user)`**

Sent when a user is logged out

`ckan.lib.signals.`**`failed_login (username)`**

This signal is sent after failed login attempt.

`ckan.lib.signals.`**`user_created (username, user)`**

This signal is sent when new user created.

`ckan.lib.signals.`**`request_password_reset (username, user)`**

This signal is sent just after mail with password reset link sent to user.

`ckan.lib.signals.`**`perform_password_reset (username, user)`**

This signal is sent when user submitted password reset form providing new password.

`ckan.lib.signals.`**`action_succeeded (action, context, data_dict, result)`**

This signal is sent when an action finished without an exception.

`ckan.lib.signals.`**`datastore_upsert (resource_id, records)`**

This signal is sent after datasetore records inserted/updated via *datastore_upsert*.

`ckan.lib.signals.`**`datastore_delete (resource_id, result, data_dict)`**

This signal is sent after successful call to *datastore_delete*.

5.13 Customizing the DataStore Data Dictionary Form

Extensions can customize the Data Dictionary form, keys available and values stored for each column using the `IDataDictionaryForm` interface.

class `ckanext.datastore.interfaces.`**`IDataDictionaryForm`**

Allow data dictionary validation and per-plugin data storage by extending the datastore_create schema and adding values to fields returned from datastore_info

`update_datastore_create_schema`(*schema: Schema*) → Schema

Return a modified schema for handling field input in the data dictionary form and datastore_create parameters.

Validators are provided a *plugin_data* dict in the context that can be used to store per-field values. Top-level keys in this dict should match the field index, second-level keys should match the plugin name and values should be a dict with string keys storing data for that plugin.

e.g. a statistics plugin that needs to store per-column information might store this with plugin_data by inserting values like:

```
{0: {'statistics': {'minimum': 34, ...}, ...}, ...}

#                    ^ the data stored for this field+plugin
#     ^ the name of the plugin
#^ 0 for the first field passed in fields
```

Values not removed from field info by validation will be available in the field *info* dict returned from *datastore_search* and *datastore_info*

`update_datastore_info_field`(*field: dict[str, Any]*, *plugin_data: dict[str, Any]*)

Return a modified version of the *datastore_info* field dict based on this field's plugin_data to provide additional information to users and existing values for new form fields in the data dictionary page.

Let's add five new keys with custom validation rules to the data dictionary fields.

With this plugin enabled each field in the Data Dictionary form will have an input for:

- an integer value
- a JSON object
- a numeric value that can only be increased when edited
- a "sticky" value that will not be removed if left blank
- a secret value that will be stored but never displayed in the form.

First extend the form template to render the form inputs:

```
{% ckan_extends %}

{% block additional_fields %}
```

(continues on next page)

(continued from previous page)

```
  {{ form.input('fields__' ~ position ~ '__an_int',
    label=_('An integer'), id='example-plugin-f' ~ position ~ 'an_int',
    value=data.get('an_int', field.get('an_int', '')),
    classes=['control-full'], error=errors.an_int) }}

  {{ form.input('fields__' ~ position ~ '__json_obj',
    label=_('JSON object'), id='example-plugin-f' ~ position ~ 'json_obj',
    value=
      data.json_obj if 'json_obj' in data else
      h.dump_json(field['json_obj']) if 'json_obj' in field else '',
    classes=['control-full'], error=errors.json_obj) }}

  {{ form.input('fields__' ~ position ~ '__only_up',
    label=_('Always increasing'), id='example-plugin-f' ~ position ~ 'only_up',
    value=data.get('only_up', field.get('only_up', '')),
    classes=['control-full'], error=errors.only_up) }}

  {{ form.input('fields__' ~ position ~ '__sticky',
    label=_('Sticky input'), id='example-plugin-f' ~ position ~ 'sticky',
    value=data.get('sticky', field.get('sticky', '')),
    classes=['control-full'], error=errors.sticky) }}

  {{ form.input('fields__' ~ position ~ '__secret',
    label=_('Secret (write-only)'),
    id='example-plugin-f' ~ position ~ 'secret',
    value='', classes=['control-full'],
    error=errors.secret) }}
{% endblock %}
```

We use the `form.input` macro to render the form fields. The name of each field starts with `fields__` and includes a `position` index because this block will be rendered once for every field in the data dictionary.

The value for each input is set to either the value from `data` the text data passed when re-rendering a form containing errors, or `field` the json value (text, number, object etc.) currently stored in the data dictionary when rendering a form for the first time.

The error for each field is set from `errors`.

Next we create a plugin to apply the template and validation rules for each data dictionary field key.

```
# encoding: utf-8

from __future__ import annotations

from typing import Any, cast
from ckan.types import Schema, ValidatorFactory
from ckan.common import CKANConfig
from ckan.lib.navl.dictization_functions import missing
from ckan.types import (
    Context, FlattenDataDict, FlattenErrorDict, FlattenKey,
)

import json
```

(continues on next page)

(continued from previous page)

```
from ckan.plugins.toolkit import (
    Invalid, get_validator, add_template_directory, _
)
from ckan import plugins
from ckanext.datastore.interfaces import IDataDictionaryForm


class ExampleIDataDictionaryFormPlugin(plugins.SingletonPlugin):
    plugins.implements(IDataDictionaryForm)
    plugins.implements(plugins.IConfigurer)

    # IConfigurer

    def update_config(self, config: CKANConfig):
        add_template_directory(config, 'templates')

    # IDataDictionaryForm

    def update_datastore_create_schema(self, schema: Schema):
        ignore_empty = get_validator('ignore_empty')
        int_validator = get_validator('int_validator')
        unicode_only = get_validator('unicode_only')
        datastore_default_current = get_validator('datastore_default_current')
        to_datastore_plugin_data = cast(
            ValidatorFactory, get_validator('to_datastore_plugin_data'))
        to_eg_iddf = to_datastore_plugin_data('example_idatadictionaryform')

        f = cast(Schema, schema['fields'])
        f['an_int'] = [ignore_empty, int_validator, to_eg_iddf]
        f['json_obj'] = [ignore_empty, json_obj, to_eg_iddf]
        f['only_up'] = [
            only_increasing, ignore_empty, int_validator, to_eg_iddf]
        f['sticky'] = [
            datastore_default_current, ignore_empty, unicode_only, to_eg_iddf]

        # use different plugin_key so that value isn't removed
        # when above fields are updated & value not exposed in
        # datastore_info
        f['secret'] = [
            ignore_empty,
            to_datastore_plugin_data('example_idatadictionaryform_secret')
        ]
        return schema

    def update_datastore_info_field(
            self, field: dict[str, Any], plugin_data: dict[str, Any]):
        # expose all our non-secret plugin data in the field
        field.update(plugin_data.get('example_idatadictionaryform', {}))
        return field


def json_obj(value: str | dict[str, Any]) -> dict[str, Any]:
```

(continues on next page)

(continued from previous page)

```
    '''accept only json objects i.e. dicts or "{...}"'''
    try:
        if isinstance(value, str):
            value = json.loads(value)
        else:
            json.dumps(value)
        if not isinstance(value, dict):
            raise TypeError
        return value
    except (TypeError, ValueError):
        raise Invalid(_('Not a JSON object'))


def only_increasing(
        key: FlattenKey, data: FlattenDataDict,
        errors: FlattenErrorDict, context: Context):
    '''once set only accept new values larger than current value'''
    value = data[key]
    field_index = key[-2]
    field_name = key[-1]
    # current values for plugin_data are available as
    # context['plugin_data'][field_index]['_current']
    current = context['plugin_data'].get(field_index, {}).get(
        '_current', {}).get('example_idatadictionaryform', {}).get(
        field_name)
    if current is None:
        return
    if value is not None and value != '' and value is not missing:
        try:
            if int(value) < current:
                errors[key].append(
                    _('Value must be larger than %d') % current)
        except ValueError:
            return  # allow int_validator to handle the error
    else:
        # keep current value when empty/missing
        data[key] = current
```

In `update_datastore_create_schema` the `to_datastore_plugin_data` factory generates a validator that will store our new keys as plugin data. The string passed is used to group keys for this plugin to allow multiple separate `IDataDictionaryForm` plugins to store data for Data Dictionary fields at the same time. It's possible to use multiple groups from the same plugin: here we use a different group for the `secret` key because we want to treat it differently.

In `update_datastore_info_field` we can add keys stored as plugin data to the `fields` objects returned by `datastore_info`. Here we add everything but the `secret` key. These values are also passed to the form template above as `field`.

CHAPTER

SIX

THEMING GUIDE

The following sections will teach you how to customize the content and appearance of CKAN pages by developing your own CKAN themes.

See also:

Getting started
If you just want to do some simple customizations such as changing the title of your CKAN site, or making some small CSS customizations, *Getting started* documents some simple configuration settings you can use.

Note: Before you can start developing a CKAN theme, you'll need a working source install of CKAN on your system. If you don't have a CKAN source install already, follow the instructions in *Installing CKAN from source* before continuing.

Note: CKAN theme development is a technical topic, for web developers. The tutorials below assume basic knowledge of:

- The Python programming language
- HTML
- CSS
- JavaScript

We also recommend familiarizing yourself with:

- Jinja2 templates
- Bootstrap
- jQuery

Note: Starting from CKAN version 2.10 the Bootstrap version used in the default CKAN theme is Bootstrap 5. For backwards compatibility, Bootstrap 3 templates will be included in CKAN core for a few versions, but they will be eventually removed so you are encouraged to update your custom theme to use Bootstrap 5. You can select which set of templates to use (Bootstrap 5 or 3) by using the *ckan.base_public_folder* and *ckan.base_templates_folder* configuration options.

6.1 Customizing CKAN's templates

CKAN pages are generated from Jinja2 template files. This tutorial will walk you through the process of writing your own template files to modify and replace the default ones, and change the layout and content of CKAN pages.

See also:

String internationalization
How to mark strings for translation in your template files.

6.1.1 Creating a CKAN extension

A CKAN theme is simply a CKAN plugin that contains some custom templates and static files, so before getting started on our CKAN theme we'll have to create an extension and plugin. For a detailed explanation of the steps below, see *Writing extensions tutorial*.

1. Use the `ckan generate extension` command as per the *Writing extensions tutorial*.
2. Create the file `ckanext-example_theme/ckanext/example_theme/plugin.py` with the following contents:

```
# encoding: utf-8

import ckan.plugins as plugins


class ExampleThemePlugin(plugins.SingletonPlugin):
    '''An example theme plugin.

    '''
    pass
```

3. Edit the `entry_points` in `ckanext-example_theme/setup.py` to look like this:

```
entry_points='''
    [ckan.plugins]
    example_theme=ckanext.example_theme.plugin:ExampleThemePlugin
''',
```

4. Run `python setup.py develop`:

```
cd ckanext-example_theme
python setup.py develop
```

5. Add the plugin to the `ckan.plugins` setting in your /etc/ckan/default/ckan.ini file:

```
ckan.plugins = stats text_view datatables_view example_theme
```

6. Start CKAN in the development web server:

```
$ ckan -c /etc/ckan/default/ckan.ini run
Starting server in PID 13961.
serving on 0.0.0.0:5000 view at http://127.0.0.1:5000
```

Open the CKAN front page in your web browser. If your plugin is in the *ckan.plugins* setting and CKAN starts without crashing, then your plugin is installed and CKAN can find it. Of course, your plugin doesn't *do* anything yet.

6.1.2 Replacing a default template file

Every CKAN page is generated by rendering a particular template. For each page of a CKAN site there's a corresponding template file. For example the front page is generated from the ckan/templates/home/index.html file, the `/about` page is generated from ckan/templates/home/about.html, the datasets page at `/dataset` is generated from ckan/templates/package/search.html, etc.

To customize pages, our plugin needs to register its own custom template directory containing template files that override the default ones. Edit the `ckanext-example_theme/ckanext/example_theme/plugin.py` file that we created earlier, so that it looks like this:

```
# encoding: utf-8

'''plugin.py

'''
from ckan.common import CKANConfig
import ckan.plugins as plugins
import ckan.plugins.toolkit as toolkit


class ExampleThemePlugin(plugins.SingletonPlugin):
    '''An example theme plugin.

    '''
    # Declare that this class implements IConfigurer.
    plugins.implements(plugins.IConfigurer)

    def update_config(self, config: CKANConfig):

        # Add this plugin's templates dir to CKAN's extra_template_paths, so
        # that CKAN will use this plugin's custom templates.
        # 'templates' is the path to the templates dir, relative to this
        # plugin.py file.
        toolkit.add_template_directory(config, 'templates')
```

This new code does a few things:

1. It imports CKAN's *plugins toolkit* module:

   ```
   import ckan.plugins.toolkit as toolkit
   ```

   The plugins toolkit is a Python module containing core functions, classes and exceptions for CKAN plugins to use. For more about the plugins toolkit, see *Writing extensions tutorial*.

2. It calls `implements()` to declare that it implements the `IConfigurer` plugin interface:

   ```
       plugins.implements(plugins.IConfigurer)
   ```

   This tells CKAN that our `ExampleThemePlugin` class implements the methods declared in the `IConfigurer` interface. CKAN will call these methods of our plugin class at the appropriate times.

3. It implements the `update_config()` method, which is the only method declared in the `IConfigurer` interface:

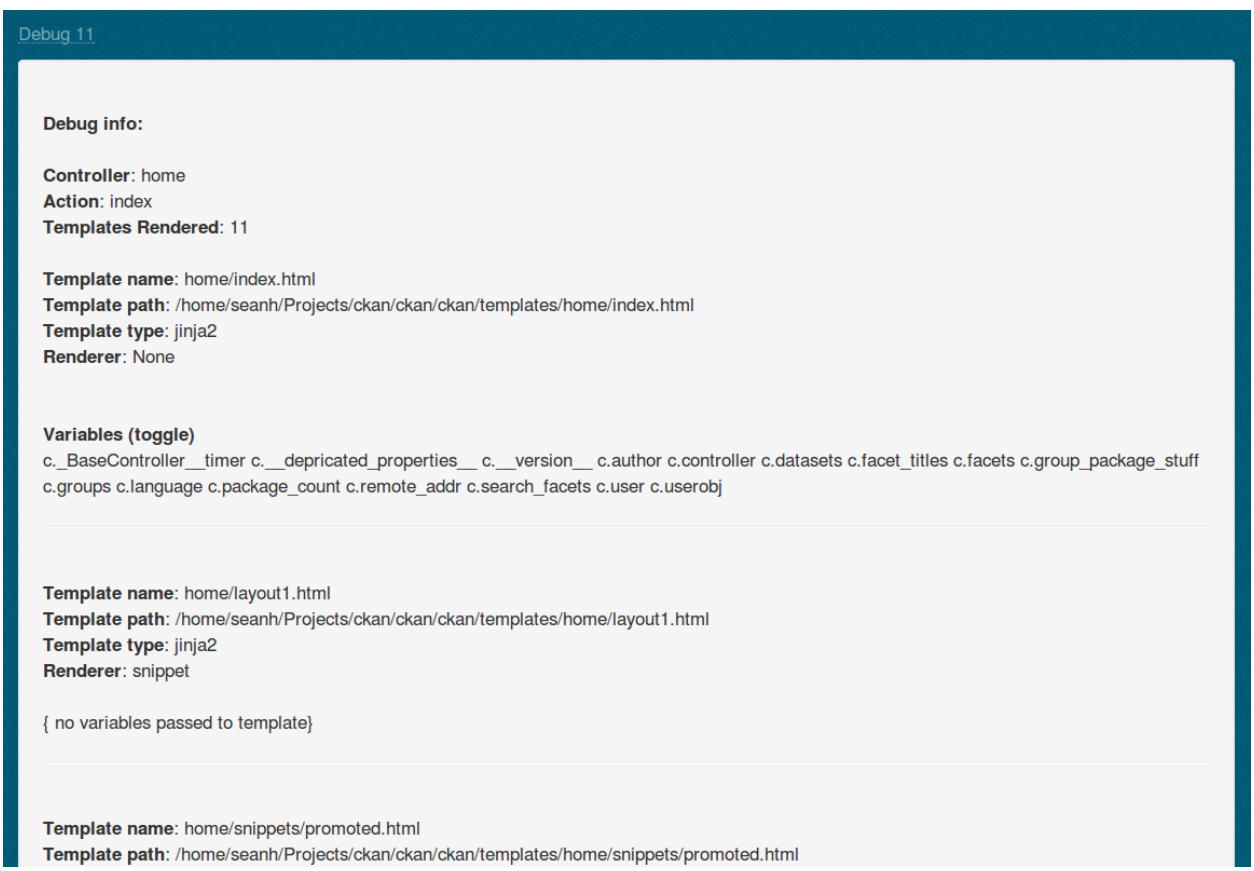
```
    def update_config(self, config: CKANConfig):

        # Add this plugin's templates dir to CKAN's extra_template_paths, so
        # that CKAN will use this plugin's custom templates.
        # 'templates' is the path to the templates dir, relative to this
        # plugin.py file.
        toolkit.add_template_directory(config, 'templates')
```

CKAN will call this method when it starts up, to give our plugin a chance to modify CKAN's configuration settings. Our `update_config()` method calls `add_template_directory()` to register its custom template directory with CKAN. This tells CKAN to look for template files in `ckanext-example_theme/ckanext/example_theme/templates` whenever it renders a page. Any template file in this directory that has the same name as one of CKAN's default template files, will be used instead of the default file.

Now, let's customize the CKAN front page. We first need to discover which template file CKAN uses to render the front page, so we can replace it. Set *debug* to `true` in your /etc/ckan/default/ckan.ini file:

```
[DEFAULT]

# WARNING: *THIS SETTING MUST BE SET TO FALSE ON A PRODUCTION ENVIRONMENT*
debug = true
```

Reload the CKAN front page in your browser, and you should see a *Debug* link in the footer at the bottom of the page. Click on this link to open the debug footer. The debug footer displays various information useful for CKAN frontend development and debugging, including the names of the template files that were used to render the current page:



The first template file listed is the one we're interested in:

```
Template name: home/index.html
Template path: /usr/lib/ckan/default/src/ckan/ckan/templates/home/index.html
```

This tells us that `home/index.html` is the root template file used to render the front page. The debug footer appears at the bottom of every CKAN page, and can always be used to find the page's template files, and other information about the page.

Note: Most CKAN pages are rendered from multiple template files. The first file listed in the debug footer is the root template file of the page. All other template files used to render the page (listed further down in the debug footer) are either included by the root file, or included by another file that is included by the root file.

To figure out which template file renders a particular part of the page you have to inspect the source code of the template files, starting with the root file.

Now let's override `home/index.html` using our plugins' custom `templates` directory. Create the `ckanext-example_theme/ckanext/example_theme/templates` directory, create a `home` directory inside the `templates` directory, and create an empty `index.html` file inside the `home` directory:

```
ckanext-example_theme/
  ckanext/
    example_theme/
      templates/
        home/
          index.html  <-- An empty file.
```

If you now restart the development web server (kill the server using Ctrl-c, then run the `ckan run` command again) and reload the CKAN front page in your web browser, you should see an empty page, because we've replaced the template file for the front page with an empty file.

Note: If you run `ckan run` without the `-r(--disable-reloader)` option, then it isn't usually necessary to restart the server after editing a Python file, a template file, your CKAN config file, or any other CKAN file. If you've added a new file or directory, however, you need to restart the server manually.

6.1.3 Jinja2

CKAN template files are written in the Jinja2 templating language. Jinja template files, such as our `index.html` file, are simply text files that, when processed, generate any text-based output format such as `HTML`, `XML`, `CSV`, etc. Most of the template files in CKAN generate `HTML`.

We'll introduce some Jinja2 basics below. Jinja2 templates have many more features than these, for full details see the Jinja2 docs.

Expressions and variables

Jinja2 *expressions* are snippets of code between `{{ ... }}` delimiters, when a template is rendered any expressions are evaluated and replaced with the resulting value.

The simplest use of an expression is to display the value of a variable, for example `{{ foo }}` in a template file will be replaced with the value of the variable `foo` when the template is rendered.

CKAN makes a number of global variables available to all templates. One such variable is `app_globals`, which can be used to access certain global attributes including some of the settings from your CKAN config file. For example, to display the value of the *ckan.site_title* setting from your config file you would put this code in any template file:

```
<p>The title of this site is: {{ app_globals.site_title }}.</p>
```

Note: The `app_globals` variable is also sometimes called `g` (an alias), you may see `g` in some CKAN templates. See *Variables and functions available to templates*.

Note: Not all config settings are available to templates via `app_globals`. The *sqlalchemy.url* setting, for example, contains your database password, so making that variable available to templates might be a security risk.

If you've added your own custom options to your config file, these will not be available in `app_globals` automatically. See *Accessing custom config settings from templates*.

Note: If a template tries to render a variable or attribute that doesn't exist, rather than crashing or giving an error message, the Jinja2 expression simply evaluates to nothing (an empty string). For example, these Jinja2 expressions will output nothing:

```
{{ app_globals.an_attribute_that_does_not_exist }}

{{ a_variable_that_does_not_exist }}
```

If, on the other hand, you try to render an attribute of a variable that doesn't exist, then Jinja2 will crash. For example, this Jinja2 expression will crash with an `UndefinedError: 'a_variable_that_does_not_exist' is undefined`:

```
{{ a_variable_that_does_not_exist.an_attribute_that_does_not_exist }}
```

See the Jinja2 variables docs for details.

Note: Jinja2 expressions can do much more than print out the values of variables, for example they can call Jinja2's global functions, CKAN's *template helper functions* and any *custom template helper functions* provided by your extension, and use any of the literals and operators that Jinja provides.

See *Variables and functions available to templates* for a list of variables and functions available to templates.

Tags

ckan.site_title is an example of a simple string variable. Some variables, such as *ckan.plugins*, are lists, and can be looped over using Jinja's `{% for %}` tag.

Jinja *tags* are snippets of code between `{% ... %}` delimiters that control the logic of the template. For example, we can output a list of the currently enabled plugins with this code in any template file:

```
<p>The currently enabled plugins are:</p>
<ul>
  {% for plugin in app_globals.plugins %}
    <li>{{ plugin }}</li>
  {% endfor %}
</ul>
```

Other boolean variables can be tested using Jinja's `{% if %}` tag:

```
{% if g.tracking_enabled %}
  <p>CKAN's page-view tracking feature is enabled.</p>
{% else %}
  <p>CKAN's page-view tracking feature is <i>not</i> enabled.</p>
{% endif %}
```

Comments

Finally, any text between `{# ... #}` delimiters in a Jinja2 template is a *comment*, and will not be output when the template is rendered:

```
{# This text will not appear in the output when this template is rendered. #}
```

6.1.4 Extending templates with `{% ckan_extends %}`

CKAN provides a custom Jinja tag `{% ckan_extends %}` that we can use to declare that our `home/index.html` template extends the default `home/index.html` template, instead of completely replacing it. Edit the empty `index.html` file you just created, and add one line:

```
{% ckan_extends %}
```

If you now reload the CKAN front page in your browser, you should see the normal front page appear again. When CKAN processes our `index.html` file, the `{% ckan_extends %}` tag tells it to process the default `home/index.html` file first.

6.1.5 Replacing template blocks with `{% block %}`

Jinja templates can contain *blocks* that child templates can override. For example, CKAN's default `home/index.html` template (one of the files used to render the CKAN front page) has a block that contains the Jinja and HTML code for the "featured group" that appears on the front page:

```
            {% block featured_group %}
            {% snippet 'home/snippets/featured_group.html' %}
            {% endblock %}
```

Note: This code calls a *template snippet* that contains the actual Jinja and HTML code for the featured group, more on snippets later.

Note: The CKAN front page supports a number of different layouts: layout1, layout2, layout3, etc. The layout can be chosen by a sysadmin using the *admin page*. This tutorial assumes your CKAN is set to use the first (default) layout.

When a custom template file extends one of CKAN's default template files using `{% ckan_extends %}`, it can replace any of the blocks from the default template with its own code by using `{% block %}`. Create the file `ckanext-example_theme/ckanext/example_theme/templates/home/index.html` with these contents:

```
{% ckan_extends %}

{% block featured_group %}
  Hello block world!
{% endblock %}
```

This file extends the default `index.html` template, and overrides the `featured_group` block. Restart the development web server and reload the CKAN front page in your browser. You should see that the featured groups section of the page has been replaced, but the rest of the page remains intact.

Note: Most template files in CKAN contain multiple blocks. To find out what blocks a template has, and which block renders a particular part of the page, you have to look at the source code of the default template files.

6.1.6 Extending parent blocks with Jinja's `{{ super() }}`

If you want to add some code to a block but don't want to replace the entire block, you can use Jinja's `{{ super() }}` tag:

```
{% ckan_extends %}

{% block featured_group %}

  <p>This paragraph will be added to the top of the
  <code>featured_group</code> block.</p>

  {# Insert the contents of the original featured_group block: #}
  {{ super() }}

  <p>This paragraph will be added to the bottom of the
  <code>featured_group</code> block.</p>

{% endblock %}
```

When the child block above is rendered, Jinja will replace the `{{ super() }}` tag with the contents of the parent block. The `{{ super() }}` tag can be placed anywhere in the block.

6.1.7 Template helper functions

Now let's put some interesting content into our custom template block. One way for templates to get content out of CKAN is by calling CKAN's *template helper functions*.

For example, let's replace the featured group on the front page with an activity stream of the site's recently created, updated and deleted datasets. Change the code in `ckanext-example_theme/ckanext/example_theme/templates/home/index.html` to this:

```
{% ckan_extends %}

{% block featured_group %}
  {{ h.recently_changed_packages_activity_stream(limit=4) }}
{% endblock %}
```

Reload the CKAN front page in your browser and you should see a new activity stream:

To call a template helper function we use a Jinja2 *expression* (code wrapped in `{{ ... }}` brackets), and we use the global variable *h* (available to all templates) to access the helper:

```
  {{ h.recently_changed_packages_activity_stream(limit=4) }}
```

To see what other template helper functions are available, look at the *template helper functions reference docs*.

6.1.8 Adding your own template helper functions

Plugins can add their own template helper functions by implementing CKAN's *ITemplateHelpers* plugin interface. (see *Writing extensions tutorial* for a detailed explanation of CKAN plugins and plugin interfaces).

Let's add another item to our custom front page: a list of the most "popular" groups on the site (the groups with the most datasets). We'll add a custom template helper function to select the groups to be shown. First, in our `plugin.py` file we need to implement *ITemplateHelpers* and provide our helper function. Change the contents of `plugin.py` to look like this:

```
# encoding: utf-8

from ckan.common import CKANConfig
import ckan.plugins as plugins
import ckan.plugins.toolkit as toolkit


def most_popular_groups():
    '''Return a sorted list of the groups with the most datasets.'''

    # Get a list of all the site's groups from CKAN, sorted by number of
    # datasets.
    groups = toolkit.get_action('group_list')(
        {}, {'sort': 'package_count desc', 'all_fields': True})

    # Truncate the list to the 10 most popular groups only.
    groups = groups[:10]

    return groups


class ExampleThemePlugin(plugins.SingletonPlugin):
    '''An example theme plugin.

    '''
    plugins.implements(plugins.IConfigurer)

    # Declare that this plugin will implement ITemplateHelpers.
    plugins.implements(plugins.ITemplateHelpers)

    def update_config(self, config: CKANConfig):

        # Add this plugin's templates dir to CKAN's extra_template_paths, so
        # that CKAN will use this plugin's custom templates.
        toolkit.add_template_directory(config, 'templates')

    def get_helpers(self):
        '''Register the most_popular_groups() function above as a template
        helper function.

        '''
        # Template helper function names should begin with the name of the
        # extension they belong to, to avoid clashing with functions from
        # other extensions.
        return {'example_theme_most_popular_groups': most_popular_groups}
```

We've added a number of new features to `plugin.py`. First, we defined a function to get the most popular groups from CKAN:

```
def most_popular_groups():
    '''Return a sorted list of the groups with the most datasets.'''

    # Get a list of all the site's groups from CKAN, sorted by number of
```

(continues on next page)

(continued from previous page)

```
    # datasets.
    groups = toolkit.get_action('group_list')(
        {}, {'sort': 'package_count desc', 'all_fields': True})

    # Truncate the list to the 10 most popular groups only.
    groups = groups[:10]

    return groups
```

This function calls one of CKAN's *action functions* to get the groups from CKAN. See *Writing extensions tutorial* for more about action functions.

Next, we called `implements()` to declare that our class now implements `ITemplateHelpers`:

```
    plugins.implements(plugins.ITemplateHelpers)
```

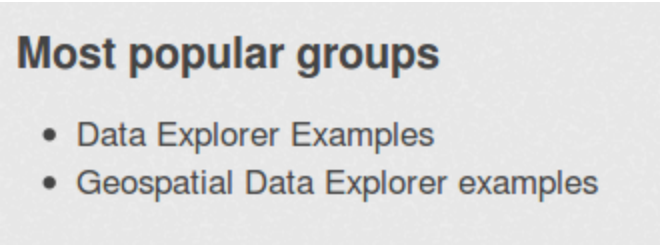
Finally, we implemented the `get_helpers()` method from `ITemplateHelpers` to register our function as a template helper:

```
    def get_helpers(self):
        '''Register the most_popular_groups() function above as a template
        helper function.

        '''
        # Template helper function names should begin with the name of the
        # extension they belong to, to avoid clashing with functions from
        # other extensions.
        return {'example_theme_most_popular_groups': most_popular_groups}
```

Now that we've registered our helper function, we need to call it from our template. As with CKAN's default template helpers, templates access custom helpers via the global variable `h`. Edit `ckanext-example_theme/ckanext/example_theme/templates/home/index.html` to look like this:

```
{% ckan_extends %}

{% block featured_group %}
  {{ h.recently_changed_packages_activity_stream(limit=4) }}
{% endblock %}

{% block featured_organization %}

  {# Show a list of the site's most popular groups. #}
  <h3>Most popular groups</h3>
  <ul>
    {% for group in h.example_theme_most_popular_groups() %}
      <li>{{ group.display_name }}</li>
    {% endfor %}
  </ul>

{% endblock %}
```

Now reload your CKAN front page in your browser. You should see the featured organization section replaced with a list of the most popular groups:



Simply displaying a list of group titles isn't very good. We want the groups to be hyperlinked to their pages, and also to show some other information about the group such as its description and logo image. To display our groups nicely, we'll use CKAN's *template snippets*...

6.1.9 Template snippets

Template snippets are small snippets of template code that, just like helper functions, can be called from any template file. To call a snippet, you use another of CKAN's custom Jinja2 tags: `{% snippet %}`. CKAN comes with a selection of snippets, which you can find in the various `snippets` directories in ckan/templates/, such as ckan/templates/snippets/ and ckan/templates/package/snippets/. For a complete list of the default snippets available to templates, see *Template snippets reference*.

`ckan/templates/group/snippets/group_list.html` is a snippet that renders a list of groups nicely (it's used to render the groups on CKAN's `/group` page and on user dashboard pages, for example):

```
{#
Display a grid of group items.

groups - A list of groups.

Example:

    {% snippet "group/snippets/group_list.html" %}

#}
{% block group_list %}
<ul class="media-grid" data-module="media-grid">
        {% block group_list_inner %}
  {% for group in groups %}
    {% snippet "group/snippets/group_item.html", group=group, position=loop.index, show_
↪capacity=show_capacity %}
  {% endfor %}
  {% endblock %}
</ul>
{% endblock %}
```

(As you can see, this snippet calls another snippet, `group_item.html`, to render each individual group.)

Let's change our `ckanext-example_theme/ckanext/example_theme/templates/home/index.html` file to call this snippet:

```
{% ckan_extends %}

{% block featured_group %}
```

(continues on next page)

(continued from previous page)

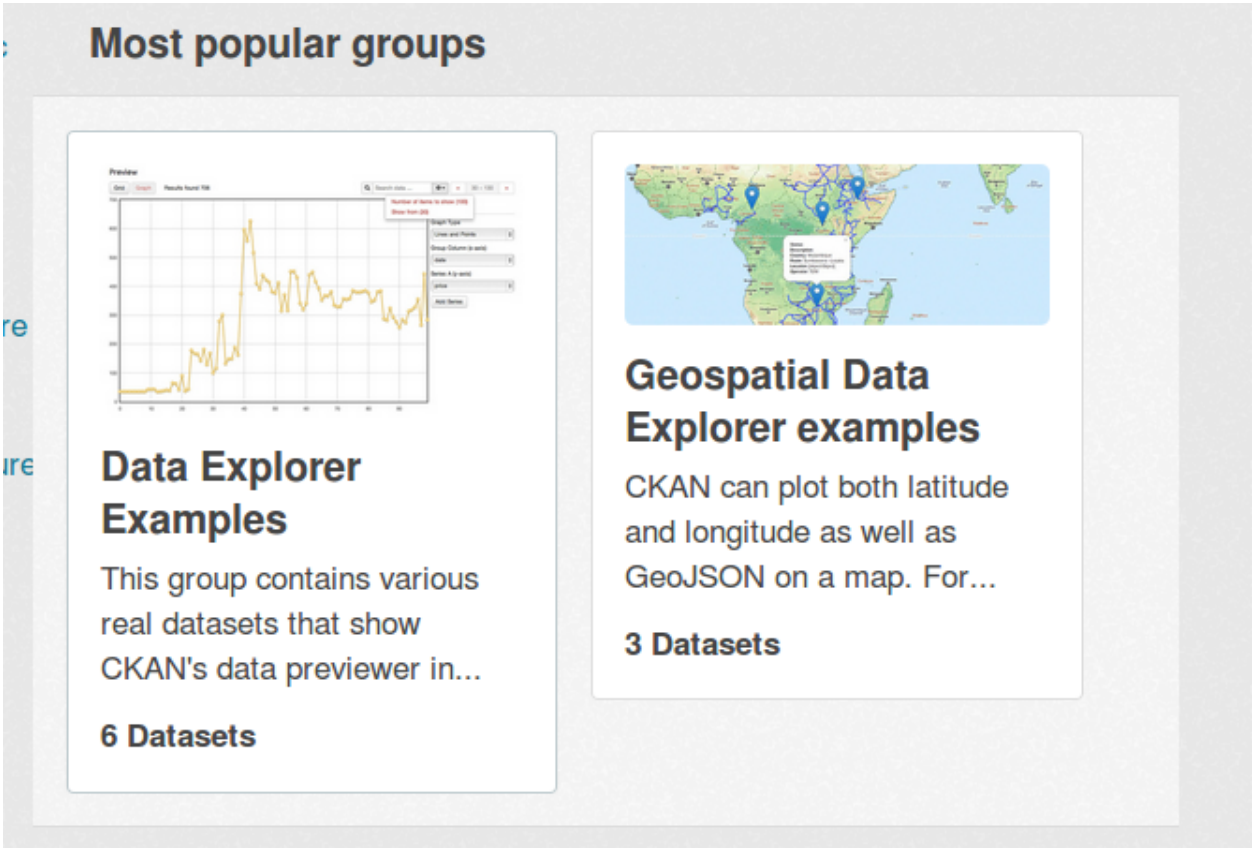
```
  {{ h.recently_changed_packages_activity_stream(limit=4) }}

{% endblock %}

{% block featured_organization %}

  <h3>Most popular groups</h3>

  {# Call the group_list.html snippet. #}
  {% snippet 'group/snippets/group_list.html',
             groups=h.example_theme_most_popular_groups() %}

{% endblock %}
```

Here we pass two arguments to the `{% snippet %}` tag:

```
  {% snippet 'group/snippets/group_list.html',
             groups=h.example_theme_most_popular_groups() %}
```

the first argument is the name of the snippet file to call. The second argument, separated by a comma, is the list of groups to pass into the snippet. After the filename you can pass any number of variables into a snippet, and these will all be available to the snippet code as top-level global variables. As in the `group_list.html` docstring above, each snippet's docstring should document the parameters it requires.

If you reload your CKAN front page in your web browser now, you should see the most popular groups rendered in the same style as the list of groups on the `/groups` page:



This style isn't really what we want for our front page, each group is too big. To render the groups in a custom style, we can define a custom snippet…

6.1.10 Adding your own template snippets

Just as plugins can add their own template helper functions, they can also add their own snippets. To add template snippets, all a plugin needs to do is add a `snippets` directory in its `templates` directory, and start adding files. The snippets will be callable from other templates immediately.

Note: For CKAN to find your plugins' snippets directories, you should already have added your plugin's custom template directory to CKAN, see *Replacing a default template file*.

Let's create a custom snippet to display our most popular groups, we'll put the `<h3>Most popular groups</h3>` heading into the snippet and make it nice and modular, so that we can reuse the whole thing on different parts of the site if we want to.

Create a new directory `ckanext-example_theme/ckanext/example_theme/templates/snippets` containing a file named `example_theme_most_popular_groups.html` with these contents:

```
{#

Renders a list of the site's most popular groups.

groups - the list of groups to render

#}
<h3>Most popular groups</h3>
<ul>
  {% for group in groups %}
    <li>
      <a href="{{ h.url_for('group_read', action='read', id=group.name) }}">
        <h3>{{ group.display_name }}</h3>
      </a>
      {% if group.description %}
        <p>
          {{ h.markdown_extract(group.description, extract_length=80) }}
        </p>
      {% else %}
        <p>{{ _('This group has no description') }}</p>
      {% endif %}
      {% if group.package_count %}
        <strong>{{ ungettext('{num} Dataset', '{num} Datasets', group.package_count).
↪format(num=group.package_count) }}</strong>
      {% else %}
        <span>{{ _('0 Datasets') }}</span>
      {% endif %}
    </li>
  {% endfor %}
</ul>
```

Note: As in the example above, a snippet should have a docstring at the top of the file that briefly documents what the

snippet does and what parameters it requires. See *Snippets should have docstrings*.

This code uses a Jinja2 `for` loop to render each of the groups, and calls a number of CKAN's template helper functions:

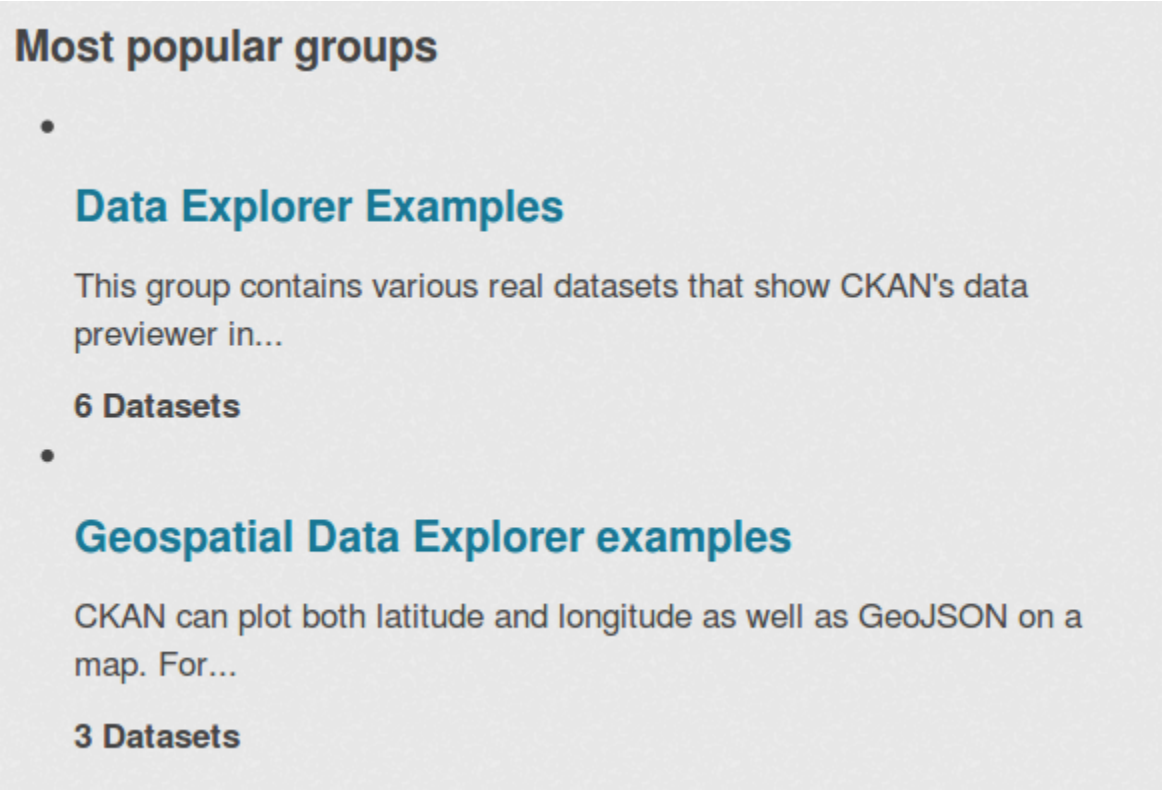
- To hyperlink each group's name to the group's page, it calls `url_for()`.
- If the group has a description, it calls `markdown_extract()` to render the description nicely.
- If the group doesn't have a description, it uses the `_()` function to mark the `'This group has no description'` message for translation. When the page is rendered in a user's web browser, this string will be shown in the user's language (if there's a translation of the string into that language).
- When rendering the group's number of datasets, it uses the `ungettext()` function to mark the message for translation with localized handling of plural forms.

The code also accesses the attributes of each group: `{{ group.name }}`, ``{{ group.display_name }}`, `{{ group.description }}`, `{{ group.package_count }}`, etc. To see what attributes a group or any other CKAN object (packages/datasets, organizations, users...) has, you can use *CKAN's API* to inspect the object. For example to find out what attributes a group has, call the `group_show()` function.

Now edit your `ckanext-example_theme/ckanext/example_theme/templates/home/index.html` file and change it to use our new snippet instead of the default one:

```
{% ckan_extends %}

{% block featured_group %}
  {{ h.recently_changed_packages_activity_stream(limit=4) }}
{% endblock %}

{% block featured_organization %}
  {% snippet 'snippets/example_theme_most_popular_groups.html',
             groups=h.example_theme_most_popular_groups()  %}
{% endblock %}
```

Restart the development web server and reload the CKAN front page and you should see the most popular groups rendered differently:



Warning: Default snippets can be overridden. If a plugin adds a snippet with the same name as one of CKAN's default snippets, the plugin's snippet will override the default snippet wherever the default snippet is used.

Also if two plugins both have snippets with the same name, one of the snippets will override the other.

To avoid unintended conflicts, we recommend that snippet filenames begin with the name of the extension they belong to, e.g. `snippets/example_theme_*.html`. See *Avoid name clashes*.

Note: Snippets don't have access to the global template context variable, `c` (see *Variables and functions available to templates*). Snippets *can* access other global variables such as `h`, `app_globals` and `request`, as well as any variables explicitly passed into the snippet by the parent template when it calls the snippet with a `{% snippet %}` tag.

6.1.11 HTML tags and CSS classes

Our additions to the front page so far don't look very good or fit in very well with the CKAN theme. Let's make them look better by tweaking our template to use the right HTML tags and CSS classes.

There are two places to look for CSS classes available in CKAN:

1. The Bootstrap 3.3.7 docs. All of the HTML, CSS and JavaScript provided by Bootstrap is available to use in CKAN.
2. CKAN's development primer page, which can be found on any CKAN site at `/testing/primer`, for example demo.ckan.org/testing/primer.

   The primer page demonstrates many of the HTML and CSS elements available in CKAN, and by viewing the source of the page you can see what HTML tags and CSS classes they use.

Edit your `example_theme_most_popular_groups.html` file to look like this:

```
{# Renders a list of the site's most popular groups. #}

<div class="box">
  <header class="module-heading">
    <h3>Most popular groups</h3>
  </header>
  <section class="module-content">
    <ul class="unstyled">
      {% for group in h.example_theme_most_popular_groups() %}
        <li>
          <a href="{{ h.url_for('group_read', action='read', id=group.name) }}">
            <h3>{{ group.display_name }}</h3>
          </a>
          {% if group.description %}
            <p>
              {{ h.markdown_extract(group.description, extract_length=80) }}
            </p>
          {% else %}
            <p>{{ _('This group has no description') }}</p>
          {% endif %}
          {% if group.packages %}
            <strong>{{ ungettext('{num} Dataset', '{num} Datasets', group.packages).
↪format(num=group.packages) }}</strong>
          {% else %}
            <span>{{ _('0 Datasets') }}</span>
          {% endif %}
        </li>
      {% endfor %}
    </ul>
  <section>
</div>
```

This simply wraps the code in a `<div class="box">`, a `<header class="module-heading">`, and a `<section class="module-content">`. We also added Bootstrap's `class="unstyled"` to the `<ul>` tag to get rid of the bullet points. If you reload the CKAN front page, the most popular groups should look much better.

To wrap your activity stream in a similar box, edit `index.html` to look like this:

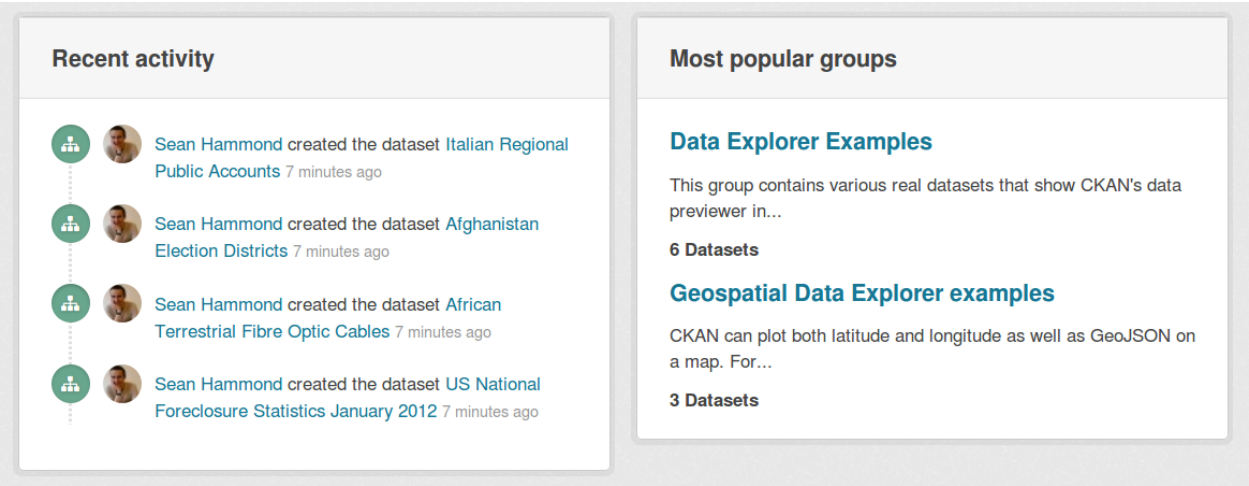
```
{% ckan_extends %}

{% block featured_group %}
  <div class="box">
    <header class="module-heading">
      <h3>Recent activity</h3>
    </header>
    <div class="module-content">
      <ul>
        <li>Activity 01</li>
        <li>Activity 02</li>
        <li>Activity 03</li>
      </ul>
    </div>
```

(continues on next page)

(continued from previous page)

```
  </div>
{% endblock %}

{% block featured_organization %}
  {% snippet 'snippets/example_theme_most_popular_groups.html' %}
{% endblock %}
```

Reload the CKAN front page, and you should see your activity stream and most popular groups looking much better:



6.1.12 Accessing custom config settings from templates

Not all CKAN config settings are available to templates via `app_globals`. In particular, if an extension wants to use its own custom config setting, this setting will not be available. If you need to access a custom config setting from a template, you can do so by wrapping the config setting in a helper function.

See also:

For more on custom config settings, see *Using custom config settings in extensions*.

Todo: I'm not sure if making config settings available to templates like this is a very good idea. Is there an alternative best practice?

Let's add a config setting, `show_most_popular_groups`, to enable or disable the most popular groups on the front page. First, add a new helper function to `plugin.py` to wrap the config setting.

```
# encoding: utf-8
from __future__ import annotations

from typing import Any, Callable
import ckan.plugins as plugins
import ckan.plugins.toolkit as toolkit
from ckan.common import CKANConfig, config
from ckan.config.declaration import Declaration, Key
```

(continues on next page)

(continued from previous page)

```
def show_most_popular_groups():
    '''Return the value of the most_popular_groups config setting.

    To enable showing the most popular groups, add this line to the
    [app:main] section of your CKAN config file::

      ckan.example_theme.show_most_popular_groups = True

    Returns ``False`` by default, if the setting is not in the config file.

    :rtype: bool

    '''
    value = config.get('ckan.example_theme.show_most_popular_groups')
    return value


def most_popular_groups():
    '''Return a sorted list of the groups with the most datasets.'''

    # Get a list of all the site's groups from CKAN, sorted by number of
    # datasets.
    groups = toolkit.get_action('group_list')(
        {}, {'sort': 'package_count desc', 'all_fields': True})

    # Truncate the list to the 10 most popular groups only.
    groups = groups[:10]

    return groups


class ExampleThemePlugin(plugins.SingletonPlugin):
    '''An example theme plugin.

    '''
    plugins.implements(plugins.IConfigurer)
    plugins.implements(plugins.IConfigDeclaration)

    # Declare that this plugin will implement ITemplateHelpers.
    plugins.implements(plugins.ITemplateHelpers)

    def update_config(self, config: CKANConfig):

        # Add this plugin's templates dir to CKAN's extra_template_paths, so
        # that CKAN will use this plugin's custom templates.
        toolkit.add_template_directory(config, 'templates')

    def get_helpers(self) -> dict[str, Callable[..., Any]]:
        '''Register the most_popular_groups() function above as a template
        helper function.

        '''
```

(continues on next page)

(continued from previous page)

```
        # Template helper function names should begin with the name of the
        # extension they belong to, to avoid clashing with functions from
        # other extensions.
        return {'example_theme_most_popular_groups': most_popular_groups,
                'example_theme_show_most_popular_groups':
                show_most_popular_groups,
                }

    # IConfigDeclaration

    def declare_config_options(self, declaration: Declaration, key: Key):
        declaration.declare_bool(
            key.ckan.example_theme.show_most_popular_groups)
```

```
def show_most_popular_groups():
    '''Return the value of the most_popular_groups config setting.

    To enable showing the most popular groups, add this line to the
    [app:main] section of your CKAN config file::

      ckan.example_theme.show_most_popular_groups = True

    Returns ``False`` by default, if the setting is not in the config file.

    :rtype: bool

    '''
    value = config.get('ckan.example_theme.show_most_popular_groups')
    return value
```

Note: Names of config settings provided by extensions should include the name of the extension, to avoid conflicting with core config settings or with config settings from other extensions. See *Avoid name clashes*.

Now we can call this helper function from our `index.html` template:

```
{% block featured_organization %}
  {% if h.example_theme_show_most_popular_groups() %}
    {% snippet 'snippets/example_theme_most_popular_groups.html' %}
  {% else %}
    {{ super() }}
  {% endif %}
{% endblock %}
```

If the user sets this config setting to `True` in their CKAN config file, then the most popular groups will be displayed on the front page, otherwise the block will fall back to its default contents.

6.2 Adding static files

You may need to add some custom *static files* to your CKAN site and use them from your templates, for example image files, PDF files, or any other static files that should be returned as-is by the webserver (as opposed to Jinja template files, which CKAN renders before returning them to the user).

By adding a directory to CKAN's *extra_public_paths* config setting, a plugin can make a directory of static files available to be used or linked to by templates. Let's add a static image file, and change the home page template to use our file as the promoted image on the front page.

See also:

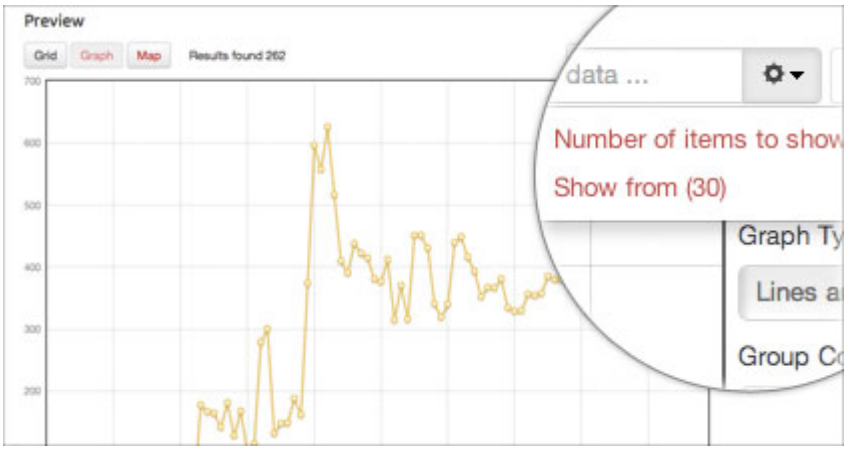
Adding CSS and JavaScript files using Webassets

> If you're adding CSS files consider using Webassets instead of *extra_public_paths*, to take advantage of extra features. See *Adding CSS and JavaScript files using Webassets*. If you're adding JavaScript modules you have to use Webassets, see *Customizing CKAN's JavaScript*.

First, create a `public` directory in your extension with a `promoted-image.jpg` file in it:

```
ckanext-example_theme/
  ckanext/
     example_theme/
        public/
           promoted-image.jpg
```

`promoted-image.jpg` should be a 420x220px JPEG image file. You could use this image file for example:



Then in `plugin.py`, register your `public` directory with CKAN by calling the `add_public_directory()` function. Add this line to the `update_config()` function:

```
    def update_config(self, config: CKANConfig):

        # Add this plugin's templates dir to CKAN's extra_template_paths, so
        # that CKAN will use this plugin's custom templates.
        toolkit.add_template_directory(config, 'templates')

        # Add this plugin's public dir to CKAN's extra_public_paths, so
        # that CKAN will use this plugin's custom static files.
        toolkit.add_public_directory(config, 'public')
```

If you now browse to 127.0.0.1:5000/promoted-image.jpg, you should see your image file.

To replace the image on the front page with your custom image, we need to override the `promoted.html` template snippet. Create the following directory and file:

```
ckanext-example_theme/
  ckanext/
    example_theme/
      templates/
        home/
          snippets/
            promoted.html
```

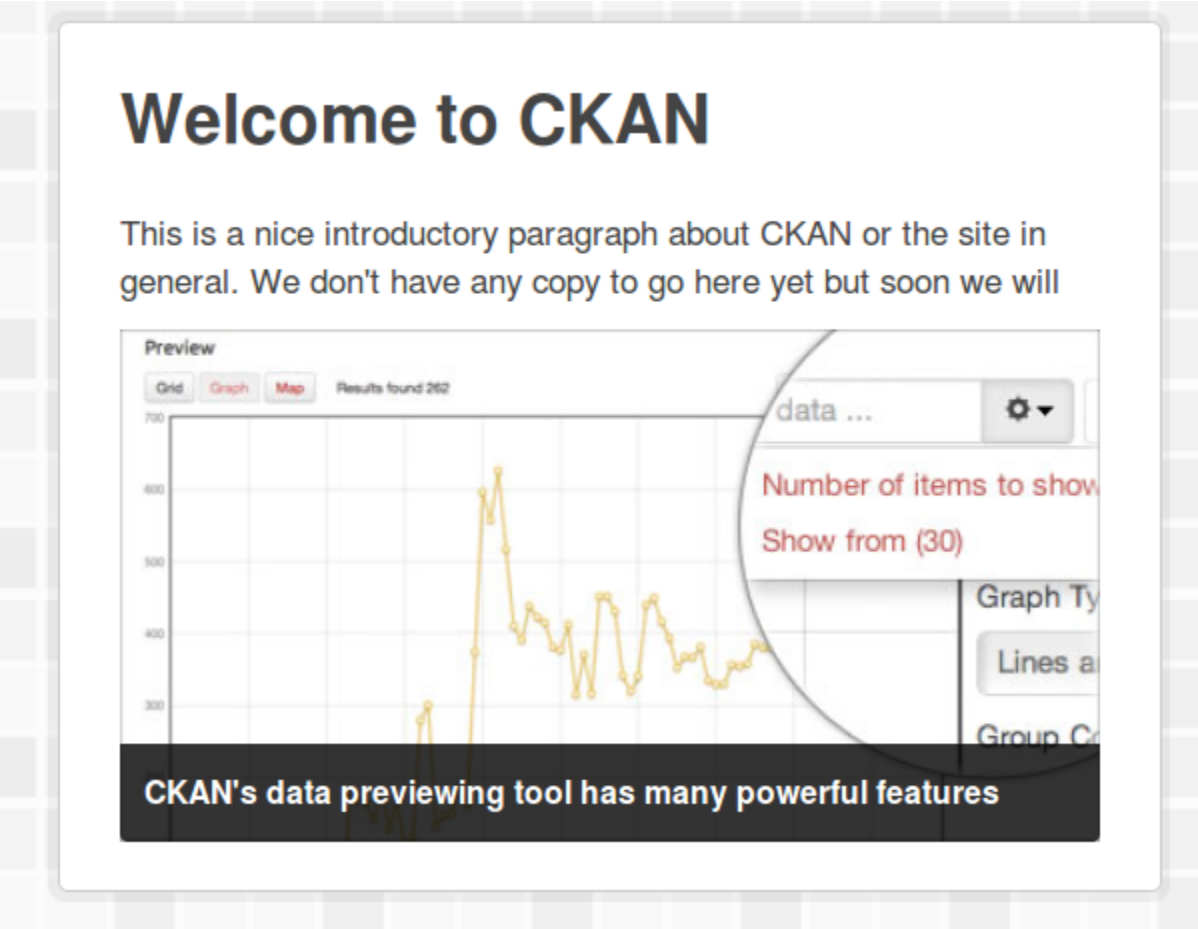
Edit your new `promoted.html` snippet, and insert these contents:

```
{% ckan_extends %}

{% block home_image_caption %}
  {{ _("CKAN's data previewing tool has many powerful features") }}
{% endblock %}

{# Replace the promoted image. #}
{% block home_image_content %}
  <a class="media-image" href="#">
    <img src="/promoted-image.jpg" alt="Featured image"
         width="420" height="220" />
  </a>
{% endblock %}
```

After calling `{% ckan_extends %}` to declare that it extends (rather than completely replaces) the default `promoted.html` snippet, this custom snippet overrides two of `promoted.html`'s template blocks. The first block replaces the caption text of the promoted image. The second block replaces the `<img>` tag itself, pointing it at our custom static image file:

```
{% block home_image_content %}
  <a class="media-image" href="#">
    <img src="/promoted-image.jpg" alt="Featured image"
         width="420" height="220" />
  </a>
{% endblock %}
```

If you now restart the development web server and reload the CKAN front page in your browser, you should see the promoted image replaced with our custom one:



6.3 Customizing CKAN's CSS

See also:

There's nothing special about CSS in CKAN, once you've got started with editing CSS in CKAN (by following the tutorial below), then you just use the usual tools and techniques to explore and hack the CSS. We recommend using your browser's web development tools to explore and experiment with the CSS, then using any good text editor to edit your extension's CSS files as needed. For example:

Firefox developer tools
: These include a Page Inspector and a Style Editor

Firebug
: Another web development toolkit for Firefox

Chrome developer tools
: Tools for inspecting and editing CSS in Google Chrome

Mozilla Developer Network's CSS section
: A good collection of CSS documentation and tutorials

Extensions can add their own CSS files to modify or extend CKAN's default CSS. Create an `example_theme.css` file in your extension's `public` directory:

```
ckanext-example_theme/
  ckanext/
    example_theme/
      public/
        example_theme.css
```

Add this CSS into the `example_theme.css` file, to change the color of CKAN's "account masthead" (the bar across the top of the site that shows the logged-in user's account info):

```
.account-masthead {
    background-color: rgb(40, 40, 40);
}
```

If you restart the development web server you should be able to open this file at http://127.0.0.1:5000/example_theme.css in a web browser.

To make CKAN use our custom CSS we need to override the `base.html` template, this is the base template which the templates for all CKAN pages extend, so if we include a CSS file in this base template then the file will be included in every page of your CKAN site. Create the file:

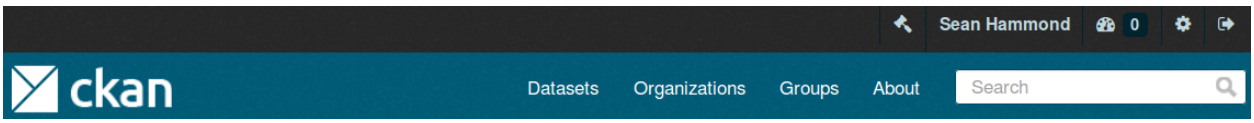
```
ckanext-example_theme/
  ckanext/
    example_theme/
      templates/
        base.html
```

and put this Jinja code in it:

```
{% ckan_extends %}

{% block styles %}
  {{ super() }}
  <link rel="stylesheet" href="/example_theme.css" />
{% endblock %}
```

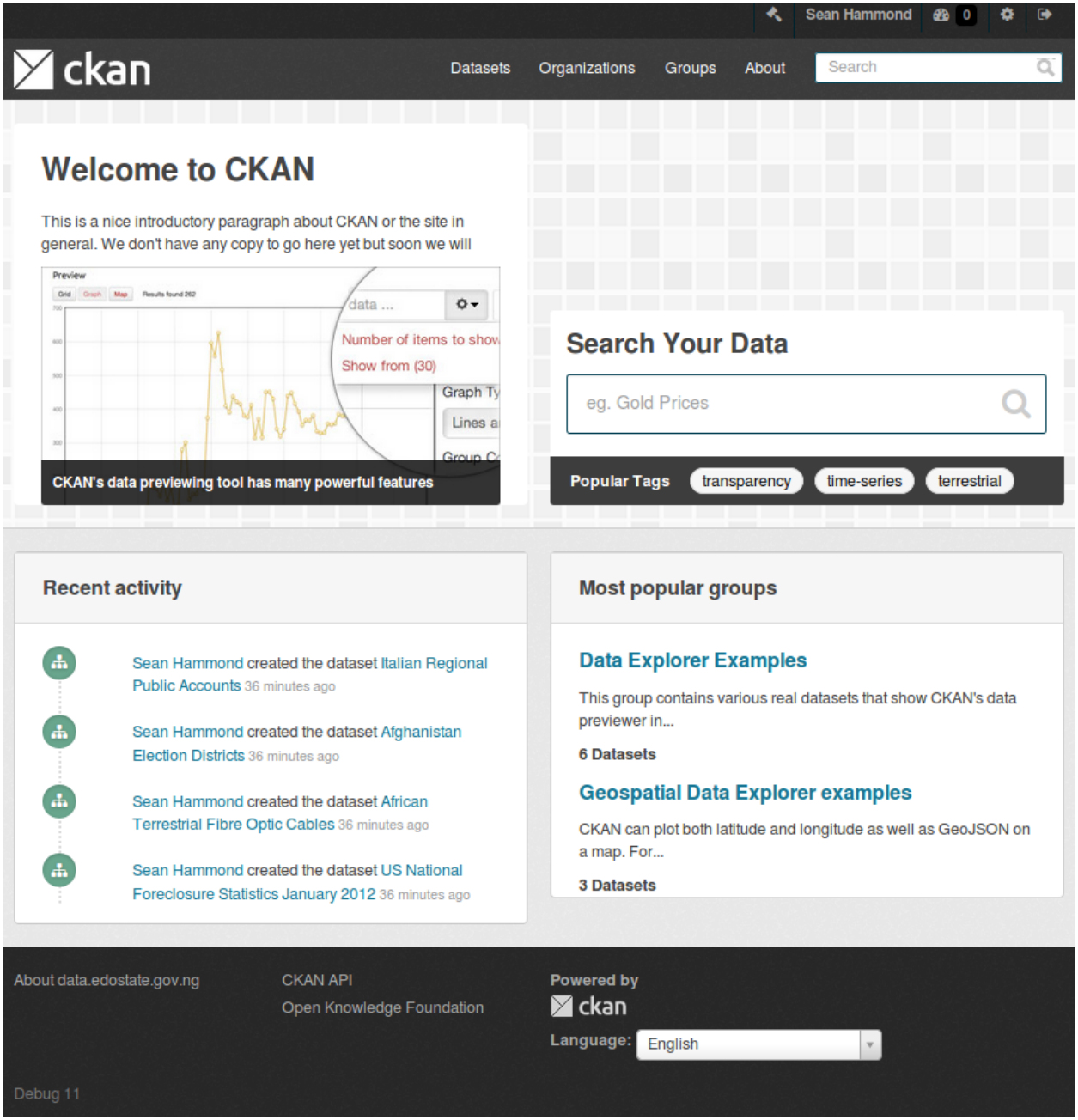
The default `base.html` template defines a `styles` block which can be extended to link to custom CSS files (any code in the styles block will appear in the <`head`> of the HTML page).

Restart the development web server and reload the CKAN page in your browser, and you should see the background color of the account masthead change:



This custom color should appear on all pages of your CKAN site.

Now that we have CKAN using our CSS file, we can add more CSS rules to the file and customize CKAN's CSS as much as we want. Let's add a bit more code to our `example_theme.css` file. This CSS implements a partial imitation of the datahub.io theme (circa 2013):




```
/* ====================================================
   The "account masthead" bar across the top of the site
   ==================================================== */

.account-masthead {
    background-color: rgb(40, 40, 40);
}
/* The "bubble" containing the number of new notifications. */
.account-masthead .account .notifications a span {
  background-color: black;
}
/* The text and icons in the user account info. */
```

(continues on next page)

(continued from previous page)

```
.account-masthead .account ul li a {
  color: rgba(255, 255, 255, 0.6);
}
/* The user account info text and icons, when the user's pointer is hovering
   over them. */
.account-masthead .account ul li a:hover {
  color: rgba(255, 255, 255, 0.7);
  background-color: black;
}


/* ========================================================================
   The main masthead bar that contains the site logo, nav links, and search
   ======================================================================== */

.masthead {
  background-color: #3d3d3d;
}
/* The "navigation pills" in the masthead (the links to Datasets,
   Organizations, etc) when the user's pointer hovers over them. */
.masthead .navigation .nav-pills li a:hover {
  background-color: rgb(48, 48, 48);
  color: white;
}
/* The "active" navigation pill (for example, when you're on the /dataset page
   the "Datasets" link is active). */
.masthead .navigation .nav-pills li.active a {
  background-color: rgb(74, 74, 74);
}
/* The "box shadow" effect that appears around the search box when it
   has the keyboard cursor's focus. */
.masthead input[type="text"]:focus {
  -webkit-box-shadow: inset 0px 0px 2px 0px rgba(0, 0, 0, 0.7);
  box-shadow: inset 0px 0px 2px 0px rgba(0, 0, 0, 0.7);
}


/* ==========================================
   The content in the middle of the front page
   ========================================== */

/* Remove the "box shadow" effect around various boxes on the page. */
.box {
  box-shadow: none;
}
/* Remove the borders around the "Welcome to CKAN" and "Search Your Data"
   boxes. */
.hero .box {
  border: none;
}
/* Change the colors of the "Search Your Data" box. */
.homepage .module-search .module-content {
```

(continues on next page)

(continued from previous page)

```
  color: rgb(68, 68, 68);
  background-color: white;
}
/* Change the background color of the "Popular Tags" box. */
.homepage .module-search .tags {
  background-color: rgb(61, 61, 61);
}
/* Remove some padding. This makes the bottom edges of the "Welcome to CKAN"
   and "Search Your Data" boxes line up. */
.module-content:last-child {
  padding-bottom: 0px;
}
.homepage .module-search {
  padding: 0px;
}
/* Add a border line between the top and bottom halves of the front page. */
.homepage [role="main"] {
  border-top: 1px solid rgb(204, 204, 204);
}


/* ====================================
   The footer at the bottom of the site
   ==================================== */

.site-footer,
body {
  background-color: rgb(40, 40, 40);
}
/* The text in the footer. */
.site-footer,
.site-footer label,
.site-footer small {
  color: rgba(255, 255, 255, 0.6);
}
/* The link texts in the footer. */
.site-footer a {
  color: rgba(255, 255, 255, 0.6);
}
```

6.4 Adding CSS and JavaScript files using Webassets

If you're adding CSS files to your theme, you can add them using Webassets rather than the simple *extra_public_paths* method described in *Adding static files*. If you're adding a JavaScript module, you *must* use Webassets.

Using Webassets to add JavaScript and CSS files takes advantage of Webassets' features, such as automatically serving minified files in production, caching and bundling files together to reduce page load times, specifying dependencies between files so that the files a page needs (and only the files it needs) are always loaded, and other tricks to optimize page load times.

Note: CKAN will only serve `*.js` and `*.css` files as Webassets resources, other types of static files (eg. image files,

PDF files) must be added using the *extra_public_paths* method described in *Adding static files*.

Adding a custom JavaScript or CSS file to CKAN using Webassets is simple. We'll demonstrate by changing our previous custom CSS example (see *Customizing CKAN's CSS*) to serve the CSS file with Webassets.

1. First, create an `assets` directory in your extension and move the CSS file from `public` into `assets`:

```
ckanext-example_theme/
  ckanext/
    example_theme/
      public/
        promoted-image.jpg
      assets/
        example_theme.css
```

2. Use CKAN's `add_resource()` function to register your assets directory with CKAN. Edit the `update_config()` method in your `plugin.py` file:

```
    def update_config(self, config: CKANConfig):

        # Add this plugin's templates dir to CKAN's extra_template_paths, so
        # that CKAN will use this plugin's custom templates.
        toolkit.add_template_directory(config, 'templates')

        # Add this plugin's public dir to CKAN's extra_public_paths, so
        # that CKAN will use this plugin's custom static files.
        toolkit.add_public_directory(config, 'public')

        # Register this plugin's assets directory with CKAN.
        # Here, 'assets' is the path to the webassets directory
        # (relative to this plugin.py file), and 'example_theme' is the name
        # that we'll use to refer to this assets directory from CKAN
        # templates.
        toolkit.add_resource('assets', 'example_theme')
```

3. Finally, edit your extension's `templates/base.html` file and use CKAN's custom Jinja2 tag `{% asset %}` instead of the normal `<link>` tag to import the file:

```
{% ckan_extends %}

{% block styles %}
  {{ super() }}

  {# Import example_theme.css using Webassets.  'example_theme/' is
    the name that the example_theme/webassets directory was registered
    with when the toolkit.add_resource() function was called.
    'example_theme' is the name to the Webassets bundle file,
    registered inside webassets.yml file. #}
  {% asset 'example_theme/example_theme' %}

{% endblock %}
```

Note: You can put `{% asset %}` tags anywhere in any template, and Webassets will insert the necessary `<style>` and `<script>` tags to include your CSS and JavaScript files. But the best place for related asset types is corresponding

`styles` and `scripts` Jinja2's block.

Assets will *not* be included on the line where the `{% asset %}` tag is.

Note: A config file *must* be used to configure how Webassets should serve each asset file (whether or not to bundle files, what order to include files in, whether to include files at the top or bottom of the page, dependencies between files, etc.) See *Assets* for details.

6.4.1 X-Sendfile

For web servers which support the *X-Sendfile* feature, you can set `ckan.webassets.use_x_sendfile` config option to `true` and configure the web server (eg Nginx) in order to serve static files in a more efficient way. Static files served under the URI `/webassets/<PATH_TO_STATIC_FILE>` are stored in the file system under the path specified by *ckan.webassets.path* the config option. If `ckan.webassets.path` is not specified, static files are stored inside a `webassests` folder defined by the *ckan.storage_path* config option.

6.5 Customizing CKAN's JavaScript

JavaScript code in CKAN is broken down into *modules*: small, independent units of JavaScript code. CKAN themes can add JavaScript features by providing their own modules. This tutorial will explain the main concepts involved in CKAN JavaScript modules and walk you through the process of adding custom modules to themes.

See also:

This tutorial assumes a basic understanding of CKAN plugins and templating, see:

- *Extending guide*
- *Customizing CKAN's templates*

See also:

This tutorial assumes a basic understanding of JavaScript and jQuery, see:

- JavaScript on the Mozilla Developer Network
- jQuery.com, including the jQuery Learning Center

See also:

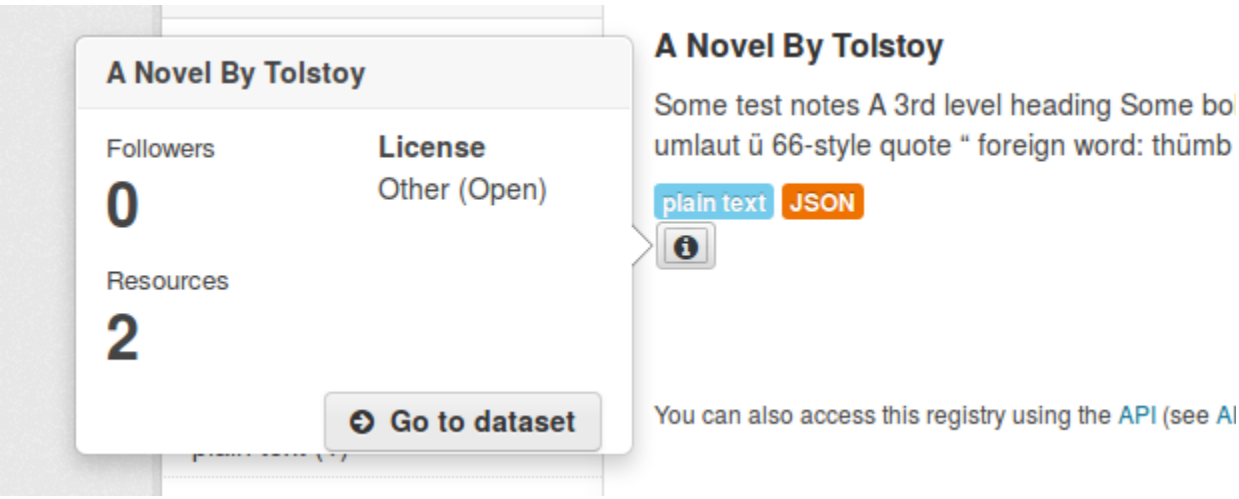
String internationalization
How to mark strings for translation in your JavaScript code.

6.5.1 Overview

The idea behind CKAN's JavaScript modules is to keep the code simple and easy to test, debug and maintain, by breaking it down into small, independent modules. JavaScript modules in CKAN don't share global variables, and don't call each other's code.

These JavaScript modules are attached to HTML elements in the page, and enhance the functionality of those elements. The idea is that an HTML element with a JavaScript module attached should still be fully functional even if JavaScript is completely disabled (e.g. because the user's web browser doesn't support JavaScript). The user experience may not be quite as nice without JavaScript, but the functionality should still be there. This is a programming technique known as *graceful degradation*, and is a basic tenet of web accessibility.

In the sections below, we'll walk you through the steps to add a new JavaScript feature to CKAN - dataset info popovers. We'll add an info button to each dataset on the datasets page which, when clicked, opens a popover containing some extra information and user actions related to the dataset:



6.5.2 Initializing a JavaScript module

To get CKAN to call some custom JavaScript code, we need to:

1. Implement a JavaScript module, and register it with CKAN. Create the file `ckanext-example_theme/ckanext/example_theme_docs/assets/example_theme_popover.js`, with these contents:

```
// Enable JavaScript's strict mode. Strict mode catches some common
// programming errors and throws exceptions, prevents some unsafe actions from
// being taken, and disables some confusing and bad JavaScript features.
"use strict";

ckan.module('example_theme_popover', function ($) {
  return {
    initialize: function () {
      console.log("I've been initialized for element: ", this.el);
    }
  };
});
```

This bit of JavaScript calls the `ckan.module()` function to register a new JavaScript module with CKAN. `ckan.module()` takes two arguments: the name of the module being registered (`'example_theme_popover'` in this example) and a function that returns the module itself. The function takes two arguments, which we'll look at later. The module is just a JavaScript object with a single attribute, `initialize`, whose value is a function that CKAN will call to initialize the module. In this example, the initialize function just prints out a confirmation message - this JavaScript module doesn't do anything interesting yet.

Note: JavaScript module names should begin with the name of the extension, to avoid conflicting with other modules. See *Avoid name clashes*.

Note: Each JavaScript module's `initialize()` function is called on DOM ready.

2. Include the JavaScript module in a page, using Assets, and apply it to one or more HTML elements on that page. We'll override CKAN's `package_item.html` template snippet to insert our module whenever a package is rendered as part of a list of packages (for example, on the dataset search page). Create the file `ckanext-example_theme/ckanext/example_theme_docs/templates/snippets/package_item.html` with these contents:

```
{% ckan_extends %}

{% block content %}
  {{ super() }}

  {# Use Webassets to include our custom JavaScript module.
    A <script> tag for the module will be inserted in the right place at the
    bottom of the page. #}
  {% asset 'example_theme/example_theme' %}

  {# Apply our JavaScript module to an HTML element. The data-module attribute,
    which can be applied to any HTML element, tells CKAN to initialize an
    instance of the named JavaScript module for the element.
    The initialize() method of our module will be called with this HTML
    element as its this.el object. #}
  <button data-module="example_theme_popover"
          class="btn"
          href="#">
    <i class="fa fa-info-circle"></i>
  </button>
{% endblock %}
```

See also:

Using data-* attributes on the Mozilla Developer Network.

If you now restart the development server and open http://127.0.0.1:5000/dataset in your web browser, you should see an extra info button next to each dataset shown. If you open a JavaScript console in your browser, you should see the message that your module has printed out.

See also:

Most web browsers come with built-in developer tools including a JavaScript console that lets you see text printed by JavaScript code to `console.log()`, a JavaScript debugger, and more. For example:

- Firefox Developer Tools
- Firebug
- Chrome DevTools

If you have more than one dataset on your page, you'll see the module's message printed once for each dataset. The `package_item.html` template snippet is rendered once for each dataset that's shown in the list, so your `<button>` element with the `data-module="example_theme_popover"` attribute is rendered once for each dataset, and CKAN creates a new instance of your JavaScript module for each of these `<button>` elements. If you view the source of your page, however, you'll see that `example_theme_popover.js` is only included with a `<script>` tag once. Assets is smart enough to deduplicate resources.

Note: JavaScript modules *must* be included as Assets resources, you can't add them to a `public` directory and include them using your own `<script>` tags.

6.5.3 this.options and this.el

Now let's start to make our JavaScript module do something useful: show a Bootstrap popover with some extra info about the dataset when the user clicks on the info button.

First, we need our Jinja template to pass some of the dataset's fields to our JavaScript module as *options*. Change `package_item.html` to look like this:

```
{% ckan_extends %}

{% block content %}
  {{ super() }}
  {% asset 'example_theme/example_theme' %}

  {# Apply our JavaScript module to an HTML <button> element.
    The additional data-module-* attributes are options that will be passed
    to the JavaScript module. #}
  <button data-module="example_theme_popover"
          data-module-title="{{ package.title }}"
          data-module-license="{{ package.license_title }}"
          data-module-num_resources="{{ package.num_resources }}">
    <i class="fa fa-info-circle"></i>
  </button>
{% endblock %}
```

This adds some `data-module-*` attributes to our `<button>` element, e.g. `data-module-title="{{ package.title }}"` (`{{ package.title }}` is a *Jinja2 expression* that evaluates to the title of the dataset, CKAN passes the Jinja2 variable `package` to our template).

> **Warning:** Although HTML 5 treats any attribute named `data-*` as a data attribute, only attributes named `data-module-*` will be passed as options to a CKAN JavaScript module. So we have to named our parameters `data-module-title` etc., not just `data-title`.

Now let's make use of these options in our JavaScript module. Change `example_theme_popover.js` to look like this:

```
"use strict";

/* example_theme_popover
 *
 * This JavaScript module adds a Bootstrap popover with some extra info about a
 * dataset to the HTML element that the module is applied to. Users can click
 * on the HTML element to show the popover.
 *
 * title - the title of the dataset
 * license - the title of the dataset's copyright license
 * num_resources - the number of resources that the dataset has.
 *
 */
ckan.module('example_theme_popover', function ($) {
  return {
    initialize: function () {
```

(continues on next page)

(continued from previous page)

```
      // Access some options passed to this JavaScript module by the calling
      // template.
      var num_resources = this.options.num_resources;
      var license = this.options.license;

      // Format a simple string with the number of resources and the license,
      // e.g. "3 resources, Open Data Commons Attribution License".
      var content = 'NUM resources, LICENSE'
        .replace('NUM', this.options.num_resources)
        .replace('LICENSE', this.options.license)

      // Add a Bootstrap popover to the HTML element (this.el) that this
      // JavaScript module was initialized on.
      this.el.popover({title: this.options.title,
                       content: content,
                       placement: 'left'});
    }
  };
});
```

Note: It's best practice to add a docstring to the top of a JavaScript module, as in the example above, briefly documenting what the module does and what options it takes. See *JavaScript modules should have docstrings*.

Any `data-module-*` attributes on the HTML element are passed into the JavaScript module in the object `this.options`:

```
      var num_resources = this.options.num_resources;
      var license = this.options.license;
```

A JavaScript module can access the HTML element that it was applied to through the `this.el` variable. To add a popover to our info button, we call Bootstap's `popover()` function on the element, passing in an options object with some of the options that Bootstrap's popovers accept:

```
// Add a Bootstrap popover to the HTML element (this.el) that this
// JavaScript module was initialized on.
this.el.popover({title: this.options.title,
                 content: content,
                 placement: 'left'});
```

See also:

For other objects and functions available to JavaScript modules, see *Objects and methods available to JavaScript modules*.

6.5.4 Default values for options

Default values for JavaScript module options can be provided by adding an `options` object to the module. If the HTML element doesn't have a `data-module-*` attribute for an option, then the default will be used instead. For example...

Todo: Think of an example to do using default values.

6.5.5 Ajax, event handling and CKAN's JavaScript sandbox

So far, we've used simple JavaScript string formatting to put together the contents of our popover. If we want the popover to contain much more complex HTML we really need to render a template for it, using the full power of *Jinja2 templates* and CKAN's *template helper functions*. Let's edit our plugin to use a Jinja2 template to render the contents of the popups nicely.

First, edit `package_item.html` to make it pass a few more parameters to the JavaScript module using `data-module-*` attributes:

```
{% ckan_extends %}

{% block content %}
  {{ super() }}

  {% asset 'example_theme/example_theme' %}

  <button data-module="example_theme_popover"
          data-module-id="{{ package.id }}"
          data-module-title="{{ package.title }}"
          data-module-license_title="{{ package.license_title }}"
          data-module-num_resources="{{ package.num_resources }}">
    <i class="fa fa-info-circle"></i>
  </button>
{% endblock %}
```

We've also added a second `{% asset %}` tag to the snippet above, to include a custom CSS file. We'll see the contents of that CSS file later.

Next, we need to add a new template snippet to our extension that will be used to render the contents of the popovers. Create this `example_theme_popover.html` file:

```
ckanext-example_theme/
  ckanext/
    example_theme/
      templates/
        ajax_snippets/
          example_theme_popover.html
```

and put these contents in it:

```
{# The contents for a dataset popover.

   id - the id of the dataset
   num_resources - the dataset's number of resources
```

(continues on next page)

(continued from previous page)

```
   license_title - the dataset's license title

#}
<div class="context-info">
  <div class="nums">
    <dl>

      <dt>{{ _('Followers') }}</dt>
      <dd>{{ h.follow_count('dataset', id) }}</dd>

      <dt>{{ _('Resources') }}</dt>
      <dd>{{ num_resources }}</dd>

    </dl>
  </div>

  <div class="license">
    <dl>
      <dt>License</dt>
      <dd>{{ license_title }}</dd>
    </dl>
  </div>

  <div class="clearfix"></div>

  {{ h.follow_button('dataset', id) }}

  <a class="btn go-to-dataset"
     href="{{ h.url_for('dataset.read', id=id) }}">
    <i class="fa fa-arrow-circle-right"></i>
    Go to dataset
  </a>

</div>
```

This is a Jinja2 template that renders some nice looking contents for a popover, containing a few bits of information about a dataset. It uses a number of CKAN's Jinja2 templating features, including marking user-visible strings for translation and calling template helper functions. See *Customizing CKAN's templates* for details about Jinja2 templating in CKAN.

Note: The new template file has to be in a `templates/ajax_snippets/` directory so that we can use the template from our JavaScript code using CKAN's `getTemplate()` function. Only templates from `ajax_snippets` directories are available from the `getTemplate()` function.

Next, edit `assets/example_theme_popover.js` as shown below. There's a lot going on in this new JavaScript code, including:

- Using Bootstrap's popover API to show and hide popovers, and set their contents.
- Using jQuery's event handling API to get our functions to be called when the user clicks on a button.
- Using a function from CKAN's *JavaScript sandbox*.

  The sandbox is a JavaScript object, available to all JavaScript modules as `this.sandbox`, that contains a col-

lection of useful functions and variables.

`this.sandbox.client` is a CKAN API client written in JavaScript, that should be used whenever a JavaScript module needs to talk to the CKAN API, instead of modules doing their own HTTP requests.

`this.sandbox.client.getTemplate()` is a function that sends an asynchronous (ajax) HTTP request (i.e. send an HTTP request from JavaScript and receive the response in JavaScript, without causing the browser to reload the whole page) to CKAN asking for a template snippet to be rendered.

Hopefully the liberal commenting in the code below makes it clear enough what's going on:

```
"use strict";

ckan.module('example_theme_popover', function ($) {
  return {
    initialize: function () {

      // proxyAll() ensures that whenever an _on*() function from this
      // JavaScript module is called, the `this` variable in the function will
      // be this JavaScript module object.
      //
      // You probably want to call proxyAll() like this in the initialize()
      // function of most modules.
      //
      // This is a shortcut function provided by CKAN, it wraps jQuery's
      // proxy() function: http://api.jquery.com/jQuery.proxy/
      $.proxyAll(this, /_on/);

      // Add a Bootstrap popover to the button. Since we don't have the HTML
      // from the snippet yet, we just set the content to "Loading..."
      this.el.popover({title: this.options.title, html: true,
                       content: this._('Loading...'), placement: 'left'});

      // Add an event handler to the button, when the user clicks the button
      // our _onClick() function will be called.
      this.el.on('click', this._onClick);
    },

    // Whether or not the rendered snippet has already been received from CKAN.
    _snippetReceived: false,

    _onClick: function(event) {

        // Send an ajax request to CKAN to render the popover.html snippet.
        // We wrap this in an if statement because we only want to request
        // the snippet from CKAN once, not every time the button is clicked.
        if (!this._snippetReceived) {
            this.sandbox.client.getTemplate('example_theme_popover.html',
                                            this.options,
                                            this._onReceiveSnippet);
            this._snippetReceived = true;
        }
    },

    // CKAN calls this function when it has rendered the snippet, and passes
```

(continues on next page)

(continued from previous page)

```
    // it the rendered HTML.
    _onReceiveSnippet: function(html) {

      // Replace the popover with a new one that has the rendered HTML from the
      // snippet as its contents.
      this.el.popover('destroy');
      this.el.popover({title: this.options.title, html: true,
                       content: html, placement: 'left'});
      this.el.popover('show');
    },

  };
});
```

Finally, we need some custom CSS to make the HTML from our new snippet look nice. In `package_item.html` above we added a `{% asset %}` tag to include a custom CSS file. Now we need to create that file, `ckanext-example_theme/ckanext/example_theme/assets/example_theme_popover.css`:

```
.dataset-list .popover .nums {

  /* We're reusing the .nums class from the dataset read page,
   * but we don't want the border, margin and padding, get rid of them. */
  border: none;
  margin: 0;
  padding: 0;

  /* We want the license and numbers to appear side by side, so float the
   * numbers list to the left and make it take up just over half of
   * the width of the popover. */
  float: left;
  width: 55%;
}

.dataset-list .popover .license {

  /* Prevent the words in the license from being wrapped mid-word. */
  word-break: keep-all;
}

.dataset-list .popover .go-to-dataset {

  /* Float the "Go to dataset" button to the right side of the popover,
   * this puts some space between the two buttons. */
  float: right;
}
```

Restart CKAN, and your dataset popovers should be looking much better.

6.5.6 Error handling

What if our JavaScript makes an Ajax request to CKAN, such as our `getTemplate()` call above, and gets an error in response? We can simulate this by changing the name of the requested template file to one that doesn't exist:

```
            this.sandbox.client.getTemplate('foobar.html',
                                            this.options,
                                            this._onReceiveSnippet);
```

If you reload the datasets page after making this change, you'll see that when you click on a popover its contents remain *Loading...*. If you have a development console open in your browser, you'll see the error response from CKAN each time you click to open a popover.

Our JavaScript module's `_onReceiveSnippet()` function is only called if the request gets a successful response from CKAN. `getTemplate()` also accepts a second callback function parameter that will be called when CKAN sends an error response. Add this parameter to the `getTemplate()` call:

```
            this.sandbox.client.getTemplate('foobar.html',
                                            this.options,
                                            this._onReceiveSnippet,
                                            this._onReceiveSnippetError);
        }
    },
```

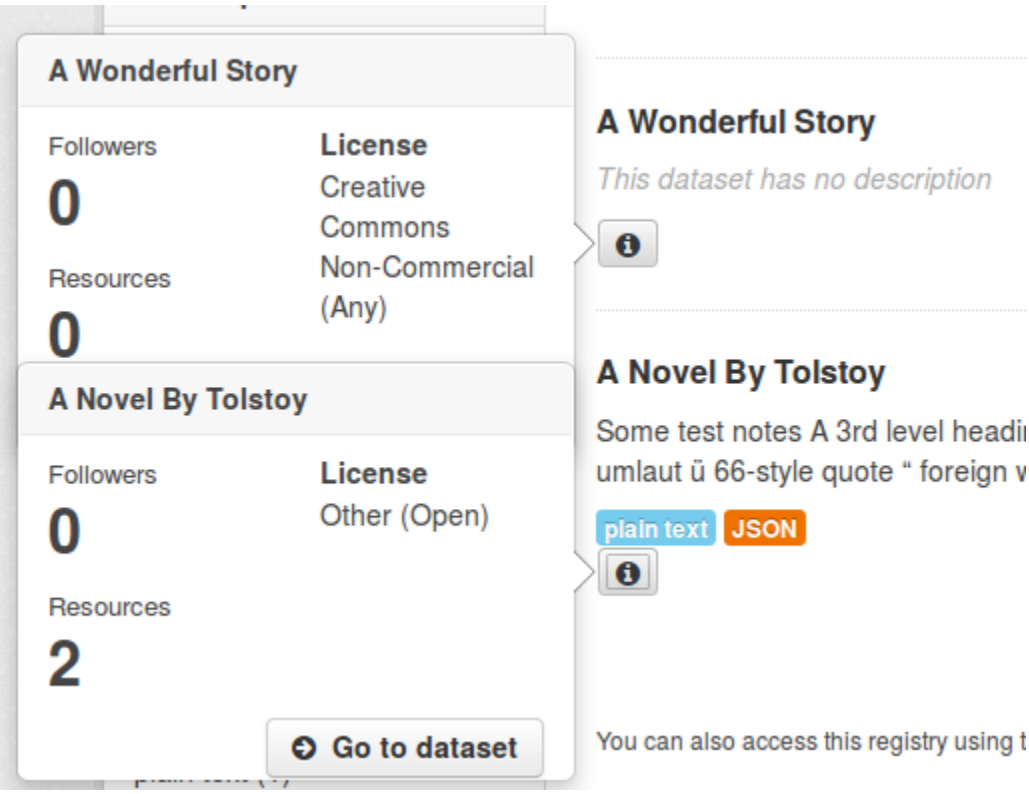
Now add the new error function to the JavaScript module:

```
    _onReceiveSnippetError: function(error) {
      this.el.popover('destroy');

      var content = error.status + ' ' + error.statusText + ' :(';
      this.el.popover({title: this.options.title, html: true,
                       content: content, placement: 'left'});

      this.el.popover('show');
      this._snippetReceived = true;
    },
```

After making these changes, you should see that if CKAN responds with an error, the contents of the popover are replaced with the error message from CKAN.

6.5.7 Pubsub

You may have noticed that, with our example code so far, if you click on the info button of one dataset on the page then click on the info button of another dataset, both dataset's popovers are shown. The first popover doesn't disappear when the second appears, and the popovers may overlap. If you click on all the info buttons on the page, popovers for all of them will be shown at once:



To make one popover disappear when another appears, we can use CKAN's `publish()` and `subscribe()` functions. These pair of functions allow different instances of a JavaScript module (or instances of different JavaScript modules) on the same page to talk to each other. The way it works is:

1. Modules can subscribe to events by calling `this.sandbox.client.subscribe()`, passing the 'topic' (a string that identifies the type of event to subscribe to) and a callback function.
2. Modules can call `this.sandbox.client.publish()` to publish an event for all subscribed modules to receive, passing the topic string and one or more further parameters that will be passed on as parameters to the receiver functions.
3. When a module calls `publish()`, any callback functions registered by previous calls to `subscribe()` with the same topic string will be called, and passed the parameters that were passed to publish.
4. If a module no longer wants to receive events for a topic, it calls `unsubscribe()`.

   All modules that subscribe to events should have a `teardown()` function that unsubscribes from the event, to prevent memory leaks. CKAN calls the `teardown()` functions of modules when those modules are removed from the page. See *JavaScript modules should unsubscribe from events in teardown()*.

Warning: Don't tightly couple your JavaScript modules by overusing pubsub. See *Don't overuse pubsub*.

Remember that because we attach our `example_theme_popover.js` module to a `<button>` element that is rendered

once for each dataset on the page, CKAN creates one instance of our module for each dataset. The only way these objects can communicate with each other so that one object can hide its popover when another object shows its popover, is by using pubsub.

Here's a modified version of our `example_theme_popover.js` file that uses pubsub to make the dataset popovers disappear whenever a new popover appears:

```
"use strict";

ckan.module('example_theme_popover', function ($) {
  return {
    initialize: function () {
      $.proxyAll(this, /_on/);
      this.el.popover({title: this.options.title, html: true,
                       content: this._('Loading...'), placement: 'left'});
      this.el.on('click', this._onClick);

      // Subscribe to 'dataset_popover_clicked' events.
      // Whenever any line of code publishes an event with this topic,
      // our _onPopoverClicked function will be called.
      this.sandbox.subscribe('dataset_popover_clicked',
                             this._onPopoverClicked);
    },

    teardown: function() {
      this.sandbox.unsubscribe('dataset_popover_clicked',
                               this._onPopoverClicked);
    },

    _snippetReceived: false,

    _onClick: function(event) {
        if (!this._snippetReceived) {
            this.sandbox.client.getTemplate('example_theme_popover.html',
                                            this.options,
                                            this._onReceiveSnippet);
            this._snippetReceived = true;
        }

        // Publish a 'dataset_popover_clicked' event for other interested
        // JavaScript modules to receive. Pass the button that was clicked as a
        // parameter to the receiver functions.
        this.sandbox.publish('dataset_popover_clicked', this.el);
    },

    // This callback function is called whenever a 'dataset_popover_clicked'
    // event is published.
    _onPopoverClicked: function(button) {

      // Wrap this in an if, because we don't want this object to respond to
      // its own 'dataset_popover_clicked' event.
      if (button != this.el) {

        // Hide this button's popover.
```

(continues on next page)

(continued from previous page)

```
        // (If the popover is not currently shown anyway, this does nothing).
        this.el.popover('hide');
      }
    },

    _onReceiveSnippet: function(html) {
      this.el.popover('destroy');
      this.el.popover({title: this.options.title, html: true,
                       content: html, placement: 'left'});
      this.el.popover('show');
    },

  };
});
```

6.5.8 jQuery plugins

CKAN provides a number of custom jQuery plugins for JavaScript modules to use by default, see *CKAN jQuery plugins reference*. Extensions can also add their own jQuery plugins, and the plugins will then be available to all JavaScript code via the `this.$` object.

See also:

How to Create a Basic Plugin
jQuery's own documentation on writing jQuery plugins. Read this for all the details on writing jQuery plugins, here we'll only provide a simple example and show you how to integrate it with CKAN.

It's a good idea to implement any JavaScript functionality not directly related to CKAN as a jQuery plugin. That way your CKAN JavaScript modules will be smaller as they'll contain only the CKAN-specific code, and your jQuery plugins will also be reusable on non-CKAN sites. CKAN core uses jQuery plugins to implement features including date formatting, warning users about unsaved changes when leaving a page containing a form without submitting the form, restricting the set of characters that can be typed into an input field, etc.

Let's add a jQuery plugin to our CKAN extension that makes our info buttons turn green when clicked.

Todo: Replace this with a more realistic example.

First we need to write the jQuery plugin itself. Create the file `ckanext-example_theme/ckanext/example_theme/assets/jquery.greenify.js` with the following contents:

```
"use strict";

(function (jQuery) {

    jQuery.fn.greenify = function() {
      this.css( "color", "green" );
      return this;
    };

})(this.jQuery);
```

If this JavaScript code looks a little confusing at first, it's probably because it's using the Immediately-Invoked Function Expression (IIFE) pattern. This is a common JavaScript code pattern in which an anonymous function is created and

then immediately called once, in a single expression. In the example above, we create an unnamed function that takes a single parameter, `jQuery`, and then we call the function passing `this.jQuery` to its `jQuery` parameter. The code inside the body of the function is the important part. Writing jQuery plugins in this way ensures that any variables defined inside the plugin are private to the plugin, and don't pollute the global namespace.

In the body of our jQuery plugin, we add a new function called `greenify()` to the `jQuery` object:

```
    jQuery.fn.greenify = function() {
      this.css( "color", "green" );
      return this;
    };
```

`jquery.fn` is the jQuery prototype object, the object that normal jQuery objects get all their methods from. By adding a method to this object, we enable any code that has a jQuery object to call our method on any HTML element or set of elements. For example, to turn all <a> elements on the page green you could do: `jQuery('a').greenify()`.

The code inside the `greenify()` function just calls jQuery's standard css() method to set the CSS `color` attribute of the element to `green`. This is just standard jQuery code, except that within a custom jQuery function you use `this` to refer to the jQuery object, instead of using `$` or `jquery` (as you would normally do when calling jQuery methods from code external to jQuery).

Our method then returns `this` to allow jQuery method chaining to be used with our method. For example, a user can set an element's CSS `color` attribute to `green` and add the CSS class `greenified` to the element in a single expression by chaining our jQuery method with another method: `$('a').greenify().addClass('greenified');`

Before we can use our `greenify()` method in CKAN, we need to import the `jquery.greenify.js` file into the CKAN page. To do this, add a `{% asset %}` tag to a template file, just as you would do to include any other JavaScript or CSS file in CKAN. Edit the `package_item.html` file:

```
{% ckan_extends %}

{% block content %}
  {{ super() }}

  {% asset 'example_theme/example_theme' %}


  <button data-module="example_theme_popover"
          data-module-id="{{ package.id }}"
          data-module-title="{{ package.title }}"
          data-module-license_title="{{ package.license_title }}"
          data-module-num_resources="{{ package.num_resources }}">
    <i class="fa fa-info-circle"></i>
  </button>
{% endblock %}
```

Now we can call the `greenify()` method from our `example_theme_popover` JavaScript module. For example, we could add a line to the `_onClick()` method in `example_theme_popover.js` so that when a dataset info button is clicked it turns green:

```
    _onClick: function(event) {

        // Make all the links on the page turn green.
        this.$('i').greenify();

        if (!this._snippetReceived) {
```

(continues on next page)

(continued from previous page)

```
            this.sandbox.client.getTemplate('example_theme_popover.html',
                                            this.options,
                                            this._onReceiveSnippet);
            this._snippetReceived = true;
        }
        this.sandbox.publish('dataset_popover_clicked', this.el);
    },
```

6.5.9 Internationalization

See *Internationalizing strings in JavaScript code*.

6.5.10 Testing JavaScript modules

Todo: Show how to write tests for the example module.

6.6 Creating dynamic user interfaces with htmx

Starting version 2.11, CKAN is shipped with htmx.

> *"htmx gives you access to AJAX, CSS Transitions, WebSockets and Server Sent Events directly in HTML, using attributes, so you can build modern user interfaces with the simplicity and power of hypertext."* – htmx.org

While not all CKAN templates have been updated to use `htmx`, you can use it in your own extensions to build modern user interfaces. htmx will be the core component in the implementation of the new CKAN UI, so you should expect more of it in future versions.

6.6.1 Overview

`htmx` is a library that allows you to use HTML attributes to make AJAX requests and update the DOM. It is a great alternative to Javascript frameworks like React or Vue, as it allows you to build dynamic user interfaces with regular flask views and Jinja2 templates, allowing templates to be overridden by themes and other extensions.

The library is very simple to use. You just need to add the `hx-*` attributes to your HTML elements to make them dynamic. For example, to make a link that makes a POST request to the `/dataset/follow/<dataset-id>` endpoint and replaces the HTML element with id `package-info` with all the HTML returned by the endpoint, you can write:

```
<a class="btn btn-danger" hx-post="{{ h.url_for('dataset.follow', id=pkg.id) }}" hx-
↪target="#package-info">
  <i class="fa-solid fa-circle-plus"></i>
  Follow
</a>
```

The example can be read as: "When the user clicks on this link, make a POST request to the `/dataset/follow/<dataset-id>` endpoint and replace the HTML element with id `package-info` with all the HTML returned by the endpoint". Notice how we are using the `hx-post` and `hx-target` attributes to define the behaviour of the link.

For a full list of the HTML attributes and their usage, check the htmx documentation.

6.6.2 Implementing new features with htmx

`htmx` give us the flexibility to implement new dynamic features in CKAN by implementing new endpoints that returns the partial HTML that we want to insert into the page. The **Follow / Unfollow** logic is a great example of this and we will explain the thought process behind it in this section.

In UI terms, the **Follow / Unfollow** logic is just a div containing a button that allows the user to follow/unfollow a dataset plus a counter that shows the number of followers. The div is displayed in the dataset page.

This is a small interactive action and we do not want a typical full refresh of the page. It doesn't make any sense to reload the whole page just to update the number of followers and the button. This is a perfect use case for `htmx`.

What we need to achieve this behaviour is:

1. A HTML structure that encapsulates the follow/unfollow UI in a single HTML element (so it can be replaced).
2. A way to trigger a call to the endpoint when the user clicks on the button and replace the element with the new content.
3. A new endpoint that covers the backed logic and returns just enough HTML to replace the HTML element.

1. HTML structure

The HTML structure is very simple: an element that contains the button and the counter. To respect the current CKAN UX we update the `package/snippets/info.html` snippet. We need to make sure that the `section` HTML element we want to replace has an id so we add it: `id="package-info"`.

```
<!-- package/snippets/info.html -->
{% block package_info %}
  {% if pkg %}
    <section id="package-info" class="module module-narrow">
    <!-- Rest of the snippet -->
    </section>
  {% endif %}
{% endblock %}
```

2. Triggering a call to the endpoint

We need to trigger a call to the endpoint when the user clicks on the button. We can do this by adding the `hx-post` attribute to the button. The `hx-post` attribute defines the URL that will be called when the user clicks on the button. In our case, we want to call the `/dataset/follow/<dataset-id>` endpoint, so we can use the `h.url_for` helper to generate the URL.

```
<a class="btn btn-danger" hx-post="{{ h.url_for('dataset.follow', id=pkg.id) }}" hx-
↪target="#package-info">
  <i class="fa-solid fa-circle-plus"></i>
  Follow
</a>
```

In addition to the `hx-post` attribute, we also need to define the `hx-target` attribute. The `hx-target` attribute defines the HTML element that will be replaced with the HTML returned by the endpoint. In our case, we want to replace the `package-info` element, so we can use the `#package-info` selector.

3. The endpoint

The last step is to implement the endpoint that will be called when the user clicks on the button. In our case, we want to call the `/dataset/follow/<dataset-id>` endpoint. This endpoint is already implemented in CKAN. We need to make sure that, under this new context, it should return only the partial HTML that we want to insert into the page instead of rendering the whole dataset page again. We achieve that by making it sure that we return the snippet that contains the HTML that we want to display, in our case `package/snippets/info.html`.

View:

```
def follow(package_type: str, id: str) -> Union[Response, str]:
    """Start following this dataset."""
    am_following: bool = False
    error_message: str = ""

    try:
        package_dict = get_action('package_show')({}, {'id': id})
    except (NotFound, NotAuthorized):
        msg = _('Dataset not found or you have no permission to view it')
        return base.abort(404, msg)

    try:
        get_action('follow_dataset')({}, {'id': id})
        am_following = True
    except ValidationError as e:
        error_message = str(e.error_dict['message'])

    extra_vars = {
        'pkg': package_dict,
        'am_following': am_following,
        'current_user': current_user,
        'error_message': error_message
        }

    return base.render('package/snippets/info.html', extra_vars)
```

Note that this endpoint is reusing the `package/snippets/info.html` that is also being called in `package/read_base.html` when calling `/dataset/<dataset-id>`. This shows how modular and reusable the CKAN templates are with `htmx`.

6.6.3 2. Accesing to HTMX request headers in CKAN

CKAN adds a new property to the CKANRequest class called `htmx` that you can use to access the htmx request headers. For example:

```
from ckan.common import request

if request.htmx:
    # do something
```

Calling `request.htmx` will return a HtmxDetails object that contains attributes for each one of the `htmx` attributes. For example, if you want to access the `hx-target` attribute, you can write:

```
from ckan.common import request
```

(continues on next page)

(continued from previous page)

```
if request.htmx:
    target = request.htmx.target
```

```
class HtmxDetails(object):
    """Object to access htmx properties from the request headers.

    This object will be added to the CKAN `request` object
    as `request.htmx`. It adds properties to easily access
    htmx's request headers defined in
    https://htmx.org/reference/#headers.
    """

    def __init__(self, request: Any):
        self.request = request

    def __bool__(self) -> bool:
        return self.request.headers.get("HX-Request") == "true"

    @property
    def boosted(self) -> bool:
        return self.request.headers.get("HX-Boosted") == "true"

    @property
    def current_url(self) -> str | None:
        return self.request.headers.get("HX-Current-URL")

    @property
    def history_restore_request(self) -> bool:
        return self.request.headers.get("HX-History-Restore-Request") == "true"

    @property
    def prompt(self) -> str | None:
        return self.request.headers.get("HX-Prompt")

    @property
    def target(self) -> str | None:
        return self.request.headers.get("HX-Target")

    @property
    def trigger(self) -> str | None:
        return self.request.headers.get("HX-Trigger")

    @property
    def trigger_name(self) -> str | None:
        return self.request.headers.get("HX-Trigger-Name")
```

6.6.4 3. htmx examples

Check the htmx examples for an overview of patterns that you can use to implement rich UX features.

6.7 Best practices for writing CKAN themes

6.7.1 Don't use `c`

As much as possible, avoid accessing the old Pylons template context `c` (or `tmpl_context`). `c` is a thread-global variable, which encourages spaghetti code that's difficult to understand and to debug. same applies for the Flask `g` object. Current uses of them in templates are to provide backwards compatibility but will be removed in the future.

Instead, have controller methods add variables to the `extra_vars` parameter of `render()`, or have the templates call *template helper functions* instead.

`extra_vars` has the advantage that it allows templates, which are difficult to debug, to be simpler and shifts logic into the easier-to-test and easier-to-debug Python code. On the other hand, template helper functions are easier to reuse as they're available to all templates and they avoid inconsistencies between the namespaces of templates that are rendered by different controllers (e.g. one controller method passes the package dict as an extra var named `package`, another controller method passes the same thing but calls it `pkg`, a third calls it `pkg_dict`).

You can use the `ITemplateHelpers` plugin interface to add custom helper functions, see *Adding your own template helper functions*.

6.7.2 Use `url_for()`

Always use `url_for()` (available to templates as `h.url_for()`) when linking to other CKAN pages, instead of hard-coding URLs like `<a href="/dataset">`. Links created with `url_for()` will update themselves if the URL routing changes in a new version of CKAN, or if a plugin changes the URL routing.

6.7.3 Use `{% trans %}`, `{% pluralize %}`, `_()` and `ungettext()`

All user-visible strings should be internationalized, see *String internationalization*.

6.7.4 Avoid name clashes

See *Avoid name clashes*.

6.7.5 JavaScript modules should have docstrings

A JavaScript module should have a docstring at the top of the file, briefly documentating what the module does and what options it takes. For example:

```
"use strict";

/* example_theme_popover
 *
 * This JavaScript module adds a Bootstrap popover with some extra info about a
 * dataset to the HTML element that the module is applied to. Users can click
```

(continues on next page)

(continued from previous page)

```
 * on the HTML element to show the popover.
 *
 * title - the title of the dataset
 * license - the title of the dataset's copyright license
 * num_resources - the number of resources that the dataset has.
 *
 */
ckan.module('example_theme_popover', function ($) {
  return {
    initialize: function () {

      // Access some options passed to this JavaScript module by the calling
      // template.
      var num_resources = this.options.num_resources;
      var license = this.options.license;

      // Format a simple string with the number of resources and the license,
      // e.g. "3 resources, Open Data Commons Attribution License".
      var content = 'NUM resources, LICENSE'
        .replace('NUM', this.options.num_resources)
        .replace('LICENSE', this.options.license)

      // Add a Bootstrap popover to the HTML element (this.el) that this
      // JavaScript module was initialized on.
      this.el.popover({title: this.options.title,
                       content: content,
                       placement: 'left'});
    }
  };
});
```

6.7.6 JavaScript modules should unsubscribe from events in `teardown()`

Any JavaScript module that calls `this.sandbox.client.subscribe()` should have a `teardown()` function that calls `unsubscribe()`, to prevent memory leaks. CKAN calls the `teardown()` functions of modules when those modules are removed from the page.

6.7.7 Don't overuse pubsub

There shouldn't be very many cases where a JavaScript module really needs to use *Pubsub*, try to only use it when you really need to.

JavaScript modules in CKAN are designed to be small and loosely-coupled, for example modules don't share any global variables and don't call each other's functions. But pubsub offers a way to tightly couple JavaScript modules together, by making modules depend on multiple events published by other modules. This can make the code buggy and difficult to understand.

6.7.8 Use {% snippet %}, not {% include %}

Always use CKAN's custom {% snippet %} tag instead of Jinja's default {% include %} tag. Snippets can only access certain global variables, and any variables explicitly passed to them by the calling template. They don't have access to the full context of the calling template, as included files do. This makes snippets more reusable, and much easier to debug.

6.7.9 Snippets should have docstrings

A snippet should have a docstring comment at the top of the file that briefly documents what the snippet does and what parameters it requires. For example:

```
{#

Renders a list of the site's most popular groups.

groups - the list of groups to render

#}
<h3>Most popular groups</h3>
<ul>
  {% for group in groups %}
    <li>
      <a href="{{ h.url_for('group_read', action='read', id=group.name) }}">
        <h3>{{ group.display_name }}</h3>
      </a>
      {% if group.description %}
        <p>
          {{ h.markdown_extract(group.description, extract_length=80) }}
        </p>
      {% else %}
        <p>{{ _('This group has no description') }}</p>
      {% endif %}
      {% if group.package_count %}
        <strong>{{ ungettext('{num} Dataset', '{num} Datasets', group.package_count).
↪format(num=group.package_count) }}</strong>
      {% else %}
        <span>{{ _('0 Datasets') }}</span>
      {% endif %}
    </li>
  {% endfor %}
</ul>
```

6.8 Custom Jinja2 tags reference

Todo: TODO

6.9 Variables and functions available to templates

The following global variables and functions are available to all CKAN templates in their top-level namespace:

Note: In addition to the global variables listed below, each template also has access to variables from a few other sources:

- Any extra variables explicitly passed into a template by the controller that rendered the template will also be available to that template, in its top-level namespace. Any variables explicitly added to the template context variable `c` will also be available to the template as attributes of `c`.

  To see which additional global variables and context attributes are available to a given template, use CKAN's *debug footer*.

- Any variables explicitly passed into a template snippet in the calling `{% snippet %}` tag will be available to the snippet in its top-level namespace. To see these variables, use the *debug footer*.
- Jinja2 also makes a number of filters, tests and functions available in each template's global namespace. For a list of these, see the Jinja2 docs.

`tmpl_context`

The Pylons template context object, a thread-safe object that the application can store request-specific variables against without the variables associated with one HTTP request getting confused with variables from another request.

`tmpl_context` is usually abbreviated to `c` (an alias).

Using `c` in CKAN is discouraged, use template helper functions instead. See *Don't use c*.

`c` is not available to snippets.

`c`

An alias for *`tmpl_context`*.

`app_globals`

The Pylons App Globals object, an instance of the `ckan.lib.app_globals.Globals` class. The application can store request-independent variables against the `app_globals` object. Variables stored against `app_globals` are shared between all HTTP requests.

`g`

An alias for *`app_globals`*.

`h`

CKAN's *template helper functions*, plus any *custom template helper functions* provided by any extensions.

`request`

The Pylons Request object, contains information about the HTTP request that is currently being responded to, including the request headers and body, URL parameters, the requested URL, etc.

response

The Pylons Response object, contains information about the HTTP response that is currently being prepared to be sent back to the user, including the HTTP status code, headers, cookies, etc.

session

The Beaker session object, which contains information stored in the user's currently active session cookie.

_()

The pylons.i18n.translation.ugettext(value) function:

> Mark a string for translation. Returns the localized unicode string of value.
>
> Mark a string to be localized as follows:
>
> ```
> _('This should be in lots of languages')
> ```

N_()

The pylons.i18n.translation.gettext_noop(value) function:

> Mark a string for translation without translating it. Returns value.
>
> Used for global strings, e.g.:
>
> ```
> foo = N_('Hello')
>
> class Bar:
>  def __init__(self):
>  self.local_foo = _(foo)
>
> h.set_lang('fr')
> assert Bar().local_foo == 'Bonjour'
> h.set_lang('es')
> assert Bar().local_foo == 'Hola'
> assert foo == 'Hello'
> ```

ungettext()

The pylons.i18n.translation.ungettext(singular, plural, n) function:

> Mark a string for translation. Returns the localized unicode string of the pluralized value.
>
> This does a plural-forms lookup of a message id. singular is used as the message id for purposes of lookup in the catalog, while n is used to determine which plural form to use. The returned message is a Unicode string.
>
> Mark a string to be localized as follows:
>
> ```
> ungettext('There is %(num)d file here', 'There are %(num)d files here',
>  n) % {'num': n}
> ```

translator

An instance of the gettext.NullTranslations class. This is for internal use only, templates shouldn't need to use this.

class actions

The `ckan.model.authz.Action` class.

Todo: Remove this? Doesn't appear to be used and doesn't look like something we want.

6.10 Objects and methods available to JavaScript modules

CKAN makes a few helpful objects and methods available for every JavaScript module to use, including:

- `this.el`, the HTML element that this instance of the object was initialized for. A jQuery element. See *this.options and this.el*.
- `this.options`, an object containing any options that were passed to the module via `data-module-*` attributes in the template. See *this.options and this.el*.
- `this.$()`, a jQuery find function that is scoped to the HTML element that the JavaScript module was applied to. For example, `this.$('a')` will return all of the <a> elements inside the module's HTML element, *not* all of the <a> elements on the entire page.

  This is a shortcut for `this.el.find()`.

  jQuery provides many useful features in an easy-to-use API, including document traversal and manipulation, event handling, and animation. See jQuery's own docs for details.
- `this.sandbox`, an object containing useful functions for all modules to use, including:
  - `this.sandbox.client`, an API client for calling the API
  - `this.sandbox.jQuery`, a jQuery find function that is not bound to the module's HTML element. `this.sandbox.jQuery('a')` will return all the <a> elements on the entire page. Using `this.sandbox.jQuery` is discouraged, try to stick to `this.$` because it keeps JavaScript modules more independent.

  See *JavaScript sandbox reference*.
- A collection of *jQuery plugins*.
- *Pubsub functions* that modules can use to communicate with each other, if they really need to.
- Bootstrap's JavaScript features, see the Bootstrap docs for details.
- The standard JavaScript `window` object. Using `window` in CKAN JavaScript modules is discouraged, because it goes against the idea of a module being independent of global context. However, there are some circumstances where a module may need to use `window` (for example if a vendor plugin that the module uses needs it).
- `this._` and `this.ngettext` for string internationalization. See *Internationalization*.
- `this.remove()`, a method that tears down the module and removes it from the page (this usually called by CKAN, not by the module itself).

6.11 Template helper functions reference

Helper functions

Consists of functions to typically be used within templates, but also available to Controllers. This module is available to templates as 'h'.

class ckan.lib.helpers.**HelperAttributeDict**

Collection of CKAN native and extension-provided helpers.

class ckan.lib.helpers.**literal**(*base: Any = ''*, *encoding: str | None = None*, *errors: str = 'strict'*)

Represents an HTML literal.

classmethod escape(*s: str | None*) → Markup

Escape a string. Calls `escape()` and ensures that for subclasses the correct type is returned.

`ckan.lib.helpers.core_helper`(*f: Helper*, *name: str | None = None*) → Helper

Register a function as a builtin helper method.

`ckan.lib.helpers.chained_helper`(*func: Helper*) → Helper

Decorator function allowing helper functions to be chained.

This chain starts with the first chained helper to be registered and ends with the original helper (or a non-chained plugin override version). Chained helpers must accept an extra parameter, specifically the next helper in the chain, for example:

```
helper(next_helper, *args, **kwargs).
```

The chained helper function may call the next_helper function, optionally passing different values, handling exceptions, returning different values and/or raising different exceptions to the caller.

Usage:

```
from ckan.plugins.toolkit import chained_helper

@chained_helper
def ckan_version(next_func, **kw):

    return next_func(**kw)
```

Parameters
func (*callable*) – chained helper function

Returns
chained helper function

Return type
callable

`ckan.lib.helpers.redirect_to`(**args: Any*, ***kw: Any*) → Response

Issue a redirect: return an HTTP response with a `302 Moved` header.

This is a wrapper for `flask.redirect()` that maintains the user's selected language when redirecting.

The arguments to this function identify the route to redirect to, they're the same arguments as `ckan.plugins.toolkit.url_for()` accepts, for example:

```
import ckan.plugins.toolkit as toolkit

# Redirect to /dataset/my_dataset.
return toolkit.redirect_to('dataset.read',
                    id='my_dataset')
```

Or, using a named route:

```
return toolkit.redirect_to('dataset.read', id='changed')
```

If given a single string as argument, this redirects without url parsing

return toolkit.redirect_to('http://example.com') return toolkit.redirect_to('/dataset') return toolkit.redirect_to('/some/other/path')

`ckan.lib.helpers.get_site_protocol_and_host()` → tuple[str, str] | tuple[None, None]

Return the protocol and host of the configured *ckan.site_url*. This is needed to generate valid, full-qualified URLs.

If *ckan.site_url* is set like this:

```
ckan.site_url = http://example.com
```

Then this function would return a tuple *('http', 'example.com')* If the setting is missing, *(None, None)* is returned instead.

`ckan.lib.helpers.url_for(*args: Any, **kw: Any)` → str

Return the URL for an endpoint given some parameters.

This is a wrapper for `flask.url_for()` and `routes.url_for()` that adds some extra features that CKAN needs.

To build a URL for a Flask view, pass the name of the blueprint and the view function separated by a period `.`, plus any URL parameters:

```
url_for('api.action', ver=3, logic_function='status_show')
# Returns /api/3/action/status_show
```

For a fully qualified URL pass the `_external=True` parameter. This takes the `ckan.site_url` and `ckan.root_path` settings into account:

```
url_for('api.action', ver=3, logic_function='status_show',
        _external=True)
# Returns http://example.com/api/3/action/status_show
```

URLs built by Pylons use the Routes syntax:

```
url_for(controller='my_ctrl', action='my_action', id='my_dataset')
# Returns '/dataset/my_dataset'
```

Or, using a named route:

```
url_for('dataset.read', id='changed')
# Returns '/dataset/changed'
```

Use `qualified=True` for a fully qualified URL when targeting a Pylons endpoint.

For backwards compatibility, an effort is made to support the Pylons syntax when building a Flask URL, but this support might be dropped in the future, so calls should be updated.

`ckan.lib.helpers.url_for_static(*args: Any, **kw: Any)` → str

Returns the URL for static content that doesn't get translated (eg CSS)

It'll raise CkanUrlException if called with an external URL

This is a wrapper for `routes.url_for()`

`ckan.lib.helpers.url_for_static_or_external(*args: Any, **kw: Any)` → str

Returns the URL for static content that doesn't get translated (eg CSS), or external URLs

`ckan.lib.helpers.is_url(*args: Any, **kw: Any)` → bool

Returns True if argument parses as a http, https or ftp URL

`ckan.lib.helpers.`**`url_is_local`**(*url: str*) → bool

Returns True if url is local

`ckan.lib.helpers.`**`full_current_url`**() → str

Returns the fully qualified current url (eg http://...) useful for sharing etc

`ckan.lib.helpers.`**`current_url`**() → str

Returns current url unquoted

`ckan.lib.helpers.`**`lang`**() → str | None

Return the language code for the current locale eg *en*

`ckan.lib.helpers.`**`strxfrm`**(*s: str*) → str

Transform a string to one that can be used in locale-aware comparisons. Override this helper if you have different text sorting needs.

`ckan.lib.helpers.`**`ckan_version`**() → str

Return CKAN version

`ckan.lib.helpers.`**`lang_native_name`**(*lang_: str | None = None*) → str | None

Return the language name currently used in it's localised form either from parameter or current environ setting

`ckan.lib.helpers.`**`is_rtl_language`**() → bool

`ckan.lib.helpers.`**`get_rtl_theme`**() → str

`ckan.lib.helpers.`**`flash_notice`**(*message: Any*, *allow_html: bool = False*) → None

Show a flash message of type notice

`ckan.lib.helpers.`**`flash_error`**(*message: Any*, *allow_html: bool = False*) → None

Show a flash message of type error

`ckan.lib.helpers.`**`flash_success`**(*message: Any*, *allow_html: bool = False*) → None

Show a flash message of type success

`ckan.lib.helpers.`**`get_flashed_messages`**(***kwargs: Any*)

Call Flask's built in get_flashed_messages

`ckan.lib.helpers.`**`link_to`**(*label: str | None*, *url: str*, ***attrs: Any*) → Markup

`ckan.lib.helpers.`**`nav_link`**(*text: str*, **args: Any*, ***kwargs: Any*) → Markup | str

Parameters

- **`class`** – pass extra class(es) to add to the `<a>` tag
- **`icon`** – name of ckan icon to use within the link
- **`condition`** – if `False` then no link is returned

`ckan.lib.helpers.`**`build_nav_main`**(**args: tuple[str, str] | tuple[str, str, list[str]] | tuple[str, str, list[str], str]*) → Markup

Build a set of menu items.

Outputs `<li><a href="...">title</a></li>`

Parameters

`args` (*`tuple[str, str, Optional[list], Optional[str]]`*) – tuples of (menu type, title) eg ('login', _('Login')). Third item specifies controllers which should be used to mark link as active. Fourth item specifies auth function to check permissions against.

Return type
str

`ckan.lib.helpers.build_nav_icon(menu_item: str, title: str, **kw: Any) → Markup`

Build a navigation item used for example in `user/read_base.html`.

Outputs `<li><a href="..."><i class="icon.."></i> title</a></li>`.

Parameters

- **menu_item** (*string*) – the name of the defined menu item defined in config/routing as the named route of the same name
- **title** (*string*) – text used for the link
- **kw** – additional keywords needed for creating url eg `id=...`

Return type
HTML literal

`ckan.lib.helpers.build_nav(menu_item: str, title: str, **kw: Any) → Markup`

Build a navigation item used for example breadcrumbs.

Outputs `<li><a href="...">title</a></li>`.

Parameters

- **menu_item** (*string*) – the name of the defined menu item defined in config/routing as the named route of the same name
- **title** (*string*) – text used for the link
- **kw** – additional keywords needed for creating url eg `id=...`

Return type
HTML literal

`ckan.lib.helpers.map_pylons_to_flask_route_name(menu_item: str)`

returns flask routes for old fashioned route names

`ckan.lib.helpers.default_group_type(type_: str) → str`

Get default group/organization type for using site-wide.

`ckan.lib.helpers.default_package_type() → str`

Get default package type for using site-wide.

`ckan.lib.helpers.humanize_entity_type(entity_type: str, object_type: str, purpose: str) → str | None`

Convert machine-readable representation of package/group type into human-readable form.

Returns capitalized *entity_type* with all underscores converted into spaces.

Example:

```
>>> humanize_entity_type('group', 'custom_group', 'add link')
'Add Custom Group'
>>> humanize_entity_type('group', 'custom_group', 'breadcrumb')
'Custom Groups'
>>> humanize_entity_type('group', 'custom_group', 'not real purpuse')
'Custom Group'
```

Possible purposes(depends on *entity_type* and change over time):

```
`add link`: "Add [object]" button on search pages
`breadcrumb`: "Home / [object]s / New" section in breadcrums
`content tab`: "[object]s | Groups | Activity" tab on details page
`create label`: "Home / ... / Create [object]" part of breadcrumb
`create title`: "Create [object] - CKAN" section of page title
`delete confirmation`: Confirmation popup when object is deleted
`description placeholder`: Placeholder for description field on form
`edit label`: "Edit [object]" label/breadcrumb/title
`facet label`: "[object]s" label in sidebar(facets/follower counters)
`form label`: "[object] Form" heading on object form page
`main nav`: "[object]s" link in the header
`view label`: "View [object]s" button on edit form
`my label`: "My [object]s" tab in dashboard
`name placeholder`: "<[object]>" section of URL preview on object form
`no any objects`: No objects created yet
`no associated label`: no gorups for dataset
`no description`: object has no description
`no label`: package with no organization
`page title`: "Title - [objec]s - CKAN" section of page title
`save label`: "Save [object]" button
`search placeholder`: "Search [object]s..." placeholder
`update label`: "Update [object]" button
`you not member`: Dashboard with no groups
```

ckan.lib.helpers.**get_facet_items_dict**(*facet: str*, *search_facets: dict[str, dict[str, Any]] | Any | None = None*, *limit: int | None = None*, *exclude_active: bool = False*) → list[dict[str, Any]]

Return the list of unselected facet items for the given facet, sorted by count.

Returns the list of unselected facet contraints or facet items (e.g. tag names like "russian" or "tolstoy") for the given search facet (e.g. "tags"), sorted by facet item count (i.e. the number of search results that match each facet item).

Reads the complete list of facet items for the given facet from search_facets, and filters out the facet items that the user has already selected.

Arguments: facet – the name of the facet to filter. search_facets – dict with search facets limit – the max. number of facet items to return. exclude_active – only return unselected facets.

ckan.lib.helpers.**has_more_facets**(*facet: str*, *search_facets: dict[str, dict[str, Any]]*, *limit: int | None = None*, *exclude_active: bool = False*) → bool

Returns True if there are more facet items for the given facet than the limit.

Reads the complete list of facet items for the given facet from search_facets, and filters out the facet items that the user has already selected.

Arguments: facet – the name of the facet to filter. search_facets – dict with search facets limit – the max. number of facet items. exclude_active – only return unselected facets.

ckan.lib.helpers.**get_param_int**(*name: str*, *default: int = 10*) → int

ckan.lib.helpers.**sorted_extras**(*package_extras: list[dict[str, Any]]*, *auto_clean: bool = False*, *subs: dict[str, str] | None = None*, *exclude: list[str] | None = None*) → list[tuple[str, Any]]

Used for outputting package extras

Parameters

- **package_extras** (*dict*) – the package extras
- **auto_clean** (*bool*) – If true capitalize and replace -_ with spaces
- **subs** (*dict {'key': 'replacement'}*) – substitutes to use instead of given keys
- **exclude** (*list of strings*) – keys to exclude

ckan.lib.helpers.**check_access**(*action: str*, *data_dict: dict[str, Any] | None = None*) → bool

ckan.lib.helpers.**linked_user**(*user: str | User*, *maxlength: int = 0*, *avatar: int = 20*) → Markup | str | None

ckan.lib.helpers.**group_name_to_title**(*name: str*) → str

ckan.lib.helpers.**truncate**(*text: str*, *length: int = 30*, *indicator: str = '...'*, *whole_word: bool = False*) → str

Truncate `text` with replacement characters.

length
The maximum length of `text` before replacement

indicator
If `text` exceeds the `length`, this string will replace the end of the string

whole_word
If true, shorten the string further to avoid breaking a word in the middle. A word is defined as any string not containing whitespace. If the entire text before the break is a single word, it will have to be broken.

Example:

```
>>> truncate('Once upon a time in a world far far away', 14)
'Once upon a...'
```

Deprecated: please use jinja filter *truncate* instead

ckan.lib.helpers.**markdown_extract**(*text: str*, *extract_length: int = 190*) → str | Markup

return the plain text representation of markdown encoded text. That is the texted without any html tags. If extract_length is 0 then it will not be truncated.

ckan.lib.helpers.**dict_list_reduce**(*list_: list[dict[str, T]]*, *key: str*, *unique: bool = True*) → list[T]

Take a list of dicts and create a new one containing just the values for the key with unique values if requested.

ckan.lib.helpers.**gravatar**(*email_hash: str*, *size: int = 100*, *default: str | None = None*) → Markup

ckan.lib.helpers.**sanitize_url**(*url: str*)

Return a sanitized version of a user-provided url for use in an <a href> or <img src> attribute, e.g.:

<a href="{{ h.sanitize_url(user_link) }}">

Sanitizing urls is tricky. This is a best-effort to produce something valid from the sort of text users might paste into a web form, not intended to cover all possible valid edge-case urls.

On parsing errors an empty string will be returned.

ckan.lib.helpers.**user_image**(*user_id: str*, *size: int = 100*) → Markup | str

ckan.lib.helpers.**pager_url**(*page: int*, *partial: str | None = None*, ***kwargs: Any*) → str

ckan.lib.helpers.**get_page_number**(*params: dict[str, Any]*, *key: str = 'page'*, *default: int = 1*) → int

Return the page number from the provided params after verifying that it is an positive integer.

If it fails it will abort the request with a 400 error.

`ckan.lib.helpers.get_display_timezone()` → tzinfo

Returns a pytz timezone for the display_timezone setting in the configuration file or UTC if not specified. :rtype: timezone

`ckan.lib.helpers.render_datetime(datetime_: datetime | None, date_format: str | None = None, with_hours: bool = False, with_seconds: bool = False)` → str

Render a datetime object or timestamp string as a localised date or in the requested format. If timestamp is badly formatted, then a blank string is returned.

Parameters

- **datetime** (*datetime or ISO string format*) – the date
- **date_format** (*string*) – a date format
- **with_hours** (*bool*) – should the *hours:mins* be shown
- **with_seconds** (*bool*) – should the *hours:mins:seconds* be shown

Return type
string

`ckan.lib.helpers.date_str_to_datetime(date_str: str)` → datetime

Convert ISO-like formatted datestring to datetime object.

This function converts ISO format date- and datetime-strings into datetime objects. Times may be specified down to the microsecond. UTC offset or timezone information may **not** be included in the string.

Note - Although originally documented as parsing ISO date(-times), this
function doesn't fully adhere to the format. This function will throw a ValueError if the string contains UTC offset information. So in that sense, it is less liberal than ISO format. On the other hand, it is more liberal of the accepted delimiters between the values in the string. Also, it allows microsecond precision, despite that not being part of the ISO format.

`ckan.lib.helpers.parse_rfc_2822_date(date_str: str, assume_utc: bool = True)` → datetime | None

Parse a date string of the form specified in RFC 2822, and return a datetime.

RFC 2822 is the date format used in HTTP headers. It should contain timezone information, but that cannot be relied upon.

If date_str doesn't contain timezone information, then the 'assume_utc' flag determines whether we assume this string is local (with respect to the server running this code), or UTC. In practice, what this means is that if assume_utc is True, then the returned datetime is 'aware', with an associated tzinfo of offset zero. Otherwise, the returned datetime is 'naive'.

If timezone information is available in date_str, then the returned datetime is 'aware', ie - it has an associated tz_info object.

Returns None if the string cannot be parsed as a valid datetime.

Note: in Python3, *email.utils* always assume UTC if there is no timezone, so *assume_utc* has no sense in this version.

`ckan.lib.helpers.time_ago_from_timestamp(timestamp: int)` → str

Returns a string like *5 months ago* for a datetime relative to now :param timestamp: the timestamp or datetime :type timestamp: string or datetime

Return type
string

`ckan.lib.helpers.dataset_display_name(package_or_package_dict: dict[str, Any] | Package)` → str

`ckan.lib.helpers.dataset_link(package_or_package_dict: dict[str, Any] | Package) → Markup`

`ckan.lib.helpers.resource_display_name(resource_dict: dict[str, Any]) → str`

`ckan.lib.helpers.resource_link(resource_dict: dict[str, Any], package_id: str, package_type: str = 'dataset') → Markup`

`ckan.lib.helpers.tag_link(tag: dict[str, Any], package_type: str = 'dataset') → Markup`

`ckan.lib.helpers.group_link(group: dict[str, Any]) → Markup`

`ckan.lib.helpers.organization_link(organization: dict[str, Any]) → Markup`

`ckan.lib.helpers.dump_json(obj: Any, **kw: Any) → str`

`ckan.lib.helpers.snippet(template_name: str, **kw: Any) → str`

This function is used to load html snippets into pages. keywords can be used to pass parameters into the snippet rendering

`ckan.lib.helpers.convert_to_dict(object_type: str, objs: list[Any]) → list[dict[str, Any]]`

This is a helper function for converting lists of objects into lists of dicts. It is for backwards compatability only.

`ckan.lib.helpers.follow_button(obj_type: str, obj_id: str) → str`

Return a follow button for the given object type and id.

If the user is not logged in return an empty string instead.

Parameters

- **obj_type** (*string*) – the type of the object to be followed when the follow button is clicked, e.g. 'user' or 'dataset'
- **obj_id** (*string*) – the id of the object to be followed when the follow button is clicked

Returns
a follow button as an HTML snippet

Return type
string

`ckan.lib.helpers.follow_count(obj_type: str, obj_id: str) → int`

Return the number of followers of an object.

Parameters

- **obj_type** (*string*) – the type of the object, e.g. 'user' or 'dataset'
- **obj_id** (*string*) – the id of the object

Returns
the number of followers of the object

Return type
int

`ckan.lib.helpers.add_url_param(alternative_url: str | None = None, controller: str | None = None, action: str | None = None, extras: dict[str, Any] | None = None, new_params: dict[str, Any] | None = None) → str`

Adds extra parameters to existing ones

controller action & extras (dict) are used to create the base url via `url_for()` controller & action default to the current ones

This can be overriden providing an alternative_url, which will be used instead.

`ckan.lib.helpers.remove_url_param(key: list[str] | str, value: str | None = None, replace: str | None = None, controller: str | None = None, action: str | None = None, extras: dict[str, Any] | None = None, alternative_url: str | None = None) → str`

Remove one or multiple keys from the current parameters. The first parameter can be either a string with the name of the key to remove or a list of keys to remove. A specific key/value pair can be removed by passing a second value argument otherwise all pairs matching the key will be removed. If replace is given then a new param key=replace will be added. Note that the value and replace parameters only apply to the first key provided (or the only one provided if key is a string).

controller action & extras (dict) are used to create the base url via `url_for()` controller & action default to the current ones

This can be overriden providing an alternative_url, which will be used instead.

`ckan.lib.helpers.debug_inspect(arg: Any) → Markup`

Output pprint.pformat view of supplied arg

`ckan.lib.helpers.groups_available(am_member: bool = False, include_dataset_count: bool = False, include_member_count: bool = False, user: str | None = None) → list[dict[str, Any]]`

Return a list of the groups that the user is authorized to edit.

Parameters
am_member – if True return only the groups the logged-in user is a member of, otherwise return all groups that the user is authorized to edit (for example, sysadmin users are authorized to edit all groups) (optional, default: False)

`ckan.lib.helpers.organizations_available(permission: str = 'manage_group', include_dataset_count: bool = False, include_member_count: bool = False, user: str | None = None) → list[dict[str, Any]]`

Return a list of organizations that the current user has the specified permission for.

`ckan.lib.helpers.member_count(group: str) → int`

Return the number of members belonging to the group

`ckan.lib.helpers.roles_translated() → dict[str, str]`

Return a dict of available roles with their translations

`ckan.lib.helpers.user_in_org_or_group(group_id: str) → bool`

Check if user is in a group or organization

`ckan.lib.helpers.escape_js(str_to_escape: str) → str`

Escapes special characters from a JS string.

Useful e.g. when you need to pass JSON to the templates

Parameters
str_to_escape – string to be escaped

Return type
string

`ckan.lib.helpers.get_pkg_dict_extra(pkg_dict: dict[str, Any], key: str, default: Any | None = None) → Any`

Returns the value for the dataset extra with the provided key.

If the key is not found, it returns a default value, which is None by default.

Parameters
pkg_dict – dictized dataset

Key
extra key to lookup

Default
default value returned if not found

`ckan.lib.helpers.get_request_param(parameter_name: str, default: Any | None = None) → Any`

This function allows templates to access query string parameters from the request. This is useful for things like sort order in searches.

`ckan.lib.helpers.html_auto_link(data: str) → str`

Linkifies HTML

tag converted to a tag link

dataset converted to a dataset link

group converted to a group link

http:// converted to a link

`ckan.lib.helpers.render_markdown(data: str, auto_link: bool = True, allow_html: bool = False) → str | Markup`

Returns the data as rendered markdown

Parameters

- **auto_link** (*bool*) – Should ckan specific links be created e.g. *group:xxx*
- **allow_html** (*bool*) – If True then html entities in the markdown data. This is dangerous if users have added malicious content. If False all html tags are removed.

`ckan.lib.helpers.format_resource_items(items: list[tuple[str, Any]]) → list[tuple[str, Any]]`

Take a resource item list and format nicely with blacklisting etc.

`ckan.lib.helpers.get_allowed_view_types(resource: dict[str, Any], package: dict[str, Any]) → list[tuple[str, str, str]]`

`ckan.lib.helpers.rendered_resource_view(resource_view: dict[str, Any], resource: dict[str, Any], package: dict[str, Any], embed: bool = False) → Markup`

Returns a rendered resource view snippet.

`ckan.lib.helpers.view_resource_url(resource_view: dict[str, Any], resource: dict[str, Any], package: dict[str, Any], **kw: Any) → str`

Returns url for resource. made to be overridden by extensions. i.e by resource proxy.

`ckan.lib.helpers.resource_view_is_filterable(resource_view: dict[str, Any]) → bool`

Returns True if the given resource view support filters.

`ckan.lib.helpers.resource_view_get_fields(resource: dict[str, Any]) → list[str]`

Returns sorted list of text and time fields of a datastore resource.

`ckan.lib.helpers.resource_view_is_iframed(resource_view: dict[str, Any]) → bool`

Returns true if the given resource view should be displayed in an iframe.

`ckan.lib.helpers.resource_view_icon(resource_view: dict[str, Any]) → str`

Returns the icon for a particular view type.

`ckan.lib.helpers.resource_view_display_preview(resource_view: dict[str, Any]) → bool`

Returns if the view should display a preview.

`ckan.lib.helpers.resource_view_full_page(resource_view: dict[str, Any]) → bool`

Returns if the edit view page should be full page.

`ckan.lib.helpers.remove_linebreaks(string: str) → str`

Remove linebreaks from string to make it usable in JavaScript

`ckan.lib.helpers.list_dict_filter(list_: list[dict[str, Any]], search_field: str, output_field: str, value: Any) → Any`

Takes a list of dicts and returns the value of a given key if the item has a matching value for a supplied key

Parameters

- **list** (*list of dicts*) – the list to search through for matching items
- **search_field** (*string*) – the key to use to find matching items
- **output_field** (*string*) – the key to use to output the value
- **value** – the value to search for

`ckan.lib.helpers.SI_number_span(number: int) → Markup`

outputs a span with the number in SI unit eg 14700 -> 14.7k

`ckan.lib.helpers.uploads_enabled() → bool`

`ckan.lib.helpers.get_featured_organizations(count: int = 1) → list[dict[str, Any]]`

Returns a list of favourite organization in the form of organization_list action function

`ckan.lib.helpers.get_featured_groups(count: int = 1) → list[dict[str, Any]]`

Returns a list of favourite group the form of organization_list action function

`ckan.lib.helpers.featured_group_org(items: list[str], get_action: str, list_action: str, count: int) → list[dict[str, Any]]`

`ckan.lib.helpers.resource_formats_default_file()`

`ckan.lib.helpers.resource_formats() → dict[str, list[str]]`

Returns the resource formats as a dict, sourced from the resource format JSON file.

Parameters

- **key** – potential user input value
- **value** – [canonical mimetype lowercased, canonical format (lowercase), human readable form]

Fuller description of the fields are described in *ckan/config/resource_formats.json*.

`ckan.lib.helpers.unified_resource_format(format: str) → str`

`ckan.lib.helpers.check_config_permission(permission: str) → list[str] | bool`

`ckan.lib.helpers.get_organization(org: str | None = None, include_datasets: bool = False) → dict[str, Any]`

`ckan.lib.helpers.license_options(existing_license_id: tuple[str, str] | None = None) → list[tuple[str, str]]`

Returns [(l.title, l.id), ...] for the licenses configured to be offered. Always includes the existing_license_id, if supplied.

`ckan.lib.helpers.get_translated(data_dict: dict[str, Any], field: str) → str | Any`

`ckan.lib.helpers.facets()` → list[str]

Returns a list of the current facet names

`ckan.lib.helpers.mail_to(email_address: str, name: str)` → Markup

`ckan.lib.helpers.clean_html(html: Any)` → str

`ckan.lib.helpers.load_plugin_helpers()` → None

(Re)loads the list of helpers provided by plugins.

`ckan.lib.helpers.sanitize_id(id_: str)` → str

Given an id (uuid4), if it has any invalid characters it raises ValueError.

`ckan.lib.helpers.get_collaborators(package_id: str)` → list[tuple[str, str]]

Return the collaborators list for a dataset

Returns a list of tuples with the user id and the capacity

`ckan.lib.helpers.can_update_owner_org(package_dict: dict[str, Any], user_orgs: list[dict[str, Any]] | None = None)` → bool

`ckan.lib.helpers.decode_view_request_filters()` → dict[str, Any] | None

`ckan.lib.helpers.check_ckan_version(min_version: str | None = None, max_version: str | None = None)`

Return `True` if the CKAN version is greater than or equal to `min_version` and less than or equal to `max_version`, return `False` otherwise.

If no `min_version` is given, just check whether the CKAN version is less than or equal to `max_version`.

If no `max_version` is given, just check whether the CKAN version is greater than or equal to `min_version`.

Parameters

- **min_version** (*string*) – the minimum acceptable CKAN version, eg. `'2.1'`
- **max_version** (*string*) – the maximum acceptable CKAN version, eg. `'2.3'`

`ckan.lib.helpers.make_login_url(login_view: str, next_url: str | None = None, next_field: str = 'next')` → str

Creates a URL for redirecting to a login page. If only *login_view* is provided, this will just return the URL for it. If *next_url* is provided, however, this will append a `next=URL` parameter to the query string so that the login view can redirect back to that URL.

`ckan.lib.helpers.csrf_input()`

6.12 Template snippets reference

Todo: Autodoc all the default template snippets here. This probably means writing a Sphinx plugin.

6.13 JavaScript sandbox reference

Todo: Autodoc the JavaScript sandbox. This will probably require writing a custom Sphinx plugin.

6.14 JavaScript API client reference

Todo: Autodoc the JavaScript client. This will probably require writing a custom Sphinx plugin.

6.15 CKAN jQuery plugins reference

CKAN adds a number of custom plugins that can be accessed by JavaScript modules via `this.sandbox.jQuery`.

CHAPTER

SEVEN

CONTRIBUTING GUIDE

CKAN is free open source software and contributions are welcome, whether they're bug reports, source code, documentation or translations. The following sections will walk you through our processes for making different kinds of contributions to CKAN:

7.1 Reporting issues

If you've found a bug in CKAN, open a new issue on CKAN's GitHub Issues (try searching first to see if there's already an issue for your bug).

If you can fix the bug yourself, please *send a pull request*!

Do not use an issue to ask how to do something - for that use StackOverflow with the 'ckan' tag.

Do not use an issue to suggest an significant change to CKAN - instead create an issue at https://github.com/ckan/ideas-and-roadmap.

7.1.1 Writing a good issue

- Describe what went wrong
- Say what you were doing when it went wrong
- If in doubt, provide detailed steps for someone else to recreate the problem.
- A screenshot is often helpful
- If it is a 500 error / ServerError / exception then it's essential to supply the full stack trace provided in the CKAN log.

7.1.2 Issues process

The CKAN Technical Team reviews new issues twice a week. They aim to assign someone on the Team to take responsibility for it. These are the sorts of actions to expect:

- If it is a serious bug and the person who raised it won't fix it then the Technical Team will aim to create a fix.
- A feature that you plan to code shortly will be happily discussed. It's often good to get the team's support for a feature before writing lots of code. You can then quote the issue number in the commit messages and branch name. (Larger changes or suggestions by non-contributers are better discussed on https://github.com/ckan/ideas-and-roadmap instead)
- Features may be marked "Good for Contribution" which means the Team is happy to see this happen, but the Team are not offering to do it.

7.1.3 Old issues

If an issue has little activity for 12 months then it should be closed. If someone is still keen for it to happen then they should comment, re-open it and push it forward.

7.2 Translating CKAN

CKAN is used in many countries, and adding a new language to the web interface is a simple process.

CKAN uses the url to determine which language is used. An example would be `/fr/dataset` would be shown in french. If CKAN is running under a directory then an example would be `/root/fr/dataset`. For custom paths check the *ckan.root_path* config option.

See also:

Developers, see *String internationalization* for how to mark strings for translation in CKAN code.

7.2.1 Supported languages

CKAN already supports numerous languages. To check whether your language is supported, look in the source at `ckan/i18n` for translation files. Languages are named using two-letter ISO language codes (e.g. `es`, `de`).

If your language is present, you can switch the default language simply by setting the `ckan.locale_default` option in your CKAN config file, as described in *Internationalisation Settings*. For example, to switch to German:

```
ckan.locale_default=de
```

See also:

Internationalisation Settings

If your language is not supported yet, the remainder of this section section provides instructions on how to prepare a translation file and add it to CKAN.

7.2.2 Adding a new language or improving an existing translation

If you want to add an entirely new language to CKAN or update an existing translation, you have two options.

- *Transifex setup*. Creating or updating translation files using Transifex, the open source translation software. To add a language you need to request it from the Transifex dashboard: https://www.transifex.com/okfn/ckan/dashboard/ Alternatively to update an existing language you need to request to join the appropriate CKAN language team. If you don't hear back from the CKAN administrators, contact them via the ckan-dev list.
- *Manual setup*. Creating translation files manually in your own branch.

Note: If you choose not to contribute your translation back via Transifex then you must ensure you make it public in another way, as per the requirements of CKAN's AGPL license.

Transifex setup

Transifex, the open translation platform, provides a simple web interface for writing translations and is widely used for CKAN internationalization.

Using Transifex makes it easier to handle collaboration, with an online editor that makes the process more accessible.

Existing CKAN translation projects can be found at: https://www.transifex.com/okfn/ckan/content/

When leading up to a CKAN release, the strings are loaded onto Transifex and ckan-dev list is emailed to encourage translation work. When the release is done, the latest translations on Transifex are checked back into CKAN.

Transifex administration

The Transifex workflow is described in the *Doing a CKAN release*

Manual setup

Note: Please keep the CKAN core developers aware of new languages created in this way.

All the English strings in CKAN are extracted into the `ckan.pot` file, which can be found in `ckan/i18n`.

Note: For information, the pot file was created with the `babel` command `python setup.py extract_messages`.

1. Preparation

This tutorial assumes you've got ckan installed as source in a virtualenv. Activate the virtualenv and cd to the ckan directory:

```
. /usr/lib/ckan/default/bin/activate
cd /usr/lib/ckan/default/src/ckan
```

2. Install Babel

You need Python's `babel` library (Debian package `python-pybabel`). Install it as follows with pip:

```
pip install --upgrade Babel
```

3. Create a 'po' file for your language

Then create a translation file for your language (a po file) using the pot file (containing all the English strings):

```
python setup.py init_catalog --locale YOUR_LANGUAGE
```

Replace `YOUR_LANGUAGE` with the two-letter ISO language code (e.g. `es`, `de`).

In future, when the pot file is updated, you can update the strings in your po file, while preserving your po edits, by doing:

```
python setup.py update_catalog --locale YOUR-LANGUAGE
```

2. Do the translation

Edit the po file and translate the strings. For more information on how to do this, see the Pylons book.

We recommend using a translation tool, such as poedit, to check the syntax is correct. There are also extensions for editors such as emacs.

3. Commit the translation

When the po is complete, create a branch in your source, then commit it to your own fork of the CKAN repo:

```
git add ckan/i18n/YOUR_LANGUAGE/LC_MESSAGES/ckan.po
git commit -m '[i18n]: New language po added: YOUR_LANGUAGE' ckan/i18n/YOUR_LANGUAGE/LC_
↪MESSAGES/ckan.po
```

NB it is not appropriate to do a Pull Request to the main ckan repo, since that takes its translations from Transifex.

4. Compile a translation

Once you have created a translation (either with Transifex or manually) you can build the po file into a `mo` file, ready for deployment.

With either method of creating the po file, it should be found in the CKAN i18n repository: `ckan/i18n/YOUR_LANGUAGE/LC_MESSAGES/ckan.po`

In this repo, compile the po file like this:

```
python setup.py compile_catalog --locale YOUR_LANGUAGE
```

As before, replace `YOUR_LANGUAGE` with your language short code, e.g. `es`, `de`.

This will result in a binary 'mo' file of your translation at `ckan/i18n/YOUR_LANGUAGE/LC_MESSAGES/ckan.mo`.

5. (optional) Deploy the translation

This section explains how to deploy your translation to your CKAN server.

Once you have a compiled translation file, copy it to your host:

```
scp ckan.mo /usr/lib/ckan/default/src/ckan/ckan/i18n/hu/LC_MESSAGES/ckan.mo
```

Adjust the path if you did not use the default location. This example is for language `hu`.

6. Configure the language

Finally, once the mo file is in place, you can switch between the installed languages using the `ckan.locale` option in the CKAN config file, as described in *Internationalisation Settings*.

7.2.3 Translations management policy

One of the aims of CKAN is to be accessible to the greatest number of users. Translating the user interface to as many languages as possible plays a huge part in this, and users are encouraged to contribute to the existing translations or submit a new one. At the same time we need to ensure the stability between CKAN releases, so the following guidelines apply when managing translations:

- About 3 weeks before a CKAN release, CKAN is branched, and the English strings are frozen, and an announcement is made on ckan-dev to call for translation work. They are given 2 weeks to translate any new strings in this release.
- During this period, translation is done on a 'resource' on Transifex which is named to match the new CKAN version. It has been created as a copy of the next most recent resource, so any new languages create or other updates done on Transifex since the last release automatically go into the new release.

7.3 Testing CKAN

If you're a CKAN developer, if you're developing an extension for CKAN, or if you're just installing CKAN from source, you should make sure that CKAN's tests pass for your copy of CKAN. This section explains how to run CKAN's tests.

CKAN's testsuite contains automated tests for both the back-end (Python) and the front-end (JavaScript). In addition, the correct functionality of the complete front-end (HTML, CSS, JavaScript) on all supported browsers should be tested manually.

See also:

CKAN coding standards for tests
: Conventions for writing tests for CKAN

7.3.1 Back-end tests

Most of CKAN's testsuite is for the backend Python code. You can run the code in a dockerized environment that replicates circleci, or you can use a virtual environment based testing.

Dockerized Tests

The `test-infrastructure` directory contains a configuration using docker compose replicating the circleci test process on the local machine.

Set up the testing environment

```
cd test-infrastructure
./setup.sh
```

This starts a docker compose environment with the supporting postgres, redis, and solr containers from the circleci test environment. The databases are initialized, and the current ckan is installed into a python container.

Run the tests

```
./execute.sh
```

Or, if you wish to run a specific test, for example `test_get_translated` in `test_helpers.py`:

```
docker compose exec ckan pytest --ckan-ini=test-core-circle-ci.ini ckan/tests/lib/test_
↪helpers.py::test_get_translated
```

Teardown

```
./teardown.sh
```

Virtual Environment based tests

Install additional dependencies

Some additional dependencies are needed to run the tests. Make sure you've created a config file at /etc/ckan/default/ckan.ini, then activate your virtual environment:

```
. /usr/lib/ckan/default/bin/activate
```

Install pytest and other test-specific CKAN dependencies into your virtual environment:

```
pip install -r /usr/lib/ckan/default/src/ckan/dev-requirements.txt
```

Set up the test databases

Create test databases:

```
sudo -u postgres createdb -O ckan_default ckan_test -E utf-8
sudo -u postgres createdb -O ckan_default datastore_test -E utf-8
```

Set the permissions:

```
ckan -c test-core.ini datastore set-permissions | sudo -u postgres psql
```

When the tests run they will use these databases, because in `test-core.ini` they are specified in the `sqlalchemy.url` and `ckan.datastore.write_url` connection strings.

You should also make sure that the *Redis database* configured in `test-core.ini` is different from your production database.

Configure Solr Multi-core

The tests assume that Solr is configured 'multi-core', whereas the default Solr set-up is often 'single-core'. You can ask Solr for its cores status:

```
curl -s 'http://127.0.0.1:8983/solr/admin/cores?action=STATUS' |python -c 'import sys;
↪import xml.dom.minidom;s=sys.stdin.read();print(xml.dom.minidom.parseString(s).
↪toprettyxml())'
```

Each core will be within a child from the `<lst name="status"` element, and contain a `<str name="instanceDir">` element.

You can also tell from your ckan config (assuming ckan is working):

```
grep solr_url |ckan.ini|
# single-core: solr_url = http://127.0.0.1:8983/solr
# multi-core:  solr_url = http://127.0.0.1:8983/solr/ckan
```

To enable multi-core:

1. Find the `instanceDir` of the existing Solr core. It is found in the output of the curl command above.

   e.g. `/usr/share/solr/` or `/opt/solr/example/solr/collection1`

2. Make a copy of that core's directory e.g.:

   ```
   sudo cp -r /usr/share/solr/ /etc/solr/ckan
   ```

3. Find your solr.xml. It is in the Solr Home directory given by this command:

   ```
   curl -s 'http://127.0.0.1:8983/solr/admin/' | grep SolrHome
   ```

4. Configure Solr with the new core by editing `solr.xml`. The 'cores' section will have one 'core' in it already and needs the second one 'ckan' added so it looks like this:

   ```
   <cores adminPath="/admin/cores" defaultCoreName="collection1">
     <core name="collection1" instanceDir="." />
     <core name="ckan" instanceDir="/etc/solr/ckan" />
   </cores>
   ```

5. Restart Solr by restarting Jetty (or Tomcat):

   ```
   sudo service jetty restart
   ```

6. Edit your main ckan config (e.g. /etc/ckan/default/ckan.ini) and adjust the solr_url to match:

   ```
   solr_url = http://127.0.0.1:8983/solr/ckan
   ```

Run the tests

To run CKAN's tests using PostgreSQL as the database, you have to give the `--ckan-ini=test-core.ini` option on the command line. This command will run the tests for CKAN core and for the core extensions:

```
pytest --ckan-ini=test-core.ini ckan/ ckanext/
```

The speed of the PostgreSQL tests can be improved by running PostgreSQL in memory and turning off durability, as described in the PostgreSQL documentation.

Common error messages

OperationalError

`OperationalError: (OperationalError) no such function: plainto_tsquery ...`

This error usually results from running a test which involves search functionality, which requires using a PostgreSQL database, but another (such as SQLite) is configured. The particular test is either missing a *@search_related* decorator or there is a mixup with the test configuration files leading to the wrong database being used.

SolrError

```
SolrError: Solr responded with an error (HTTP 404): [Reason: None]
<html><head><meta content="text/html; charset=ISO-8859-1" http-equiv="Content-Type" />
↪<title>Error 404 NOT_FOUND</title></head><body><h2>HTTP ERROR 404</h2><p>Problem␣
↪accessing /solr/ckan/select/. Reason:<pre>    NOT_FOUND</pre></p><hr /><i><small>
↪Powered by Jetty://</small></i>``
```

This means your solr_url is not corresponding with your SOLR. When running tests, it is usually easiest to change your set-up to match the default solr_url in test-core.ini. Often this means switching to multi-core - see *Configure Solr Multi-core*.

7.3.2 Front-end tests

Front-end testing consists of both automated tests (for the JavaScript code) and manual tests (for the complete front-end consisting of HTML, CSS and JavaScript).

Automated JavaScript tests

The JS tests are written using the Cypress test framework. First you need to install the necessary packages:

```
sudo apt-get install npm nodejs-legacy
sudo npm install
```

To run the tests, make sure that a test server is running:

```
. /usr/lib/ckan/default/bin/activate
ckan -c |ckan.ini| run
```

Once the test server is running switch to another terminal and execute the tests:

```
npx cypress run
```

Manual tests

All new CKAN features should be coded so that they work in the following browsers:

- Internet Explorer: 11, 10, 9 & 8
- Firefox: Latest + previous version
- Chrome: Latest + previous version

Install browser virtual machines

In order to test in all the needed browsers you'll need access to all the above browser versions. Firefox and Chrome should be easy whatever platform you are on. Internet Explorer is a little trickier. You'll need Virtual Machines.

We suggest you use https://github.com/xdissent/ievms to get your Internet Explorer virtual machines.

Testing methodology

Firstly we have a primer page. If you've touched any of the core front-end code you'll need to check if the primer is rendering correctly. The primer is located at: http://localhost:5000/testing/primer

Secondly whilst writing a new feature you should endeavour to test in at least in your core browser and an alternative browser as often as you can.

Thirdly you should fully test all new features that have a front-end element in all browsers before making your pull request into CKAN master.

Common front-end pitfalls & their fixes

Here's a few of the most common front end bugs and a list of their fixes.

Reserved JS keywords

Since IE has a stricter language definition in JS it really doesn't like you using JS reserved keywords method names, variables, etc… This is a good list of keywords not to use in your JavaScript:

https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Reserved_Words

```
/* These are bad */
var a = {
  default: 1,
  delete: function() {}
};

/* These are good */
var a = {
  default_value: 1,
  remove: function() {}
};
```

Unclosed JS arrays / objects

Internet Explorer doesn't like it's JS to have unclosed JS objects and arrays. For example:

```
/* These are bad */
var a = {
  b: 'c',
};
var a = ['b', 'c', ];

/* These are good */
var a = {
  c: 'c'
};
var a = ['b', 'c'];
```

7.4 Writing commit messages

We use the version control system git for our code and documentation, so when contributing code or docs you'll have to commit your changes to git and write a git commit message. Generally, follow the commit guidelines from the Pro Git book:

- Try to make each commit a logically separate, digestible changeset.
- The first line of the commit message should concisely summarise the changeset.
- Optionally, follow with a blank line and then a more detailed explanation of the changeset.
- Use the imperative present tense as if you were giving commands to the codebase to change its behaviour, e.g. *Add tests for...*, *make xyzzy do frotz...*, this helps to make the commit message easy to read.

If your commit has an issue in the CKAN issue tracker put the issue number at the start of the first line of the commit message like this: `[#123]`. This makes the CKAN release manager's job much easier!

Here's an example of a good CKAN commit message:

```
[#607] Allow reactivating deleted datasets

Currently if a dataset is deleted and users navigate to the edit form,
there is no state field and the delete button is still shown.

After this change, the state dropdown is shown if the dataset state is
not active, and the delete button is not shown.
```

If your PR provides change that should be mentioned in changelog(generally, any PR is good to mention), consider creating "changelog fragment". It's a file inside **changes** folder in the root of the repository, which will be used for generating changelog when preparing new CKAN release. This file must follow naming convention `{issue number}.{change type}`, where **issue number** is a identified of issue or PR in the CKAN issue tracker and type is one of the following, depending on change type:

- **migration** - fragment intorduces migration guide for existing CKAN instances
- **bugfix** - some issue was fixed.
- **removal** - function/class/module was removed or deprecated
- **misc** - another small changes, like additional logging or code-style fixes

The same PR can provide multiple fragments with different type. For example, removing some code for issue with number **1234** in tracker, following files can be added to **changes** folder:

```
# changes/1234.removal
Module ``xxx`` marked as deprecated.

# changes/1234.migration
Replace all import from ``xxx`` with corresponding imports from ``yyy``.
```

7.5 Making a pull request

Once you've written some CKAN code or documentation, you can submit it for review and merge into the central CKAN git repository by making a pull request. This section will walk you through the steps for making a pull request.

Note: Except in some special cases, all pull requests should target the `master` branch. The tech team will backport the change to the relevant release branches (or ask you to submit a separate pull request against a release branch), but all changes should be present in the `master` branch first so they don't get lost in future versions.

1. Create a git branch

   Each logically separate piece of work (e.g. a new feature, a bug fix, a new docs page, or a set of improvements to a docs page) should be developed on its own branch forked from the master branch.

   The name of the branch should include the issue number (if this work has an issue in the CKAN issue tracker), and a brief one-line synopsis of the work, for example:

   ```
   2298-add-sort-by-controls-to-search-page
   ```

2. Fork CKAN on GitHub

   Sign up for a free account on GitHub and fork CKAN, so that you have somewhere to publish your work.

   Add your CKAN fork to your local CKAN git repo as a git remote. Replace `USERNAME` with your GitHub username:

   ```
   git remote add my_fork https://github.com/USERNAME/ckan
   ```

3. Commit and push your changes

   Commit your changes on your feature branch, and push your branch to GitHub. For example, make sure you're currently on your feature branch then run these commands:

   ```
   git add doc/my_new_feature.rst
   git commit -m "Add docs for my new feature"
   git push my_fork my_branch
   ```

   When writing your git commit messages, try to follow the *Writing commit messages* guidelines.

4. Send a pull request

   Once your work on a branch is complete and is ready to be merged into the master branch, create a pull request on GitHub. A member of the CKAN team will review your work and provide feedback on the pull request page. The reviewer may ask you to make some changes. Once your pull request has passed the review, the reviewer will merge your code into the master branch and it will become part of CKAN!

   When submitting a pull request:

- Your branch should contain one logically separate piece of work, and not any unrelated changes.
- You should have good commit messages, see *Writing commit messages*.
- Your branch should contain new or changed tests for any new or changed code, and all the CKAN tests should pass on your branch, see *Testing CKAN*.
- Your pull request shouldn't lower our test coverage. You can check it at our *coveralls page <https://coveralls.io/r/ckan/ckan>*. If for some reason you can't avoid lowering it, explain why on the pull request.
- Your branch should contain new or updated documentation for any new or updated code, see *Writing documentation*.
- Your branch should be up to date with the master branch of the central CKAN repo, so pull the central master branch into your feature branch before submitting your pull request.

  For long-running feature branches, it's a good idea to pull master into the feature branch periodically so that the two branches don't diverge too much.

7.6 Reviewing and merging a pull request

Of course it's not possible to give an exact recipe for reviewing a pull request, you simply have to assess the code and decide whether you're happy with it. Nonetheless, here is an incomplete list of things to look for:

- Does the pull request contain one logically separate piece of work (e.g. one new feature, bug fix, etc. per pull request)?
- Does the pull request follow the guidelines for *writing commit messages*?
- Is the branch up to date - have the latest commits from master been pulled into the branch?
- Does the pull request contain new or updated tests for any new or updated code, and do the tests follow *CKAN's testing coding standards*?
- Do all the CKAN tests pass, on the new branch?
- Does the pull request contain new or updated docs for any new or updated features, and do the docs follow *CKAN's documentation guidelines*?
- Does the new code follow CKAN's code architecture and the various coding standards for Python, JavaScript, etc.?
- If the new code contains changes to the database schema, does it have a *database migration*?
- Does the code contain any changes that break backwards-incompatibility? If so, is the breakage necessary or do the benefits of the change justify the breakage? Have the breaking changes been added to the *changelog*?

  Backwards-compability needs to be considered when making changes that break the interfaces that CKAN provides to third-party code, including API clients, plugins and themes.

  In general, any code that's documented in the reference sections of the *API*, *extensions* or *theming* needs to be considered. For example this includes changes to the API actions, the plugin interfaces or plugins toolkit, the converter and validator functions (which are used by plugins), the custom Jinja2 tags and variables available to Jinja templates, the template helper functions, the core template files and their blocks, the sandbox available to JavaScript modules (including custom jQuery plugins and the JavaScript CKAN API client), etc.
- Does the new code add any dependencies to CKAN (e.g. new third-party Python modules imported)? If so, is the new dependency justified and has it been added following the right process? See *Upgrading CKAN's dependencies*.

7.6.1 Merging a pull request

Once you've reviewed a pull request and you're happy with it, you need to merge it into the master branch. You should do this using the `--no-ff` option in the `git merge` command. For example:

```
git checkout feature-branch
git pull origin feature-branch
git checkout master
git pull origin master
git merge --no-ff feature-branch
git push origin master
```

Before doing the `git push`, it's a good idea to check that all the tests are passing on your master branch (if the latest commits from master have already been pulled into the feature branch on github, then it may be enough to check that all tests passed for the latest commit on this branch on Circle CI).

Also before doing the `git push`, it's a good idea to use `git log` and/or `git diff` to check the difference between your local master branch and the remote master branch, to make sure you only push the changes you intend to push:

```
git log ...origin/master
git diff ..origin/master
```

7.7 Writing documentation

This section gives some guidelines to help us to write consistent and good quality documentation for CKAN.

Documentation isn't source code, and documentation standards don't need to be followed as rigidly as coding standards do. In the end, some documentation is better than no documentation, it can always be improved later. So the guidelines below are soft rules.

Having said that, we suggest just one hard rule: **no new feature (or change to an existing feature) should be missing from the docs** (but see *todo*).

See also:

Jacob Kaplan-Moss's Writing Great Documentation
A series of blog posts about writing technical docs, a lot of our guidelines were based on this.

See also:

The quickest and easiest way to contribute documentation to CKAN is to sign up for a free GitHub account and simply edit the CKAN Wiki. Docs started on the wiki can make it onto docs.ckan.org later. If you do want to contribute to docs.ckan.org directly, follow the instructions on this page.

Tip: Use the reStructuredText markup format when creating a wiki page, since reStructuredText is the format that docs.ckan.org uses, this will make moving the documentation from the wiki into docs.ckan.org later easier.

7.7.1 Getting started

This section will walk you through downloading the source files for CKAN's docs, editing them, and submitting your work to the CKAN project.

CKAN's documentation is created using Sphinx, which in turn uses Docutils (reStructuredText is part of Docutils). Some useful links to bookmark:

- Sphinx's reStructuredText Primer
- reStructuredText cheat sheet
- reStructuredText quick reference
- Sphinx Markup Constructs is a full list of the markup that Sphinx adds on top of Docutils.

The source files for the docs are in the doc directory of the CKAN git repo. The following sections will walk you through the process of making changes to these source files, and submitting your work to the CKAN project.

Install CKAN into a virtualenv

Create a Python virtual environment (virtualenv), activate it, install CKAN into the virtual environment, and install the dependencies necessary for building CKAN. In this example we'll create a virtualenv in a folder called `pyenv`. Run these commands in a terminal:

```
virtualenv --no-site-packages pyenv
. pyenv/bin/activate
pip install -e 'git+https://github.com/ckan/ckan.git#egg=ckan'
cd pyenv/src/ckan/
pip install -r dev-requirements.txt
pip install -r requirements.txt
```

Build the docs

You should now be able to build the CKAN documentation locally. Make sure your virtual environment is activated, and then run this command:

```
sphinx-build doc build/sphinx
```

Now you can open the built HTML files in `build/sphinx/html`, e.g.:

```
firefox build/sphinx/html/index.html
```

Edit the reStructuredText files

To make changes to the documentation, use a text editor to edit the `.rst` files in `doc/`. Save your changes and then build the docs again (`sphinx-build doc build/sphinx`) and open the HTML files in a web browser to preview your changes.

Once your docs are ready to submit to the CKAN project, follow the steps in *Making a pull request*.

7.7.2 How the docs are organized

It's important that the docs have a clear, simple and extendable structure (and that we keep it that way as we add to them), so that both readers and writers can easily answer the questions: If you need to find the docs for a particular feature, where do you look? If you need to add a new page to the docs, where should it go?

As /index explains, the documentation is organized into several guides, each for a different audience: a user guide for web interface users, an extending guide for extension developers, a contributing guide for core contributors, etc. These guides are ordered with the simplest guides first, and the most advanced last.

In the source, each one of these guides is a subdirectory with its own `index.rst` containing its own `.. toctree::` directive that lists all of the other files in that subdirectory. The root toctree just lists each of these `*/index.rst` files.

When adding a new page to the docs, the first question to ask yourself is: who is this page for? That should tell you which subdirectory to put your page in. You then need to add your page to that subdirectory's `index.rst` file.

Within each guide, the docs are broken up by topic. For example, the extending guide has a page for the writing extensions tutorial, a page about testing extensions, a page for the plugins toolkit reference, etc. Again, the topics are ordered with the simplest first and the most advanced last, and reference pages generally at the very end.

The changelog is one page that doesn't fit into any of the guides, because it's relevant to all of the different audiences and not only to one particular guide. So the changelog is simply a top-level page on its own. Hopefully we won't need to add many more of these top-level pages. If you're thinking about adding a page that serves two or more audiences at once, ask yourself whether you can break that up into separate pages and put each into one of the guides, then link them together using *seealso* boxes.

Within a particular page, for example a new page documenting a new feature, our suggestion for what sections the page might have is:

1. **Overview**: a conceptual overview of or introduction to the feature. Explain what the feature provides, why someone might want to use it, and introduce any key concepts users need to understand. This is the **why** of the feature.

   If it's developer documentation (extension writing, theming, API, or core developer docs), maybe put an architecture guide here.

2. **Tutorials**: tutorials and examples for how to setup the feature, and how to use the feature. This is the **how**.
3. **Reference**: any reference docs such as config options or API functions.
4. **Troubleshooting**: common error messages and problems, FAQs, how to diagnose problems.

Subdirectories

Some of the guides have subdirectories within them. For example *Maintainer's guide* contains a subdirectory *Installing CKAN* that collects together the various pages about installing CKAN with its own `doc/maintaining/installing/index.rst` file.

While subdirectories are useful, we recommend that you **don't put further subdirectories inside the subdirectories**, try to keep it to at most two levels of subdirectories inside the `doc` directory. Keep it simple, otherwise the structure becomes confusing, difficult to get an overview of and difficult to navigate.

Linear ordering

Keep in mind that Sphinx requires the docs to have a simple, linear ordering. With HTML pages it's possible to design structure where, for example, someone reads half of a page, then clicks on a link in the middle of the page to go and read another page, then goes back to the middle of the first page and continues reading where they left off. While technically you can do this in Sphinx as well, it isn't a good idea, things like the navigation links, table of contents, and PDF version will break, users will end up going in circles, and the structure becomes confusing.

So the pages of our Sphinx docs need to have a simple linear ordering - one page follows another, like in a book.

7.7.3 Sphinx

This section gives some useful tips about using Sphinx.

Don't introduce any new Sphinx warnings

When you build the docs, Sphinx prints out warnings about any broken cross-references, syntax errors, etc. We aim not to have any of these warnings, so when adding to or editing the docs make sure your changes don't introduce any new ones.

It's best to delete the `build` directory and completely rebuild the docs, to check for any warnings:

```
rm -rf build; sphinx-build doc build/sphinx
```

Maximum line length

As with Python code, try to limit all lines to a maximum of 79 characters.

versionadded and versionchanged

Use Sphinx's `versionadded` and `versionchanged` directives to mark new or changed features. For example:

```
================
Tag vocabularies
================

.. versionadded:: 1.7

CKAN sites can have *tag vocabularies*, which are a way of grouping related
tags together into custom fields.

...
```

With `versionchanged` you usually need to add a sentence explaining what changed (you can also do this with `versionadded` if you want):

```
=============
Authorization
=============

.. versionchanged:: 2.0
```

(continues on next page)

(continued from previous page)

```
   Previous versions of CKAN used a different authorization system.

CKAN's authorization system controls which users are allowed to carry out
which...
```

Cross-references and links

Whenever mentioning another page or section in the docs, an external website, a configuration setting, or a class, exception or function, etc. try to cross-reference it. Using proper Sphinx cross-references is better than just typing things like "see above/below" or "see section foo" because Sphinx cross-refs are hyperlinked, and because if the thing you're referencing to gets moved or deleted Sphinx will update the cross-reference or print a warning.

Cross-referencing to another file

Use `:doc:` to cross-reference to other files by filename:

```
See :doc:`configuration`
```

If the file you're editing is in a subdir within the `doc` dir, you may need to use an absolute reference (starting with a `/`):

```
See :doc:`/configuration`
```

See Cross-referencing documents for details.

Cross-referencing a section within a file

Use `:ref:` to cross-reference to particular sections within the same or another file. First you have to add a label before the section you want to cross-reference to:

```
.. _getting-started:

---------------
Getting started
---------------
```

then from elsewhere cross-reference to the section like this:

```
See :ref:`getting-started`.
```

see Cross-referencing arbitrary locations.

Cross-referencing to CKAN config settings

Whenever you mention a CKAN config setting, make it link to the docs for that setting in *Configuration Options* by using `:ref:` and the name of the config setting:

```
:ref:`ckan.site_title`
```

This works because all CKAN config settings are documented in *Configuration Options*, and every setting has a Sphinx label that is exactly the same as the name of the setting, for example:

```
.. _ckan.site_title:

ckan.site_title
^^^^^^^^^^^^^^^

Example::

ckan.site_title = Open Data Scotland

Default value:  ``CKAN``

This sets the name of the site, as displayed in the CKAN web interface.
```

If you add a new config setting to CKAN, make sure to document like this it in *Configuration Options*.

Cross-referencing to a Python object

Whenever you mention a Python function, method, object, class, exception, etc. cross-reference it using a Sphinx domain object cross-reference. See *Referencing other code objects with :py:*.

Changing the link text of a cross-reference

With `:doc:` `:ref:` and other kinds of link, if you want the link text to be different from the title of the thing you're referencing, do this:

```
:doc:`the theming document </theming>`

:ref:`the getting started section <getting-started>`
```

Cross-referencing to an external page

The syntax for linking to external URLs is slightly different from cross-referencing, you have to add a trailing underscore:

```
`Link text <http://example.com/>`_
```

or to define a URL once and then link to it in multiple places, do:

```
This is `a link`_ and this is `a link`_ and this is
`another link <a link>`_.
```

(continues on next page)

(continued from previous page)

```
.. _a link: http://example.com/
```

see Hyperlinks for details.

Substitutions

Substitutions are a useful way to define a value that's needed in many places (eg. a command, the location of a file, etc.) in one place and then reuse it many times.

You define the value once like this:

```
.. |ckan.ini| replace:: /etc/ckan/default/ckan.ini
```

and then reuse it like this:

```
Now open your |ckan.ini| file.
```

`|ckan.ini|` will be replaced with the full value `/etc/ckan/default/ckan.ini`.

Substitutions can also be useful for achieving consistent spelling and capitalization of names like reStructuredText, PostgreSQL, Nginx, etc.

The `rst_epilog` setting in `doc/conf.py` contains a list of global substitutions that can be used from any file.

Substitutions can't immediately follow certain characters (with no space in-between) or the substitution won't work. If this is a problem, you can insert an escaped space, the space won't show up in the generated output and the substitution will work:

```
pip install -e 'git+\ |git_url|'
```

Similarly, certain characters are not allowed to immediately follow a substitution (without a space) or the substitution won't work. In this case you can just escape the following characters, the escaped character will show up in the output and the substitution will work:

```
pip install -e 'git+\ |git_url|\#egg=ckan'
```

Also see *Parsed literals* below for using substitutions in code blocks.

Parsed literals

Normally things like links and substitutions don't work within a literal code block. You can make them work by using a `parsed-literal` block, for example:

```
Copy your development.ini file to create a new production.ini file::

.. parsed-literal::

   cp |development.ini| |production.ini|
```

autodoc

We try to use autodoc to pull documentation from source code docstrings into our Sphinx docs, wherever appropriate. This helps to avoid duplicating documentation and also to keep the documentation closer to the code and therefore more likely to be kept up to date.

Whenever you're writing reference documentation for modules, classes, functions or methods, exceptions, attributes, etc. you should probably be using autodoc. For example, we use autodoc for the *Action API reference*, the *Plugin interfaces reference*, etc.

For how to write docstrings, see *Docstrings*.

todo

No new feature (or change to an existing feature) should be missing from the docs. It's best to document new features or changes as you implement them, but if you really need to merge something without docs then at least add a todo directive to mark where docs need to be added or updated (if it's a new feature, make a new page or section just to contain the `todo`):

```
=====================================
CKAN's builtin social network feature
=====================================

.. todo::

   Add docs for CKAN's builtin social network for data hackers.
```

deprecated

Use Sphinx's deprecated directive to mark things as deprecated in the docs:

```
.. deprecated:: 3.1
   Use :func:`spam` instead.
```

seealso

Often one page of the docs is related to other pages of the docs or to external pages. A seealso block is a nice way to include a list of related links:

```
.. seealso::

   :doc:`The DataStore extension <datastore>`
     A CKAN extension for storing data.

   CKAN's `demo site <https://demo.ckan.org/>`_
     A demo site running the latest CKAN beta version.
```

Seealso boxes are particularly useful when two pages are related, but don't belong next to each other in the same section of the docs. For example, we have docs about how to upgrade CKAN, these belong in the maintainer's guide because they're for maintainers. We also have docs about how to do a new release, these belong in the contributing guide because they're for developers. But both sections are about CKAN releases, so we link each to the other using seealso boxes.

7.7.4 Code examples

If you're going to paste example code into the docs, or add a tutorial about how to do something with code, then:

1. The code should be in standalone Python, HTML, JavaScript etc. files, not pasted directly into the `.rst` files. You then pull the code into your `.rst` file using a Sphinx `.. literalinclude::` directive (see example below).
2. The code in the standalone files should be a complete working example, with tests. Note that not all of the code from the example needs to appear in the docs, you can include just parts of it using `.. literalinclude::`, but the example code needs to be complete so it can be tested.

This is so that we don't end up with a lot of broken, outdated examples and tutorials in the docs because breaking changes have been made to CKAN since the docs were written. If your example code has tests, then when someone makes a change in CKAN that breaks your example those tests will fail, and they'll know they have to fix their code or update your example.

The *plugins tutorial* is an example of this technique. ckanext/example_iauthfunctions is a complete and working example extension. The tests for the extension are in ckanext/example_iauthfunctions/tests. Different parts of the reStructuredText file for the tutorial pull in different parts of the example code as needed, like this:

```
.. literalinclude:: ../../ckanext/example_iauthfunctions/plugin_v3.py
   :start-after: # We have the logged-in user's user name, get their user id.
   :end-before: # Finally, we can test whether the user is a member of the curators␣
↪group.
```

`literalinclude` has a few useful options for pulling out just the part of the code that you want. See the Sphinx docs for literalinclude for details.

You may notice that ckanext/example_iauthfunctions contains multiple versions of the same example plugin, `plugin_v1.py`, `plugin_v2.py`, etc. This is because the tutorial walks the user through first making a trivial plugin, and then adding more and more advanced features one by one. Each step of the tutorial needs to have its own complete, standalone example plugin with its own tests.

For more examples, look into the source files for other tutorials in the docs.

7.7.5 Style

This section covers things like what tone to use, how to capitalize section titles, etc. Having a consistent style will make the docs nice and easy to read and give them a complete, quality feel.

Use American spelling

Use American spellings everywhere: organization, authorization, realize, customize, initialize, color, etc. There's a list here: https://wiki.ubuntu.com/EnglishTranslation/WordSubstitution

Spellcheck

Please spellcheck documentation before merging it into master!

Commonly used terms

CKAN
Should be written in ALL-CAPS.

email
Use email not e-mail.

PostgreSQL, SQLAlchemy, Nginx, Python, SQLite, JavaScript, etc.
These should always be capitalized as shown above (including capital first letters for Python and Nginx even when they're not the first word in a sentence). `doc/conf.py` defines substitutions for each of these so you don't have to remember them, see *Substitutions*.

Web site
Two words, with Web always capitalized

frontend
Not front-end

command line
Two words, not commandline or command-line (this is because we want to be like Neal Stephenson)

CKAN config file or configuration file
Not settings file, ini file, etc. Also, the **config file** contains **config options** such as `ckan.site_id`, and each config option is **set** to a certain **setting** or **value** such as `ckan.site_id = demo.ckan.org`.

Section titles

Capitalization in section titles should follow the same rules as in normal sentences: you capitalize the first word and any proper nouns.

This seems like the easiest way to do consistent capitalization in section titles because it's a capitalization rule that we all know already (instead of inventing a new one just for section titles).

Right:

- Installing CKAN from package
- Getting started
- Command line interface
- Writing extensions
- Making an API request
- You're done!
- Libraries available to extensions

Wrong:

- Installing CKAN from Package
- Getting Started
- Command Line Interface
- Writing Extensions
- Making an API Request
- You're Done!
- Libraries Available To Extensions

For lots of examples of this done right, see Django's table of contents.

In Sphinx, use the following section title styles:

```
===============
Top-level title
===============

------------------
Second-level title
------------------

Third-level title
=================

Fourth-level title
------------------
```

If you need more than four levels of headings, you're probably doing something wrong, but see: http://docutils.sourceforge.net/docs/ref/rst/restructuredtext.html#sections

Be conversational

Write in a friendly, conversational and personal tone:

- Use contractions like don't, doesn't, it's etc.
- Use "we", for example *"We'll publish a call for translations to the ckan-dev and ckan-discuss mailing lists, announcing that the new version is ready to be translated"* instead of *"A call for translations will be published"*.
- Refer to the reader personally as "you", as if you're giving verbal instructions to someone in the room: *"First, you'll need to do X. Then, when you've done Y, you can start working on Z"* (instead of stuff like *"First X must be done, and then Y must be done…"*).

Write clearly and concretely, not vaguely and abstractly

Politics and the English Language has some good tips about this, including:

1. Never use a metaphor, simile, or other figure of speech which you are used to seeing in print.
2. Never use a long word where a short one will do.
3. If it's possible to cut out a word, always cut it out.
4. Never use the passive when you can be active.
5. Never use a foreign phrase, scientific word or jargon word if you can think of an everyday English equivalent.

This will make your meaning clearer and easier to understand, especially for people whose first language isn't English.

Facilitate skimming

Readers skim technical documentation trying to quickly find what's important or what they need, so break walls of text up into small, visually identifiable pieces:

- Use lots of inline markup:

  ```
  *italics*
  **bold**
  ``code``
  ```

  For code samples or filenames with variable parts, uses Sphinx's :samp: and :file: directives.

- Use lists to break up text.
- Use `.. note::` and `.. warning::`, see Sphinx's paragraph-level markup.

  (reStructuredText actually supports lots more of these: `attention`, `error`, `tip`, `important`, etc. but most Sphinx themes only style `note` and `warning`.)

- Break text into short paragraphs of 5-6 sentences each max.
- Use section and subsection headers to visualize the structure of a page.

7.8 Projects for beginner CKAN developers

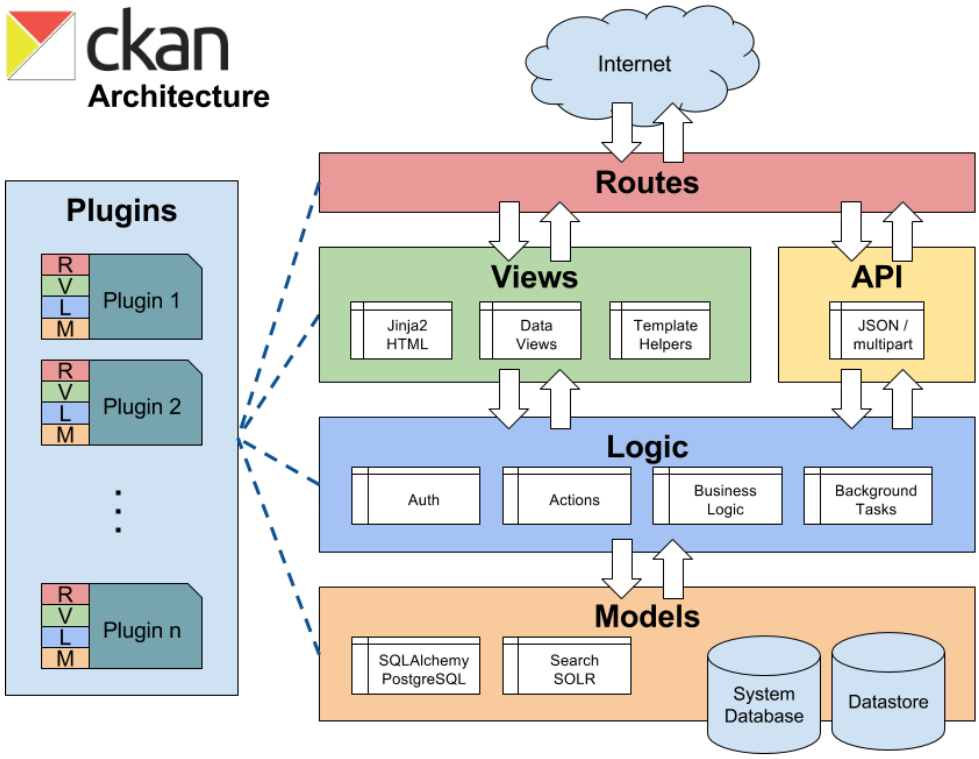
The 'Good for Contribution' label on Github lists issues which are feasible for people who aren't intimately familiar with CKAN's internals. Many of them are things which would be extremely helpful if they got done, but the core team never seems to get around to them. They're all busy wrestling with the problems that do require familiarity with the internals. We hope this will make it easier for more people to assist the CKAN project, by giving new developers places to jump in.

These issues vary from simple text changes to more complicated code changes that require more knowledge of Python and CKAN. Do not despair if any individual task seems daunting; there's probably an easier one. If you have no programming skills, we can still use your help with documentation or translation.

If you wish to take up an issue make sure to keep in touch with the team on the Github issue itself, the ckan-dev mailing list, or the CKAN chat on Gitter.

7.9 CKAN code architecture

This section documents our CKAN-specific coding standards, which are guidelines for writing code that is consistent with the intended design and architecture of CKAN.



7.9.1 Blueprints

CKAN is based on Flask and built using Blueprints.

Default blueprints are defined along views in `ckan.views` and extended with the *`ckan.plugins.interfaces.IBlueprint`* plugin interface.

7.9.2 Views

Views process requests by reading and updating data with action function and return a response by rendering Jinja2 templates. CKAN views are defined in `ckan.views` and templates in `ckan.templates`.

Views and templates may use `logic.check_access` or *`ckan.lib.helpers.check_access()`* to hide links or render helpful errors but action functions, not views, are responsible for actually enforcing permissions checking.

Plugins define new views by adding or updating routes. For adding templates or helper functions from a plugin see *Theming guide* and *Adding your own template helper functions*.

Template helper functions

Template helper functions are used for code that is reused frequently or code that is too complicated to be included in the templates themselves.

Template helpers should never perform expensive queries or update data.

`ckan.lib.helpers` contains helper functions that can be used from `ckan.controllers` or from templates. When developing for ckan core, only use the helper functions found in `ckan.lib.helpers.__allowed_functions__`.

Always go through the action functions

Whenever some code, for example in `ckan.lib` or `ckan.controllers`, wants to get, create, update or delete an object from CKAN's model it should do so by calling a function from the `ckan.logic.action` package, and *not* by accessing `ckan.model` directly.

Use `get_action()`

Don't call `logic.action` functions directly, instead use `get_action()`. This allows plugins to override action functions using the `IActions` plugin interface. For example:

```
ckan.logic.get_action('group_activity_list')(...)
```

Instead of

```
ckan.logic.action.get.group_activity_list(...)
```

Views and templates may check authorization to avoid rendering

Don't pass ORM objects to templates

Don't pass SQLAlchemy ORM objects (e.g. `ckan.model.User` objects) to templates (for example by adding them to *c*, passing them to `render()` in the `extra_vars` dict, returning them from template helper functions, etc.)

Using ORM objects in the templates often creates SQLAlchemy "detached instance" errors that cause 500 Server Errors and can be difficult to debug.

7.9.3 Logic

Logic includes action functions, auth functions, background tasks and business logic.

Action functions have a uniform interface accepting a dictionary of simple strings lists, dictionaries or files (wrapped in a `cgi.FieldStorage` objects). They return simple dictionaries or raise one of a small number of exceptions including `ckan.logic.NotAuthorized`, `ckan.logic.NotFound` and `ckan.logic.ValidationError`.

Plugins override action functions with the *`ckan.plugins.interfaces.IActions`* interface and auth functions with the *`ckan.plugins.interfaces.IAuthFunctions`* interface.

Action functions are exposed in the API

The functions in `ckan.logic.action` are exposed to the world as the *API guide*. The API URL for an action function is automatically generated from the function name, for example `ckan.logic.action.create.package_create()` is exposed at `/api/action/package_create`. See Steve Yegge's Google platforms rant for some interesting discussion about APIs.

All publicly visible functions in the `ckan.logic.action.{create,delete,get,update}` namespaces will be exposed through the *API guide*. **This includes functions imported** by those modules, **as well as any helper functions** defined within those modules. To prevent inadvertent exposure of non-action functions through the action api, care should be taken to:

1. Import modules correctly (see *Imports*). For example:

```
import ckan.lib.search as search

search.query_for(...)
```

2. Hide any locally defined helper functions:

```
def _a_useful_helper_function(x, y, z):
   '''This function is not exposed because it is marked as private```
   return x+y+z
```

3. Bring imported convenience functions into the module namespace as private members:

```
_get_or_bust = logic.get_or_bust
```

Auth functions and `check_access()`

Each action function defined in `ckan.logic.action` should use its own corresponding auth function defined in `ckan.logic.auth`. Instead of calling its auth function directly, an action function should go through `ckan.logic.check_access` (which is aliased `_check_access` in the action modules) because this allows plugins to override auth functions using the `IAuthFunctions` plugin interface. For example:

```
def package_show(context, data_dict):
    _check_access('package_show', context, data_dict)
```

`check_access` will raise an exception if the user is not authorized, which the action function should not catch. When this happens the user will be shown an authorization error in their browser (or will receive one in their response from the API).

`logic.get_or_bust()`

The `data_dict` parameter of logic action functions may be user provided, so required files may be invalid or absent. Naive Code like:

```
id = data_dict['id']
```

may raise a `KeyError` and cause CKAN to crash with a 500 Server Error and no message to explain what went wrong. Instead do:

```
id = _get_or_bust(data_dict, "id")
```

which will raise `ValidationError` if `"id"` is not in `data_dict`. The `ValidationError` will be caught and the user will get a 400 Bad Request response and an error message explaining the problem.

Validation and `ckan.logic.schema`

Logic action functions can use schema defined in `ckan.logic.schema` to validate the contents of the `data_dict` parameters that users pass to them.

An action function should first check for a custom schema provided in the context, and failing that should retrieve its default schema directly, and then call `_validate()` to validate and convert the data. For example, here is the validation code from the `user_create()` action function:

```
schema = context.get('schema') or ckan.logic.schema.default_user_schema()
session = context['session']
validated_data_dict, errors = _validate(data_dict, schema, context)
if errors:
    session.rollback()
    raise ValidationError(errors)
```

7.9.4 Models

Ideally SQLAlchemy should only be used within `ckan.model` and not from other packages such as `ckan.logic`. For example instead of using an SQLAlchemy query from the logic package to retrieve a particular user from the database, we add a `get()` method to `ckan.model.user.User`:

```
@classmethod
def get(cls, user_id):
    query = ...
    .
    .
    .
    return query.first()
```

Now we can call this method from the logic package.

7.9.5 Deprecation

- Anything that may be used by *extensions*, *themes* or *API clients* needs to maintain backward compatibility at call-site. For example: action functions, template helper functions and functions defined in the plugins toolkit.
- The length of time of deprecation is evaluated on a function-by-function basis. At minimum, a function should be marked as deprecated during a point release.
- To deprecate a function use the `ckan.lib.maintain.deprecated()` decorator and add "deprecated" to the function's docstring:

```
@maintain.deprecated("helpers.get_action() is deprecated and will be removed "
                     "in a future version of CKAN. Instead, please use the "
                     "extra_vars param to render() in your controller to pass "
                     "results from action functions to your templates.")
def get_action(action_name, data_dict=None):
    '''Calls an action function from a template. Deprecated in CKAN 2.3.'''
```

(continues on next page)

(continued from previous page)

```
    if data_dict is None:
        data_dict = {}
    return logic.get_action(action_name)({}, data_dict)
```

- Any deprecated functions should be added to an *API changes and deprecations* section in the *Changelog* entry for the next release (do this before merging the deprecation into master)
- Keep the deprecation messages passed to the decorator short, they appear in logs. Put longer explanations of why something was deprecated in the changelog.

7.10 CSS coding standards

Note: For CKAN 2.0 we use Sass as a pre-processor for our core CSS. View Front-end Documentation for more information on this subject.

7.10.1 Formatting

All CSS documents must use **two spaces** for indentation and files should have no trailing whitespace. Other formatting rules:

- Use soft-tabs with a two space indent.
- Use double quotes.
- Use shorthand notation where possible.
- Put spaces after `:` in property declarations.
- Put spaces before `{` in rule declarations.
- Use hex color codes `#000` unless using `rgba()`.
- Always provide fallback properties for older browsers.
- Use one line per property declaration.
- Always follow a rule with one line of whitespace.
- Always quote `url()` and `@import()` contents.
- Do not indent blocks.

For example:

```
.media {
  overflow: hidden;
  color: #fff;
  background-color: #000; /* Fallback value */
  background-image: linear-gradient(black, grey);
}

.media .img {
  float: left;
  border: 1px solid #ccc;
```

(continues on next page)

(continued from previous page)

```
}

.media .img img {
  display: block;
}

.media .content {
  background: #fff url("../images/media-background.png") no-repeat;
}
```

7.10.2 Naming

All ids, classes and attributes must be lowercase with hyphens used for separation.

```
/* GOOD */
.dataset-list {}

/* BAD */
.datasetlist {}
.datasetList {}
.dataset_list {}
```

7.10.3 Comments

Comments should be used liberally to explain anything that may be unclear at first glance, especially IE workarounds or hacks.

```
.prose p {
  font-size: 1.1666em /* 14px / 12px */;
}

.ie7 .search-form {
  /*
    Force the item to have layout in IE7 by setting display to block.
    See: http://reference.sitepoint.com/css/haslayout
  */
  display: inline-block;
}
```

7.10.4 Modularity and specificity

Try keep all selectors loosely grouped into modules where possible and avoid having too many selectors in one declaration to make them easy to override.

```
/* Avoid */
ul#dataset-list {}
ul#dataset-list li {}
ul#dataset-list li p a.download {}
```

Instead here we would create a dataset "module" and styling the item outside of the container allows you to use it on it's own e.g. on a dataset page:

```
.dataset-list {}
.dataset-list-item {}
.dataset-list-item .download {}
```

In the same vein use classes make the styles more robust, especially where the HTML may change. For example when styling social links:

```
<ul class="social">
  <li><a href="">Twitter</a></li>
  <li><a href="">Facebook</a></li>
  <li><a href="">LinkedIn</a></li>
</ul>
```

You may use pseudo selectors to keep the HTML clean:

```
.social li:nth-child(1) a {
  background-image: url(twitter.png);
}

.social li:nth-child(2) a {
  background-image: url(facebook.png);
}

.social li:nth-child(3) a {
  background-image: url(linked-in.png);
}
```

However this will break any time the HTML changes for example if an item is added or removed. Instead we can use class names to ensure the icons always match the elements (Also you'd probably sprite the image :).

```
.social .twitter {
  background-image: url(twitter.png);
}

.social .facebook {
  background-image: url(facebook.png);
}

.social .linked-in {
  background-image: url(linked-in.png);
}
```

Avoid using tag names in selectors as this prevents re-use in other contexts.

```
/* Cannot use this class on an <ol> or <div> element */
ul.dataset-item {}
```

Also ids should not be used in selectors as it makes it far too difficult to override later in the cascade.

```
/* Cannot override this button style without including an id */
.btn#download {}
```

7.11 HTML coding standards

See also:

String internationalization
How to mark strings for translation.

7.11.1 Formatting

All HTML documents must use **two spaces** for indentation and there should be no trailing whitespace. HTML5 syntax must be used and all attributes must use double quotes around attributes.

```
<video autoplay="autoplay" poster="poster_image.jpg">
  <source src="foo.ogg" type="video/ogg">
</video>
```

HTML5 elements should be used where appropriate reserving `<div>` and `<span>` elements for situations where there is no semantic value (such as wrapping elements to provide styling hooks).

7.11.2 Doctype and layout

All documents must be using the HTML5 doctype and the `<html>` element should have a `"lang"` attribute. The `<head>` should also at a minimum include `"viewport"` and `"charset"` meta tags.

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <title>Example Site</title>
  </head>
  <body></body>
</html>
```

7.11.3 Forms

Form fields must always include a `<label>` element with a `"for"` attribute matching the `"id"` on the input. This helps accessibility by focusing the input when the label is clicked, it also helps screen readers match labels to their respective inputs.

```
<label for="field-email">email</label>
<input type="email" id="field-email" name="email" value="" />
```

Each `<input>` should have an `"id"` that is unique to the page. It does not have to match the `"name"` attribute.

Forms should take advantage of the new HTML5 input types where they make sense to do so, placeholder attributes should also be included where relevant. Including these can provided enhancements in browsers that support them such as tailored inputs and keyboards.

```
<div>
  <label for="field-email">Email</label>
  <input type="email" id="field-email" name="email" value="name@example.com">
</div>
<div>
  <label for="field-phone">Phone</label>
  <input type="phone" id="field-phone" name="phone" value="" placeholder="+44 077 12345␣
↪678">
</div>
<div>
  <label for="field-url">Homepage</label>
  <input type="url" id="field-url" name="url" value="" placeholder="http://example.com">
</div>
```

Wufoo provides an excellent reference for these attributes.

7.11.4 Including meta data

Classes should ideally only be used as styling hooks. If you need to include additional data in the HTML document, for example to pass data to JavaScript, then the HTML5 `data-` attributes should be used.

```
<a class="btn" data-format="csv">Download CSV</a>
```

These can then be accessed easily via jQuery using the `.data()` method.

```
jQuery('.btn').data('format'); //=> "csv"

// Get the contents of all data attributes.
jQuery('.btn').data(); => {format: "csv"}
```

One thing to note is that the JavaScript API for datasets will convert all attribute names into camelCase. So `"data-file-format"` will become `fileFormat`.

For example:

```
<a class="btn" data-file-format="csv">Download CSV</a>
```

Will become:

```
jQuery('.btn').data('fileFormat'); //=> "csv"
jQuery('.btn').data(); => {fileFormat: "csv"}
```

Ideally you should be using CKAN's JavaScript module format for defining how JavaScript is initiated and interacts with the DOM.

7.11.5 Targeting Internet Explorer

Targeting lower versions of Internet Explorer (IE), those below version 9, should be handled by the stylesheets. Small fixes should be provided inline using the `.ie` specific class names. Larger fixes may require a separate stylesheet but try to avoid this if at all possible.

Adding IE specific classes:

```
<!DOCTYPE html>
<!--[if lt IE 7]> <html lang="en" class="ie ie6"> <![endif]-->
<!--[if IE 7]>    <html lang="en" class="ie ie7"> <![endif]-->
<!--[if IE 8]>    <html lang="en" class="ie ie8"> <![endif]-->
<!--[if gt IE 8]><!--> <html lang="en"> <!--<![endif]-->
```

Note: Only add lines for classes that are actually being used.

These can then be used within the CSS:

```
.clear:before,
.clear:after {
    content: "";
    display: table;
}

.clear:after {
    clear: both;
}

.ie7 .clear {
    zoom: 1; /* For IE 6/7 (trigger hasLayout) */
}
```

7.11.6 i18n

Don't include line breaks within <p> blocks. ie do this:

```
<p>Blah foo blah</p>
<p>New paragraph, blah</p>
```

And **not**:

```
<p>Blah foo blah
   New paragraph, blah</p>
```

7.12 JavaScript coding standards

See also:

String internationalization
How to mark strings for translation.

7.12.1 Formatting

All JavaScript documents must use **two spaces** for indentation. This is contrary to the OKFN Coding Standards but matches what's in use in the current code base.

Coding style must follow the idiomatic.js style but with the following exceptions.

Note: Idiomatic is heavily based upon Douglas Crockford's style guide which is recommended by the OKFN Coding Standards.

White space

Two spaces must be used for indentation at all times. Unlike in idiomatic whitespace must not be used _inside_ parentheses between the parentheses and their Contents.

```
// BAD: Too much whitespace.
function getUrl( full ) {
  var url = '/styleguide/javascript/';
  if ( full ) {
    url = 'http://okfn.github.com/ckan' + url;
  }
  return url;
}

// GOOD:
function getUrl(full) {
  var url = '/styleguide/javascript/';
  if (full) {
    url = 'http://okfn.github.com/ckan' + url;
  }
  return url;
}
```

Note: See section 2.D.1.1 of idiomatic for more examples of this syntax.

Quotes

Single quotes should be used everywhere unless writing JSON or the string contains them. This makes it easier to create strings containing HTML.

```
jQuery('<div id="my-div" />').appendTo('body');
```

Object properties need not be quoted unless required by the interpreter.

```
var object = {
  name: 'bill',
  'class': 'user-name'
};
```

Variable declarations

One `var` statement must be used per variable assignment. These must be declared at the top of the function in which they are being used.

```
// GOOD:
var good = 'string';
var alsoGood = 'another';

// GOOD:
var good = 'string';
var okay = [
  'hmm', 'a bit', 'better'
];

// BAD:
var good = 'string',
    iffy = [
  'hmm', 'not', 'great'
];
```

Declare variables at the top of the function in which they are first used. This avoids issues with variable hoisting. If a variable is not assigned a value until later in the function then it it okay to define more than one per statement.

```
// BAD: contrived example.
function lowercaseNames(names) {
  var names = [];

  for (var index = 0, length = names.length; index < length; index += 1) {
    var name = names[index];
    names.push(name.toLowerCase());
  }

  var sorted = names.sort();
  return sorted;
}

// GOOD:
function lowercaseNames(names) {
```

(continues on next page)

(continued from previous page)

```
  var names = [];
  var index, sorted, name;

  for (index = 0, length = names.length; index < length; index += 1) {
    name = names[index];
    names.push(name.toLowerCase());
  }

  sorted = names.sort();
  return sorted;
}
```

7.12.2 Naming

All properties, functions and methods must use lowercase camelCase:

```
var myUsername = 'bill';
var methods = {
  getSomething: function () {}
};
```

Constructor functions must use uppercase CamelCase:

```
function DatasetSearchView() {
}
```

Constants must be uppercase with spaces delimited by underscores:

```
var env = {
  PRODUCTION:  'production',
  DEVELOPMENT: 'development',
  TESTING:     'testing'
};
```

Event handlers and callback functions should be prefixed with "on":

```
function onDownloadClick(event) {}

jQuery('.download').click(onDownloadClick);
```

Boolean variables or methods returning boolean functions should prefix the variable name with "is":

```
function isAdmin() {}

var canEdit = isUser() && isAdmin();
```

Note: Alternatives are "has", "can" and "should" if they make more sense

Private methods should be prefixed with an underscore:

```
View.extend({
  "click": "_onClick",
  _onClick: function (event) {
  }
});
```

Functions should be declared as named functions rather than assigning an anonymous function to a variable.

```
// GOOD:
function getName() {
}

// BAD:
var getName = function () {
};
```

Named functions are generally easier to debug as they appear named in the debugger.

7.12.3 Comments

Comments should be used to explain anything that may be unclear when you return to it in six months time. Single line comments should be used for all inline comments that do not form part of the documentation.

```
// Export the function to either the exports or global object depending
// on the current environment. This can be either an AMD module, CommonJS
// module or a browser.
if (typeof module.define === 'function' && module.define.amd) {
  module.define('broadcast', function () {
    return Broadcast;
  });
} else if (module.exports) {
  module.exports = Broadcast;
} else {
  module.Broadcast = Broadcast;
}
```

7.12.4 JSHint

All JavaScript should pass JSHint before being committed. This can be installed using `npm` (which is bundled with node) by running:

```
$ npm -g install jshint
```

Each project should include a jshint.json file with appropriate configuration options for the tool. Most text editors can also be configured to read from this file.

7.12.5 Documentation

For documentation we use a simple markup format to document all methods. The documentation should provide enough information to show the reader what the method does, arguments it accepts and a general example of usage. Also for API's and third party libraries, providing links to external documentation is encouraged.

The formatting is as follows:

```
/* My method description. Should describe what the method does and where
 * it should be used.
 *
 * param1 - The method params, one per line (default: null)
 * param2 - A default can be provided in brackets at the end.
 *
 * Example
 *
 *   // Indented two spaces. Should give a common example of use.
 *   client.getTemplate('index.html', {limit: 1}, function (html) {
 *     module.el.html(html);
 *   });
 *
 * Returns describes what the object returns.
 */
```

For example:

```
/* Loads an HTML template from the CKAN snippet API endpoint. Template
 * variables can be passed through the API using the params object.
 *
 * Optional success and error callbacks can be provided or these can
 * be attached using the returns jQuery promise object.
 *
 * filename - The filename of the template to load.
 * params   - An optional object containing key/value arguments to be
 *            passed into the template.
 * success  - An optional success callback to be called on load. This will
 *            recieve the HTML string as the first argument.
 * error    - An optional error callback to be called if the request fails.
 *
 * Example
 *
 *   client.getTemplate('index.html', {limit: 1}, function (html) {
 *     module.el.html(html);
 *   });
 *
 * Returns a jqXHR promise object that can be used to attach callbacks.
 */
```

7.12.6 Testing

For testing we use Cypress.

Tests are run from the cypress directory. We use the BDD interface (`describe()`, `it()` etc.).

Generally we try and have the core functionality of all libraries and modules unit tested.

7.12.7 Best practices

Forms

All forms should work without JavaScript enabled. This means that they must submit `application/x-www-form-urlencoded` data to the server and receive an appropriate response. The server should check for the `X-Requested-With: XMLHTTPRequest` header to determine if the request is an ajax one. If so it can return an appropriate format, otherwise it should issue a 303 redirect.

The one exception to this rule is if a form or button is injected with JavaScript after the page has loaded. It's then not part of the HTML document and can submit any data format it pleases.

Ajax

Note: Calls to the CKAN API from JavaScript should be done through the CKAN client.

Ajax requests can be used to improve the experience of submitting forms and other actions that require server interactions. Nearly all requests will go through the following states.

1. User clicks button.
2. JavaScript intercepts the click and disables the button (add `disabled` attr).
3. A loading indicator is displayed (add class `.loading` to button).
4. The request is made to the server.
5. a) On success the interface is updated.
   b) On error a message is displayed to the user if there is no other way to resolve the issue.
6. The loading indicator is removed.
7. The button is re-enabled.

Here's a possible example for submitting a search form using jQuery.

```
jQuery('#search-form').submit(function (event) {
  var form = $(this);
  var button = $('[type=submit]', form);

  // Prevent the browser submitting the form.
  event.preventDefault();

  button.prop('disabled', true).addClass('loading');

  jQuery.ajax({
    type: this.method,
```

(continues on next page)

(continued from previous page)

```
    data: form.serialize(),
    success: function (results) {
      updatePageWithResults(results);
    },
    error: function () {
      showSearchError('Sorry we were unable to complete this search');
    },
    complete: function () {
      button.prop('disabled', false).removeClass('loading');
    }
  });
});
```

This covers possible issues that might arise from submitting the form as well as providing the user with adequate feedback that the page is doing something. Disabling the button prevents the form being submitted twice and the error feedback should hopefully offer a solution for the error that occurred.

Event handlers

When using event handlers to listen for browser events it's a common requirement to want to cancel the default browser action. This should be done by calling the `event.preventDefault()` method:

```
jQuery('button').click(function (event) {
  event.preventDefault();
});
```

It is also possible to return `false` from the callback function. Avoid doing this as it also calls the `event.stopPropagation()` method which prevents the event from bubbling up the DOM tree. This prevents other handlers listening for the same event. For example an analytics click handler attached to the `<body>` element.

Also jQuery (1.7+) now provides the .on() and .off() methods as alternatives to `.bind()`, `.unbind()`, `.delegate()` and `.undelegate()` and they should be preferred for all tasks.

Templating

Small templates that will not require customisation by the instance can be placed inline. If you need to create multi-line templates use an array rather than escaping newlines within a string:

```
var template = [
  '<li>',
  '<span></span>',
  '</li>'
].join('');
```

Always *localise text strings* within your template. If you are including them inline this can be done with jQuery:

```
jQuery(template).find('span').text(this._('This is my text string'));
```

Larger templates can be loaded in using the CKAN snippet API. Modules get access to this functionality via the `sandbox.client` object:

```
initialize: function () {
  var el = this.el;
  this.sandbox.client.getTemplate('dataset.html', function (html) {
    el.html(html);
  });
}
```

The primary benefits of this is that the localisation can be done by the server and it keeps the JavaScript modules free from large strings.

7.13 Python coding standards

For Python code style follow PEP 8 plus the guidelines below.

Some good links about Python code style:

- Guide to Python from Hitchhiker's
- Google Python Style Guide

See also:

String internationalization
How to mark strings for translation.

7.13.1 Use single quotes

Use single-quotes for string literals, e.g. `'my-identifier'`, *but* use double-quotes for strings that are likely to contain single-quote characters as part of the string itself (such as error messages, or any strings containing natural language), e.g. `"You've got an error!"`.

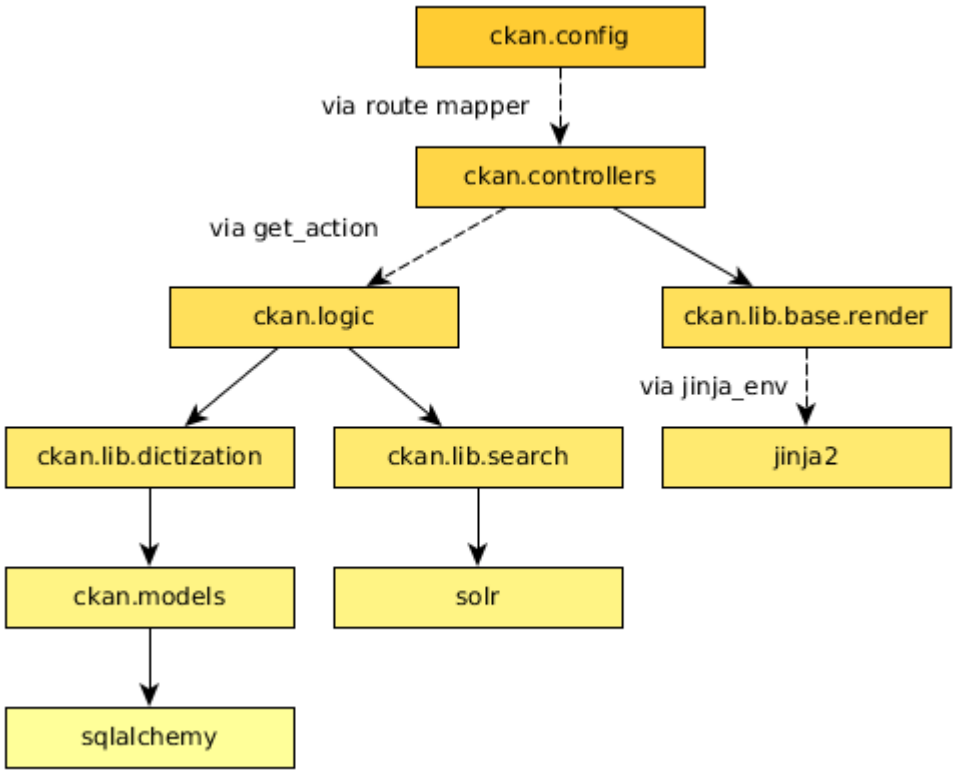
Single-quotes are easier to read and to type, but if a string contains single-quote characters then double-quotes are better than escaping the single-quote characters or wrapping the string in double single-quotes.

We also use triple single-quotes for docstrings, see *Docstrings*.

7.13.2 Imports

- Avoid creating circular imports by only importing modules more specialized than the one you are editing.

  CKAN often uses code imported into a data structure instead of importing names directly. For example CKAN controllers only use `get_action` to access logic functions. This allows customization by CKAN plugins.



- Don't use `from module import *`. Instead list the names you need explicitly:

```
from module import name1, name2
```

  Use parenthesis around the names if they are longer than one line:

```
from module import (name1, name2, ...
    name12, name13)
```

  Most of the current CKAN code base imports just the modules and then accesses names with `module.name`. This allows circular imports in some cases and may still be necessary for existing code, but is not recommended for new code.

- Make all imports at the start of the file, after the module docstring. Imports should be grouped in the following order:

  1. Standard library imports
  2. Third-party imports
  3. CKAN imports

7.13.3 Logging

We use the Python standard library's logging module to log messages in CKAN, e.g.:

```
import logging
...
logger = logging.getLogger(__name__)
...
logger.debug('some debug message')
```

When logging:

- Keep log messages short.
- Don't include object representations in the log message. It *is* useful to include a domain model identifier where appropriate.
- Choose an appropriate log-level (DEBUG, INFO, ERROR, WARNING or CRITICAL, see Python's Logging HOWTO).

7.13.4 String formatting

Don't use the old `%s` style string formatting, e.g. `"i am a %s" % sub`. This kind of string formatting is not helpful for internationalization.

Use the new .format() method instead, and give meaningful names to each replacement field, for example:

```
_(' ... {foo} ... {bar} ...').format(foo='foo-value', bar='bar-value')
```

7.13.5 Unicode handling

CKAN strives to only use Unicode internally (via the `unicode` type) and to convert to/from ASCII at the interface to other systems and libraries if necessary.

See also:

Unicode handling
Details on Unicode handling in CKAN

7.13.6 Docstrings

We want CKAN's docstrings to be clear and easy to read for programmers who are smart and competent but who may not know a lot of CKAN technical jargon and whose first language may not be English. We also want it to be easy to maintain the docstrings and keep them up to date with the actual behaviour of the code as it changes over time. So:

- All modules and all public functions, classes and methods exported by a module should normally have docstrings (see PEP 257).
- Keep docstrings short, describe only what's necessary and no more.
- Keep docstrings simple: use plain, concise English.
- Try to avoid repetition.

PEP 257 (Docstring Conventions)

Generally, follow PEP 257 for docstrings. We'll only describe the ways that CKAN differs from or extends PEP 257 below.

CKAN docstrings deviate from PEP 257 in a couple of ways:

- We use `'''triple single quotes'''` around docstrings, not `"""triple double quotes"""` (put triple single quotes around one-line docstrings as well as multi-line ones, it makes them easier to expand later)
- We use Sphinx domain object cross-references to cross-reference to other code objects (see below)
- We use Sphinx directives for documenting parameters, exceptions and return values (see below)

7.13.7 Referencing other code objects with `:py:`

If you want to refer to another Python or JavaScript module, function or class etc. in a docstring (or from a `.rst` file), use Sphinx domain object cross-references, for example:

```
See :py:mod:`ckan.lib.helpers`.

See :py:func:`ckan.logic.action.create.package_create`.

See :py:class:`ckan.logic.NotFound`.
```

For the full list of types of cross-reference, see the Sphinx docs.

Note: These kinds of cross-references can also be used to reference other types of object besides Python objects, for example JavaScript objects or even command-line scripts and options and environment variables. See the Sphinx docs for the full details.

Cross-referencing objects like this means that Sphinx will style the reference with the right CSS, and hyperlink the reference to the docs for the referenced object. Sphinx can also generate error messages when non-existent objects are referenced, which helps to keep the docs up to date as the code changes.

Tip: Sphinx will render a cross-reference like `:py:func:`ckan.logic.action.create.package_create` as the full name of the function: *`ckan.logic.action.create.package_create()`*. If you want the docs to contain only the local name of the function (e.g. just *`package_create()`*), put a ~ at the start:

```
:py:func:`~ckan.logic.action.create.package_create`
```

(But you should always use the fully qualified name in your docstring or `*.rst` file.)

Documenting exceptions raised with `:raises`

There are a few guidelines that CKAN code should follow regarding exceptions:

1. **All public functions that CKAN exports for third-party code to use should document any exceptions they raise**. See below for how to document exceptions raised.

   For example the template helper functions in *ckan.lib.helpers*, anything imported into *ckan.plugins.toolkit*, and all of the action API functions defined in `ckan.logic.action`, should list exceptions raised in their docstrings.

   This is because CKAN themes, extensions and API clients need to be able to call CKAN code without crashing, so they need to know what exceptions they should handle (and extension developers shouldn't have to understand the CKAN core source code).

2. On the other hand, **internal functions that are only used within CKAN shouldn't list exceptions in their docstrings**.

   This is because it would be difficult to keep all the exception lists up to date with the actual code behaviour, so the docstrings would become more misleading than useful.

3. **Code should only raise exceptions from within its allowed set**.

   Each module in CKAN has a set of zero or more exceptions, defined somewhere near the module, that code in that module is allowed to raise. For example `ckan/logic/__init__.py` defines a number of exception types for code in `ckan/logic/` to use. CKAN code should never raise exceptions types defined elsewhere in CKAN, in third-party code or in the Python standard library.

4. **All code should catch any exceptions raised by called functions**, and either handle the exception, re-raise the exception (if it's from the code's set of allowed exception types), or wrap the exception in an allowed exception type and re-raise it.

   This is to make it easy for a CKAN core developer to look at the source code of an internal function, scan it for the keyword `raise`, and see what types of exception the function may raise, so they know what exceptions they need to catch if they're going to call the function. Developers shouldn't have to read the source of all the functions that a function calls (and the functions they call…) to find out what exceptions they needs to catch to call a function without crashing.

Todo: Insert examples of how to re-raise and how to wrap-and-re-raise an exception.

Use `:raises:` to document exceptions raised by public functions. The docstring should say what type of exception is raised and under what conditions. Use `:py:class:` to reference exception types. For example:

```
def member_list(context, data_dict):
    '''Return the members of a group.

    ... (parameters and return values documented here) ...

    :raises: :py:class:`ckan.logic.NotFound`: if the group doesn't exist

    '''
```

Sphinx field lists

Use Sphinx field lists for documenting the parameters, exceptions and returns of functions:

- Use `:param` and `:type` to describe each parameter
- Use `:returns` and `:rtype` to describe each return
- Use `:raises` to describe each exception raised

Example of a short docstring:

```
@property
def packages(self):
    '''Return a list of all packages that have this tag, sorted by name.

    :rtype: list of ckan.model.package.Package objects

    '''
```

Example of a longer docstring:

```
@classmethod
def search_by_name(cls, search_term, vocab_id_or_name=None):
    '''Return all tags whose names contain a given string.

    By default only free tags (tags which do not belong to any vocabulary)
    are returned. If the optional argument ``vocab_id_or_name`` is given
    then only tags from that vocabulary are returned.

    :param search_term: the string to search for in the tag names
    :type search_term: string
    :param vocab_id_or_name: the id or name of the vocabulary to look in
        (optional, default: None)
    :type vocab_id_or_name: string

    :returns: a list of tags that match the search term
    :rtype: list of ckan.model.tag.Tag objects

    '''
```

The phrases that follow `:param foo:`, `:type foo:`, or `:returns:` should not start with capital letters or end with full stops. These should be short phrases and not full sentences. If more detail is required put it in the function description instead.

Indicate optional arguments by ending their descriptions with `(optional)` in brackets. Where relevant also indicate the default value: `(optional, default: 5)`.

You can also use a little inline reStructuredText markup in docstrings, e.g. `*stars for emphasis*` or ``` ``double-backticks for literal text`` ```

Action API docstrings

Docstrings from CKAN's action API are processed with autodoc and included in the API chapter of CKAN's documentation. The intended audience of these docstrings is users of the CKAN API and not (just) CKAN core developers.

In the Python source each API function has the same two arguments (`context` and `data_dict`), but the docstrings should document the keys that the functions read from `data_dict` and not `context` and `data_dict` themselves, as this is what the user has to POST in the JSON dict when calling the API.

Where practical, it's helpful to give examples of param and return values in API docstrings.

CKAN datasets used to be called packages and the old name still appears in the source, e.g. in function names like `package_list()`. When documenting functions like this write dataset not package, but the first time you do this put package after it in brackets to avoid any confusion, e.g.

```
def package_show(context, data_dict):
    '''Return the metadata of a dataset (package) and its resources.
```

Example of a ckan.logic.action API docstring:

```
def vocabulary_create(context, data_dict):
    '''Create a new tag vocabulary.

    You must be a sysadmin to create vocabularies.

    :param name: the name of the new vocabulary, e.g. ``'Genre'``
    :type name: string
    :param tags: the new tags to add to the new vocabulary, for the format of
        tag dictionaries see ``tag_create()``
    :type tags: list of tag dictionaries

    :returns: the newly-created vocabulary
    :rtype: dictionary

    '''
```

7.13.8 Some helpful tools for Python code quality

There are various tools that can help you to check your Python code for PEP8 conformance and general code quality. We recommend using them.

- pep8 checks your Python code against some of the style conventions in PEP 8. As mentioned above, only perform style clean-ups on master to help avoid spurious merge conflicts.
- pylint analyzes Python source code looking for bugs and signs of poor quality.
- pyflakes also analyzes Python programs to detect errors.
- flake8 combines both pep8 and pyflakes into a single tool.
- Syntastic is a Vim plugin with support for flake8, pyflakes and pylint.

7.14 String internationalization

All user-facing Strings in CKAN Python, JavaScript and Jinja2 code should be internationalized, so that our translators can then localize the strings for each of the many languages that CKAN supports. This guide shows CKAN developers how to internationalize strings, and what to look for regarding string internationalization when reviewing a pull request.

Note: *Internationalization* (or i18n) is the process of marking strings for translation, so that the strings can be extracted from the source code and given to translators. *Localization* (l10n) is the process of translating the marked strings into different languages.

See also:

Translating CKAN
If you want to translate CKAN, this page documents the process that translators follow to localize CKAN into different languages.

Doing a CKAN release
The processes for extracting internationalized strings from CKAN and uploading them to Transifex to be translated, and for downloading the translations from Transifex and loading them into CKAN to be displayed are documented on this page.

Note: Much of the existing code in CKAN was written before we had these guidelines, so it doesn't always do things as described on this page. When writing new code you should follow the guidelines on this page, not the existing code.

7.14.1 Internationalizating strings in Jinja2 templates

Most user-visible strings should be in the Jinja2 templates, rather than in Python or JavaScript code. This doesn't really matter to translators, but it's good for the code to separate logic and content. Of course this isn't always possible. For example when error messages are delivered through the API, there's no Jinja2 template involved.

The preferred way to internationalize strings in Jinja2 templates is by using the trans tag from Jinja2's i18n extension, which is available to all CKAN core and extension templates and snippets.

Most of the following examples are taken from the Jinja2 docs.

To internationalize a string put it inside a `{% trans %}` tag:

```
<p>{% trans %}This paragraph is translatable.{% endtrans %}</p>
```

You can also use variables from the template's namespace inside a `{% trans %}`:

```
<p>{% trans %}Hello {{ user }}!{% endtrans %}</p>
```

(Only variable tags are allowed inside trans tags, not statements.)

You can pass one or more arguments to the `{% trans %}` tag to bind variable names for use within the tag:

```
 <p>{% trans user=user.username %}Hello {{ user }}!{% endtrans %}</p>

{% trans book_title=book.title, author=author.name %}
This is {{ book_title }} by {{ author }}
{% endtrans %}
```

To handle different singular and plural forms of a string, use a `{% pluralize %}` tag:

```
{% trans count=list|length %}
There is {{ count }} {{ name }} object.
{% pluralize %}
There are {{ count }} {{ name }} objects.
{% endtrans %}
```

(In English the first string will be rendered if `count` is 1, the second otherwise. For other languages translators will be able to provide their own strings for different values of `count`.)

The first variable in the block (`count` in the example above) is used to determine which of the singular or plural forms to use. Alternatively you can explicitly specify which variable to use:

```
{% trans ..., user_count=users|length %}
   ...
{% pluralize user_count %}
   ...
{% endtrans %}
```

The `{% trans %}` tag is preferable, but if you need to pluralize a string within a Jinja2 expression you can use the `_()` and `ungettext()` functions:

```
{% set hello = _('Hello World!') %}
```

To use variables in strings, use Python format string syntax and then call the `.format()` method on the string that `_()` returns:

```
{% set hello = _('Hello {name}!').format(name=user.name) %}
```

Singular and plural forms are handled by `ungettext()`:

```
{% set text = ungettext(
       '{num} apple', '{num} apples', num_apples).format(num=num_apples) %}
```

Note: There are also `gettext()` and `ngettext()` functions available to templates, but we recommend using `_()` and `ungettext()` for consistency with CKAN's Python code. This deviates from the Jinja2 docs, which do use `gettext()` and `ngettext()`.

`_()` is not an alias for `gettext()` in CKAN's Jinja2 templates, `_()` is the function provided by Pylons, whereas `gettext()` is the version provided by Jinja2, their behaviors are not exactly the same.

7.14.2 Internationalizing strings in Python code

CKAN uses the `_()` and `ngettext()` functions from the Flask-Babel library to internationalize strings in Python code.

Note: Code running on Pylons will use the functions provided by the pylons.i18n.translation module, but their behaviour is the same.

Core CKAN modules should import `_()` and `ungettext()` from `ckan.common`, i.e. `from ckan.common import _, ungettext` (don't import `flask_babel._()` or `pylons.i18n.translation._()` directly, for example).

CKAN plugins should import *ckan.plugins.toolkit* and use `ckan.plugins.toolkit._()` and `ckan.plugins.toolkit.ungettext()`, i.e. do `import ckan.plugins.toolkit as toolkit` and then use `toolkit._()` and `toolkit.ungettext()` (see *Plugins toolkit reference*).

To internationalize a string pass it to the `_()` function:

```
my_string = _("This paragraph is translatable.")
```

To use variables in a string, call the `.format()` method on the translated string that `_()` returns:

```
hello = _("Hello {user}!").format(user=user.name)

book_description = _("This is { book_title } by { author }").format(
    book_title=book.title, author=author.name)
```

To handle different plural and singular forms of a string, use `ungettext()`:

```
translated_string = ungettext(
    "There is {count} {name} object.",
    "There are {count} {name} objects.",
    num_objects).format(count=count, name=name)
```

7.14.3 Internationalizing strings in JavaScript code

Each *CKAN JavaScript module* offers the methods `_` and `ngettext`. The `ngettext` function is used to translate a single string which has both a singular and a plural form, whereas `_` is used to translate a single string only:

```
this.ckan.module('i18n-demo', function($) {
    return {
        initialize: function () {
            console.log(this._('Translate me!'));
            console.log(this.ngettext('%(num)d item', '%(num)d items', 3));
        }
    };
};
```

To translate a fixed singular string, use `_`. It returns the translation of the string for the currently selected locale. If the current locale doesn't provide a translation for the string then it is returned unchanged.

```
this._('Something that should be translated')
```

Placeholders are supported via sprintf-syntax, the corresponding values are passed via another parameter:

```
this._("My name is %(name)s and I'm from %(hometown)s.",
       {name: 'Sarah', hometown: 'Cape Town'})
```

`ngettext` allows you to translate a string that may be either singular or plural, depending on some variable:

```
this.ngettext('Deleted %(num)d item',
              'Deleted %(num)d items',
              items.length)
```

If `items.length` is 1 then the translation for the first argument will be returned, otherwise that of the second argument. `num` is a magical placeholder that is automatically provided by `ngettext` and contains the value of the third parameter.

Like _, `ngettext` can take additional placeholders:

```
this.ngettext("I'm %(name)s and I'm %(num)d year old",
              "I'm %(name)s and I'm %(num)d years old",
              age,
              {name: 'John'})
```

Note: CKAN's JavaScript code automatically downloads the appropriate translations at request time from the CKAN server. Since CKAN 2.7 the corresponding translation files are regenerated automatically if necessary when CKAN starts.

You can also regenerate the translation files manually using `ckan translation js`:

```
python setup.py extract_messages  # Extract translatable strings
# Update .po files as desired
python setup.py compile_catalog   # Compile .mo files for Python/Jinja
ckan -c /etc/ckan/default/ckan.ini translation js         # Compile JavaScript catalogs
```

Note: Prior to CKAN 2.7, JavaScript modules received a similar but different _ function for string translation as a parameter. This is still supported but deprecated and will be removed in a future release.

7.14.4 General guidelines for internationalizing strings

Below are some guidelines to follow when marking your strings for translation. These apply to strings in Jinja2 templates or in Python or JavaScript code. These are mostly meant to make life easier for translators, and help to improve the quality of CKAN's translations:

- Leave as much HTML and other code out of the translation string as possible.

  For example, don't include surrounding `<p>...</p>` tags in the marked string. These aren't necessary for the translator to do the translation, and if the translator accidentally changes them in the translation string the HTML will be broken.

  Good:

  ```
  <p>{% trans %}Don't put HTML tags inside translatable strings{% endtrans %}</p>
  ```

  Bad (`<p>` tags don't need to be in the translation string):

  ```
  mystring = _("<p>Don't put HTML tags inside translatable strings</p>")
  ```

- But don't split a string into separate strings.

  Translators need as much context as possible to translate strings well, and if you split a string up into separate strings and mark each for translation separately, translators must translate each of these separate strings in isolation. Also, some languages may need to change the order of words in a sentence or even change the order of sentences in a paragraph, splitting into separate strings makes assumptions about word order.

  It's better to leave HTML tags or other code in strings than to split a string. For example, it's often best to leave HTML `<a>` tags in rather than split a string.

  Good:

```
_("Don't split a string containing some <b>markup</b> into separate strings.")
```

Bad (text will be difficult to translate or untranslatable):

```
_("Don't split a string containing some ") + "<b>" + _("markup") + </b> + _("into␣
↪separate strings.")
```

- You can split long strings over multiple lines using parentheses to avoid long lines, Python will concatenate them into a single string:

  Good:

```
_("This is a really long string that would just make this line far too "
  "long to fit in the window")
```

- Leave unnecessary whitespace out of translatable strings, but do put punctuation into translatable strings.
- Try not to make translators translate strings that don't need to be translated.

  For example, `'templates'` is the name of a directory, it doesn't need to be marked for translation.

- Mark singular and plural forms of strings correctly.

  In Jinja2 templates this means using `{% trans %}` and `{% pluralize %}` or `ungettext()`. In Python it means using `ungettext()`. See above for examples.

  Singular and plural forms work differently in different languages. For example English has singular and plural nouns, but Slovenian has singular, dual and plural.

  Good:

```
num_people = 4
translated_string = ungettext(
    'There is one person here',
    'There are {num_people} people here',
    num_people).format(num_people=num_people)
```

  Bad (this assumes that all languages have the same plural forms as English):

```
if num_people == 1:
    translated_string = _('There is one person here')
else:
    translated_string = _(
        'There are {num_people} people here'.format(num_people=num_people))
```

- Don't use old-style %s string formatting in Python, use the new .format() method instead.

  Strings formatted with `.format()` give translators more context. The `.format()` method is also more expressive, and is the preferred way to format strings in Python 3.

  Good:

```
"Welcome to {site_title}".format(site_title=site_title)
```

  Bad (not enough context for translators):

```
"Welcome to %s" % site_title
```

- Use descriptive names for replacement fields in strings.

  This gives translators more context.

  Good:

```
"Welcome to {site_title}".format(site_title=site_title)
```

  Bad (not enough context for translators):

```
"Welcome to {0}".format(site_title)
```

  Worse (doesn't work in Python 2.6):

```
"Welcome to {}".format(site_title)
```

- Use `TRANSLATORS:` comments to provide extra context for translators for difficult to find, very short, or obscure strings.

  For example, in Python:

```
# TRANSLATORS: This is a helpful comment.
_("This is an ambiguous string")
```

  In Jinja2:

```
{# TRANSLATORS: This heading is displayed on the user's profile page. #}
<h1>{% trans %}Heading{% endtrans %}</h1>
```

  In JavaScript:

```
// TRANSLATORS: "Manual" refers to the user manual
_("Manual")
```

  These comments end up in the `ckan.pot` file and translators will see them when they're translating the strings (Transifex shows them, for example).

  Note: The comment must be on the line before the line with the `_()`, `ungettext()` or `{% trans %}`, and must start with the exact string `TRANSLATORS:` (in upper-case and with the colon). This string is configured in `setup.cfg`.

Todo: Explain how to use *message contexts*, where the same exact string may appear in two different places in the UI but have different meanings.

For example "filter" can be a noun or a verb in English, and may need two different translations in another language. Currently if the string `_("filter")` appears in different places in CKAN this will only produce one string to be translated in the `ckan.pot` file.

I think the right way to handle this with gettext is using `msgctxt`, but it looks like babel doesn't support it yet.

Todo: Explain how we internationalize dates, currencies and numbers (e.g. different positioning and separators used for decimal points in different languages).

7.15 Unicode handling

This document explains how Unicode and related issues are handled in CKAN. For a general introduction to Unicode and Unicode handling in Python 2 please read the Python 2 Unicode HOWTO. Since Unicode handling differs greatly between Python 2 and Python 3 you might also be interested in the Python 3 Unicode HOWTO.

CKAN uses the six module to provide simultaneous compatibility with Python 2 and Python 3. All **strs** are Unicode in Python 3 so the builtins `unicode` and `basestring` have been removed so there are a few general rules to follow:

1. Change all calls to `basestring()` into calls to `six.string_types()`
2. Change remaining instances of `basestring` to `six.string_types`
3. Change all instances of `(str, unicode)` to `six.string_types`
4. Change all calls to `unicode()` into calls to `six.text_type()`
5. Change remaining instances of `unicode` to `six.text_type`

These rules do not apply in every instance so some thought needs to be given about the context around these changes.

Note: This document describes the intended future state of Unicode handling in CKAN. For historic reasons, some existing code does not yet follow the rules described here.

New code should always comply with the rules in this document. Exceptions must be documented.

7.15.1 Overall Strategy

CKAN only uses Unicode internally (`six.text_type` on both Python 2 and Python 3). Conversion to/from ASCII strings happens on the boundary to other systems/libraries if necessary.

7.15.2 Encoding of Python files

Files containing Python source code (`*.py`) must be encoded using UTF-8, and the encoding must be declared using the following header:

```
# encoding: utf-8
```

This line must be the first or second line in the file. See PEP 263 for details.

7.15.3 String literals

String literals are string values given directly in the source code (as opposed to strings variables read from a file, received via argument, etc.). In Python 2, string literals by default have type `str`. They can be changed to `unicode` by adding a `u` prefix. In addition, the `b` prefix can be used to explicitly mark a literal as `str`:

```
x = "I'm a str literal"
y = u"I'm a unicode literal"
z = b"I'm also a str literal"
```

In Python 3, all `str` are Unicode and `str` and `bytes` are explicitly different data types so:

```
x = "I'm a str literal"
y = u"I'm also a str literal"
z = b"I'm a bytes literal"
```

In CKAN, every string literal must carry either a `u` or a `b` prefix. While the latter is redundant in Python 2, it makes the developer's intention explicit and eases a future migration to Python 3.

This rule also holds for *raw strings*, which are created using an `r` prefix. Simply use `ur` instead:

```
m = re.match(ur'A\s+Unicode\s+pattern')
```

For more information on string prefixes please refer to the Python documentation.

Note: The `unicode_literals` future statement is *not* used in CKAN.

7.15.4 Best Practices

Use `io.open` to open text files

When opening text (not binary) files you should use io.open instead of `open`. This allows you to specify the file's encoding and reads will return Unicode instead of ASCII:

```
import io

with io.open(u'my_file.txt', u'r', encoding=u'utf-8') as f:
    text = f.read()  # contents is automatically decoded
                     # to Unicode using UTF-8
```

Text files should be encoded using UTF-8 if possible.

Normalize strings before comparing them

For many characters, Unicode offers multiple descriptions. For example, a small latin `e` with an acute accent (`é`) can either be specified using its dedicated code point (U+00E9) or by combining the code points for `e` (U+0065) and the accent (U+0301). Both variants will look the same but are different from a numerical point of view:

```
>>> x = u'\N{LATIN SMALL LETTER E WITH ACUTE}'
>>> y = u'\N{LATIN SMALL LETTER E}\N{COMBINING ACUTE ACCENT}'
>>> print x, y
é e
>>> print repr(x), repr(y)
u'\xe9' u'e\u0301'
>>> x == y
False
```

Therefore, if you want to compare two Unicode strings based on their characters you need to normalize them first using unicodedata.normalize:

```
>>> from unicodedata import normalize
>>> x_norm = normalize(u'NFC', x)
>>> y_norm = normalize(u'NFC', y)
```

(continues on next page)

(continued from previous page)

```
>>> print x_norm, y_norm
é é
>>> print repr(x_norm), repr(y_norm)
u'\xe9' u'\xe9'
>>> x_norm == y_norm
True
```

Use the Unicode flag in regular expressions

By default, the character classes of Python's re module (\w, \d, ...) only match ASCII-characters. For example, \w (alphanumeric character) does, by default, not match ö:

```
>>> print re.match(ur'^\w$', u'ö')
None
```

Therefore, you need to explicitly activate Unicode mode by passing the re.U flag:

```
>>> print re.match(ur'^\w$', u'ö', re.U)
<_sre.SRE_Match object at 0xb60ea2f8>
```

Note: Some functions (e.g. `re.split` and `re.sub`) take additional optional parameters before the flags, so you should pass the flag via a keyword argument:

```
replaced = re.sub(ur'\W', u'_', original, flags=re.U)
```

The type of the values returned by `re.split`, `re.MatchObject.group`, etc. depends on the type of the input string:

```
>>> re.split(ur'\W+', b'Just a string!', flags=re.U)
['Just', 'a', 'string', '']

>>> re.split(ur'\W+', u'Just some Unicode!', flags=re.U)
[u'Just', u'some', u'Unicode', u'']
```

Note that the type of the *pattern string* does not influence the return type.

Filenames

Like all other strings, filenames should be stored as Unicode strings internally. However, some filesystem operations return or expect byte strings, so filenames have to be encoded/decoded appropriately. Unfortunately, different operating systems use different encodings for their filenames, and on some of them (e.g. Linux) the file system encoding is even configurable by the user.

To make decoding and encoding of filenames easier, the `ckan.lib.io` module therefore contains the functions `decode_path` and `encode_path`, which automatically use the correct encoding:

```
import io
import json

from ckan.lib.io import decode_path
```

(continues on next page)

(continued from previous page)

```
# __file__ is a byte string, so we decode it
MODULE_FILE = decode_path(__file__)
print(u'Running from ' + MODULE_FILE)

# The functions in os.path return unicode if given unicode
MODULE_DIR = os.path.dirname(MODULE_FILE)
DATA_FILE = os.path.join(MODULE_DIR, u'data.json')

# Most of Python's built-in I/O-functions accept Unicode filenames as input
# and encode them automatically
with io.open(DATA_FILE, encoding='utf-8') as f:
    data = json.load(f)
```

Note that almost all Python's built-in I/O-functions accept Unicode filenames as input and encode them automatically, so using `encode_path` is usually not necessary.

The return type of some of Python's I/O-functions (e.g. os.listdir and os.walk) depends on the type of their input: If passed byte strings they return byte strings and if passed Unicode they automatically decode the raw filenames to Unicode before returning them. Other functions exist in two variants that return byte strings (e.g. os.getcwd) and Unicode (os.getcwdu), respectively.

> **Warning:** Some of Python's I/O-functions may return *both* byte and Unicode strings for *a single* call. For example, os.listdir will normally return Unicode when passed Unicode, but filenames that cannot be decoded using the filesystem encoding will still be returned as byte strings!
>
> Note that if the filename of an existing file cannot be decoded using the filesystem's encoding then the environment Python is running in is most probably incorrectly set up.

The instructions above are meant for the names of existing files that are obtained using Python's I/O functions. However, sometimes one also wants to create new files whose names are generated from unknown sources (e.g. user input). To make sure that the generated filename is safe to use and can be represented using the filesystem's encoding use `ckan.lib.munge.munge_filename`:

```
>> ckan.lib.munge.munge_filename(u'Data from Linköping (year: 2016).txt')
u'data-from-linkoping-year-2016.txt'
```

Note: `munge_filename` will remove a leading path from the filename.

7.16 Testing coding standards

All new code, or changes to existing code, should have new or updated tests before being merged into master. This document gives some guidelines for developers who are writing tests or reviewing code for CKAN.

See also:

Testing CKAN
: How to set up your development environment to run CKAN's test suite

Testing code that uses background jobs
: How to handle asynchronous background jobs in your tests

7.16.1 Guidelines for writing tests

We want the tests in `ckan.tests` to be:

Fast

- Don't share setup code between tests (e.g. in test class `setup()` or `setup_class()` methods, saved against the `self` attribute of test classes, or in test helper modules).

  Instead use fixtures that create test objects and pass them as parameters, and inject into every method only the required fixtures.

- Where appropriate, use the `monkeypatch` fixture to avoid pulling in other parts of CKAN (especially the database).

Independent

- Each test module, class and method should be able to be run on its own.
- Tests shouldn't be tightly coupled to each other, changing a test shouldn't affect other tests.

Clear

It should be quick and easy to see what went wrong when a test fails, or to see what a test does and how it works if you have to debug or update a test. If you think the test or helper method isn't clear by itself, add docstrings.

You shouldn't have to figure out what a complex test method does, or go and look up a lot of code in other files to understand a test method.

- Tests should follow the canonical form for a pytest, see *Recipe for a test method*.
- Write lots of small, simple test methods not a few big, complex tests.
- Each test method should test just One Thing.
- The name of a test method should clearly explain the intent of the test. See *Naming test methods*.

Easy to find

It should be easy to know where to add new tests for some new or changed code, or to find the existing tests for some code.

- See *How should tests be organized?*
- See *Naming test methods*.

Easy to write

Writing lots of small, clear and simple tests that all follow similar recipes and organization should make tests easy to write, as well as easy to read.

The follow sections give some more specific guidelines and tips for writing CKAN tests.

How should tests be organized?

The organization of test modules in `ckan.tests` mirrors the organization of the source modules in `ckan`:

```
ckan/
  tests/
    controllers/
      test_package.py <-- Tests for ckan/controllers/package.py
      ...
    lib/
      test_helpers.py <-- Tests for ckan/lib/helpers.py
      ...
```

(continues on next page)

(continued from previous page)

```
    logic/
      action/
        test_get.py
        ...
      auth/
        test_get.py
        ...
      test_converters.py
      test_validators.py
    migration/
      versions/
        test_001_add_existing_tables.py
        ...
    model/
      test_package.py
      ...
    ...
```

There are a few exceptional test modules that don't fit into this structure, for example PEP8 tests and coding standards tests. These modules can just go in the top-level `ckan/tests/` directory. There shouldn't be too many of these.

Naming test methods

The name of a test method should clearly explain the intent of the test.

Test method names are printed out when tests fail, so the user can often see what went wrong without having to look into the test file. When they do need to look into the file to debug or update a test, the test name helps to clarify the test.

Do this even if it means your method name gets really long, since we don't write code that calls our test methods there's no advantage to having short test method names.

Some modules in CKAN contain large numbers of loosely related functions. For example, *`ckan.logic.action.update`* contains all functions for updating things in CKAN. This means that `ckan.tests.logic.action.test_update` is going to contain an even larger number of test functions.

So as well as the name of each test method explaining the intent of the test, tests should be grouped by a test class that aggregates tests against a model entity or action type, for instance:

```
class TestPackageCreate(object):
    # ...
    def test_it_validates_name(self):
        # ...

    def test_it_validates_url(self):
        # ...


class TestResourceCreate(object)
    # ...
    def test_it_validates_package_id(self):
        # ...

# ...
```

Good test names:

- `TestUserUpdate.test_update_with_id_that_does_not_exist`
- `TestUserUpdate.test_update_with_no_id`
- `TestUserUpdate.test_update_with_invalid_name`

Bad test names:

- `test_user_update`
- `test_update_pkg_1`
- `test_package`

Recipe for a test method

The Pylons Unit Testing Guidelines give the following recipe for all unit test methods to follow:

1. Set up the preconditions for the method / function being tested.
2. Call the method / function exactly one time, passing in the values established in the first step.
3. Make assertions about the return value, and / or any side effects.
4. Do absolutely nothing else.

Most CKAN tests should follow this form. Here's an example of a simple action function test demonstrating the recipe:

```
def test_user_update_name(self):
    """Test that updating a user's name works successfully."""

    # The canonical form of a test has four steps:
    # 1. Setup any preconditions needed for the test.
    # 2. Call the function that's being tested, once only.
    # 3. Make assertions about the return value and/or side-effects of
    #    of the function that's being tested.
    # 4. Do nothing else!

    # 1. Setup.
    user = factories.User()
    user["name"] = "updated"

    # 2. Make assertions about the return value and/or side-effects.
    with pytest.raises(logic.ValidationError):
        helpers.call_action("user_update", **user)
```

How detailed should tests be?

Generally, what we're trying to do is test the *interfaces* between modules in a way that supports modularization: if you change the code within a function, method, class or module, if you don't break any of that code's tests you should be able to expect that CKAN as a whole will not be broken.

As a general guideline, the tests for a function or method should:

- Test for success:
  - Test the function with typical, valid input values
  - Test with valid, edge-case inputs
  - If the function has multiple parameters, test them in different combinations
- Test for failure:
  - Test that the function fails correctly (e.g. raises the expected type of exception) when given likely invalid inputs (for example, if the user passes an invalid user_id as a parameter)
  - Test that the function fails correctly when given bizarre input
- Test that the function behaves correctly when given unicode characters as input
- Cover the interface of the function: test all the parameters and features of the function

Creating test objects: `ckan.tests.factories`

This is a collection of factory classes for building CKAN users, datasets, etc.

Factories can be either used directly or via corresponding pytest fixtures to create any objects that are needed for the tests. These factories are written using `factory_boy`:

https://factoryboy.readthedocs.org/en/latest/

These are not meant to be used for the actual testing, e.g. if you're writing a test for the *`user_create()`* function then call *`call_action()`*, don't test it via the *`User`* factory or `user_factory()` fixture.

Usage:

```
# Create a user with the factory's default attributes, and get back a
# user dict:
def test_creation():
    user_dict = factories.User()

# or

def test_creation(user_factory):
    user_dict = user_factory()

# You can create a second user the same way. For attributes that can't be
# the same (e.g. you can't have two users with the same name) a new value
# will be generated each time you use the factory:
def test_creation():
    user_dict = factories.User()
    another_user_dict = factories.User()

# Create a user and specify your own user name and email (this works
```

(continues on next page)

(continued from previous page)

```
# with any params that CKAN's user_create() accepts):
def test_creation():
    custom_user_dict = factories.User(name='bob', email='bob@bob.com')

# Get a user dict containing the attributes (name, email, password, etc.)
# that the factory would use to create a user, but without actually
# creating the user in CKAN:
def test_creation():
    user_attributes_dict = vars(factories.User.stub())

# If you later want to create a user using these attributes, just pass them
# to the factory:
def test_creation():
    user = factories.User(**user_attributes_dict)

# If you just need random user, you can get ready-to-use dictionary inside
# your test by requiring `user` fixture (just drop `_factory` suffix):
def test_creation(user):
    assert isinstance(user, dict)
    assert "name" in user

# If you need SQLAlchemy model object instead of the plain dictionary, call
# `model` method of the corresponding factory. All arguments has the same
# effect as if they were passed directly to the factory:
def test_creation():
    user = factories.User.model(name="bob")
    assert isinstance(user, model.User)


# In order to create your own factory:
# * inherit from :py:class:`~ckan.tests.factories.CKANFactory`
# * create `Meta` class inside it, with the two properties:
#   * model: corresponding SQLAlchemy model
#   * action: API action that can create instances of the model
# * define any extra attributes
# * register factory as a fixture using :py:func:`~pytest_factoryboy.register`
import factory
from pytest_factoryboy import register
from ckan.tests.factories import CKANFactory

@register
class RatingFactory(CKANFactory):

    class Meta:
        model = ckanext.ext.model.Rating
        action = "rating_create"

    # These are the default params that will be used to create new ratings
    value = factory.Faker("pyint")
    comment = factory.Faker("text")
    approved = factory.Faker("boolean")
```

Factory-fixtures are generated using `pytest-factoryboy`:

https://pytest-factoryboy.readthedocs.io/en/latest/

class ckan.tests.factories.**CKANOptions**

CKANFactory options.

Parameters

- **action** – name of the CKAN API action used for entity creation
- **primary_key** – name of the entity's property that can be used for retrieving entity object from database

class ckan.tests.factories.**CKANFactory**(***kwargs*)

Extension of SQLAlchemy factory.

Creates entities via CKAN API using an action specified by the *Meta.action*.

Provides `model` method that returns created model object instead of the plain dictionary.

Check factoryboy's documentation for more details: https://factoryboy.readthedocs.io/en/stable/orms.html#sqlalchemy

classmethod api_create(*data_dict*)

Create entity via API call.

classmethod model(***kwargs*)

Create entity via API and retrieve result directly from the DB.

class ckan.tests.factories.**User**(***kwargs*)

A factory class for creating CKAN users.

class ckan.tests.factories.**Resource**(***kwargs*)

A factory class for creating CKAN resources.

class ckan.tests.factories.**ResourceView**(***kwargs*)

A factory class for creating CKAN resource views.

Note: if you use this factory, you need to load the *image_view* plugin on your test class (and unload it later), otherwise you will get an error.

Example:

```
@pytest.mark.ckan_config("ckan.plugins", "image_view")
@pytest.mark.usefixtures("with_plugins")
def test_resource_view_factory():
    ...
```

class ckan.tests.factories.**Sysadmin**(***kwargs*)

A factory class for creating sysadmin users.

class ckan.tests.factories.**Group**(***kwargs*)

A factory class for creating CKAN groups.

class ckan.tests.factories.**Organization**(***kwargs*)

A factory class for creating CKAN organizations.

class ckan.tests.factories.**Dataset**(***kwargs*)

A factory class for creating CKAN datasets.

class ckan.tests.factories.**Vocabulary**(***kwargs*)

A factory class for creating tag vocabularies.

class ckan.tests.factories.**Tag**(***kwargs*)

A factory class for creating tag vocabularies.

class ckan.tests.factories.**MockUser**(***kwargs*)

A factory class for creating mock CKAN users using the mock library.

class ckan.tests.factories.**SystemInfo**(***kwargs*)

A factory class for creating SystemInfo objects (config objects stored in the DB).

class ckan.tests.factories.**APIToken**(***kwargs*)

A factory class for creating CKAN API Tokens

class ckan.tests.factories.**UserWithToken**(***kwargs*)

A factory class for creating CKAN users with an associated API token.

class ckan.tests.factories.**SysadminWithToken**(***kwargs*)

A factory class for creating CKAN sysadmin users with an associated API token.

Test helper functions: `ckan.tests.helpers`

This is a collection of helper functions for use in tests.

We want to avoid sharing test helper functions between test modules as much as possible, and we definitely don't want to introduce a complex hierarchy of test class subclasses, etc.

We want to reduce the amount of "travel" that a reader needs to undertake to understand a test method – reducing the number of other files they need to go and read to understand what the test code does. And we want to avoid tightly coupling test modules to each other by having them share code.

But some test helper functions just increase the readability of tests so much and make writing tests so much easier, that it's worth having them despite the potential drawbacks.

New in CKAN 2.9: Consider using *Pytest fixtures* whenever possible for setting up the initial state of a test or to create helpers objects like client apps.

ckan.tests.helpers.**reset_db**()

Reset CKAN's database.

Rather than use this function directly, use the `clean_db` fixture either for all tests in a class:

```
@pytest.mark.usefixtures("clean_db")
class TestExample(object):

    def test_example(self):
```

or for a single test:

```
class TestExample(object):

    @pytest.mark.usefixtures("clean_db")
    def test_example(self):
```

If a test class uses the database, then it may call this function in its `setup()` method to make sure that it has a clean database to start with (nothing left over from other test classes or from previous test runs).

If a test class doesn't use the database (and most test classes shouldn't need to) then it doesn't need to call this function.

Returns
None

`ckan.tests.helpers.call_action(action_name: str, context=None, **kwargs)`

Call the named `ckan.logic.action` function and return the result.

This is just a nicer way for user code to call action functions, nicer than either calling the action function directly or via `ckan.logic.get_action()`.

For example:

```
user_dict = call_action('user_create', name='seanh',
                        email='seanh@seanh.com', password='pass')
```

Any keyword arguments given will be wrapped in a dict and passed to the action function as its `data_dict` argument.

Note: this skips authorization! It passes 'ignore_auth': True to action functions in their `context` dicts, so the corresponding authorization functions will not be run. This is because ckan.tests.logic.action tests only the actions, the authorization functions are tested separately in ckan.tests.logic.auth. See the *testing guidelines* for more info.

This function should eventually be moved to `ckan.logic.call_action()` and the current `ckan.logic.get_action()` function should be deprecated. The tests may still need their own wrapper function for `ckan.logic.call_action()`, e.g. to insert `'ignore_auth': True` into the `context` dict.

Parameters

- **action_name** (*string*) – the name of the action function to call, e.g. `'user_update'`
- **context** (*dict*) – the context dict to pass to the action function (optional, if no context is given a default one will be supplied)

Returns
the dict or other value that the action function returns

`ckan.tests.helpers.call_auth(auth_name: str, context, **kwargs) → bool`

Call the named `ckan.logic.auth` function and return the result.

This is just a convenience function for tests in *`ckan.tests.logic.auth`* to use.

Usage:

```
result = helpers.call_auth('user_update', context=context,
                           id='some_user_id',
                           name='updated_user_name')
```

Parameters

- **auth_name** (*string*) – the name of the auth function to call, e.g. `'user_update'`
- **context** (*dict*) – the context dict to pass to the auth function, must contain `'user'` and `'model'` keys, e.g. `{'user': 'fred', 'model': my_mock_model_object}`

Returns
the 'success' value of the authorization check, e.g. `{'success': True}` or `{'success': False, msg: 'important error message'}` or just `{'success': False}`

Return type
bool

class ckan.tests.helpers.**CKANCliRunner**(*charset: str = 'utf-8', env: Mapping[str, str | None] | None = None, echo_stdin: bool = False, mix_stderr: bool = True*)

invoke(**args, **kwargs*)

Invokes a command in an isolated environment. The arguments are forwarded directly to the command line script, the *extra* keyword arguments are passed to the `main()` function of the command.

This returns a `Result` object.

Parameters

- **cli** – the command to invoke
- **args** – the arguments to invoke. It may be given as an iterable or a string. When given as string it will be interpreted as a Unix shell command. More details at `shlex.split()`.
- **input** – the input data for *sys.stdin*.
- **env** – the environment overrides.
- **catch_exceptions** – Whether to catch any other exceptions than `SystemExit`.
- **extra** – the keyword arguments to pass to `main()`.
- **color** – whether the output should contain color codes. The application can still override this explicitly.

Changed in version 8.0: The result object has the `return_value` attribute with the value returned from the invoked command.

Changed in version 4.0: Added the `color` parameter.

Changed in version 3.0: Added the `catch_exceptions` parameter.

Changed in version 3.0: The result object has the `exc_info` attribute with the traceback if available.

class ckan.tests.helpers.**CKANResponse**(*response: Iterable[bytes] | bytes | Iterable[str] | str | None = None, status: int | str | HTTPStatus | None = None, headers: Mapping[str, str | Iterable[str]] | Iterable[tuple[str, str]] | None = None, mimetype: str | None = None, content_type: str | None = None, direct_passthrough: bool = False*)

class ckan.tests.helpers.**CKANTestApp**(*app*)

A wrapper around flask.testing.Client

It adds some convenience methods for CKAN

class ckan.tests.helpers.**CKANTestClient**(*application: WSGIApplication, response_wrapper: type[Response] | None = None, use_cookies: bool = True, allow_subdomain_redirects: bool = False*)

open(**args, **kwargs*)

Generate an environ dict from the given arguments, make a request to the application using it, and return the response.

Parameters

- **args** – Passed to `EnvironBuilder` to create the environ for the request. If a single arg is passed, it can be an existing `EnvironBuilder` or an environ dict.

- **buffered** – Convert the iterator returned by the app into a list. If the iterator has a `close()` method, it is called automatically.
- **follow_redirects** – Make additional requests to follow HTTP redirects until a non-redirect status is returned. `TestResponse.history` lists the intermediate responses.

Changed in version 2.1: Removed the `as_tuple` parameter.

Changed in version 2.0: The request input stream is closed when calling `response.close()`. Input streams for redirects are automatically closed.

Changed in version 0.5: If a dict is provided as file in the dict for the `data` parameter the content type has to be called `content_type` instead of `mimetype`. This change was made for consistency with `werkzeug.FileWrapper`.

Changed in version 0.5: Added the `follow_redirects` parameter.

class `ckan.tests.helpers.`**FunctionalTestBase**

A base class for functional test classes to inherit from.

Deprecated: Use the `app`, `clean_db`, `ckan_config` and `with_plugins` ref:*fixtures* as needed to create functional test classes, eg:

```
@pytest.mark.ckan_config('ckan.plugins', 'image_view')
@pytest.mark.usefixtures('with_plugins')
@pytest.mark.usefixtures('clean_db')
class TestDatasetSearch(object):

    def test_dataset_search(self, app):

        url = h.url_for('dataset.search')
        response = app.get(url)
```

Allows configuration changes by overriding _apply_config_changes and resetting the CKAN config after your test class has run. It creates a CKANTestApp at self.app for your class to use to make HTTP requests to the CKAN web UI or API. Also loads plugins defined by _load_plugins in the class definition.

If you're overriding methods that this class provides, like setup_class() and teardown_class(), make sure to use super() to call this class's methods at the top of yours!

setup()

Reset the database and clear the search indexes.

class `ckan.tests.helpers.`**RQTestBase**

Base class for tests of RQ functionality.

setup()

Delete all RQ queues and jobs.

all_jobs()

Get a list of all RQ jobs.

enqueue(*job=None*, **args*, ***kwargs*)

Enqueue a test job.

class `ckan.tests.helpers.`**FunctionalRQTestBase**

Base class for functional tests of RQ functionality.

setup()

Delete all RQ queues and jobs.

ckan.tests.helpers.**change_config**(*key*, *value*)

Decorator to temporarily change CKAN's config to a new value

This allows you to easily create tests that need specific config values to be set, making sure it'll be reverted to what it was originally, after your test is run.

Usage:

```
@helpers.change_config('ckan.site_title', 'My Test CKAN')
def test_ckan_site_title(self):
    assert config['ckan.site_title'] == 'My Test CKAN'
```

Parameters

- **key** (*string*) – the config key to be changed, e.g. `'ckan.site_title'`
- **value** (*string*) – the new config key's value, e.g. `'My Test CKAN'`

See also:

The context manager *changed_config()*

ckan.tests.helpers.**changed_config**(*key*, *value*)

Context manager for temporarily changing a config value.

Allows you to temporarily change the value of a CKAN configuration option. The original value is restored once the context manager is left.

Usage:

```
with changed_config(u'ckan.site_title', u'My Test CKAN'):
    assert config[u'ckan.site_title'] == u'My Test CKAN'
```

See also:

The decorator *change_config()*

ckan.tests.helpers.**recorded_logs**(*logger=None*, *level=10*, *override_disabled=True*, *override_global_level=True*)

Context manager for recording log messages.

Parameters

- **logger** – The logger to record messages from. Can either be a `logging.Logger` instance or a string with the logger's name. Defaults to the root logger.
- **level** (*int*) – Temporary log level for the target logger while the context manager is active. Pass `None` if you don't want the level to be changed. The level is automatically reset to its original value when the context manager is left.
- **override_disabled** (*bool*) – A logger can be disabled by setting its `disabled` attribute. By default, this context manager sets that attribute to `False` at the beginning of its execution and resets it when the context manager is left. Set `override_disabled` to `False` to keep the current value of the attribute.
- **override_global_level** (*bool*) – The `logging.disable` function allows one to install a global minimum log level that takes precedence over a logger's own level. By default, this

context manager makes sure that the global limit is at most `level`, and reduces it if necessary during its execution. Set `override_global_level` to `False` to keep the global limit.

Returns
A recording log handler that listens to `logger` during the execution of the context manager.

Return type
RecordingLogHandler

Example:

```
import logging

logger = logging.getLogger(__name__)

with recorded_logs(logger) as logs:
    logger.info(u'Hello, world!')

logs.assert_log(u'info', u'world')
```

class `ckan.tests.helpers.`**`RecordingLogHandler`**(**args*, ***kwargs*)

Log handler that records log messages for later inspection.

You can inspect the recorded messages via the `messages` attribute (a dict that maps log levels to lists of messages) or by using `assert_log`.

This class is rarely useful on its own, instead use *recorded_logs()* to temporarily record log messages.

emit(*record*)

Do whatever it takes to actually log the specified logging record.

This version is intended to be implemented by subclasses and so raises a NotImplementedError.

assert_log(*level*, *pattern*, *msg=None*)

Assert that a certain message has been logged.

Parameters

- **pattern** (*string*) – A regex which the message has to match. The match is done using `re.search`.
- **level** (*string*) – The message level (`'debug'`, ...).
- **msg** (*string*) – Optional failure message in case the expected log message was not logged.

Raises
AssertionError – If the expected message was not logged.

clear()

Clear all captured log messages.

class `ckan.tests.helpers.`**`FakeSMTP`**

Mock *SMTP* client, catching all the messages.

sendmail(*from_addr*, *to_addrs*, *msg*, *mail_options=()*, *rcpt_options=()*)

Just store message inside current instance.

Pytest fixtures

This is a collection of pytest fixtures for use in tests.

All fixtures below available anywhere under the root of CKAN repository. Any external CKAN extension should be able to include them by adding next lines under root *conftest.py*

```
# -*- coding: utf-8 -*-

pytest_plugins = [
    u'ckan.tests.pytest_ckan.ckan_setup',
    u'ckan.tests.pytest_ckan.fixtures',
]
```

There are three type of fixtures available in CKAN:

- Fixtures that have some side-effect. They don't return any useful value and generally should be injected via `pytest.mark.usefixtures`. Ex.: *with_plugins*, *clean_db*, *clean_index*.
- Fixtures that provide value. Ex. *app*
- Fixtures that provide factory function. They are rarely needed, so prefer using 'side-effect' or 'value' fixtures. Main use-case when one may use function-fixture - late initialization or repeatable execution(ex.: cleaning database more than once in a single test). But presence of these fixtures in test usually signals that is's a good time to refactor this test.

Deeper explanation can be found in official documentation

class ckan.tests.pytest_ckan.fixtures.**UserFactory**(***kwargs*)

class ckan.tests.pytest_ckan.fixtures.**ResourceFactory**(***kwargs*)

class ckan.tests.pytest_ckan.fixtures.**ResourceViewFactory**(***kwargs*)

class ckan.tests.pytest_ckan.fixtures.**GroupFactory**(***kwargs*)

class ckan.tests.pytest_ckan.fixtures.**PackageFactory**(***kwargs*)

class ckan.tests.pytest_ckan.fixtures.**VocabularyFactory**(***kwargs*)

class ckan.tests.pytest_ckan.fixtures.**TagFactory**(***kwargs*)

class ckan.tests.pytest_ckan.fixtures.**SystemInfoFactory**(***kwargs*)

class ckan.tests.pytest_ckan.fixtures.**APITokenFactory**(***kwargs*)

class ckan.tests.pytest_ckan.fixtures.**SysadminFactory**(***kwargs*)

class ckan.tests.pytest_ckan.fixtures.**OrganizationFactory**(***kwargs*)

ckan.tests.pytest_ckan.fixtures.**ckan_config**(*request*, *monkeypatch*)

Allows to override the configuration object used by tests

Takes into account config patches introduced by the `ckan_config` mark.

If you just want to set one or more configuration options for the scope of a test (or a test class), use the `ckan_config` mark:

```
@pytest.mark.ckan_config('ckan.auth.create_unowned_dataset', True)
def test_auth_create_unowned_dataset():

    # ...
```

To use the custom config inside a test, apply the `ckan_config` mark to it and inject the `ckan_config` fixture:

```
@pytest.mark.ckan_config(u"some.new.config", u"exists")
def test_ckan_config_mark(ckan_config):
    assert ckan_config[u"some.new.config"] == u"exists"
```

If the change only needs to be applied locally, use the `monkeypatch` fixture

```
@pytest.mark.usefixtures("with_request_context")
def test_deleting_a_key_delets_it_on_flask_config(monkeypatch, ckan_config):
    monkeypatch.setitem(ckan_config, u"ckan.site_title", u"Example title")
    del ckan_config[u"ckan.site_title"]
    assert u"ckan.site_title" not in flask.current_app.config
```

`ckan.tests.pytest_ckan.fixtures.`**`make_app`**(*ckan_config*)

Factory for client app instances.

Unless you need to create app instances lazily for some reason, use the `app` fixture instead.

`ckan.tests.pytest_ckan.fixtures.`**`app`**(*make_app*)

Returns a client app instance to use in functional tests

To use it, just add the `app` parameter to your test function signature:

```
def test_dataset_search(self, app):

    url = h.url_for('dataset.search')

    response = app.get(url)
```

`ckan.tests.pytest_ckan.fixtures.`**`cli`**(*ckan_config*)

Provides object for invoking CLI commands from tests.

This is subclass of *click.testing.CliRunner*, so all examples from Click docs are valid for it.

`ckan.tests.pytest_ckan.fixtures.`**`reset_db`**()

Callable for resetting the database to the initial state.

If possible use the `clean_db` fixture instead.

`ckan.tests.pytest_ckan.fixtures.`**`reset_index`**()

Callable for cleaning search index.

If possible use the `clean_index` fixture instead.

`ckan.tests.pytest_ckan.fixtures.`**`reset_redis`**()

Callable for removing all keys from Redis.

Accepts redis key-pattern for narrowing down the list of items to remove. By default removes everything.

This fixture removes all the records from Redis on call:

```
def test_redis_is_empty(reset_redis):
    redis = connect_to_redis()
    redis.set("test", "test")

    reset_redis()
    assert not redis.get("test")
```

If only specific records require removal, pass a pattern to the fixture:

```
def test_redis_is_empty(reset_redis):
    redis = connect_to_redis()
    redis.set("AAA-1", 1)
    redis.set("AAA-2", 2)
    redis.set("BBB-3", 3)

    reset_redis("AAA-*")
    assert not redis.get("AAA-1")
    assert not redis.get("AAA-2")

    assert redis.get("BBB-3") is not None
```

`ckan.tests.pytest_ckan.fixtures.`**`clean_redis`**(*reset_redis*)

Remove all keys from Redis.

This fixture removes all the records from Redis:

```
@pytest.mark.usefixtures("clean_redis")
def test_redis_is_empty():
    assert redis.keys("*") == []
```

If test requires presence of some initial data in redis, make sure that data producer applied **after** `clean_redis`:

```
@pytest.mark.usefixtures(
    "clean_redis",
    "fixture_that_adds_xxx_key_to_redis"
)
def test_redis_has_one_record():
    assert redis.keys("*") == [b"xxx"]
```

`ckan.tests.pytest_ckan.fixtures.`**`clean_db`**(*reset_db*)

Resets the database to the initial state.

This can be used either for all tests in a class:

```
@pytest.mark.usefixtures("clean_db")
class TestExample(object):

    def test_example(self):
```

or for a single test:

```
class TestExample(object):
```

(continues on next page)

(continued from previous page)

```
    @pytest.mark.usefixtures("clean_db")
    def test_example(self):
```

ckan.tests.pytest_ckan.fixtures.**migrate_db_for**()

Apply database migration defined by plugin.

In order to use models defined by extension extra tables may be required. In such cases database migrations(that were generated by *ckan generate migration -p PLUGIN_NAME*) can be applied as per example below:

```
@pytest.mark.usefixtures("clean_db")
def test_migrations_applied(migrate_db_for):
    migrate_db_for("my_plugin")
    assert model.Session.bind.has_table("my_plugin_custom_table")
```

ckan.tests.pytest_ckan.fixtures.**clean_index**(*reset_index*)

Clear search index before starting the test.

ckan.tests.pytest_ckan.fixtures.**with_plugins**(*ckan_config*)

Load all plugins specified by the `ckan.plugins` config option at the beginning of the test(and disable any plugin which is not listed inside `ckan.plugins`). When the test ends (including fail), it will unload all the plugins.

```
@pytest.mark.ckan_config("ckan.plugins", "image_view")
@pytest.mark.usefixtures("non_clean_db", "with_plugins")
def test_resource_view_factory():
    resource_view1 = factories.ResourceView()
    resource_view2 = factories.ResourceView()
    assert resource_view1[u"id"] != resource_view2[u"id"]
```

Use this fixture if test relies on CKAN plugin infrastructure. For example, if test calls an action or helper registered by plugin XXX:

```
@pytest.mark.ckan_config("ckan.plugins", "XXX")
@pytest.mark.usefixtures("with_plugin")
def test_action_and_helper():
    assert call_action("xxx_action")
    assert tk.h.xxx_helper()
```

It will not work without `with_plugins`. If `XXX` plugin is not loaded, `xxx_action` and `xxx_helper` do not exist in CKAN registries.

But if the test above use direct imports instead, `with_plugins` is optional:

```
def test_action_and_helper():
    from ckanext.xxx.logic.action import xxx_action
    from ckanext.xxx.helpers import xxx_helper

    assert xxx_action()
    assert xxx_helper()
```

Keep in mind, that generally it's a bad idea to import helpers and actions directly. If **every** test of extension requires standard set of plugins, specify these plugins inside test config file(`test.ini`):

```
ckan.plugins = essential_plugin another_plugin_required_by_every_test
```

And create an autouse-fixture that depends on `with_plugins` inside the main `conftest.py` (`ckanext/ext/tests/conftest.py`):

```
@pytest.fixture(autouse=True)
def load_standard_plugins(with_plugins):
    ...
```

This will automatically enable `with_plugins` for every test, even if it's not required explicitely.

`ckan.tests.pytest_ckan.fixtures.`**`test_request_context`**(*app*)

Provide function for creating Flask request context.

`ckan.tests.pytest_ckan.fixtures.`**`with_request_context`**(*test_request_context*)

Execute test inside requests context

`ckan.tests.pytest_ckan.fixtures.`**`mail_server`**(*monkeypatch*)

Catch all outcome mails.

`ckan.tests.pytest_ckan.fixtures.`**`with_test_worker`**(*monkeypatch*)

Worker that doesn't create forks.

`ckan.tests.pytest_ckan.fixtures.`**`with_extended_cli`**(*ckan_config*, *monkeypatch*)

Enables effects of IClick.

Without this fixture, only CLI command that came from plugins specified in real config file are available. When this fixture enabled, changing *ckan.plugins* on test level allows to update list of available CLI command.

`ckan.tests.pytest_ckan.fixtures.`**`reset_db_once`**(*reset_db*)

Internal fixture that cleans DB only the first time it's used.

`ckan.tests.pytest_ckan.fixtures.`**`non_clean_db`**(*reset_db_once*)

Guarantees that DB is initialized.

This fixture either initializes DB if it hasn't been done yet or does nothing otherwise. If there is some data in DB, it stays intact. If your tests need empty database, use *clean_db* instead, which is much slower, but guarantees that there are no data left from the previous test session.

Example:

```
@pytest.mark.usefixtures("non_clean_db")
def test_example():
    assert factories.User()
```

class `ckan.tests.pytest_ckan.fixtures.`**`FakeFileStorage`**(*stream: IO[bytes]*, *filename: str*)

`ckan.tests.pytest_ckan.fixtures.`**`create_with_upload`**(*clean_db*, *ckan_config*, *monkeypatch*, *tmpdir*)

Shortcut for creating resource/user/org with upload.

Requires content and name for newly created object. By default is using *resource_create* action, but it can be changed by passing named argument *action*.

Upload field if configured by passing *upload_field_name* named argument. Default value: *upload*.

In addition, accepts named argument *context* which will be passed to *ckan.tests.helpers.call_action* and arbitrary number of additional named arguments, that will be used as resource properties.

Example:

```
def test_uploaded_resource(create_with_upload):
    dataset = factories.Dataset()
    resource = create_with_upload(
        "hello world", "file.txt", url="http://data",
        package_id=dataset["id"])
    assert resource["url_type"] == "upload"
    assert resource["format"] == "TXT"
    assert resource["size"] == 11
```

Mocking: the `mock` library

We use the mock library to replace parts of CKAN with mock objects. This allows a CKAN function to be tested independently of other parts of CKAN or third-party libraries that the function uses. This generally makes the test simpler and faster (especially when `ckan.model` is mocked out so that the tests don't touch the database). With mock objects we can also make assertions about what methods the function called on the mock object and with which arguments.

Note: Overuse of mocking is discouraged as it can make tests difficult to understand and maintain. Mocking can be useful and make tests both faster and simpler when used appropriately. Some rules of thumb:

- Don't mock out more than one or two objects in a single test method.
- Don't use mocking in more functional-style tests. For example the action function tests in *`ckan.tests.logic.action`* and the frontend tests in *`ckan.tests.controllers`* are functional tests, and probably shouldn't do any mocking.
- Do use mocking in more unit-style tests. For example the authorization function tests in *`ckan.tests.logic.auth`*, the converter and validator tests in *`ckan.tests.logic.auth`*, and most (all?) lib tests in *`ckan.tests.lib`* are unit tests and should use mocking when necessary (often it's possible to unit test a method in isolation from other CKAN code without doing any mocking, which is ideal).

  In these kind of tests we can often mock one or two objects in a simple and easy to understand way, and make the test both simpler and faster.

A mock object is a special object that allows user code to access any attribute name or call any method name (and pass any parameters) on the object, and the code will always get another mock object back:

```
>>> import unittest.mock as mock
>>> my_mock = mock.MagicMock()
>>> my_mock.foo
<MagicMock name='mock.foo' id='56032400'>
>>> my_mock.bar
<MagicMock name='mock.bar' id='54093968'>
>>> my_mock.foobar()
<MagicMock name='mock.foobar()' id='54115664'>
>>> my_mock.foobar(1, 2, 'barfoo')
<MagicMock name='mock.foobar()' id='54115664'>
```

When a test needs a mock object to actually have some behavior besides always returning other mock objects, it can set the value of a certain attribute on the mock object, set the return value of a certain method, specify that a certain method should raise a certain exception, etc.

You should read the mock library's documentation to really understand what's going on, but here's an example of a test from `ckan.tests.logic.auth.test_update` that tests the `user_update()` authorization function and mocks out

ckan.model:

```
def test_user_update_user_cannot_update_another_user():
    """Users should not be able to update other users' accounts."""

    # 1. Setup.

    # Make a mock ckan.model.User object, Fred.
    fred = factories.MockUser()

    # Make a mock ckan.model object.
    mock_model = mock.MagicMock()
    # model.User.get(user_id) should return Fred.
    mock_model.User.get.return_value = fred

    # Put the mock model in the context.
    # This is easier than patching import ckan.model.
    context = {"model": mock_model}

    # The logged-in user is going to be Bob, not Fred.
    context["user"] = "bob"

    # 2. Call the function that's being tested, once only.

    # Make Bob try to update Fred's user account.
    params = {"id": fred.id, "name": "updated_user_name"}

    # 3. Make assertions about the return value and/or side-effects.

    with pytest.raises(logic.NotAuthorized):
        helpers.call_auth("user_update", context=context, **params)

    # 4. Do nothing else!
```

The following sections will give specific guidelines and examples for writing tests for each module in CKAN.

Note: When we say that *all* functions should have tests in the sections below, we mean all *public* functions that the module or class exports for use by other modules or classes in CKAN or by extensions or templates.

Private helper methods (with names beginning with _) never have to have their own tests, although they can have tests if helpful.

Writing `ckan.logic.action` tests

All action functions should have tests.

Most action function tests will be high-level tests that both test the code in the action function itself, and also indirectly test the code in `ckan.lib`, `ckan.model`, `ckan.logic.schema` etc. that the action function calls. This means that most action function tests should *not* use mocking.

Tests for action functions should use the *`ckan.tests.helpers.call_action()`* function to call the action functions.

One thing *`call_action()`* does is to add `ignore_auth: True` into the `context` dict that's passed to the action function, so that CKAN will not call the action function's authorization function. The tests for an action function *don't* need to cover authorization, because the authorization functions have their own tests in *`ckan.tests.logic.auth`*. But action function tests *do* need to cover validation, more on that later.

Action function tests *should* test the logic of the actions themselves, and *should* test validation (e.g. that various kinds of valid input work as expected, and invalid inputs raise the expected exceptions).

Here's an example of a simple `ckan.logic.action` test:

```
def test_user_update_name(self):
    """Test that updating a user's name works successfully."""

    # The canonical form of a test has four steps:
    # 1. Setup any preconditions needed for the test.
    # 2. Call the function that's being tested, once only.
    # 3. Make assertions about the return value and/or side-effects of
    #    of the function that's being tested.
    # 4. Do nothing else!

    # 1. Setup.
    user = factories.User()
    user["name"] = "updated"

    # 2. Make assertions about the return value and/or side-effects.
    with pytest.raises(logic.ValidationError):
        helpers.call_action("user_update", **user)
```

Todo: Insert the names of all tests for `ckan.logic.action.update.user_update`, for example, to show what level of detail things should be tested in.

Writing `ckan.logic.auth` tests

All auth functions should have tests.

Most auth function tests should be unit tests that test the auth function in isolation, without bringing in other parts of CKAN or touching the database. This requires using the `mock` library to mock `ckan.model`, see *Mocking: the mock library*.

Tests for auth functions should use the *`ckan.tests.helpers.call_auth()`* function to call auth functions.

Here's an example of a simple `ckan.logic.auth` test:

```
def test_user_update_user_cannot_update_another_user():
    """Users should not be able to update other users' accounts."""

    # 1. Setup.

    # Make a mock ckan.model.User object, Fred.
    fred = factories.MockUser()

    # Make a mock ckan.model object.
    mock_model = mock.MagicMock()
    # model.User.get(user_id) should return Fred.
    mock_model.User.get.return_value = fred

    # Put the mock model in the context.
    # This is easier than patching import ckan.model.
    context = {"model": mock_model}

    # The logged-in user is going to be Bob, not Fred.
    context["user"] = "bob"

    # 2. Call the function that's being tested, once only.

    # Make Bob try to update Fred's user account.
    params = {"id": fred.id, "name": "updated_user_name"}

    # 3. Make assertions about the return value and/or side-effects.

    with pytest.raises(logic.NotAuthorized):
        helpers.call_auth("user_update", context=context, **params)

    # 4. Do nothing else!
```

Writing converter and validator tests

All converter and validator functions should have unit tests.

Although these converter and validator functions are tested indirectly by the action function tests, this may not catch all the converters and validators and all their options, and converters and validators are not only used by the action functions but are also available to plugins. Having unit tests will also help to clarify the intended behavior of each converter and validator.

CKAN's action functions call `ckan.lib.navl.dictization_functions.validate()` to validate data posted by the user. Each action function passes a schema from `ckan.logic.schema` to `validate()`. The schema gives `validate()` lists of validation and conversion functions to apply to the user data. These validation and conversion functions are defined in *`ckan.logic.validators`*, *`ckan.logic.converters`* and *`ckan.lib.navl.validators`*.

Most validator and converter tests should be unit tests that test the validator or converter function in isolation, without bringing in other parts of CKAN or touching the database. This requires using the `mock` library to mock `ckan.model`, see *Mocking: the mock library*.

When testing validators, we often want to make the same assertions in many tests: assert that the validator didn't modify the `data` dict, assert that the validator didn't modify the `errors` dict, assert that the validator raised `Invalid`, etc. Decorator functions are defined at the top of validator test modules like `ckan.tests.logic.test_validators` to make these common asserts easy. To use one of these decorators you have to:

1. Define a nested function inside your test method, that simply calls the validator function that you're trying to test.
2. Apply the decorators that you want to this nested function.
3. Call the nested function.

Here's an example of a simple validator test that uses this technique:

```
def test_user_name_validator_with_non_string_value():
    """user_name_validator() should raise Invalid if given a non-string
    value.

    """
    non_string_values = [
        13,
        23.7,
        100,
        1.0j,
        None,
        True,
        False,
        ("a", 2, False),
        [13, None, True],
        {"foo": "bar"},
        lambda x: x ** 2,
    ]

    # Mock ckan.model.
    mock_model = mock.MagicMock()
    # model.User.get(some_user_id) needs to return None for this test.
    mock_model.User.get.return_value = None

    key = ("name",)
    for non_string_value in non_string_values:
        data = validator_data_dict()
        data[key] = non_string_value
        errors = validator_errors_dict()
        errors[key] = []

        @t.does_not_modify_data_dict
        @raises_invalid
        def call_validator(*args, **kwargs):
            return validators.user_name_validator(*args, **kwargs)

        call_validator(key, data, errors, context={"model": mock_model})
```

No tests for `ckan.logic.schema.py`

We *don't* write tests for the schemas defined in `ckan.logic.schema`. The validation done by the schemas is instead tested indirectly by the action function tests. The reason for this is that CKAN actually does validation in multiple places: some validation is done using schemas, some validation is done in the action functions themselves, some is done in dictization, and some in the model. By testing all the different valid and invalid inputs at the action function level, we catch it all in one place.

Writing `ckan.controllers` tests

Controller tests probably shouldn't use mocking.

Todo: Write the tests for one controller, figuring out the best way to write controller tests. Then fill in this guidelines section, using the first set of controller tests as an example.

Some things have been decided already:

- All controller methods should have tests
- Controller tests should be high-level tests that work by posting simulated HTTP requests to CKAN URLs and testing the response. So the controller tests are also testing CKAN's templates and rendering - these are CKAN's front-end tests.

  For example, maybe we use a testapp and then use beautiful soup to parse the HTML?
- In general the tests for a controller shouldn't need to be too detailed, because there shouldn't be a lot of complicated logic and code in controller classes. The logic should be handled in other places such as `ckan.logic` and `ckan.lib`, where it can be tested easily and also shared with other code.
- The tests for a controller should:
  - Make sure that the template renders without crashing.
  - Test that the page contents seem basically correct, or test certain important elements in the page contents (but don't do too much HTML parsing).
  - Test that submitting any forms on the page works without crashing and has the expected side-effects.
  - When asserting side-effects after submitting a form, controller tests should user the *`ckan.tests.helpers.call_action()`* function. For example after creating a new user by submitting the new user form, a test could call the *`user_show()`* action function to verify that the user was created with the correct values.

Warning: Some CKAN controllers *do* contain a lot of complicated logic code. These controllers should be refactored to move the logic into `ckan.logic` or `ckan.lib` where it can be tested easily. Unfortunately in cases like this it may be necessary to write a lot of controller tests to get this code's behavior into a test harness before it can be safely refactored.

Writing `ckan.model` tests

All model methods should have tests.

Todo: Write the tests for one `ckan.model` module, figuring out the best way to write model tests. Then fill in this guidelines section, using the first set of model tests as an example.

Writing `ckan.lib` tests

All lib functions should have tests.

Todo: Write the tests for one `ckan.lib` module, figuring out the best way to write lib tests. Then fill in this guidelines section, using the first

We probably want to make these unit tests rather than high-level tests and mock out `ckan.model`, so the tests are really fast and simple.

Note that some things in lib are particularly important, e.g. the functions in *`ckan.lib.helpers`* are exported for templates (including extensions) to use, so all of these functions should really have tests and docstrings. It's probably worth focusing on these modules first.

Writing `ckan.plugins` tests

The plugin interfaces in *`ckan.plugins.interfaces`* are not directly testable because they don't contain any code, *but*:

- Each plugin interface should have an example plugin in `ckan.ckanext` and the example plugin should have its own functional tests.
- The tests for the code that calls the plugin interface methods should test that the methods are called correctly.

For example *`ckan.logic.action.get.package_show()`* calls `ckan.plugins.interfaces.IDatasetForm.read()`, so the *`package_show()`* tests should include tests that `read()` is called at the right times and with the right parameters.

Everything in *`ckan.plugins.toolkit`* should have tests, because these functions are part of the API for extensions to use. But *`toolkit`* imports most of these functions from elsewhere in CKAN, so the tests should be elsewhere also, in the test modules for the modules where the functions are defined.

Other than the plugin interfaces and plugins toolkit, any other code in `ckan.plugins` should have tests.

Writing `ckan.ckanext` tests

Within extensions, follow the same guidelines as for CKAN core. For example if an extension adds an action function then the action function should have tests, etc.

7.17 Frontend development guidelines

7.17.1 Templating

Within CKAN 2.0 we moved out templating to use Jinja2 from Genshi. This was done to provide a more flexible, extensible and most importantly easy to understand templating language.

Some useful links to get you started.

- Jinja2 Homepage
- Jinja2 Developer Documentation
- Jinja2 Template Documentation

Legacy Templates

Existing Genshi templates have been moved to the *templates_legacy* directory and will continue to be served if no file with the same name is located in *templates*. This should ensure backward compatibility until instances are able to upgrade to the new system.

The lookup path for templates is as follows. Give the template path "user/index.html":

1. Look in the template directory of each loaded extension.
2. Look in the template_legacy directory for each extension.
3. Look in the main ckan template directory.
4. Look in the template_legacy directory.

CKAN will automatically determine the template engine to use.

File Structure

The file structure for the CKAN templates is pretty much the same as before with a directory per controller and individual files per action.

With Jinja2 we also have the ability to use snippets which are small fragments of HTML code that can be pulled in to any template. These are kept in a snippets directory within the same folder as the actions that are using them. More generic snippets are added to templates/snippets.

```
templates/
  base.html             # A base template with just core HTML structure
  page.html             # A base template with default page layout
  header.html           # The site header.
  footer.html           # The site footer.
  snippets/             # A folder of generic sitewide snippets
  home/
    index.html          # Template for the index action of the home controller
    snippets/           # Snippets for the home controller
  user/
    ...
templates_legacy/
  # All ckan templates
```

Using the templating system

Jinja2 makes heavy use of template inheritance to build pages. A template for an action will tend to inherit from *page.html*:

```
{% extends "page.html" %}
```

Each parent defines a number of blocks that can be overridden to add content to the page. *page.html* defines majority of the markup for a standard page. Generally only `{% block primary_content %}` needs to be extended:

```
{% extends "page.html" %}

{% block page_content.html %}
  <h1>My page title</h1>
  <p>This content will be added to the page</p>
{% endblock %}
```

Most template pages will define enough blocks so that the extending page can customise as little or as much as required.

Internationalisation

See *Internationalizating strings in Jinja2 templates*.

Conventions

There are a few common conventions that have evolved from using the language.

Includes

Note: Includes should be avoided as they are not portable use {% snippet %} tags whenever possible.

Snippets of text that are included using `{% include %}` should be kept in a directory called _snippets_. This should be kept in the same directory as the code that uses it.

Generally we use the `{% snippet %}` helper in all theme files unless the parents context must absolutely be available to the snippet. In which case the usage should be clearly documented.

Snippets

Note: {% snippet %} tags should be used in favour of h.snippet()

Snippets are essentially middle ground between includes and macros in that they are includes that allow a specific context to be provided (includes just receive the parent context).

These should be preferred to includes at all times as they make debugging much easier.

Macros

Macros should be used very sparingly to create custom generators for very generic snippets of code. For example macros/form.html has macros for creating common form fields.

They should generally be avoided as they are hard to extend and customise.

Templating within extensions

When you need to add or customize a template from within an extension you need to tell CKAN that there is a template directory that it can call from. Within your `update_config` method for the extension you'll need to add a `extra_template_paths` to the `config`.

Custom Control Structures

We've provided a few additional control structures to make working with the templates easier. Other helpers can still be used using the `h` object as before.

ckan_extends

```
{% ckan_extends %}
```

This works in a very similar way to `{% extend %}` however it will load the next template up in the load path with the same name.

For example if you wish to remove the breadcrumb from the user profile page in your own site. You would locate the template you wish to override.

```
ckan/templates/user/read.html
```

And create a new one in your theme extension.

```
ckanext-mytheme/ckanext/mytheme/templates/user/read.html
```

In this new file you would pull in the core template using `{% ckan_extends %}`:

```
{% ckan_extends %}
```

This will now render the current user/read page but we can override any portion that we wish to change. In this case the `breadcrumb` block.

```
{% ckan_extends %}

{# Remove the breadcrumb #}
{% block breadcrumb %}{% endblock %}
```

This function works recursively and so is ideal for extensions that wish to add a small snippet of functionality to the page.

Note: {% ckan_extend %} only extends templates of the same name.

snippet

```
{% snippet [filepath], [arg1=arg1], [arg2=arg2]... %}
```

Snippets work very much like Jinja2's `{% include %}` except that that do not inherit the parent templates context. This means that all variables must be explicitly passed in to the snippet. This makes debugging much easier.

```
{% snippet "package/snippets/package_form.html", data=data, errors=errors %}
```

url_for

```
{% url_for [arg1=arg1], [arg2=arg2]... %}
```

Works exactly the same as `h.url_for()`:

```
<a href="{% url_for controller="home", action="index" %}">Home</a>
```

link_for

```
{% link_for text, [arg1=arg1], [arg2=arg2]... %}
```

Works exactly the same as `h.link_for()`:

```
<li>{% link_for _("Home"), controller="home", action="index" %}</li>
```

url_for_static

```
{% url_for_static path %}
```

Works exactly the same as `h.url_for_static()`:

```
<script src="{% url_for_static "/javascript/home.js" %}"></script>
```

Form Macros

For working with forms we have provided some simple macros for generating common fields. These will be suitable for most forms but anything more complicated will require the markup to be written by hand.

The macros can be imported into the page using the `{% import %}` command.

```
{% import 'macros/form.html' as form %}
```

The following fields are provided:

form.input()

Creates all the markup required for an input element. Handles matching labels to inputs, error messages and other useful elements.

```
name        - The name of the form parameter.
id          - The id to use on the input and label. Convention is to prefix with 'field-
↪'.
label       - The human readable label.
value       - The value of the input.
placeholder - Some placeholder text.
type        - The type of input eg. email, url, date (default: text).
error       - A list of error strings for the field or just true to highlight the field.
classes     - An array of classes to apply to the control-group.
attrs       - Dictionary of extra tag attributes
is_required - Boolean of whether this input is required for the form to validate
```

Examples:

```
{% import 'macros/form.html' as form %}
{{ form.input('title', label=_('Title'), value=data.title, error=errors.title) }}
```

form.checkbox()

Builds a single checkbox input.

```
name        - The name of the form parameter.
id          - The id to use on the input and label. Convention is to prefix with 'field-
↪'.
label       - The human readable label.
value       - The value of the input.
checked     - If true the checkbox will be checked
error       - An error string for the field or just true to highlight the field.
classes     - An array of classes to apply to the control-group.
attrs       - Dictionary of extra tag attributes
is_required - Boolean of whether this input is required for the form to validate
```

Example:

```
{% import 'macros/form.html' as form %}
{{ form.checkbox('remember', checked=true) }}
```

form.select()

Creates all the markup required for an select element. Handles matching labels to inputs and error messages.

A field should be a dict with a "value" key and an optional "text" key which will be displayed to the user. `{"value": "my-option", "text": "My Option"}`. We use a dict to easily allow extension in future should extra options be required.

```
name        - The name of the form parameter.
id          - The id to use on the input and label. Convention is to prefix with 'field-
↪'.
label       - The human readable label.
options     - A list/tuple of fields to be used as <options>.
selected    - The value of the selected <option>.
error       - A list of error strings for the field or just true to highlight the field.
classes     - An array of classes to apply to the control-group.
attrs       - Dictionary of extra tag attributes
is_required - Boolean of whether this input is required for the form to validate
```

Examples:

```
{% import 'macros/form.html' as form %}
{{ form.select('year', label=_('Year'), options=[{'name':2010, 'value': 2010},{'name':␣
↪2011, 'value': 2011}], selected=2011, error=errors.year) }}
```

form.textarea()

Creates all the markup required for a plain textarea element. Handles matching labels to inputs, selected item and error messages.

```
name        - The name of the form parameter.
id          - The id to use on the input and label. Convention is to prefix with 'field-
↪'.
label       - The human readable label.
value       - The value of the input.
placeholder - Some placeholder text.
error       - A list of error strings for the field or just true to highlight the field.
classes     - An array of classes to apply to the control-group.
attrs       - Dictionary of extra tag attributes
is_required - Boolean of whether this input is required for the form to validate
```

Examples:

```
{% import 'macros/form.html' as form %}
{{ form.textarea('desc', id='field-description', label=_('Description'), value=data.desc,
↪ error=errors.desc) }}
```

form.markdown()

Creates all the markup required for a Markdown textarea element. Handles matching labels to inputs, selected item and error messages.

```
name        - The name of the form parameter.
id          - The id to use on the input and label. Convention is to prefix with 'field-
↪'.
label       - The human readable label.
value       - The value of the input.
placeholder - Some placeholder text.
```

(continues on next page)

(continued from previous page)

```
error       - A list of error strings for the field or just true to highlight the field.
classes     - An array of classes to apply to the control-group.
attrs       - Dictionary of extra tag attributes
is_required - Boolean of whether this input is required for the form to validate
```

Examples:

```
{% import 'macros/form.html' as form %}
{{ form.markdown('desc', id='field-description', label=_('Description'), value=data.desc,
↪ error=errors.desc) }}
```

form.prepend()

Creates all the markup required for an input element with a prefixed segment. These are useful for showing url slugs and other fields where the input information forms only part of the saved data.

```
name        - The name of the form parameter.
id          - The id to use on the input and label. Convention is to prefix with 'field-
↪'.
label       - The human readable label.
prepend     - The text that will be prepended before the input.
value       - The value of the input.
              which will use the name key as the value.
placeholder - Some placeholder text.
error       - A list of error strings for the field  or just true to highlight the field.
classes     - An array of classes to apply to the control-group.
attrs       - Dictionary of extra tag attributes
is_required - Boolean of whether this input is required for the form to validate
```

Examples:

```
{% import 'macros/form.html' as form %}
{{ form.prepend('slug', id='field-slug', prepend='/dataset/', label=_('Slug'),␣
↪value=data.slug, error=errors.slug) }}
```

form.custom()

Creates all the markup required for an custom key/value input. These are usually used to let the user provide custom meta data. Each "field" has three inputs one for the key, one for the value and a checkbox to remove it. So the arguments for this macro are nearly all tuples containing values for the (key, value, delete) fields respectively.

```
name        - A tuple of names for the three fields.
id          - An id string to be used for each input.
label       - The human readable label for the main label.
values      - A tuple of values for the (key, value, delete) fields. If delete
              is truthy the checkbox will be checked.
placeholder - A tuple of placeholder text for the (key, value) fields.
error       - A list of error strings for the field or just true to highlight the field.
classes     - An array of classes to apply to the control-group.
```

(continues on next page)

(continued from previous page)

```
attrs       - Dictionary of extra tag attributes
is_required - Boolean of whether this input is required for the form to validate
```

Examples:

```
{% import 'macros/form.html' as form %}
{{ form.custom(
     names=('custom_key', 'custom_value', 'custom_deleted'),
     id='field-custom',
     label=_('Custom Field'),
     values=(extra.key, extra.value, extra.deleted),
     error='')
}}
```

form.autoform()

Builds a form from the supplied form_info list/tuple.

```
form_info       - A list of dicts describing the form field to build.
data            - The form data object.
errors          - The form errors object.
error_summary   - The form errors object.
```

Example

```
{% set form_info = [
    {'name': 'ckan.site_title', 'control': 'input', 'label': _('Site Title'),
↪'placeholder': ''},
    {'name': 'ckan.theme', 'control': 'select', 'options': styles, 'label': _('Style'),
↪'placeholder': ''},
    {'name': 'ckan.site_description', 'control': 'input', 'label': _('Site Tag Line'),
↪'placeholder': ''},
    {'name': 'ckan.site_logo', 'control': 'input', 'label': _('Site Tag Logo'),
↪'placeholder': ''},
    {'name': 'ckan.site_about', 'control': 'markdown', 'label': _('About'), 'placeholder
↪': _('About page text')},
    {'name': 'ckan.site_intro_text', 'control': 'markdown', 'label': _('Intro Text'),
↪'placeholder': _('Text on home page')},
    {'name': 'ckan.site_custom_css', 'control': 'textarea', 'label': _('Custom CSS'),
↪'placeholder': _('Customisable css inserted into the page header')},
    ] %}

{% import 'macros/form.html' as form %}
{{ form.autoform(form_info, data, errors) }}
```

7.17.2 Assets

Note: Assets are only supported on CKAN 2.9 and above. If you are using CKAN <= 2.8, refer to the legacy Fanstatic resources documentation.

Assets are .css and .js files that may be included in an html page. Assets are included in the page by using the `{% asset %}` tag. CKAN then uses Webassets to serve these assets.

```
{% asset 'library_name/asset_name' %}
```

Assets are grouped into libraries and the full asset name consists of `<library name>/<asset name>`. For example:

```
{% asset 'my_webassets_library/my_javascript_file.js' %}
```

It is important to note that these assets will be added to the page as defined by the assets configuration, not in the location of the `{% asset %}` tag. Duplicate assets will not be added and any dependencies will be included as well as the assets, all in the correct order (see below for details).

Extensions can add new libraries to CKAN using a helper function defined in the :doc:` plugins-toolkit <plugins-toolkit>`. See below.

In debug mode assets are served un-minified and un-bundled (ie each asset is served separately). In non-debug mode the files are served minified and bundled (where allowed).

Note: When adding js and css files to the repository, they should be supplied as un-minified files. Minified files will be created automatically when serving them.

Assets within extensions

To add an asset from an extension, use the `add_resource(path, name)` function:

```
ckan.plugins.toolkit.add_resource('path/to/my/webassets/library/dir',
       'my_webassets_library')
```

The first argument is the path to the asset directory relative to the file calling the function (generally `plugin.py`). The second argument, is the name of the library (to be used by templates when they want to include an asset from the library using the `{% asset %}` tag as, so for instance `my_webassets_library` in the example shown above).

webassets.yml

The `webassets.yml` file is used to define the assets in a directory and its sub-folders. Here is an example file. Each section is described below

```
# Example webassets.yml file

select2-css:
  contents:
    - select2/select2.css
  output: my_webassets_library/%(version)s_select2.css
  filters: cssrewrite
```

(continues on next page)

(continued from previous page)

```
jquery:
  contents:
    - jquery.js
  filters: rjsmin
  output: my_webassets_library/%(version)s_jquery.js

vendor:
  contents:
    - jed.js
    - moment-with-locales.js
    - select2/select2.js
  filters: rjsmin
  output: my_webassets_library/%(version)s_vendor.js
  extra:
    preload:
      - my_webassets_library/select2-css
      - my_webassets_library/jquery
```

Top level items

These are names of the available assets

select2-css

This asset should be added via `{% asset 'my_webassets_library/select2-css' %}`.

jquery

This asset should be added via `{% asset 'my_webassets_library/jquery' %}`.

vendor

This asset should be added via `{% asset 'my_webassets_library/vendor' %}`. If it is present in the template, **select2-css** and **jquery** can be omitted (because they are explicitly defined in `vendor.extra.preload`)

[contents] (required)

List of relative paths to source files that will be used to generate final asset.

Important: An asset *must* only include files of the same type. I.e, one cannot mix JS and CSS files in the same asset.

[output] (optional)

Assets will be compiled the first time they are included in a template. Output files are located either on the path specified by the `ckan.webassets.path` config directive or at `{{ ckan.storage_path }}/webassets` if the former is not provided. The file specified by the **output** option will be created there. If it's not provided, the file will be created in a `webassets-external` sub-folder. The `%(version)s` fragment is a dynamic part that will be replaced with a hash of the generated file content. This technique is useful to address a number of cache issues for static files.

[filters] (optional)

These are the pre-processors that are applied to the file before producing the final asset. `cssrewrite` for CSS and `rjsmin` for JS are supported out of the box. Details and other options can be found in the Webassets documentation

[extra] (optional)

Additional configuration details. Currently, only one option is supported: `preload`.

preload

Defines list of assets in format `asset_library/asset_name`, that must be included into HTML output *before* the current asset.

7.17.3 Creating a new template

This is a brief tutorial covering the basics of building a common template.

Extending a base template

Firstly we need to extend a parent template to provide us with some basic page structure. This can be any other HTML page however the most common one is `page.html` which provides the full CKAN theme including header and footer.

```
{% extends "page.html" %}
```

The `page.html` template provides numerous blocks that can be extended. It's worth spending a few minutes getting familiar with what's available. The most common blocks that we'll be using are those ending with "content".

- `primary_content`: The main content area of the page.
- `secondary_content`: The secondary content (sidebar) of the page.
- `breadcrumb_content`: The contents of the breadcrumb navigation.
- `actions_content`: The content of the actions bar to the left of the breadcrumb.

Primary Content

For now we'll add some content to the main content area of the page.

```
{% block primary_content %}
  {{ super() }}

  {% block my_content %}
    <h2>{{ _('This is my content heading') }}</h2>
    <p>{{ _('This is my content') }}</p>
  {% endblock %}
{% endblock %}
```

Notice we've wrapped our own content in a block. This allows other templates to extend and possibly override this one and is extremely useful for making a them more customisable.

Secondary Content

Secondary content usually compromises of reusable modules which are pulled in as snippets. Snippets are also very useful for keeping the templates clean and allowing theme extensions to override them.

```
{% block primary_content %}
  {{ super() }}

  {% block my_sidebar_module %}
    {% snippet "snippets/my-sidebar-module.html" %}
  {% endblock %}
{% endblock %}
```

Breadcrumb and Actions

There is a consistent breadcrumb running through all the pages and often it is useful to provide additional actions that a related to the page.

```
{% block breadcrumb_content %}
  <li class="active">{% link_for _('Viewing Dataset'), named_route=pkg.type ~ '.read',␣
↪id=pkg.id %}</li>
{% endblock %}

{% block actions_content %}
  {{ super() }}
  <li class="active">{% link_for _('New Dataset'), named_route='dataset.new', class_='btn
↪', icon='plus' %}</li>
{% endblock %}
```

Scripts and Stylesheets

Currently scripts and stylesheets can be added by using the `{% asset %}` tag which manages script loading for us.

```
{% asset 'my-extension/main-css' %}
{% asset 'my-extension/main-js' %}
```

Summary

And that's about all there is to it be sure to check out `base.html` and `page.html` to see all the tags available for extension.

7.17.4 Template Blocks

These blocks can be extended by child templates to replace or extend common CKAN functionality.

Usage

There are currently two base templates *base.html* which provides the bare HTML structure such as title, head and body tags as well as hooks for adding links, stylesheets and scripts. *page.html* defines the content structure and is the template that you'll likely want to use.

To extend a template simply create a new template file and call `{% extend %}` then define the blocks that you wish to override.

7.17.5 Blocks in page.html

page.html extends the "page" block in *base.html* and provides the basic page structure for primary and secondary content.

header

Override the header on a page by page basis by extending this block. If making site wide header changes it is preferable to override the header.html file:

```
{% block header %}
  {% include "custom_header.html" %}
{% endblock %}
```

content

The content block allows you to replace the entire content section of the page with your own markup if needed:

```
{% block content %}
  <div class="custom-content">
    {% block custom_block %}{% endblock %}
  </div>
{% endblock %}
```

toolbar

The toolbar is for content to be added at the top of the page such as the breadcrumb navigation. You can remove/replace this by extending this block:

```
{# Remove the toolbar from this page. #}
{% block toolbar %}{% endblock %}
```

breadcrumb

Add a breadcrumb to the page by extending this element:

```
{% block breadcrumb %}
  {% include "breadcrumb.html" %}
{% endblock %}
```

primary

This block can be used to remove the entire primary content element:

```
{% block primary %}{% endblock %}
```

primary_content

The primary_content block can be used to add content to the page. This is the main block that is likely to be used within a template:

```
{% block primary_content %}
  <h1>My page content</h1>
  <p>Some content for the page</p>
{% endblock %}
```

secondary

This block can be used to remove the entire secondary content element:

```
{% block secondary %}{% endblock %}
```

secondary_content

The secondary_content block can be used to add content to the sidebar of the page. This is the main block that is likely to be used within a template:

```
{% block secondary_content %}
  <h2>A sidebar item</h2>
  <p>Some content for the item</p>
{% endblock %}
```

footer

Override the footer on a page by page basis by extending this block:

```
{% block footer %}
  {% include "custom_footer.html" %}
{% endblock %}
```

If making site wide header changes it is preferable to override the *footer.html*. Adding scripts should use the "scripts" block instead.

7.17.6 Blocks in base.html

doctype

Allows the DOCTYPE to be set on a page by page basis:

```
{% block doctype %}<!DOCTYPE html>{% endblock %}
```

htmltag

Allows custom attributes to be added to the <html> tag:

```
{% block htmltag %}<html lang="en-gb" class="no-js">{% endblock %}
```

headtag

Allows custom attributes to be added to the <head> tag:

```
{% block headtag %}<head data-tag="No idea what you'd add here">{% endblock %}
```

bodytag

Allows custom attributes to be added to the <body> tag:

```
{% block bodytag %}<body class="full-page">{% endblock %}
```

meta

Add custom meta tags to the page. Call `super()` to get the default tags such as charset, viewport and generator:

```
{% block meta %}
  {{ super() }}
  <meta name="author" value="Joe Bloggs" />
  <meta name="description" value="My website description" />
{% endblock %}
```

title

Add a custom title to the page by extending the title block. Call `super()` to get the default page title:

```
{% block title %}My Subtitle - {{ super() }}{% endblock %}
```

links

The links block allows you to add additional content before the stylesheets such as rss feeds and favicons in the same way as the meta block:

```
{% block link %}
  <meta rel="shortcut icon" href="custom_icon.png" />
{% endblock %}
```

styles

The styles block allows you to add additional stylesheets to the page in the same way as the meta block. Use `` super() `` to include the default stylesheets before or after your own:

```
{% block styles %}
  {{ super() }}
  <link rel="stylesheet" href="/base/css/custom.css" />
{% endblock %}
```

page

The page block allows you to add content to the page. Most of the time it is recommended that you extend one of the page.html templates in order to get the site header and footer. If you need a clean page then this is the block to use:

```
{% block page %}
  <div>Some other page content</div>
{% endblock %}
```

scripts

The scripts block allows you to add additional scripts to the page. Use the `super()` function to load the default scripts before/after your own:

```
{% block scripts %}
  {{ super() }}
  <script src="/base/js/custom.js"></script>
{% endblock %}
```

7.17.7 Building a JavaScript Module

CKAN makes heavy use of modules to add additional functionality to the page. Essentially all a module consists of is an object with an `.initialize()` and `.teardown()` method.

Here we will go through the basic functionality of building a simple module that sends a "favourite" request to the server when the user clicks a button.

HTML

The idea behind modules is that the element should already be in the document when the page loads. For example our favourite button will work just fine without our module JavaScript loaded.

```
<form action="/favourite" method="post" data-module="favorite">
  <button class="btn" name="package" value="101">Submit</button>
</form>
```

Here it's the `data-module="favorite"` that tells the CKAN module loader to create a new instance for this element.

JavaScript

Modules reside in the *javascript/modules* directory and should share the same name as the module. We use hyphens to delimit spaces in both filenames and modules.

```
/javascript/modules/favorite.js
```

A module can be created by calling `ckan.module()`:

```
ckan.module('favorite', function (jQuery) {
  return {};
});
```

We pass in the module name and a factory function that should return our module object. This factory gets passed a local jQuery object and a translation object.

Note: In order to include a module for page render inclusion within an extension it is recommended that you use `{% asset %}` within your templates. See the Assets Documentation

Initialisation

Once ckan has found an element on the page it creates a new instance of your module and if present calls the `.initialize()` method.

```
ckan.module('favorite', function (jQuery) {
  return {
    initialize: function () {
      console.log('I've been called for element: %o', this.el);
    }
  };
});
```

Here we can set up event listeners and other setup functions.

```
initialize: function () {
  // Grab our button and assign it to a property of our module.
  this.button = this.$('button');

  // Watch for our favourite button to be clicked.
  this.button.on('submit', jQuery.proxy(this._onClick, this));
},
_onClick: function (event) {}
```

Event Handling

Now we create our click handler for the button:

```
_onClick: function (event) {
  event.preventDefault();
  this.favorite();
}
```

And this calls a `.favorite()` method. It's generally best not to do too much in event handlers it means that you can't use the same functionality elsewhere.

```
favorite: function () {
  // The client on the sandbox should always be used to talk to the api.
  this.sandbox.client.favoriteDataset(this.button.val());
}
```

Internationalisation

See *Internationalizing strings in JavaScript code*.

Notifications

This submits the dataset to the API but ideally we want to tell the user what we're doing.

```
favorite: function () {
  this.button.text('Loading');

  // The client on the sandbox should always be used to talk to the api.
  var request = this.sandbox.client.favoriteDataset(this.button.val())
  request.done(jQuery.proxy(this._onSuccess, this));
},
_onSuccess: function () {
  // Notify allows global messages to be displayed to the user.
  this.sandbox.notify('Done', 'success');
}
```

Options

Displaying an id to the user isn't very friendly. We can use the `data-module` attributes to pass options through to the module.

```
<form action="/favourite" method="post" data-module="favorite" data-module-dataset="my␣
↪dataset">
```

This will override the defaults in the options object.

```
ckan.module('favorite', function (jQuery) {
  return {
    options: {
      dataset: ''
    }
    initialize: function () {
      console.log('this dataset is: %s', this.options.dataset);
      //=> "this dataset is: my dataset"
    }
  };
});
```

Error handling

When ever we make an Ajax request we want to make sure that we notify the user if the request fails. Again we can use `this.sandbox.notify()` to do this.

```
favorite: function () {
  // The client on the sandbox should always be used to talk to the api.
  var request = this.sandbox.client.favoriteDataset(this.button.val())
  request.done(jQuery.proxy(this._onSuccess, this));
  request.fail(jQuery.proxy(this._onError, this));
},
_onError: function () {
  // Notify allows global messages to be displayed to the user.
  this.sandbox.notify('An error occurred!', 'error');
}
```

Module Scope

You may have noticed we keep making calls to `jQuery.proxy()` within these methods. This is to ensure that `this` when the callback is called is the module it belongs to.

We have a shortcut method called `jQuery.proxyAll()` that can be used in the `.initialize()` method to do all the binding at once. It can accept method names or simply a regexp.

```
initialize: function () {
  jQuery.proxyAll(this, '_onSuccess');

  // Same as:
  this._onSuccess = jQuery.proxy(this, '_onSuccess');
```

(continues on next page)

(continued from previous page)

```
  // Even better do all methods starting with _on at once.
  jQuery.proxyAll(this, /_on/);
}
```

Publish/Subscribe

Sometimes we want modules to be able to talk to each other in order to keep the page state up to date. The sandbox has the `.publish()` and `.subscribe()` methods for just this cause.

For example say we had a counter up in the header that showed how many favourite datasets the user had. This would be incorrect when the user clicked the ajax button. We can publish an event when the favorite button is successful.

```
_onSuccess: function () {
  // Notify allows global messages to be displayed to the user.
  this.sandbox.notify(message, 'success');

  // Tell other modules about this event.
  this.sandbox.publish('favorite', this.button.val());
}
```

Now in our other module 'user-favorite-counter' we can listen for this.

```
ckan.module('user-favorite-counter', function (jQuery) {
  return {
    initialize: function () {
      jQuery.proxyAll(this, /_on/);
      this.sandbox.subscribe('favorite', this._onFavorite);
    },
    teardown: function () {
      // We must always unsubscribe on teardown to prevent memory leaks.
      this.sandbox.unsubscribe('favorite', this._onFavorite);
    },
    incrementCounter: function () {
      var count = this.el.text() + 1;
      this.el.text(count);
    },
    _onFavorite: function (id) {
      this.incrementCounter();
    }
  };
});
```

Unit Tests

Every module has unit tests. These use Cypress to assert the expected functionality of the module.

7.17.8 Install frontend dependencies

The front end stylesheets are written using Sass (this depends on node.js being installed on the system)

Instructions for installing Node.js can be found on the Node.js website. Please check the ones relevant to your own distribution

On Ubuntu, run the following to install Node.js official repository and the node package:

```
curl -sL https://deb.nodesource.com/setup_13.x | sudo -E bash -
sudo apt-get install -y nodejs
```

Note: If you use the package on the default Ubuntu repositories (eg `sudo apt-get install nodejs`), the node binary will be called `nodejs`. This will prevent the CKAN Sass script to work properly, so you will need to create a link to make it work:

```
ln -s /usr/bin/nodejs /usr/bin/node
```

For more information, refer to the Node.js instructions.

Dependencies can then be installed via the node package manager (npm). We use `gulp` to make our Sass compiler a watcher style script.

`cd` into the CKAN source folder (eg /usr/lib/ckan/default/src/ckan) and run:

```
$ npm install
```

You may need to use `sudo` depending on your CKAN install type.

7.17.9 File structure

All front-end files to be served via a web server are located in the `public` directory (in the case of the new CKAN base theme it's `public/base`).

```
css/
  main.css
scss/
  main.scss
  _ckan.scss
  ...
javascript/
  main.js
  utils.js
  components/
  ...
vendor/
  jquery.js
  jquery.plugin.js
```

(continues on next page)

(continued from previous page)

```
  underscore.js
  bootstrap.css
  ...
```

All files and directories should be lowercase with hyphens used to separate words.

css
Should contain any site specific CSS files including compiled production builds generated by Sass.

scss
Should contain all the scss files for the site. Additional vendor styles should be added to the *vendor* directory and included in main.scss.

javascript
Should contain all website files. These can be structured appropriately. It is recommended that *main.js* be used as the bootstrap filename that sets up the page.

vendor
Should contain all external dependencies. These should not contain version numbers in the filename. This information should be available in the header comment of the file. Library plugins should be prefixed with the library name. If a dependency has many files (such as bootstrap) then the entire directory should be included as distributed by the maintainer.

7.17.10 Stylesheets

Because all the stylesheets are using Sass we need to compile them before beginning development by running:

```
$ npm run watch
```

This will watch for changes to all of the scss files and automatically rebuild the CSS for you. To quit the script press `ctrl-c`. If you need sourcemaps for debugging, set *DEBUG* environment variable. I.e:

```
$ DEBUG=1 npm run watch
```

There are many Sass files which attempt to group the styles in useful groups. The main two are:

main.scss:
This contains *all* the styles for the website including dependancies and local styles. The only files that are excluded here are those that are conditionally loaded such as IE only CSS and large external apps (like some preview plugins) that only appear on a single page.

ckan.scss:
This includes all the local ckan stylesheets.

Note: Whenever a CSS change effects `main.scss` it's important than after the merge into master that a `$ npm run build` should be run and committed.

There is a basic pattern primer available at: http://localhost:5000/testing/primer/ that shows all the main page elements that make up the CKAN core interface.

7.17.11 JavaScript

The core of the CKAN JavaScript is split up into three areas.

- Core (such as i18n, pub/sub and API clients)
- Modules (small HTML components or widgets)
- jQuery Plugins (very small reusable components)

Core

Everything in the CKAN application lives on the `ckan` namespace. Currently there are four main components that make up the core.

- Modules
- Publisher/Subscriber
- Client
- i18n/Jed

Modules

Modules are the core of the CKAN website, every component that is interactive on the page should be a module. These are then initialized by including a `data-module` attribute on an element on the page. For example:

```
<select name="format" data-module="autocomplete"></select>
```

The idea is to create small isolated components that can easily be tested. They should ideally not use any global objects, all functionality should be provided to them via a "sandbox" object.

There is a global factory that can be used to create new modules and jQuery and Localisation methods are available via `this.sandbox.jQuery` and `this.sandbox.translate()` respectively. To save typing these two common objects we can take advantage of JavaScript closures and use an alternative module syntax that accepts a factory function.

```
ckan.module('my-module', function (jQuery) {
  return {
    initialize: function () {
      // Called when a module is created.
      // jQuery and translate are available here.
    },
    teardown: function () {
      // Called before a module is removed from the page.
    }
  }
});
```

Note: A guide on creating your own modules is located in the *Building a JavaScript Module* guide.

Publisher/subscriber

There is a simple pub/sub module included under `ckan.pubsub` it's methods are available to modules via `this.sandbox.publish/subscribe/unsubscribe`. This can be used to publish messages between modules.

Modules should use the publish/subscribe methods to talk to each other and allow different areas of the UI to update where relevant.

```
ckan.module('language-picker', function (jQuery) {
  return {
    initialize: function () {
      var sandbox = this.sandbox;
      this.el.on('change', function () {
        sandbox.publish('change:lang', this.selected);
      });
    }
  }
});

ckan.module('language-notifier', function (jQuery) {
  return {
    initialize: function () {
      this.sandbox.subscribe('change:lang', function (lang) {
        alert('language is now ' + lang);
      });
    }
  }
});
```

Client

Ideally no module should use jQuery.ajax() to make XHR requests to the CKAN API, all functionality should be provided via the client object.

```
ckan.module('my-module', function (jQuery) {
  return {
    initialize: function () {
      this.sandbox.client.getCompletions(this.options.completionsUrl);
    }
  }
});
```

Internationalization

See *Internationalizing strings in JavaScript code*.

Life cycle

CKAN modules are intialised on dom ready. The `ckan.module.initialize()` will look for all elements on the page with a `data-module` attribute and attempt to create an instance.

```
<select name="format" data-module="autocomplete" data-module-key="id"></select>
```

The module will be created with the element, any options object extracted from `data-module-*` attributes and a new sandbox instance.

Once created the modules `initialize()` method will be called allowing the module to set themselves up.

Modules should also provide a `teardown()` method this isn't used at the moment except in the unit tests to restore state but may become useful in the future.

jQuery plugins

Any functionality that is not directly related to ckan should be packaged up in a jQuery plug-in if possible. This keeps the modules containing only ckan specific code and allows plug-ins to be reused on other sites.

Examples of these are `jQuery.fn.slug()`, `jQuery.fn.slugPreview()` and `jQuery.proxyAll()`.

Unit tests

Every core component, module and plugin should have a set of unit tests. Tests can be filtered using the `grep={regexp}` query string parameter.

Each file has a description block for it's top level object and then within that a nested description for each method that is to be tested:

```
describe('ckan.module.MyModule()', function () {
  describe('.initialize()', function () {
    it('should do something...', function () {
      // assertions.
    });
  });

  describe('.myMethod(arg1, arg2, arg3)', function () {
  });
});
```

The `` `.beforeEach()` `` and `` `.afterEach()` `` callbacks can be used to setup objects for testing (all blocks share the same scope so test variables can be attached):

```
describe('ckan.module.MyModule()', function () {
  before(() => {
      // Open CKAN front page
      cy.visit('/');

      // Pull the class out of the registry.
      cy.window().then(win => {
        // make module available as this.MyModule
        cy.wrap(win.ckan.module.registry['my-module']).as('MyModule');
        win.jQuery('<div id="fixture">').appendTo(win.document.body)
```

(continues on next page)

(continued from previous page)

```
      })
    });

  beforeEach(function () {
    // window object is needed to access the javascript objects
    cy.window().then(win => {
      // Create a test element.
      this.el = win.jQuery('<div />');

      // Create a test sandbox.
      this.sandbox = win.ckan.sandbox();

      // Create a test module.
      this.module = new this.MyModule(this.el, {}, this.sandbox);
    });
  });

  afterEach(function () {
    // Clean up.
    this.module.teardown();
  });
});
```

Templates can also be loaded using the `.loadFixture()` method that is available in all test contexts. Tests can be made asynchronous by using promises (Cypress returns a promise in almost all functions):

```
describe('ckan.module.MyModule()', function () {

  before(function (done) {
    cy.visit('/');

    // Add a fixture element to page
    cy.window().then(win => {
        win.jQuery('<div id="fixture">').appendTo(win.document.body)
    })

    // Load the template once.
    cy.loadFixture('my-template.html').then((template) => {
        cy.wrap(template).as('template');
    });
  });

  beforeEach(function () {
    // Assign the template to the module each time.
    cy.window().then(win => {
        win.jQuery('#fixture').html(this.template).children();
    });
  });
```

7.18 Database migrations

When changes are made to the model classes in `ckan.model` that alter CKAN's database schema, a migration script has to be added to migrate old CKAN databases to the new database schema when they upgrade their copies of CKAN. These migration scripts are kept in `ckan.migration.versions`.

When you upgrade a CKAN instance, as part of the upgrade process you run any necessary migration scripts with the *ckan db upgrade* command.

A migration script should be checked into CKAN at the same time as the model changes it is related to.

To create a new migration script, use CKAN CLI:

```
ckan -c /etc/ckan/default/ckan.ini generate migration -m "Add account table"
```

Update the generated file, because it doesn't contain any actual changes, only placeholders for *upgrade* and *downgrade* steps. For more details see: https://alembic.sqlalchemy.org/en/latest/tutorial.html#create-a-migration-script

Rename the file to include a prefix numbered one higher than the previous one, like the others in `ckan/migration/versions/`.

7.18.1 Manual checking

As a diagnostic tool, you can manually compare the database as created by the model code and the migrations code:

```
# Database created by model
ckan -c |ckan.ini| db clean
ckan -c |ckan.ini| db create-from-model
sudo -u postgres pg_dump -s -f /tmp/model.sql ckan_default

# Database created by migrations
ckan -c |ckan.ini| db clean
ckan -c |ckan.ini| db init
sudo -u postgres pg_dump -s -f /tmp/migrations.sql ckan_default

sudo -u postgres diff /tmp/migrations.sql /tmp/model.sql
```

7.18.2 Troubleshooting

If you are working on a branch that adds new database migrations and merge the most recent commits from master, you might find the following error when running the tests (or manually upgrading the database):

```
        if len(current_heads) > 1:
            raise MultipleHeads(
                current_heads,
>               "%s@head" % branch_label if branch_label else "head")
E           CommandError: Multiple head revisions are present for given argument 'head';␣
↪please specify a specific target revision, '<branchname>@head' to narrow to a specific␣
↪head, or 'heads' for all heads

../../local/lib/python2.7/site-packages/alembic/script/revision.py:271: CommandError
```

This means that your current alembic history has two heads, because a new database migration was also added in master in the meantime. To check which migrations need adjusting, go to the `ckan/migrations` folder and run:

```
alembic history
```

You should see a `branchpoint` revision and two `head` revisions, like in this example:

```
d4d9be9189fe -> 588d7cfb9a41 (head), Add metadata_modified filed to Resource
d4d9be9189fe -> f789f233226e (head), Add package_member_table
01afcadbd8c0 -> d4d9be9189fe (branchpoint), Remove activity.revision_id
0ffc0b277141 -> 01afcadbd8c0, resource package_id index
980dcd44de4b -> 0ffc0b277141, group_extra group_id index
23c92480926e -> 980dcd44de4b, delete migrate version table
```

In this case `d4d9be9189fe` was the latest common migration, and changes in master introduced `588d7cfb9a41`, while we had already added `f789f233226e`.

The easiest fix is to manually set the down revision in our branch migration to the most recent one in master:

```
diff --git a/ckan/migration/versions/f789f233226e_add_package_member_table.py b/ckan/
↪migration/versions/f789f233226e_add_package_member_table.py
index 5628d1350..ade2dd07f 100644
--- a/ckan/migration/versions/f789f233226e_add_package_member_table.py
+++ b/ckan/migration/versions/f789f233226e_add_package_member_table.py
@@ -10,7 +10,7 @@ import sqlalchemy as sa

 # revision identifiers, used by Alembic.
 revision = 'f789f233226e'
-down_revision = u'd4d9be9189fe'
+down_revision = u'588d7cfb9a41'
 branch_labels = None
 depends_on = None
```

This will give us a linear history once again:

```
588d7cfb9a41 -> f789f233226e (head), Add package_member_table
d4d9be9189fe -> 588d7cfb9a41, Add metadata_modified filed to Resource
01afcadbd8c0 -> d4d9be9189fe, Remove activity.revision_id
0ffc0b277141 -> 01afcadbd8c0, resource package_id index
980dcd44de4b -> 0ffc0b277141, group_extra group_id index
23c92480926e -> 980dcd44de4b, delete migrate version table
```

In more complex scenarios like two migrations updating the same tables, you can use the alembic merge command.

7.19 Upgrading CKAN's dependencies

The Python modules that CKAN depends on are pinned to specific versions, so we can guarantee that whenever anyone installs CKAN, they'll always get the same versions of the Python modules in their virtual environment.

Our dependencies are defined in three files:

requirements.in
: This file is only used to create a new version of the `requirements.txt` file when upgrading the dependencies. Contains our direct dependencies only (not dependencies of dependencies) with loosely defined versions. For example, `python-dateutil>=1.5.0,<2.0.0`.

requirements.txt
This is the file that people actually use to install CKAN's dependencies into their virtualenvs. It contains every dependency, including dependencies of dependencies, each pinned to a specific version. For example, `simplejson==3.3.1`.

dev-requirements.txt
Contains those dependencies only needed by developers, not needed for production sites. These are pinned to a specific version. For example, `factory-boy==2.1.1`.

We haven't created a `dev-requirements.in` file because we have too few dev dependencies, we don't update them often, and none of them have a known incompatible version.

7.19.1 Steps to upgrade

These steps will upgrade all of CKAN's dependencies to the latest versions that work with CKAN:

1. Create a new virtualenv: `virtualenv --no-site-packages upgrading`
2. Install the requirements with unpinned versions: `pip install -r requirements.in`
3. Save the new dependencies versions: `pip freeze > requirements.txt`. We have to do this before installing the other dependencies so we get only what was in `requirements.in`
4. Install CKAN: `python setup.py develop`
5. Install the development dependencies: `pip install -r dev-requirements.txt`
6. Run the tests to make sure everything still works (see *Testing CKAN*).
   - If not, try to fix the problem. If it's too complicated, pinpoint which dependency's version broke our tests, find an older version that still works, and add it to `requirements.in` (i.e., if `python-dateutil` 2.0.0 broke CKAN, you'd add `python-dateutil>=1.5.0,<2.0.0`). Go back to step 1.
7. Navigate a bit on CKAN to make sure the tests didn't miss anything. Review the dependencies changes and their changelogs. If everything seems fine, go ahead and make a pull request (see *Making a pull request*).

7.20 Doing a CKAN release

This section documents the steps followed by the development team to do a new CKAN release.

See also:

Upgrading CKAN
An overview of the different kinds of CKAN release, and the process for upgrading a CKAN site to a new version.

7.20.1 Process overview

Changed in version 2.6.

The process of a new release starts with the creation of a new release development branch. A release development branch is the one that will be stabilized and eventually become the actual released version. Release branches are always named `dev-vM.m`, after the *major and minor versions* they include. Major and minor versions are always branched from master. When the release is actually published a patch version number is added and the release is tagged in the form `ckan-M.m.p`. All backports are cherry-picked on the `dev-vM.m` branch.

```
+--+----------------------------------------+------------>  Master
   |                                        |
   +-----+------------+------>  dev-v2.6    +------->  dev-v2.7
         |            |
     ckan-2.6.0    ckan-2.6.1
```

Additionally, the `release-vM.m-latest` branches always contain the latest published release for that version (eg `2.6.1` on the example above).

Note: Prior to CKAN 2.6, release branches were named `release-vM.m.p`, after the *major, minor and patch versions* they included, and patch releases were always branched from the most recent tip of the previous patch release branch (tags were created with the same convention). Starting from CKAN 2.6, the convention is the one described above.

```
+--+----------------------------------------+------------>  Master
   |                                        |
   +----------------->  release-v2.4.0      +---------->  release-v2.5.0
                      |
                      +--------->  release-v2.4.1
                                 |
                                 +------>  release-v2.4.2
```

Once a release branch has been created there is generally a three-four week period until the actual release. During this period the branch is tested and fixes cherry-picked. The whole process is described in the following sections.

7.20.2 Doing the beta release

Beta releases are branched off a certain point in master and will eventually become stable releases.

Turn this file into a github issue with a checklist using this command:

```
echo 'Full instructions here: https://github.com/ckan/ckan/blob/master/doc/contributing/
↪release-process.rst'; egrep '^(\#\.|Doing|Leading|Preparing)' doc/contributing/release-
↪process.rst | sed 's/^\([^#]\)/\n## \1/g' | sed 's/\#\./* [ ]/g' |sed 's/::/./g'
```

1. Create a new release branch:

   ```
   git checkout -b dev-v2.7
   ```

   Update `ckan/__init__.py` to change the version number to the new version with a *b* after it, e.g. *2.7.0b* (Make sure to include 0 as the patch version number). Commit the change and push the new branch to GitHub:

   ```
   git commit -am "Update version number"
   git push origin dev-v2.7
   ```

   You will probably need to update the same file on master to increase the version number, in this case ending with an *a* (for alpha).

   During the beta process, all changes to the release branch must be cherry-picked from master (or merged from special branches based on the release branch if the original branch was not compatible).

   As in the master branch, if some commits involving CSS changes are cherry-picked from master, the sass compiling command needs to be run on the release branch. This will update the `main.css` file:

```
npm run build
git commit -am "Rebuild CSS"
git push
```

2. Update beta.ckan.org to run new branch.

   The beta staging site (http://beta.ckan.org, currently on s084) must be set to track the latest beta release branch to allow user testing. This site is automatically updated nightly.

   Check the message on the front page reflects the current version. Edit it as a syadmin here: http://beta.ckan.org/ckan-admin/config

3. Announce the branch and ask for help testing on beta.ckan.org on ckan-dev.

4. Create the documentation branch from the release branch. This branch should be named just with the minor version and nothing else (eg `2.7`, `2.8`, etc). We will use this branch to build the documentation in Read the Docs on all patch releases for this version.

5. Make latest translation strings available on Transifex.

   During beta, a translation freeze is in place (ie no changes to the translatable strings are allowed). Strings need to be extracted and uploaded to Transifex:

   a. Install the Transifex CLI.

   b. Create a `~/.transifexrc` file if necessary with your login details (To generate the token, go to the Transifex user settings page):

   ```
   [https://www.transifex.com]
   api_hostname  = https://api.transifex.com
   hostname      = https://www.transifex.com
   username      = api
   password      = ADD_YOUR_TOKEN_HERE
   rest_hostname = https://rest.api.transifex.com
   token         = ADD_YOUR_TOKEN_HERE
   ```

   c. Extract new strings from the CKAN source code into the `ckan.pot` file. The pot file is a text file that contains the original, untranslated strings extracted from the CKAN source code.:

   ```
   python setup.py extract_messages
   ```

   The po files are text files, one for each language CKAN is translated to, that contain the translated strings next to the originals. Translators edit the po files (on Transifex) to update the translations. We never edit the po files locally.

   c. Get the latest translations (of the previous CKAN release) from Transifex, in case any have changed since:

   ```
   tx pull --all --minimum-perc=5 --force
   ```

   (This ignores any language files which less than 5% translation - which is the bare minimum we require)

   e. Update the `ckan.po` files with the new strings from the `ckan.pot` file:

   ```
   python setup.py update_catalog --no-fuzzy-matching
   ```

   Any new or updated strings from the CKAN source code will get into the po files, and any strings in the po files that no longer exist in the source code will be deleted (along with their translations).

   We use the `--no-fuzzy-matching` option because fuzzy matching often causes problems with Babel and Transifex.

If you get this error for a new translation:

> babel.core.UnknownLocaleError: unknown locale 'crh'

then it's Transifex appears to know about new languages before Babel does. Just delete that translation locally - it may be ok with a newer Babel in later CKAN releases.

f. Run msgfmt checks:

```
find ckan/i18n/ -name "*.po"| xargs -n 1 msgfmt -c
```

You must correct any errors or you will not be able to send these to Transifex.

A common problem is that Transifex adds to the end of a po file as comments any extra strings it has, but msgfmt doesn't understand them. Just delete these lines.

g. Run our script that checks for mistakes in the ckan.po files:

```
ckan -c |ckan.ini| translation check-po ckan/i18n/*/LC_MESSAGES/ckan.po
```

If the script finds any mistakes then at some point before release you will need to correct them, but it doesn't need to be done now, since the priority is to announce the call for translations.

When it is done, you must do the correction on Transifex and then run the tx pull command again, don't edit the files directly. Repeat until the script finds no mistakes.

h. Edit `.tx/config`, on line 4 to set the Transifex 'resource' to the new major release name (if different). For instance v2.10.0, v2.10.1 and v2.10.2 all share: `[o:okfn:p:ckan:r:2-10]`.

i. Create a new resource in the CKAN project on Transifex by pushing the new pot and po files:

```
tx push --source --translations --force
```

Because it reads the new version number in the `.tx/config` file, tx will create a new resource on Transifex rather than updating an existing resource (updating an existing resource, especially with the `--force` option, can result in translations being deleted from Transifex).

If you get a 'msgfmt' error, go back to the step where msgfmt is run.

j. On Transifex give the new resource a more friendly name. Go to the resource e.g. https://www.transifex.com/okfn/ckan/2-5/ and the settings are accessed from the triple dot icon "...". Keep the slug like "2-4", but change the name to be like "CKAN 2.5".

k. Update the `ckan.mo` files by compiling the po files:

```
python setup.py compile_catalog
```

The mo files are the files that CKAN actually reads when displaying strings to the user.

l. Commit all the above changes to git and push them to GitHub:

```
git add ckan/i18n/*.mo ckan/i18n/*.po
git commit -am "Update strings files before CKAN X.Y.Z call for translations"
git push
```

6. Send an annoucement email with a call for translations.

Send an email to the ckan-dev list, tweet from @CKANproject and send a transifex announcement from: https://www.transifex.com/okfn/ckan/announcements/ . Make sure to post a link to the correct Transifex resource (like this one) and tell users that they can register on Transifex to contribute. Give a deadline in two weeks time.

7. Create deb packages.

   Ideally do this once a week. Create the deb package with the latest release branch, using `betaX` iterations. Deb packages are built using Ansible scripts located at the following repo:

   https://github.com/ckan/ckan-packaging

   The repository contains further instructions on how to run the scripts, but essentially you need to generate the packages (one for precise and one for trusty) on your local machine and upload them to the Amazon S3 bucket.

   To generate the packages, run:

```
./ckan-package -v 2.x.y -i betaX
```

   To upload the files to the S3 bucket, you will need the relevant credentials and to install the Amazon AWS command line interface

   Make sure to upload them to the *build* folder, so they are not mistaken by the stable ones:

```
aws s3 cp python-ckan_2.5.0-precisebeta1_amd64.deb s3://packaging.ckan.org/build/
↪python-ckan_2.5.0-precisebeta1_amd64.deb
```

   Now the .deb files are available at https://packaging.ckan.org/build/ invite people on ckan-dev to test them.

7.20.3 Leading up to the release

1. Update the CHANGELOG.txt with the new version changes.

   - Check that all merged PRs have corresponding fragment inside `changes/` folder. Name of every fragment is following format `{issue number}.{fragment type}`, where *issue number* is GitHub issue id and *fragment type* is one of *migration*, *removal*, *bugfix* or *misc* depending on change introduced by PR. Missing fragments can be created using *towncrier create –edit {issue number}.{fragment type}* command. The following gist has a script that uses the GitHub API to aid in getting the merged issues between releases:

     https://gist.github.com/amercader/4ec55774b9a625e815bf

     But dread found changed the first step slightly to get it to work:

```
git log --pretty=format:%s --reverse --no-merges release-v2.4.2...release-v2.5.
↪0 -- | grep -Pzo "^\[#\K[0-9]+" | sort -u -n > issues_2.5.txt
```

     When all fragments are ready, make a test build:

```
towncrier build --draft
```

     And check output. If no problems identified, compile updated changelog:

```
towncrier build --yes
```

     You'll be asked, whether it's ok to remove source fragments. Feel free to answer "yes" - all changes will be automatically inserted into changelog, so there is no sense in keeping those files. Don't forget to commit changes afterwards.

2. A week before the translations will be closed send a reminder email.

3. Once the translations are closed, sync them from Transifex.

   Pull the updated strings from Transifex:

```
tx pull --all --minimum-perc=5 --force
```

Check and compile them as before:

```
    ckan -c |ckan.ini| translation check-po ckan/i18n/*/LC_MESSAGES/ckan.po
    python setup.py compile_catalog

The compilation shows the translation percentage. Compare this with the new
languages directories added to ckan/i18n::

    git status
```

`git add` any new ones. (If all is well, you won't see any that are under 5% translated.)

Now push:

```
git commit -am "Update translations from Transifex"
git push
```

4. A week before the actual release, announce the upcoming release(s).

   Send an email to the ckan-announce mailing list, so CKAN instance maintainers can be aware of the upcoming releases. List any patch releases that will be also available. Here's an example email.

7.20.4 Doing the final release

Once the release branch has been thoroughly tested and is stable we can do a release.

1. Run the most thorough tests:

```
pytest --ckan-ini=test-core.ini ckan/tests
```

2. Review the CHANGELOG to check it is complete.
3. Check that the docs compile correctly:

```
rm build/sphinx -rf
sphinx-build doc build/sphinx
```

4. Remove the beta letter in the version number.

   The version number is in `ckan/__init__.py` (eg 2.5.0b -> 2.5.0) and commit the change:

```
git commit -am "Update version number for release X.Y.Z"
```

5. Tag the repository with the version number.

   Make sure to push it to GitHub afterwards:

```
git tag -a -m '[release]: Release tag' ckan-X.Y.Z
git push --tags
```

6. Create and deploy the final deb package.

   Move it to the root of the publicly accessible folder of the packaging server from the */build* folder.

   Make sure to rename it so it follows the deb packages name convention:

```
python-ckan_Major.minor_amd64.deb
```

Note that we drop any patch version or iteration from the package name.

7. Upload the release to PyPI:

```
python setup.py sdist upload
```

You will need a PyPI account with admin permissions on the ckan package, and your credentials should be defined on a `~/.pypirc` file such as:

```
[distutils]
index-servers =
    pypi

[pypi]
username: <user-name>
password: <password>
```

For more info, see: here

If running in Vagrant you may get error `error: Operation not permitted` due to failure to create a hard link. The solution is to add a line at the top of setup.py:

```
# Avoid problem releasing to pypi from vagrant
import os
if os.environ.get('USER', '') == 'vagrant':
    del os.link
```

as described here: https://stackoverflow.com/questions/7719380/python-setup-py-sdist-error-operation-not-permitted

If you upload a bad package, then you can remove it from PyPI however you must use a new version number next time.

8. Build new Docker images for the new version in the following repos:

   - openknowledge/docker-ckan -> `openknowledge/ckan-base:{Major:minor}` and `openknowledge/ckan-dev:{Major:minor}` (ping @amercader for this one)
   - ckan/ckan-solr -> `ckan/ckan-solr:{Major:minor}-solr{solr-version}`
   - ckan/ckan-postgres-dev -> `ckan/ckan-postgres-dev:{Major:minor}`

9. Enable the new version of the docs on Read the Docs.

   (You will need an admin account.)

   a. Make sure the documentation branch is up to date with the latest changes in the corresponding `dev-vX.Y` branch.

   b. If this is the first time a minor version is released, go to the Read The Docs versions page and make the relevant release 'active' (make sure to use the documentation branch, ie X.Y, not the development branch, ie dev-vX.Y).

   c. If it is the latest stable release, set it to be the Default Version and check it is displayed on http://docs.ckan.org.

10. Write a CKAN blog post and announce it to ckan-announce & ckan-dev & twitter.

    CKAN blog here: <http://ckan.org/wp-admin>`_

- Example blog
- Example email

Tweet from @CKANproject

11. Cherry-pick the i18n changes from the release branch onto master.

    We don't generally merge or cherry-pick release branches into master, but the files in ckan/i18n are an exception. These files are only ever changed on release branches following the *Doing the beta release* instructions above, and after a release has been finalized the changes need to be cherry-picked onto master.

    To find out what i18n commits there are on the release-v* branch that are not on master, do:

    ```
    git log master..dev-v* ckan/i18n
    ```

    Then `checkout` the master branch, do a `git status` and a `git pull` to make sure you have the latest commits on master and no local changes. Then use `git cherry-pick` when on the master branch to cherry-pick these commits onto master. You should not get any merge conflicts. Run the `check-po` command again just to be safe, it should not report any problems. Run CKAN's tests, again just to be safe. Then do `git push origin master`.

7.20.5 Preparing patch releases

1. Announce the release date & time with a week's notice on ckan-announce.

   Often this will be part of the announcement of a CKAN major/minor release. But if patches go out separately then they will need their own announcement.

2. Update `ckan/__init__.py` with the incremented patch number e.g. *2.5.1* becomes *2.5.2*. Commit the change and push the new branch to GitHub:

   ```
   git commit -am "Update version number"
   git push origin release-v2.5.2
   ```

3. Cherry-pick PRs marked for back-port.

   These are usually marked on Github using the `Backport Pending` labels and the relevant labels for the versions they should be cherry-picked to (eg `Backport 2.5.3`). Remember to look for PRs that are closed i.e. merged. Remove the `Backport Pending` label once the cherry-picking has been done (but leave the version ones).

4. Ask the tech team if there are security fixes or other fixes to include.

5. Update the CHANGELOG.

7.20.6 Doing the patch releases

1. Review the CHANGELOG to check it is complete.

2. Tag the repository with the version number.

   Make sure to push it to GitHub afterwards:

   ```
   git tag -a -m '[release]: Release tag' ckan-X.Y.Z
   git push --tags
   ```

3. Create and deploy the final deb package.

   Create using ckan-packaging checkout e.g.:

```
./ckan-package -v 2.5.2 -i 1
```

Make sure to rename the deb files so it follows the deb packages name convention:

```
python-ckan_Major.minor_amd64.deb
```

Note that we drop the patch version and iteration number from the package name.

Move it to the root of the publicly accessible folder of the packaging server from the */build* folder, replacing the existing file for this minor version.

4. Upload the release to PyPI:

```
python setup.py sdist upload
```

5. Make sure the documentation branch (`X.Y`) is up to date with the latest changes in the corresponding `dev-vX.Y` branch.

6. Write a CKAN blog post and announce it to ckan-announce & ckan-dev & twitter.

   Often this will be part of the announcement of a CKAN major/minor release. But if patches go out separately then they will need their own announcement.

CHAPTER

EIGHT

CHANGELOG

8.1 v.2.10.4 2024-03-13

8.1.1 Migration notes

- The default format for accepted uploads for user, groups and organization images is now limited to PNG, GIF anf JPG. If you need to add additional foramts you can use the *ckan.upload.user.mimetypes* and *ckan.upload.group.mimetypes*) (#7028)
- Public user registration is disabled by default, ie users can not create new accounts from the UI. With this default value, new users can be created by being invited by an organization admin, being created directly by a sysadmin in the `/user/register` endpoint or being created in the CLI using `ckan user add`. To allow public registration see *ckan.auth.create_user_via_web*, but it's strongly encouraged to put some measures in place to avoid spam. (#7028) (#7208)

8.1.2 Minor changes

- Define allowed alternative Solr query parsers via the *ckan.search.solr_allowed_query_parsers* config option (#8053)

8.1.3 Bugfixes

- CVE-2024-27097: fixed potential log injection in reset user endpoint.
- use custom group type from the activity object if it's not supplied, eg on user activity streams (#7980)
- Removes extra <<<HEAD from resources list template (#7998)
- CKAN does not start without `beaker.session.validate_key` option introduced in v2.10.3 (#8023)
- Editing of resources unavailable from package view page. (#8025)
- Pass custom package types through to the 'new resource' activity item (#8034)
- Fix Last Modified sort parameter for bulk-process page (#8048)
- Detect XLSX mimetypes correctly in uploader (#8088)
- Remove nginx cache as configuration from documentation (#8031)
- Fix *clean_db* fixtures breaking when tables are missing (#8054)
- Fix JS error in flash message when adding a Member (#8104)

8.2 v.2.10.3 2023-12-13

8.2.1 Minor changes

- New sites now default to cookie-based sessions (the default value for `beaker.session.type` is now `cookie`. The `beaker.session.samesite` configuration option has been introduced, allowing you to specify the `SameSite` attribute for session cookies. This attribute determines how cookies are sent in cross-origin requests, enhancing security and privacy.

  Note: When using cookie-based sessions, it is now required to set `beaker.session.validate_key` appropriately.

- Skip interactive mode of `ckan user setpass` using `-p/--password` option. (#7530)
- Added support for Solr 9. Users of the official Docker images can use the `ckan/ckan-solr:2.10-solr9` tag. (#7693)
- Update requirements to support more Python versions (#7935)
- Add tooltips when links are truncated, to show the full text. (#7743)
- Added pages to confirm User delete and Dataset Collaborator delete. Fixed cancellation of Group Member delete. (#7813)
- The `validators` attribute of a declared config option makes tries to parse arguments to validators as python literals. If **all** arguments can be parsed, they are passed to a validator factory with original types. If at least one argument is not a valid Python literal, all values are passed as a string (this was the previous behavior). Space characters are still not allowed inside arguments, use the `\\x20` symbol if you need a space in a literal (#7615):

```
# Not changed
`validators: v(xxx)` # v("xxx")
`validators: v("xxx",yyy)` # v("xxx", "yyy")
`validators: v(1,2,none)` # v("1", "2", "none")
`validators: v("hello\\x20world")` # v("hello world")

# Changed
`validators: v("xxx")` # v("xxx")
`validators: v("xxx",1)` # v("xxx", 1)
`validators: v(1,2,None)` # v(1, 2, None)
```

- Automatically add the `not_empty` validator to any config option declared with `required: true` (#7658)

8.2.2 Bugfixes

- CVE-2023-50248: fix potential out of memory error when submitting the dataset form with a specially-crafted field.
- Fix `deprecated` decorator (#7939)
- Fix for missing Tag facets on Home page (#7520)
- Fix errors when running the *ckan db upgrade* command (#7681)
- Fix datastore_search + downloading datastore resources as json with null values (#6713)

- `CONFIG_FROM_ENV_VARS` takes precedence over config file and extensions but those settings are not normalized. (#7502)
- Fixed server not recognizing SSL settings in configuration .ini file (#7758)
- Fix error when indexing a full ISO date with timezone info (#7775)
- Aligned *member_create* with *group_member_save* to prevent possible member duplication. (#7804)
- datastore-only resources now have a visible download button on the resource page (#7806)
- update resource `datastore_active` with a single statement on `datastore_create/delete` (#7832)
- Fixed Octet Streaming for Datastore Dump requests. (#7839)
- Fixed restricting anonymous users in actions to check user in context. (#7871)
- Empty string in `beaker.session.timeout` produces an error instead of never-expiring session (#7881)
- Updated Bootstrap alert-error class to alert-danger (#7901)
- Changed dataset query to check for `+state:` in the `fq_list` as well as the *fq* parameter before forcing `state:active` (#7905)
- View modules use pluggable `ckan.plugins.toolkit.h` instead of *ckan.lib.helpers* (#7923)
- Fix HTML5 validation failing on resource uploads (#7925)
- Fixed issues with the `ckan views create` CLI sub-command. (#7944)
- Improve handling of date fields in Solr (#7775)
- Fix URL validator does not support ":" for specifying ports (#7891)
- Fix user_show for `ckan.auth.public_user_details` (#7866)
- Add missing translations to aria-label attributes (#7947)
- Catch AttributeErrors in license retrieval (#7931)
- Fix downloading datastore resources as json with null values in json columns (#7545)

8.3 v.2.10.2

Unreleased

8.4 v.2.10.1 2023-05-24

8.4.1 Bug fixes

- CVE-2023-32321: fix potential path traversal, remote code execution, information disclosure and DOS vulnerabilities via crafted resource ids.
- Redirect on password reset form error now maintains root_path and locale (#7006)
- Fix display of Popular snippet (#7205)
- Fixes missing CSRF token when trying to remove a group from a package. (#7417)
- `IMiddleware` implementations produce an error mentioning missing `app.after_request` attribute. (#7426)
- Application hangs during startup when using config chains. (#7427)

- Fix exception in `license_list` action (#7454)
- In tests, templates from `ckan.plugins` set by the config file are used even if these plugins are disabled for the test via `pytest.mark.ckan_config("ckan.plugins", "")` (#7483)
- Fix usage of `defer_commit` in context in create actions for users, datasets, organizations and groups.
- `model.Dashboard.get()` no longer creates a dashboard object under the hood if it does not exist in the database (#7487)
- "Groups" link in the header is not translated. (#7500)
- Names are now quoted in From and To addresses in emails, meaning that site titles with commas no longer break email clients. (#7508)
- Pagination widget is not styled in Bootstrap 5 templates. (#7528)
- Fix missing resource URL on update resource with uploaded file (#7449)
- Fix custom macro styles (#7461)
- Fix mobile layout styles (#7467)
- Fix fontawesome icons, replace unavailable FA v3 icons (#7474)
- Fix promote sysadmin layout (#7476)
- Fix markdown macros regression (#7485)
- Set session scope for migrate_db_for fixture (#7563)

8.4.2 Migration notes

- The default storage backend for the session data used by the Beaker library uses the Python `pickle` module, which is considered unsafe. While there is no direct known vulnerability using this vector, a safer alternative is to store the session data in the client-side cookie. This will probably be the default behaviour in future CKAN versions:

```
# ckan.ini

beaker.session.type = cookie
beaker.session.data_serializer = json
# Use a long, random string for this setting
beaker.session.validate_key = CHANGE_ME

beaker.session.httponly = True
beaker.session.secure = True
beaker.session.samesite = Lax
# or Strict, depending on your setup
```

Note: You might need to install an additional library that can provide AES encryption, e.g. `pip install cryptography`

8.5 v.2.10.0 2023-02-15

8.5.1 Overview

- CKAN 2.10 supports Python 3.7 to 3.10
- This version requires a requirements upgrade on source installations
- This version requires a database upgrade
- This version does not require a Solr schema upgrade if you are already using the 2.9 schema, but it is recommended to upgrade to the 2.10 Solr schema.
- Make sure to check the *Migration notes*

8.5.2 Major features

- Added **CSRF protection** to the frontend forms to protect against Cross-Site Request Forgery attacks. This feature is enabled by default in CKAN core, extensions are excluded from the CSRF protection to give time to update them, but CSRF protection will be enforced in the future. To enforce the CSRF protection in extensions you can use the *ckan.csrf_protection.ignore_extensions* setting. See the *CSRF section* in the extension best practices for more information on how to enable it. (#6920)
- Refactored the **Authentication logic** to use Flask-login instead of repoze.who. This has implications on how login sessions are managed (e.g. when and why users might be logged out) and will affect all plugins that modify the standard authentication process. Please check the *Migration notes* section below to learn more (#6560).
- **Configuration declaration**: declare configuration options to ensure validation and default values. All declared CKAN configuration options are validated and converted to the expected type during the application startup. See the *Migration notes* section below to understand the changes involved and check the *documentation*. (#6467)
- Add **Signals** support to allow subscriptor-based features in extensions. See *Signals* (#5359)
- Add **Blanket implementations**: decorators providing common implementations of simple interfaces to reduce boilerplate in plugins. See the `blanket()` method in the *Plugins toolkit reference* (#5169)
- Add CLI commands for API Token management (#5868)
- The CKAN source code is fully typed now (#5924)
- Add extensible snippet for resource uploads (#6226)
- Migrated to **Bootstrap 5** from v3 for the default CKAN theme. Bootstrap v3 templates are still available for use by specifying the base template folder in the configuration (#6307):

```
ckan.base_public_folder=public-bs3
ckan.base_templates_folder=templates-bs3
```

- Removed the **Docker** related files from the main CKAN repository. A brand new official Docker setup can be found at the ckan/ckan-docker repository. (#7370)
- Added new command `ckan shell` that opens an interactive python shell with the Flask's application context preloaded (among other useful objects). (#6919)
- Added new sub-commands to the `search-index` command (#7044 and #7175):
  - `list-orphans` lists all public package IDs which exist in the solr index, but do not exist in the database.
  - `clear-orphans` clears the search index for all the public orphaned packages.
  - `list-unindexed` lists all ununindexed packages

- Add new group command: `clean`. Add `clean users` command to delete users containing images with formats not supported in `ckan.upload.user.mimetypes` config option. (#7241)
- Activities now receive the full dict of the object they refer to in their `data` section. This allows greater flexibility when creating custom activities from plugins. (#6557)
- Site maintainers can choose to completely ignore cookie based by using `ckan.auth.enable_cookie_auth_in_api`. When set to False, all API requests must use *API Tokens*. Note that this is likely to break some existing JS modules from the frontend that perform API calls, so it should be used with caution. (#7088)
- CKAN now records the last time a user was active on the site. The minimum interval between records can be controlled with the *ckan.user.last_active_interval* config option. (#6466)
- `BaseModel` class for declarative SQLAlchemy models added to `ckan.plugins.toolkit`. Models extending `BaseModel` class are attached to the SQLAlchemy's metadata object automatically:

```
from ckan.plugins import toolkit

class ExtModel(toolkit.BaseModel):

    __tablename__ = "ext_model"
    id = Column(String(50), primary_key=True)
    ... (`#7351 <https://github.com/ckan/ckan/pull/7351>`_)
```

- Add dev containers / GitHub Codespaces config (See the documentation

8.5.3 Minor changes

- Test factories extends SQLAlchemy factory, are available via fixtures and produce more random entities using faker library. (#6335)
- Migrated preprocessor from LESS to SCSS for preliminary work for Bootstrap upgrade. (#6175)
- Add `ckan.plugins.core.plugin_loaded` to the core helpers as `plugin_loaded` (#7011)
- Make HTTP response returned on a private dataset if not authorized configurable (#6641)
- Allow `_id` for `datastore_upsert` unique key (#6793)
- Add functionality to `user_show` to fetch own details when logged in without passing id (#5490)
- `datastore_info` now returns more detailed info. It returns database-level metadata in addition to rowcount (aliases, id, size, index_size, db_size and table_type), and the data dictionary with database-level schemata (native_type, index_name, is_index, notnull & uniquekey). See the documentation at `datastore_info()` (#5831)
- `datastore_info` now works with aliases, and can be used to dereference aliases. (#5832)
- Document new `ckan.download_proxy` config value for extensions that download external URLs (#xloader-127)
- Add *organization_followee_count* to the get api (#2628)
- Environment variables prefixed with *CKAN_* can be used as variables inside config file via `option = %(CKAN_***)s` (#6192)
- CLI command `less` is now renamed to `sass` as the preprocessor was changed in #6175. (#6287)
- Support including file attachments when sending emails (#6535)

- Reworked the JavaScript for the view filters to allow for special characters as well as colons and pipes, which previously caused errors. Added a new helper (`decode_view_request_filters()`) to easily decode the new flattened filter string. (#6747)
- Add an index on column resource_id in table resource_view. (#7134)
- Non-sysadmin users are no longer able to change their own state (#6956)
- The "rank" field is no longer returned in datastore_search results unless explicitly defined in the fields parameter (#6961)
- Upgrade requirements to the latest version whenever possible (#7064)
- Create a `fresh_context()` function to allow cleaning the `context` dict preserving some common values (`user`, `model`, etc) (#7112)
- Add `--quiet` option to `ckan user token add` command to mak easier to integrate with automated scripts (#7217)
- Updated and documented input param for `api_token_list` from `user` to `user_id`. `user` is still supported for backwards compatibility but it might be removed in the future. (#7344)

8.5.4 Bugfixes

- Stable default ordering when consuming resource content from datastore (#2317)
- Fix missing activities from UI when internal processes are run by ignored users (#5699)
- Fix the datapusher trigger in case of resource_update via API (#5727)
- package_revise now returns some errors in normal keys instead of under 'message' (#5888)
- Allow multi-level config inheritance (#6000)
- Fix Chinese locales. Note that the URLs for the *zh_CN* and *zh_TW* locales have changed but there are redirects in place, eg http://localhost:5000/zh_CN/dataset -> http://localhost:5000/zh_Hans_CN/dataset (#6008)
- Fix performance bottleneck in activity queries (#6028)
- Keep repeatable facets inside pagination links (#6084)
- Consistent CLI behavior when when no command provided and when using *–help* options (#6120)
- Variables from extended config files (`use = config:...`) have lower precedence. In the following example:

  ```
  ;; a.ini
  output = %(var)s

  ;; b.ini
  use = config:a.ini
  var = B

  ;; c.ini
  use = config:b.ini
  var = C
  ```

  final value of the `output` config option will be `C`. (#6192)
- Restore error traceback for *search-index rebuild -i* CLI command (#6329)
- Prevent Traceback to logged for HTTP Exception until debug is true Add the HTTP status Code in logging for HTTP requests (#6340)

- Improve rendering data types in resource view (#6356)
- Snippet names rendered into HTML as comments in non-debug mode. (#6406)
- h.remove_url_param fail with minimal set of params (#6414)
- Type of uploads for group and user image can be restricted via the *ckan.upload.{object_type}.types* and *ckan.upload.{object_type}.mimetypes* config options (eg *ckan.upload.group.types*, *ckan.upload.user.mimetypes*) (#6477)
- `*_patch` actions call their `*_update` equivalents via `get_action` allowing plugins to override them consistently (#6519)
- Fixed and simplified organization and group forms breadcrumb inheritance (#6637)
- Ensure that locale exists on i18n JS API (#6698)
- Configuration options that were used to specify a CSS file with a base theme have been removed. Use the altenatives below in order to specify an _asset_ (see *Adding CSS and JavaScript files using Webassets*) with a base theme for application (#6817): * `ckan.main_css` replaced by *ckan.theme* * `ckan.i18n.rtl_css` replaced by *ckan.i18n.rtl_theme*
- prepare_dataset_blueprint: support dataset type (#7031)
- Changed default sort key for group and user lists from ASCII Alphebitized to new *strxfrm* helper, resulting in human-readable alphebitization. (#7039)
- Fix resource file size not updating with resource_patch (#7075)
- Revert Flask requirement from 2.2.2 to 2.0.3. (#7082)
- restore original plugin template directory order after update_config order change (#7085)
- Fix urls containing unicode encoded in hex (#7107)
- Fix a bug that causes CKAN to only register the first blueprint of plugins. (#7108)
- remove old deleted resources on package_update so that performance is consistent over time (no longer degrading) (#7119)
- Beaker session config variables need to be initialised in a newly generated ckan config file (#7133)
- Fixed broken organization delete form (#7150)
- Fix the current year reference for CKAN documentation (#7153)
- Fix bootstrap 3 webassets files to point to valid assets. (#7161)
- Fix the display of the License select element in the Dataset form. (#7162)
- Build CSS files with latest updates. (#7163)
- Fix activity stream icon on Boostrap 5. Migrate activity CSS classes to the extension folder. (#7169)
- Fix 404 error when selecting the same date in the changes view (#7191)
- Fix display of Popular snippet. Removes old *ckan-icon* scss class. (#7205)
- Fix icons and alignment in resource datastore tab. (#7247)
- Make heading semantic in bug report template (#7186)
- Add title attribute to iframe (#7187)
- Fix color contrast in dashboard buttons for web accesibility (#7193)
- Make skip to content visible for keyboard-only user (#7194)
- Fix color contrast issue in add dataset page (#7195)

- Fix color contrast of delete button in user edit page for web accesibility (#7199)

8.5.5 Migration notes

- Changes in the authenticated users management (logged in users): The old `auth_tkt` cookie created by repoze.who does not exist anymore. Flask-login stores the logged-in user identifier in the Flask session. CKAN uses Beaker to manage the session, and the default session backend stores this session information as files on the server (on `/tmp`). This means that **if the session data is deleted in the server, all users will be logged out of the site**. This can happen for instance:
  - if the CKAN container is redeployed in a Docker / cloud setup and the session directory is not persisted
  - if the sessions are periodically cleaned by an external script

  Here's a summary of the behaviour changes between CKAN versions:

Action	CKAN < 2.10	CKAN >= 2.10
Clear cookies	User logged out	User logged out (If `remember_me` cookie is deleted)
Clear server sessions	User still logged in	User logged out

  The way to keep the old behaviour with the Beaker backend is to store the session data in the cookie itself (note that this stores *all* session data, not just the user identifier). This will probably be the default behaviour in future CKAN versions:

```
# ckan.ini
beaker.session.type = cookie
beaker.session.validate_key = CHANGE_ME

beaker.session.httponly = True
beaker.session.secure = True
beaker.session.samesite = Lax # or Strict
```

  Alternatively you can configure another persistent backend for the sessions in the server, like an SQL Database or Redis (see the Beaker configuration for details).

- It is recommended that you review the *Session settings* and *Flask-Login Remember me cookie settings* to make sure they cover your security requirements.
- Due to the newly introduced *Config declaration*, all declared CKAN configuration options are validated and converted to the expected type during the application startup:

```
debug = config.get("debug")

# CKAN <= v2.9
assert type(debug) is str
assert debug == "false" # or any value that is specified in the config file

# CKAN >= v2.10
assert type(debug) is bool
assert debug is False # or ``True``
```

  The `aslist`, `asbool`, `asint` converters from `ckan.plugins.toolkit` will keep the current behaviour:

```
# produces the same result in v2.9 and v2.10
assert tk.asbool(config.get("debug")) is False
assert tk.asint(config.get("ckan.devserver.port")) == 5000
assert tk.aslist(config.get("ckan.plugins")) == ["stats"]
```

If you are using custom logic, the code requires a review. For example, the following code will produce an `AttributeError` exception, because `ckan.plugins` is converted into a list during the application's startup:

```
# AttributeError
plugins = config.get("ckan.plugins").split()
```

Depending on the desired backward compatibility, one of the following expressions can be used instead:

```
# if both v2.9 and v2.10 are supported
plugins = tk.aslist(config.get("ckan.plugins"))

# if only v2.10 is supported
plugins = config.get("ckan.plugins")
```

The second major change affects default values for configuration options. Starting from CKAN 2.10, the majority of the config options have a declared default value. It means that whenever you invoke `config.get` method, the *declared default* value is returned instead of `None`. Example:

```
# CKAN v2.9
assert config.get("search.facets.limit") is None

# CKAN v2.10
assert config.get("search.facets.limit") == 10
```

The second argument to `config.get` should be only used to get the value of a missing *undeclared* option:

```
assert config.get("not.declared.and.missing.from.config", 1) == 1
```

The above is the same for any extension that *declares* its config options using `IConfigDeclaration` interface or `config_declarations` blanket. (#6467)

- Public registration of users has been disabled by default (#7210)
- User and group/org image upload formats have been restricted by default (#7210)
- The activites feature has been extracted into a separate `activity` plugin. To keep showing the activities in the UI and enable the activity related API actions you need to add the `activity` plugin to the *ckan.plugins* config option. This change doesn't affect activities already stored in the DB. They are still available once the plugin is enabled. Note that some imports have changed (#6790):

```
`ckan.model.Activity` -> `ckanext.activity.model.Activity`
```

- Users of the Xloader or DataPusher need to provide a valid API Token in their configurations using the `ckanext.xloader.api_token` or `ckan.datapusher.api_token` keys respectively. (#7139)
- Only user-defined functions can be used as validators. An attempt to use a mock-object, built-in function or class will cause a `TypeError`. (#6048)
- The language code for the Norwegian language has been updated from `no` to `nb_NO`. There are redirects in place from the old code to the new one for localized URLs, but please update your links. If you were using the old `no` code in a config option like `ckan.default_locale` or `ckan.locales_offered` you will need to update the value to `nb_NO`. (#6746)

- *toolkit.aslist* now converts any iterable other than `list` and *tuple* into a `list`: `list(value)`. Before, such values were just wrapped into a list, i.e: `[value]` (#7257).

Table 1: Short overview of changes

Expresion	Before	After
`aslist([1,2])`	`[1, 2]`	`[1, 2]`
`aslist({1,2})`	`[{1, 2}]`	`[1, 2]`
`aslist({1: "one", 2: "two"})`	`[{1: "one", 2: "two"}]`	`[1, 2]`
`aslist(range(1,3))`	`[range(1, 3)]`	`[1, 2]`

8.5.6 Removals and deprecations

- Legacy API keys are no longer supported for Authentication and have been removed from the UI. API Tokens should be used instead. See *Authentication and API tokens* for more details (#6247)
- `build_nav_main()`, `build_nav_icon()` and `build_nav()` helpers no longer support Pylons route syntax. eg use `dataset.search` instead of `controller=dataset, action=search`. (#6263)
- The following old helper functions have been removed and are no longer available: `submit()`, `radio()`, `icon_url()`, `icon_html()`, `icon()`, `resource_icon()`, `format_icon()`, `button_attr()`, `activity_div()` (#6272)
- The following methods are deprecated and should be replaced with their respective new versions in the plugin interfaces:
  - *ckan.plugins.interfaces.IResourceController*:
    - change `before_create` to `before_resource_create`
    - change `after_create` to `after_resource_create`
    - change `before_update` to `before_resource_update`
    - change `after_update` to `after_resource_update`
    - change `before_delete` to `before_resource_delete`
    - change `after_delete` to `after_resource_delete`
    - change `before_show` to `before_resource_show`
  - *ckan.plugins.interfaces.IPackageController*:
    - change `after_create` to `after_dataset_create`
    - change `after_update` to `after_dataset_update`
    - change `after_delete` to `after_dataset_delete`
    - change `after_show` to `after_dataset_show`
    - change `before_search` to `before_dataset_search`
    - change `after_search` to `after_dataset_search`
    - change `before_index` to `before_dataset_index`

  (#6501)

- The `ckan seed` command has been removed in favour of `ckan generate fake-data` for generating test entities in the database. Refer to `ckan generate fake-data --help` for some usage examples. (#6504)
- The `IRoutes` interface has been removed since it was part of the old Pylons architecture. (#6594)
- Remove `ckan.cache_validated_datasets` config (#6628)
- Remove `ckan.search.automatic_indexing` config (#6639)
- The `PluginMapperExtension` has been removed since it was no longer used in core and it had a deprecated dependency. (#6648)
- Remove deprecated `fields` parameter in `resource_search` method. (#6687)
- The `ISession` interface has been removed from CKAN. To extend SQLAlchemy use event listeners instead. (#6699)
- `unselected_facet_items` helper has been removed. You can use `get_facet_items_dict` with `exclude_active=True` instead. (#6765)
- The Recline-based view plugins (`recline_view`, `recline_grid_view`, `recline_graph_view` and `recline_map_view`) are deprecated and will be removed in future versions. Check *Data preview and visualization* for alternatives. (#7078)
- The requirement-setuptools.txt file has been removed (#7271)
- `ckan.route_after_login` renamed to `ckan.auth.route_after_login` (#7350)

8.6 v.2.9.11 2024-03-13

8.6.1 Minor changes

- Define allowed alternative Solr query parsers via the *ckan.search.solr_allowed_query_parsers* config option (#8053). Note that the 2.9 version of this patch does not use pyparsing to parse the local parameters string, so some limitations are in place, mainly that no quotes are allowed in the local paramaters definition.
- Get default formats for DataStore views from config (#8095)

8.6.2 Bugfixes

- CVE-2024-27097: fixed potential log injection in reset user endpoint.
- Fixed Octet Streaming for Datastore Dump requests. (#7899)
- Fix Password Reset Keys with multiple accounts (#8079)
- Detect XLSX mimetypes correctly in uploader (#8088)

8.7 v.2.9.10 2023-12-13

8.7.1 Bugfixes

- CVE-2023-50248: fix potential out of memory error when submitting the dataset form with a specially-crafted field.
- Update resource datastore_active with a single statement (#7833)
- Fix downloading datastore resources as json with null values in json columns (#7545)
- Fix errors when running the *ckan db upgrade* command (#7681)
- Fix `deprecated` decorator (#7939)
- Changed dataset query to check for `+state:` in the `fq_list` as well as the *fq* parameter before forcing `state:active` (#7905)

8.8 v.2.9.9 2023-05-24

8.8.1 Bugfixes

- CVE-2023-32321: fix potential path traversal, remote code execution, information disclosure and DOS vulnerabilities via crafted resource ids.
- Names are now quoted in From and To addresses in emails, meaning that site titles with commas no longer break email clients. (#7508)

8.8.2 Migration notes

- The default storage backend for the session data used by the Beaker library uses the Python `pickle` module, which is considered unsafe. While there is no direct known vulnerability using this vector, a safer alternative is to store the session data in the client-side cookie. This will probably be the default behaviour in future CKAN versions:

```
    # ckan.ini
    beaker.session.type = cookie
beaker.session.data_serializer = json
    beaker.session.validate_key = CHANGE_ME

    beaker.session.httponly = True
    beaker.session.secure = True
    beaker.session.samesite = Lax
# or Strict, depending on your setup
```

8.9 v.2.9.8 2023-02-15

8.9.1 Major changes

- Disable public registration of users by default (#7210)
- Restrict user and group/org image upload formats by default (#7210)

8.9.2 Minor changes

- Add dev containers / GitHub Codespaces config for CKAN 2.9 (See the documentation
- Add new group command: `clean`. Add `clean users` command to delete users containing images with formats not supported in `ckan.upload.user.mimetypes` config option (#7241)
- Set the `resource` blueprint to not auto register. (#7374)
- `prepare_dataset_blueprint`: support dataset type (#7031)
- Add `--quiet` option to `ckan user token add` command to mak easier to integrate with automated scripts (#7217)

8.9.3 Bugfixes

- Fix `package_update` performance (#7219)
- Fix `_()` function override (#7232)
- Fix 404 when selecting the same date in the changes view (#7192)
- Enable DateTime to be returned through Actions, allowing `datapusher_status` to be accessed through the API. (#7110)
- Fixed broken organization delete form (#7150)

8.10 v.2.9.7 2022-10-26

8.10.1 Bugfixes

- CVE-2022-43685: fix potential user account takeover via user create
- Fix Datatables view download format selector (#7147)
- Revert deletions included in 2.9.6 as part of #6187 (#7118)

8.11 v.2.9.6 2022-09-28

Note: This release includes requirements upgrades to address security issues

8.11.1 Bugfixes

- Fixes incorrectly encoded url current_url (#6685)
- Check if locale exists on i18n JS API (#6698)
- Add `csrf_input()` helper for cross-CKAN version compatibilty (#7016)
- Fix not empty validator (#6658)
- Use `get_action()` in patch actions to allow custom logic (#6519)
- Allow to extend organization_facets (#6682)
- Expose check_ckan_version to templates (#6741)
- Allow get_translated helper to fall back to base version of a language (#6815)
- Fix server error in tag autocomplete when vocabulary does not exist (#6820)
- Check if locale exists on i18n JS API (#6698)
- Fix updating a non-existing resource causes an internal sever error (#6928)
- Remove extra comma (#6774)
- Fix test data creation issues (#6805)
- Fix for updating non-existing resource
- Avoid storing the session on each request (#6954)
- Return zero results instead of raising NotFound when vocabulary does not exist
- Fix the datapusher trigger in case of resource_update via API (#5727)
- Consistent CLI behavior when when no command provided and when using *–help* options (#6120)
- Fix regression when validating resource subfields (#6546)
- Fix resource file size not updating with resource_patch (#7075)
- Prevent non-sysadmin users to change their own state (#6956)
- Use user id in auth cookie rather than name
- Reorder resource view button: allow translation (#6089)
- Optmize temp dir creation on uploads (#6578)
- Exclude site_user from user_listi (#6618)
- Fix race condition in creating the default site user (#6638)
- gettext not for metadata fields (#6660)
- Include root_path in activity email notifications (#6743)
- Extract translations from emails (#5857)
- Use the headers Reply-to value if its set in the extensions (#6838)
- Improve error when downloading resource (#6832)

- `ckan_config` test mark works with request context (#6868)
- Fix caching logic on logged in users (#6864)
- Fix member delete (#6892)
- Concurrent-safe resource updates (#6439)
- Fix error when listing tokens in the CLI in py2 (#6789)

8.11.2 Minor changes

- The `ckan.main_css` and `ckan.i18.rtl_css` settings, which were not working, have been replaced by *ckan.theme* and *ckan.i18n.rtl_theme* respectively. Both expect the name of an *asset* with a base theme for the application (#6817)
- The type of uploads for group and user image can be restricted via the *ckan.upload.{object_type}.types* and *ckan.upload.{object_type}.mimetypes* config options (eg *ckan.upload.group.types*, *ckan.upload.user.mimetypes*) (#6477)
- Allow to use PDB and IDE debuggers (#6798)
- Unpin pytz, upgrade zope.interface (#6665)
- Update sqlparse version
- Bump markdown requirement to support Python 3.9
- Update psycopg2 to support PostgreSQL 12
- Add auth functions for 17 actions that didn't have them before (#7045)
- Add no-op `csrf_input()` helper to help extensions with cross-CKAN version suport (#7030)

8.12 v.2.9.5 2022-01-19

8.12.1 Major features

- Solr 8 support. Starting from version 2.9.5, CKAN supports Solr versions 6 and 8. Support for Solr 6 will be dropped in the next CKAN minor version (2.10). Note that if you want to use Solr 8 you need to use the `ckan/config/solr/schema.solr8.xml` file, or alternatively you can use the `ckan/ckan-solr:2.9-solr8` Docker image which comes pre-configured. (#6530)

8.12.2 Bugfixes

- Consistent CLI behavior when no command is provided and when using *–help* (#6120)
- Fix regression when validating resource subfields (#6546)
- Fix user create/edit email validators (#6399)
- Error opening JS translations on Python 2 (#6531)
- Set logging level to error in error mail handler (#6577)
- Add RootPathMiddleware to flask stack to support non-root installs running on python 3 (#6556)
- Use correct auth function when editing organizations (#6622)
- Fix invite user with existing email error (#5880)

- Accept empty string in one of validator (#6612)

8.12.3 Minor changes

- Add timeouts to requests calls (see *ckan.requests.timeout*) (#6408)
- Types of file uploads for group and user imags can be restricted via the *ckan.upload.{object_type}.types* and *ckan.upload.{object_type}.mimetypes* config options (eg *ckan.upload.group.types*, *ckan.upload.user.mimetypes*) (#6477)
- Allow children elements on select2 lists (#6503)
- Enable `minimumInputLength` and fix loading message in select2 (#6554)

8.13 v.2.9.4 2021-09-22

Note: This release includes requirements upgrades to address security issues

8.13.1 Bugfixes

- Don't show snippet names in non-debug mode (#6406)
- Show job title on job start/finish log messages (#6387)
- Fix unpriviledged users being able to access bulk process (#6290)
- Allow UTF-8 in JS translations (#6051)
- Handle Traceback Exception for HTTP and HTTP status Code in logging (#6340)
- Fix object list validation output (#6149)
- Coerce query string keys/values before passing to quote() (#6099)
- Fix datetime formatting when listing user tokens on py2. (#6319)
- Fix Solr HTTP basic auth cred handling (#6286)
- Remove not accessed user object in resource_update (#6220)
- Fix for g.__timer (#6207)
- Fix guard clause on has_more_facets, #6190 (#6190)
- Fix page render errors when search facets are not defined (#6181)
- Fix exception when using solr_user and solr_password on Py3 (#6179)
- Fix pagination links for custom org types (#6162)
- Fixture for plugin DB migrations (#6139)
- Render activity timestamps with title= attribute (#6109)
- Fix db init error in alembic (#5998)
- Fix user email validator when using name as id parameter (#6113)
- Fix DataPusher error during resource_update (#5597)
- render_datetime helper does not respect ckan.display_timezone configuration (#6252)
- Fix SQLAlchemy configuration for DataStore (#6087)

- Don't cache license translations across requests (#5586)
- Fix tracking.js module preventing links to be opened in new tabs (#6386)
- Fix deleted org/group feeds (#6368)
- Fix runaway preview height (#6284)
- Stable default ordering when consuming resource content from datastore (#2317)
- Several documentation fixes and improvements

8.14 v.2.9.3 2021-05-19

8.14.1 Bugfixes

- Fix Chinese locales. Note that the URLs for the *zh_CN* and *zh_TW* locales have changed but there are redirects in place, eg http://localhost:5000/zh_CN/dataset -> http://localhost:5000/zh_Hans_CN/dataset (#6008)
- Fix performance bottleneck in activity queries (#6028)
- Keep repeatable facets inside pagination links (#6084)
- Ensure order of plugins in PluginImplementations (#5965)
- Fix for Datastore file dump extension (#5593)
- Allow package activity migration on py3 (#5930)
- Fix TemplateSyntaxError in snippets/changes/license.html (#5972)
- Remove hardcoded logging level (#5941)
- Include extra files into ckanext distribution (#5995)
- Fix db init in docker as the directory is not empty (#6027)
- Fix sqlalchemy configuration, add doc (#5932)
- Fix issue with purging custom entity types (#5859)
- Only load view filters on templates that need them
- Sanitize user image url
- Allow installation of requirements without any additional actions using pip (#5408)
- Include requirements files in Manifest (#5726)
- Dockerfile: pin pip version (#5929)
- Allow uploaders to only override asset / resource uploading (#6088)
- Catch TypeError from invalid thrown by dateutils (#6085)
- Display proper message when sysadmin password is incorect (#5911)
- Use external library to parse view filter params
- Fix auth error when deleting a group/org (#6006)
- Fix datastore_search language parameter (#5974)
- make SQL function whitelist case-insensitive unless quoted (#5969)
- Fix Explore button not working (#3720)

- remove unused var in task_status_update (#5861)
- Prevent guessing format and mimetype from resource urls without path (#5852)
- Multiple documentation improvements

8.14.2 Minor changes

- Support for setting host and port on the ini file (#5939)
- Allow to set path to INI file in the WSGI script (#5987)
- Allow multi-level config inheritance (#6000)

8.15 v.2.9.2 2021-02-10

General notes:

- Note: To use PostgreSQL 12 on CKAN 2.9 you need to upgrade psycopg2 to at least 2.8.4 (more details in #5796)

8.15.1 Major features

- Add CLI commands for API Token management (#5868)

8.15.2 Bugfixes

- Persist attributes in chained functions (#5751)
- Fix install documentation (#5618)
- Fix exception when passing limit to organization (#5789)
- Fix for adding directories from plugins if partially string matches existing values (#5836)
- Fix upload log activity sorting (#5827)
- Textview: escape text formats (#5814)
- Add allow_partial_update to fix losing users (#5734)
- Set default group_type to group in group_create (#5693)
- Use user performing the action on activity context on user_update (#5743)
- New block in nav links in user dashboard (#5804)
- Update references to DataPusher documentation
- Fix JavaScript error on Edge (#5782)
- Fix error when deleting resource with missing datastore table (#5757)
- ensure HTTP_HOST is bytes under python2 (#5714)
- Don't set old_filename when updating groups (#5707)
- Filter activities from user at the database level (#5698)
- Fix user_list ordering (#5667)

- Allowlist for functions in datastore_search_sql (see *ckan.datastore.sqlsearch.allowed_functions_file*)
- Fix docker install (#5381)
- Fix Click requirement conflict (#5539)
- Return content-type header on downloads if mimetype is (#5670)
- Fix missing activities from UI when internal processes are run by ignored users (#5699)
- Replace 'paster' occurrences with 'ckan' in docs (#5700)
- Include requirements files in Manifest (#5726)
- Fix order which plugins are returned by PluginImplementations changing (#5731)
- Raise NotFound when creating a non-existing collaborator (#5759)
- Restore member edit page (#5767)
- Don't add –ckan-ini pytest option if already added (by pytest-ckan) (#5774)
- Update organization_show package limit docs (#5784)
- Solve encoding errors in changes templates (#5785)

8.15.3 Minor changes

- Add aria attribute and accessible screen reader text to the mobile nav button. (#5555)
- Remove jinja2 blocks from robots.txt (#5648)
- Allow to run the development server using SSL (#5825)
- Update extension template, migrate tests to GitHub Actions (#5797)

8.16 v.2.9.1 2020-10-21

General notes:

- Note: This version requires a database upgrade with `ckan db upgrade` (You should always backup your database first)

8.16.1 Bugfixes

- Restore *stats* extension with reduced functionality (#5215)
- Allow IAuthenticator methods to return responses (#5259)
- Emit activities when updating datasets in bulk (#5479)
- Catch IndexError from date parsing during dataset indexation (#5535)
- Remove foreign keys relationships in revision tables to avoid purge errors (#5542)
- Fix fullscreen for resource webpageview (#5552)
- Fix skip to content link hiding on screen readers (#5556)
- Fix KeyErrors in change list detection (#5562)
- Fix instantiation of smtp on python 3.8 (#5595)

- Fix *unflatten* function and DataDictionary/package extras update bug (#5611)
- Fix managing resources by collaborators (#5620)
- package_revise: allow use by normal users (#5637)
- Fix reloader option on ckan run command (#5639)
- Allow config-tool to be used with an incomplete config file (#5647)

8.16.2 Minor changes

- Add aria attribute and accessible screen reader text to the mobile nav button. (#5555)
- Remove jinja2 blocks from robots.txt (#5648)

8.17 v.2.9.0 2020-08-05

8.17.1 Migration notes

- This version does require a requirements upgrade on source installations
- This version does require a database upgrade
- This version does not require a Solr schema upgrade if you are already using the 2.8 schema, but it is recommended to upgrade to the 2.9 Solr schema.
- This version requires changes to the `who.ini` configuration file. If your setup doesn't use the one bundled with this repo, you will have to manually change the following lines:

  ```
  use = ckan.lib.auth_tkt:make_plugin
  ```

  to:

  ```
  use = ckan.lib.repoze_plugins.auth_tkt:make_plugin
  ```

  And also:

  ```
  use = repoze.who.plugins.friendlyform:FriendlyFormPlugin
  ```

  to:

  ```
  use = ckan.lib.repoze_plugins.friendly_form:FriendlyFormPlugin
  ```

  Otherwise, if you are using symbolinc link to `who.ini` under vcs, no changes required. (#4796)
- All the static CSS/JS files must be bundled via a *webassets.yml* file, as opposed to the previously used, optional *resource.config* file. Check the Assets documentation for more details. (#4614)
- When `ckan.cache_enabled` is set to `False` (default) all requests include the `Cache-control: private` header. If `ckan.cache_enabled` is set to `True`, when the user is not logged in and there is no session data, a `Cache-Control: public` header will be added. For all other requests the `Cache-control: private` header will be added. Note that you will also need to set the `ckan.cache_expires` config option to allow caching of requests. (#4781)

- A full history of dataset changes is now displayed in the Activity Stream to admins, and optionally to the public. By default this is enabled for new installs, but disabled for sites which upgrade (just in case the history is sensitive). When upgrading, open data CKANs are encouraged to make this history open to the public, by setting this in production.ini: `ckan.auth.public_activity_stream_detail = true` (#3972)
- When upgrading from previous CKAN versions, the Activity Stream needs a migrate_package_activity.py running for displaying the history of dataset changes. This can be performed while CKAN is running or stopped (whereas the standard *paster db upgrade* migrations need CKAN to be stopped). Ideally it is run before CKAN is upgraded, but it can be run afterwards. If running previous versions or this version of CKAN, download and run migrate_package_activity.py like this:

```
cd /usr/lib/ckan/default/src/ckan/
wget https://raw.githubusercontent.com/ckan/ckan/2.9/ckan/migration/migrate_package_
↪activity.py
wget https://raw.githubusercontent.com/ckan/ckan/2.9/ckan/migration/revision_legacy_
↪code.py
python migrate_package_activity.py -c /etc/ckan/production.ini
```

  Future versions of CKAN are likely to need a slightly different procedure. Full info about this migration is found here: https://github.com/ckan/ckan/wiki/Migrate-package-activity (#4784)
- The *CKAN configuration file* default name has been changed to `ckan.ini` across the documentation regardless of the environment. You can use any name including the legacy `development.ini` and `production.ini` but to keep in sync with the documentation is recommended to update the name.
- The old *paster* CLI has been removed in favour of the new *ckan* command. In most cases the commands and subcommands syntax is the same, but the `-c` or `--config` parameter to point to the ini file needs to provided immediately after the *ckan* command, eg:

```
ckan -c /etc/ckan/default/ckan.ini sysadmin
```

- The minimum PostgreSQL version required starting from this version is 9.5 (#5458)

8.17.2 Major features

- Python 3 support. CKAN nows supports Python 3.6, 3.7 and 3.8 (Overview). Check this page for support on how to migrate existing extensions to Python 3.
- Dataset collaborators: In addition to traditional organization-based permissions, CKAN instances can also enable the dataset collaborators feature, which allows dataset-level authorization. This provides more granular control over who can access and modify datasets that belong to an organization, or allows authorization setups not based on organizations. It works by allowing users with appropriate permissions to give permissions to other users over individual datasets, regardless of what organization they belong to. To learn more about how to enable it and the different configuration options available, check the documentation on *Dataset collaborators*. (#5346)
- API Tokens: an alternative to API keys. Tokens can be created and removed on demand (check *Authentication and API tokens*) and there is no restriction on the maximum number of tokens per user. Consider using tokens instead of API keys and create a separate token for each use-case instead of sharing the same token between multiple clients. By default API Tokens are JWT, but alternative formats can be implemented using *ckan.plugins.interfaces.IApiToken* interface. (#5146)
- Safe dataset updates with `package_revise`: This is a new API action for safe concurrent changes to datasets and resources. `package_revise` allows assertions about current package metadata, selective update and removal of fields at any level, and multiple file uploads in a single call. See the documentation at `package_revise()` (#4618)
- Refactor frontend assets management to use webassets, including support for *X-Sendfile* (#4614)

- Users can now upload or link to custom profile pictures. By default, if a user picture is not provided it will fall back to gravatar. Alternatively, gravatar can be completely disabled by setting `ckan.gravatar_default = disabled`. In that case a placeholder image is shown instead, which can be customized by overriding the `templates/user/snippets/placeholder.html` template. (#5272)
- Add *plugin_extras* field allowing extending User object for internal use (#5382)

8.17.3 Minor changes

- New command for running database migrations from extensions. See *Don't automatically modify the database structure* for details, (#5150)
- For navl schemas, the 'default' validator no longer applies the default when the value is False, 0, [] or {} (#4448)
- Use alembic instead of sqlalchemy-migrate for managing database migrations (#4450)
- If you've customized the schema for package_search, you'll need to add to it the limiting of `row`, as per default_package_search_schema now does. (#4484)
- Several logic functions now have new upper limits to how many items can be returned, notably `group_list`, `organization_list` when `all_fields=true`, `datastore_search` and `datastore_search_sql`. These are all configurable. (#4562)
- Give users the option to define which page they want to be redirected to after logging in via *ckan.route_after_login* config variable. (#4770)
- Add cache control headers to flask (#4781)
- Create recline_view on ods files by default (#4936)
- Replase nosetests with pytest (#4996)
- Make creating new tags in autocomplete module optional (#5012)
- Allow reply to emails (#5024)
- Improve and reorder resource_formats.json (#5034)
- Email unique validator (#5100)
- Preview for multimedia files (#5103)
- Allow extensions to define Click commands (#5112)
- Add organization and group purge (#5127)
- HTML emails (#5132)
- Unified workflow for creating/applying DB migrations from extensions (#5150)
- Use current package_type for urls (#5189)
- Werkzeug dev server improvements (#5195)
- Allow passing arguments to the RQ enqueue_call function (#5208)
- Add option to configure labels of next/prev page button and pager format. (#5223)
- DevServer: threaded mode and extra files (#5303)
- Make default sorting configurable (#5314)
- Allow initial values in group form (#5345)
- Make ckan more accessible (#5360)
- Update date formatters (#5376)

- Allow multiple *ext_** params in search views (#5398)
- Always 404 on non-existing user lookup (#5464)

8.17.4 Bugfixes

- 500 error when calling *resource_search* by *last_modified* (#4130)
- Action function "datastore_search" would calculate the total, even if you set `include_total=False`. (#4448)
- Emails not sent from flask routes (#4711)
- Admin of organization can add himself as a member/editor to the organization and lose admin rights (#4821)
- Error when posting empty array with type json using datastore_create (#4826)
- ValueError when you configure exception emails (#4831)
- Dataset counts incorrect on Groups listing (#4987)
- Fix broken layout in organization bulk_process (#5147)
- Index template with template path instead of numeric index (#5172)
- Add metadata_modified field to resource (#5236)
- Send the right URL of CKAN to datapusher (#5281)
- Multiline translation strings not translated (#5339)
- Allow repeaded params in h.add_url_param (#5373)
- Accept timestamps with seconds having less than 6 decimals (#5417)
- RTL css fixes (#5420)
- Prevent account presence exposure when *ckan.auth.public_user_details = false* (#5432)
- *ckan.i18n_directory* config option ignored in Flask app. (#5436)
- Allow lists in resource extras (#5453)

8.17.5 Removals and deprecations

- Revision and History UI is removed: */revision/** & */dataset/{id}/history* in favour of */dataset/changes/* visible in the Activity Stream. `model.ActivityDetail` is no longer used and will be removed in the next CKAN release. (#3972)
- `c.action` and `c.controller` variables should be avoided. `ckan.plugins.toolkit.get_endpoint` can be used instead. This function returns tuple of two items(depending on request handler): 1. Flask blueprint name / Pylons controller name 2. Flask view name / Pylons action name In some cases, Flask blueprints have names that are differs from their Pylons equivalents. For example, 'package' controller is divided between 'dataset' and 'resource' blueprints. For such cases you may need to perform additional check of returned value:

```
>>> if toolkit.get_endpoint()[0] in ['dataset', 'package']:
>>>     do_something()
```

  In this code snippet, will be called if current request is handled via Flask's dataset blueprint in CKAN>=2.9, and, in the same time, it's still working for Pylons package controller in CKAN<2.9 (#4319)

- The following logic functions have been removed (#4627): * `dashboard_activity_list_html` * `organization_activity_list_html` * `user_activity_list_html` * `package_activity_list_html` * `group_activity_list_html` * `organization_activity_list_html` * `recently_changed_packages_activity_list_html` * `dashboard_activity_list_html` * `activity_detail_list`
- Remove Bootstrap 2 templates (#4779)
- Extensions that add CLI commands should note the deprecation of `ckan.lib.cli.CkanCommand` and all other helpers in ckan.lib.cli. Extensions should instead implement CLIs using the new IClick interface. (#5112)
- Remove paster CLI (#5264)

8.18 v.2.8.12 2022-10-26

8.18.1 Bugfixes

- CVE-2022-43685: fix potential user account takeover via user create

8.19 v.2.8.11 2022-09-28

Fixes:

- Fixes incorrectly encoded url current_url (#6685)
- Check if locale exists on i18n JS API (#6698)
- Add `csrf_input()` helper for cross-CKAN version compatibilty (#7016)
- Fix not empty validator (#6658)
- Use `get_action()` in patch actions to allow custom logic (#6519)
- Allow to extend organization_facets (#6682)
- Expose check_ckan_version to templates (#6741)
- Allow get_translated helper to fall back to base version of a language (#6815)
- Fix server error in tag autocomplete when vocabulary does not exist (#6820)
- Check if locale exists on i18n JS API (#6698)
- Fix updating a non-existing resource causes an internal sever error (#6928)

8.20 v.2.8.10 2022-01-19

Fixes:

- Add timeouts to requests calls (see *ckan.requests.timeout*) (#6408)
- Fix user create/edit email validators (#6399)
- Allow children elements on select2 lists (#6503)

8.21 v.2.8.9 2021-09-22

Fixes:

- render_datetime helper does not respect ckan.display_timezone configuration (#6252)
- Fix SQLAlchemy configuration for DataStore (#6087)
- Don't cache license translations across requests (#5586)
- Fix tracking.js module preventing links to be opened in new tabs (#6386)
- Fix deleted org/group feeds (#6368)
- Fix runaway preview height (#6284)
- Fix unreliable ordering of DataStore results (#2317)

8.22 v.2.8.8 2021-05-19

- Fix Chinese locales (#4413)
- Allow installation of requirements without any additional actions using pip (#5408)
- Include requirements files in Manifest (#5726)
- Dockerfile: pin pip version (#5929)
- Allow uploaders to only override asset / resource uploading (#6088)
- Catch TypeError from invalid thrown by dateutils (#6085)
- Display proper message when sysadmin password is incorect (#5911)
- Use external library to parse view filter params
- Fix auth error when deleting a group/org (#6006)
- Fix datastore_search language parameter (#5974)
- make SQL function whitelist case-insensitive unless quoted (#5969)
- Fix Explore button not working (#3720)
- remove unused var in task_status_update (#5861)
- Prevent guessing format and mimetype from resource urls without path (#5852)

8.23 v.2.8.7 2021-02-10

General notes: * Note: To use PostgreSQL 12 on CKAN 2.8 you need to upgrade SQLAlchemy to 1.2.17 and vdm to 0.15 (more details in #5796)

Fixes:

- Persist attributes in chained functions (#5751)
- Fix install documentation (#5618)
- Fix exception when passing limit to organization (#5789)
- Fix for adding directories from plugins if partially string matches existing values (#5836)

- Fix upload log activity sorting (#5827)
- Textview: escape text formats (#5814)
- Add allow_partial_update to fix losing users (#5734)
- Set default group_type to group in group_create (#5693)
- Use user performing the action on activity context on user_update (#5743)
- New block in nav links in user dashboard (#5804)
- Update references to DataPusher documentation
- Fix JavaScript error on Edge (#5782)
- Fix error when deleting resource with missing datastore table (#5757)
- ensure HTTP_HOST is bytes under python2 (#5714)
- Don't set old_filename when updating groups (#5707)
- Filter activities from user at the database level (#5698)
- Fix user_list ordering (#5667)
- Allowlist for functions in datastore_search_sql (see *ckan.datastore.sqlsearch.allowed_functions_file*)

8.24 v.2.8.6 2020-10-21

Fixes: * Allow IAuthenticator methods to return responses (#5259) * Fix skip to content link hiding on screen readers (#5556) * Fix unflattening of dataset extras (#5602) * Fix minified JS files in 2.7 (#5557) * Send the right URL of CKAN to datapusher (#5281) * Fix fullscreen for resource webpageview (#5552) * PackageSearchIndex.index_package(): catch IndexError from date parsing (#5535) * Fix collapsible menu in mobile view (#5448) * Refactor query string parsing module

8.25 v.2.8.5 2020-08-05

Fixes:

- Add RTL support (#5413)
- Fix UnicodeDecodeError on abort fucntion (#4829)
- Improve and reorder resource_formats.json (#5034)
- Allow passing arguments to the RQ enqueue_call function (#5208)
- Fix dashboard follower filter (#5412)
- Update dictionary.html for bs2 version (#5365)
- Prevent password reset exposing account presence (#5431)
- Add class dropdown to 'New view' menu (#5470)
- Update jQuery to 3.5.0 (#5364)
- Fix dashboard activity filter (#5424)
- Prevent account presence exposure when ckan.auth.public_user_details = false (#5432)
- Fix resource upload filename fetching in IE (#5438)

- Unflatten: allow nesting >1 level (#5444)
- Allow lists in resource extras (#5453)
- Only add error to tag_errors if not empty (#5454)
- Fix order_by param in user_list action (#5342)
- Fix for Resources validation errors display (#5335)

8.26 v.2.8.4 2020-04-15

General notes:

- Note: This version does not requires a requirements upgrade on source installations
- Note: This version does not requires a database upgrade
- Note: This version does not require a Solr schema upgrade
- Note: This version includes changes in the way the `SameSite` flag is set on the `auth_tkt` authorization cookie. The new default setting for it is `SameSite=Lax`, which aligns with the behaviour of all major browsers. If for some reason you need a different value, you can set it via the *who.samesite* configuration option. You can find more information on the `SameSite` attribute here.

Fixes:

- Fix for number of datasets displayed on the My organizations tab (#3580)
- Allow chaining of core actions (#4509)
- Password reset request - generally tighten it up (#4636)
- Fix start option in data_dict (#4920)
- Add missing get_action calls in activity actions (#4967)
- Fix datetime comparison in resource_dict_save (#5033)
- Fix wrong _ function reference in user blueprint (#5046)
- Allow vocabulary_id in /api/2/util/tag/autocomplete (#5071)
- Fetch less data for *get_all_entity_ids* (#5201)
- Show error in text view if xhr failed (#5271)
- Fix code injection in autocomplete module (#5064)
- Check for the existence of tracking summary data before attempting to load it (#5030)
- Disable streaming for pylons requests (#4431)
- Filter revisions shown according to dataset permissions
- Fix wrong resource URL after ValidationErrors (#5152)
- Update JS vendor libraries
- Samesite support in auth cookie (#5255)
- Handle missing resources in case we have a race condition with the DataPusher (#3980)
- Add the g object to toolkit
- Use returned facets in group controller (#2713)

- Updated translations
- Fix broken translation in image view placeholder (#5099)

8.27 v.2.8.3 2019-07-03

General notes:

- Note: This version does not requires a requirements upgrade on source installations
- Note: This version does not requires a database upgrade
- Note: This version does not require a Solr schema upgrade

Fixes:

- Fix *include_total* in *datastore_search* (#4446)
- Fix problem with reindex-fast (#4352)
- Fix *ValueError* in *url_validator* (#4629)
- Strip local path when uploading file in IE (#4608)
- Increase size of h1 headings to 1.8em (#4665)
- Fix broken div nesting in the *user/read_base.html* (#4672)
- *package_search* parameter *fl* accepts list-like values (#4464)
- Use *chained_auth_function* with core auth functions (#4491)
- Allow translation of custom licenses (#4594)
- Fix delete button links (#4598)
- Fix hardcoded root paths (#4662)
- Fix reCaptcha (#4732)
- Fix incremented follower-counter (#4767)
- Fix breadcrumb on /datasets (#4405)
- Fix *root_path* when using mod_wsgi (#4452)
- Correctly insert root_path for urls generated with _external flag (#4722)
- Make reorder resources button translatable (#4838)
- Fix *feeds* urls generation (#4854)
- More robust auth functions for *resource_view_show* (#4827)
- Allow to customize the DataProxy URL (#4874)
- Allow custom CKAN callback URL for the DataPusher (#4878)
- Add *psycopg>=2.8* support (#4841)

8.28 v.2.8.2 2018-12-12

General notes:

- This version requires a requirements upgrade on source installations
- Note: This version does not requires a database upgrade
- Note: This version does not require a Solr schema upgrade

Fixes:

- Strip full URL on uploaded resources before saving to DB (#4382)
- Fix user not being defined in check_access function (#4574)
- Remove html5 shim from stats extension (#4236)
- Fix for datastore_search distinct=true option (#4236)
- Fix edit slug button (#4379)
- Don't re-register plugin helpers on flask_app (#4414)
- Fix for Resouce View Re-order (#4416)
- autocomplete.js: fix handling of comma key codes (#4421)
- Flask patch update (#4426)
- Allow plugins to define multiple blueprints (#4495)
- Fix i18n API encoding (#4505)
- Allow to defined legacy route mappings as a dict in config (#4521)
- group_patch does not reset packages (#4557)

8.29 v.2.8.1 2018-07-25

General notes:

- Note: This version does not requires a requirements upgrade on source installations
- Note: This version does not requires a database upgrade
- Note: This version does not require a Solr schema upgrade

Fixes:

- "Add Filter" Performance Issue (#4162)
- Error handler update (#4257)
- "New view" button does not work (#4260)
- Upload logo is not working (#4262)
- Unable to pip install ckan (#4271)
- The "License" Icon in 2.8 is wrong (#4272)
- Search - input- border color is overly specific in CSS (#4273)
- Site logo image does not scale down when very large (#4283)

- Validation Error on datastore_search when sorting timestamp fields (#4288)
- Undocumented changes breaking error_document_template (#4303)
- Internal server error when viewing /dashboard when logged out (#4305)
- Missing c.action attribute in 2.8.0 templates (#4310)
- [multilingual] AttributeError: '_Globals' object has no attribute 'fields' (#4338)
- *search* legacy route missing (#4346)

8.30 v.2.8.0 2018-05-09

General notes:

- This version requires a requirements upgrade on source installations
- This version requires a database upgrade
- This version requires a Solr schema upgrade
- This version requires re-running the `datastore set-permissions` command (assuming you are using the DataStore). See: *Set permissions*

  Otherwise new and updated datasets will not be searchable in DataStore and the logs will contain this error:

  ```
  ProgrammingError: (psycopg2.ProgrammingError) function populate_full_text_
  ↪trigger() does not exist
  ```

  CKAN developers should also re-run set-permissions on the test database: *Set up the test databases*
- There are several old features being officially deprecated starting from this version. Check the *Deprecations* section to be prepared.

Major changes:

- New revamped frontend templates based on Bootstrap 3, see "Changes and deprecations" (#3547)
- Allow datastore_search_sql on private datasets (#2562)
- New Flask blueprints migrated from old Pylons controllers: user, dashboard, feeds, admin and home (#3927, #3870, #3775, #3762)
- Improved support for custom groups and organization types (#4032)
- Hide user details to anonymous users (#3915)

Minor changes:

- Allow chaining of authentication functions (#3679)
- Show custom dataset types in search pages (#3807)
- Overriding datastore authorization system (#3679)
- Standardize on url_for (#3831)
- Deprecate notify_after_commit (#3633)
- _mail_recipient header override (#3781)
- Restrict access to member forms (#3684)
- Clean up template rendering code (#3923)

- Permission labels are indexed by type text in SOLR (#3863)
- CLI commands require a Flask test request context (#3760)
- Allow IValidator to override existing validators (#3865)
- Shrink datastore_create response size (#3810)
- Stable version URLs CKAN for documentation (#4209)
- API Documentation update (#4136)
- Documentation of Data Dictionary (#3989)
- Remove datastore legacy mode (#4041)
- Map old Pylons routes to Flask ones (#4066)

Bug fixes:

- File uploads don't work on new Flask based API (#3869)
- {% ckan_extends %} not working on templates served by Flask (#4044)
- Problems in background workers with non-core database relations (#3606)
- Render_datetime can't handle dates before year 1900 (#2228)
- DatapusherPlugin implementation of notify() can call 'datapusher_submit' multiple times (#2334)
- Dataset creation page generates incorrect URLs with Chrome autocomplete (#2501)
- Search buttons need accessible labels (#2550)
- Column name length limit for datastore upload (#2804)
- #2373: Do not validate packages or resources from database to views (#3016)
- Creation of dataset - different behaviour between Web API & CKAN Interface functionality (#3528)
- Redirecting to same page in non-root hosted ckan adds extra root_path to url (#3499)
- Beaker 1.8.0 exception when the code is served from OSX via Vagrant (#3512)
- Add "Add Dataset" button to user's and group's page (#2794)
- Some links in CKAN is not reachable (#2898)
- Exception when specifying a directory in the ckan.i18n_directory option (#3539)
- Resource view filter user filters JS error (#3590)
- Recaptcha v1 will stop working 2018-3-31 (#4061)
- "Testing coding standards" page in docs is missing code snippets (#3635)
- Followers count not updated immediately on UI (#3639)
- Increase jQuery version (#3665)
- Search icon on many pages is not properly vertically aligned (#3654)
- Datatables view can't be used as a default view (#3669)
- Resource URL is not validated on create/update (#3660)
- Upload to Datastore tab shows incorrect time at Upload Log (#3588)
- Filter results button is not working (#3593)
- Broken link in "Upgrading CKAN's dependencies" doc page (#3637)

- Default logo image not properly saved (#3656)
- Activity test relies on datetime.now() (#3644)
- Info block text for Format field not properly aligned in resource form page (#3663)
- Issue upon creating new organization/group through UI form (#3661)
- In API docs "package_create" lists "owner_org" as optional (#3647)
- Embed modal window not working (#3731)
- Frontent build command does not work on master (#3688)
- Loading image duplicated (#3716)
- Datastore set-up error - logging getting in the way (#3694)
- Registering a new account redirects to an unprefixed url (#3834)
- Exception in search page when not authorized (#4081)
- Datastore full-text-search column is populated by postgres trigger rather than python (#3785)
- Datastore dump results are not the same as data in database (#4150)
- Adding filter at resoruce preview doesn't work while site is setup with ckan.root_path param (#4140)
- No such file or directory: '/usr/lib/ckan/default/src/ckan/requirement-setuptools.txt' during installation from source (#3641)
- Register user form missing required field indicators (#3658)
- Datastore full-text-search column is populated by postgres trigger rather than python (#3786)
- Add missing major changes to change log (#3799)
- Paster/CLI config-tool requires _get_test_app which in turn requires a dev-only dependency (#3806)
- Change log doesn't mention necessary Solr scheme upgrade (#3851)
- TypeError: expected byte string object, value of type unicode found (#3921)
- CKAN's state table clashes with PostGIS generated TIGER state table (#3929)
- [Docker] entrypoint initdb.d sql files copied to root (#3939)
- DataStore status page throws TypeError - Bleach upgrade regression (#3968)
- Source install error with who.ini (#4020)
- making a JSONP call to the CKAN API returns the wrong mime type (#4022)
- Deleting a resource sets datastore_active=False to all resources and overrides their extras (#4042)
- Deleting first Group and Organization custom field is not possible (#4094)

Changes and deprecations:

- The default templates included in CKAN core have been updated to use Bootstrap 3. Extensions implementing custom themes are encouraged to update their templates, but they can still make CKAN load the old Bootstrap 2 templates during the transition using the following configuration options:

```
ckan.base_public_folder = public-bs2
ckan.base_templates_folder = templates-bs2
```

- The API versions 1 and 2 (also known as the REST API), ie `/api/rest/*` have been completely removed in favour of the version 3 (action API, `/api/action/*`).

- The old Celery based background jobs have been removed in CKAN 2.8 in favour of the new RQ based jobs (http://docs.ckan.org/en/latest/maintaining/background-tasks.html). Extensions can still of course use Celery but they will need to handle the management themselves.
- After introducing dataset blueprint, *h.get_facet_items_dict* takes search_facets as second argument. This change is aimed to reduce usage of global variables in context. For a while, it has default value of None, in which case, *c.search_facets* will be used. But all template designers are strongly advised to specify this argument explicitly, as in future it'll become required.
- The `ckan.recaptcha.version` config option is now removed, since v2 is the only valid version now (#4061)

8.31 v.2.7.12 2021-09-22

Fixes:

- Fix tracking.js module preventing links to be opened in new tabs (#6384)
- Fix deleted org/group feeds (#6367)
- Fix runaway preview height (#6283)
- Fix unreliable ordering of DataStore results (#2317)

8.32 v.2.7.11 2021-05-19

Fixes:

- Allow uploaders to only override asset / resource uploading (#6088)
- Catch TypeError from invalid thrown by dateutils (#6085)
- Use external library to parse view filter params
- Fix auth error when deleting a group/org (#6006)
- Fix datastore_search language parameter (#5974)
- make SQL function whitelist case-insensitive unless quoted (#5969)
- Fix Explore button not working (#3720)
- "New view" button fix (#4260)
- remove unused var in task_status_update (#5861)
- Prevent guessing format and mimetype from resource urls without path (#5852)

8.33 v.2.7.10 2021-02-10

Fixes:

- Fix install documentation (#5618)
- Fix exception when passing limit to organization (#5789)
- Fix for adding directories from plugins if partially string matches existing values (#5836)
- Fix upload log activity sorting (#5827)
- Textview: escape text formats (#5814)
- Add allow_partial_update to fix losing users (#5734)
- Set default group_type to group in group_create (#5693)
- Use user performing the action on activity context on user_update (#5743)
- New block in nav links in user dashboard (#5804)
- Update references to DataPusher documentation
- Fix JavaScript error on Edge (#5782)
- Fix error when deleting resource with missing datastore table (#5757)
- ensure HTTP_HOST is bytes under python2 (#5714)
- Don't set old_filename when updating groups (#5707)
- Filter activities from user at the database level (#5698)
- Fix user_list ordering (#5667)
- Allow list for functions in datastore_search_sql (see *ckan.datastore.sqlsearch.allowed_functions_file*)

8.34 v.2.7.9 2020-10-21

Fixes:

- Fix unflattening of dataset extras (#5602)
- Fix minified JS files in 2.7 (#5557)
- Send the right URL of CKAN to datapusher (#5281)
- Fix fullscreen for resource webpageview (#5552)
- PackageSearchIndex.index_package(): catch IndexError from date parsing (#5535)
- Fix collapsible menu in mobile view (#5448)
- Refactor query string parsing module

8.35 v.2.7.8 2020-08-05

Fixes:

- Fix UnicodeDecodeError on abort fucntion (#4829)
- Improve and reorder resource_formats.json (#5034)
- Allow passing arguments to the RQ enqueue_call function (#5208)
- Fix dashboard follower filter (#5412)
- Update dictionary.html for bs2 version (#5365)
- Prevent password reset exposing account presence (#5431)
- Add class dropdown to 'New view' menu (#5470)
- Update jQuery to 3.5.0 (#5364)
- Fix dashboard activity filter (#5424)
- Prevent account presence exposure when ckan.auth.public_user_details = false (#5432)
- Fix resource upload filename fetching in IE (#5438)
- Unflatten: allow nesting >1 level (#5444)
- Allow lists in resource extras (#5453)
- Only add error to tag_errors if not empty (#5454)
- Fix order_by param in user_list action (#5342)
- Fix for Resources validation errors display (#5335)

8.36 v.2.7.7 2020-04-15

General notes:

- Note: This version does not requires a requirements upgrade on source installations
- Note: This version does not requires a database upgrade
- Note: This version does not require a Solr schema upgrade
- Note: This version includes changes in the way the `SameSite` flag is set on the `auth_tkt` authorization cookie. The new default setting for it is `SameSite=Lax`, which aligns with the behaviour of all major browsers. If for some reason you need a different value, you can set it via the *who.samesite* configuration option. You can find more information on the `SameSite` attribute here.

Fixes:

- Fix for number of datasets displayed on the My organizations tab (#3580)
- Password reset request - generally tighten it up (#4636)
- Add missing get_action calls in activity actions (#4967)
- Fix datetime comparison in resource_dict_save (#5033)
- Allow vocabulary_id in /api/2/util/tag/autocomplete (#5071)
- Fetch less data for *get_all_entity_ids* (#5201)

- Show error in text view if xhr failed (#5271)
- Fix code injection in autocomplete module (#5064)
- Check for the existence of tracking summary data before attempting to load it (#5030)
- Fix broken translation in image view placeholder (#5099)
- Filter revisions shown according to dataset permissions
- Update JS vendor libraries
- Use returned facets in group controller (#2713)
- Samesite support in auth cookie (#5255)
- Handle missing resources in case we have a race condition with the DataPusher (#3980)
- Add the g object to toolkit

8.37 v.2.7.6 2019-07-03

General notes:

- Note: This version does not requires a requirements upgrade on source installations
- Note: This version does not requires a database upgrade
- Note: This version does not require a Solr schema upgrade

Fixes:

- Fix problem with reindex-fast (#4352)
- Fix *include_total* in *datastore_search* (#4446)
- Fix *ValueError* in *url_validator* (#4629)
- Strip local path when uploading file in IE (#4608)
- Increase size of h1 headings to 1.8em (#4665)
- Fix broken div nesting in the *user/read_base.html* (#4672)
- Use *get_action* to call activity actions (#4684)
- Make reorder resources button translatable (#4838)
- More robust auth functions for *resource_view_show* (#4827)
- Allow to customize the DataProxy URL (#4874)
- Allow custom CKAN callback URL for the DataPusher (#4878)

8.38 v2.7.5 2018-12-12

- Strip full URL on uploaded resources before saving to DB (#4382)
- Fix for datastore_search distinct=true option (#4236)
- Fix edit slug button (#4379)
- Don't re-register plugin helpers on flask_app (#4414)
- Fix for Resouce View Re-order (#4416)
- autocomplete.js: fix handling of comma key codes (#4421)
- Flask patch update (#4426)
- Allow plugins to define multiple blueprints (#4495)
- Fix i18n API encoding (#4505)
- Allow to defined legacy route mappings as a dict in config (#4521)
- group_patch does not reset packages (#4557)

8.39 v2.7.4 2018-05-09

- Adding filter at resoruce preview doesn't work while site is setup with ckan.root_path param (#4140)
- Datastore dump results are not the same as data in database (#4150)

8.40 v2.7.3 2018-03-15

General notes:

- As with all patch releases this one does not include requirement changes. However in some scenarios you might encounter the following error while installing or upgrading this version of CKAN:

  ```
  Error: could not determine PostgreSQL version from '10.2'
  ```

  This is due to a bug in the psycopg2 version pinned to the release. To solve it, upgrade psycopg2 with the following command:

  ```
  pip install --upgrade psycopg2==2.8.2
  ```

- This release does not require a Solr schema upgrade, but if you are having the issues described in #3863 (datasets wrongly indexed in multilingual setups), you can upgrade the Solr schema and reindex to solve them.
- #3422 (implemented in #3425) introduced a major bug where if a resource was deleted and the DataStore was active extras from all resources on the site where changed. This is now fixed as part of this release but if your database is already affected you will need to run a script to restore the extras to their previous state. Remember, you only need to run the script if all the following are true:
  1. You are currently running CKAN 2.7.0 or 2.7.2, and
  2. You have enabled the DataStore, and

3. One or more resources with data on the DataStore have been deleted (or you suspect they might have been)

If all these are true you can run the following script to restore the extras to their previous state:

https://github.com/ckan/ckan/blob/dev-v2.7/scripts/4042_fix_resource_extras.py

This issue is described in #4042

Fixes:

- Fix toggle bars header icon (#3880)
- Change CORS header keys and values to string instead of unicode (#3855)
- Fix cors header when all origins are allowed (#3898)
- Update SOLR schema.xml reference in Dockerfile
- Build local SOLR container by default
- Create datastore indexes only if they are not exist
- Properly close file responses
- Use javascript content-type for jsonp responses (#4022)
- Add Data Dictionary documentation (#3989)
- Fix SOLR index delete_package implementation
- Add second half of DataStore set-permissions command(Docs)
- Fix extras overriding for removed resources (#4042)
- Return a 403 if not authorized on the search page (#4081)
- Add support for user/pass for Solr as ENV var
- Change permission_labels type to string in schema.xml (#3863)
- Disallow solr local parameters
- Improve text view rendering
- Update Orgs/Groups logic for custom fields delete and update (#4094)
- Upgrade Solr Docker image

8.41 v2.7.2 2017-09-28

- Include missing minified JavaScript files

8.42 v2.7.1 2017-09-27

- add field_name to image_upload macro when uploading resources (#3766)
- Add some missing major changes to change log. (#3799)
- _mail_recipient header override (#3781)
- skip url parsing in redirect (#3499)
- Fix multiple errors in i18n of JS modules (#3590)

- Standardize on url_for on popup (#3831)

8.43 v2.7.0 2017-08-02

General notes:

- Starting from this version, CKAN requires at least Postgres 9.3
- Starting from this version, CKAN requires a Redis database. Please refer to the new ckan.redis.url configuration option.
- This version requires a requirements upgrade on source installations
- This version requires a database upgrade
- This version requires a Solr schema upgrade
- There are several old features being officially deprecated starting from this version. Check the *Deprecations* section to be prepared.

Major changes:

- New datatables_view resource view plugin for tabular data (#3444)
- IDataStoreBackend plugins for replacing the default DataStore Postgres backend (#3437)
- datastore_search new result formats and performance improvements (#3523)
- PL/PGSQL triggers for DataStore tables (#3428)
- DataStore dump CLI commands (#3384)
- Wrap/override actions defined in other plugins (#3494)
- DataStore table data dictionary stored as postgres comments (#3414)
- Common session object for Flask and Pylons (#3208)
- Rename deleted datasets when they conflict with new ones (#3370)
- DataStore dump more formats: CSV, TSV, XML, JSON; BOM option (#3390)
- Common requests code for Flask and Pylons so you can use Flask views via the new IBlueprint interface (#3212)
- Generate complete datastore dump files (#3344)
- A new system for asynchronous background jobs (#3165)
- Chaining of action functions (#3494)

Minor changes:

- Renamed example theme plugin (#3576)
- Localization support for groups (#3559)
- Create new resource views when format changes (#3515)
- Email field validation (#3568)
- datastore_run_triggers sysadmin-only action to apply triggers to existing data (#3565)
- Docs updated for Ubuntu 16.04 (#3544)
- Upgrade leaflet to 0.7.7 (#3534)

- Datapusher CLI always-answer-yes option (#3524)
- Added docs for all plugin interfaces (#3519)
- DataStore dumps nested columns as JSON (#3487)
- Faster/optional datastore_search total calculation (#3467)
- Faster group_activity_query (#3466)
- Faster query performance (#3430)
- Marked remaining JS strings translatable (#3423)
- Upgrade font-awesome to 4.0.3 (#3400)
- group/organization_show include_dataset_count option (#3385)
- image_formats config option for image viewer (#3380)
- click may now be used for CLI interfaces: use load_config instead of CkanCommand (#3384)
- package_search option to return only names/ids (#3427)
- user_list all_fields option (#3353)
- Error controller may now be overridden (#3340)
- Plural translations in JS (#3211)
- Support JS translations in extensions (#3272)
- Requirements upgraded (#3305)
- Dockerfile updates (#3295)
- Fix activity test to use utcnow (#3644)
- Changed required permission from 'update' to 'manage_group' (#3631)
- Catch invalid sort param exception (#3630)
- Choose direction of recreated package relationship depending on its type (#3626)
- Fix render_datetime for dates before year 1900 (#3611)
- Fix KeyError in 'package_create' (#3027)
- Allow slug preview to work with autocomplete fields (#2501)
- Fix filter results button not working for organization/group (#3620)
- Allow underscores in URL slug preview on create dataset (#3612)
- Fallback to po file translations on `h.get_translated()` (#3577)
- Fix Fanstatic URL on non-root installs (#3618)
- Fixed escaping issues with `helpers.mail_to` and datapusher logs
- Autocomplete fields are more responsive - 300ms timeout instead of 1s (#3693)
- Fixed dataset count display for groups (#3711)
- Restrict access to form pages (#3684)
- Render_datetime can handle dates before year 1900 (#2228)

API changes:

- `organization_list_for_user` (and the `h.organizations_available()` helper) now return all organizations a user belongs to regardless of capacity (Admin, Editor or Member), not just the ones where she is an administrator (#2457)
- `organization_list_for_user` (and the `h.organizations_available()` helper) now default to not include package_count. Pass include_dataset_count=True if you need the package_count values.
- `resource['size']` will change from string to long integer (#3205)
- Font Awesome has been upgraded from version 3.2.1 to 4.0.3 .Please refer to https://github.com/FortAwesome/Font-Awesome/wiki/Upgrading-from-3.2.1-to-4 to upgrade your code accordingly if you are using custom themes.

Deprecations:

- The API versions 1 and 2 (also known as the REST API, ie `/api/rest/*` will removed in favour of the version 3 (action API, `/api/action/*`), which was introduced in CKAN 2.0. The REST API will be removed on CKAN 2.8.
- The default theme included in CKAN core will switch to use Bootstrap 3 instead of Bootstrap 2 in CKAN 2.8. The current Bootstrap 2 based templates will still be included in the next CKAN versions, so existing themes will still work. Bootstrap 2 templates will be eventually removed though, so instances are encouraged to update their themes using the available documentation (https://getbootstrap.com/migration/)
- The activity stream related actions ending with `*_list` (eg `package_activity_list`) and `*_html` (eg `package_activity_list_html`) will be removed in CKAN 2.8 in favour of more efficient alternatives and are now deprecated.
- The legacy revisions controller (ie `/revisions/*`) will be completely removed in CKAN 2.8.
- The old Celery based background jobs will be removed in CKAN 2.8 in favour of the new RQ based jobs (http://docs.ckan.org/en/latest/maintaining/background-tasks.html). Extensions can still of course use Celery but they will need to handle the management themselves.

8.44 v.2.6.9 2020-04-15

General notes:

- Note: This version does not requires a requirements upgrade on source installations
- Note: This version does not requires a database upgrade
- Note: This version does not require a Solr schema upgrade

Fixes:

- Fix for number of datasets displayed on the My organizations tab (#3580)
- Fix datetime comparison in resource_dict_save (#5033)
- Fetch less data for *get_all_entity_ids* (#5201)
- Show error in text view if xhr failed (#5271)
- Allow vocabulary_id in /api/2/util/tag/autocomplete (#5071)
- Fix code injection in autocomplete module (#5064)
- Fix broken translation in image view placeholder (#5099)
- Filter revisions shown according to dataset permissions
- Update JS vendor libraries

- Use returned facets in group controller (#2713)
- Samesite support in auth cookie (#5255)
- Handle missing resources in case we have a race condition with the DataPusher (#3980)
- Add the g object to toolkit

8.45 v.2.6.8 2019-07-03

General notes:

- Note: This version does not requires a requirements upgrade on source installations
- Note: This version does not requires a database upgrade
- Note: This version does not require a Solr schema upgrade

Fixes:

- Fix broken div nesting in the *user/read_base.html* (#4672)
- Strip local path when uploading file in IE (#4608)
- Increase size of h1 headings to 1.8em (#4665)
- Fix *ValueError* in *url_validator* (#4629)
- More robust auth functions for *resource_view_show* (#4827)
- Allow to customize the DataProxy URL (#4874)
- Allow custom CKAN callback URL for the DataPusher (#4878)

8.46 v2.6.7 2018-12-12

- Fix for Resouce View Re-order (#4416)
- autocomplete.js: fix handling of comma key codes (#4421)
- group_patch does not reset packages (#4557)

8.47 v2.6.6 2018-05-09

- Adding filter at resoruce preview doesn't work while site is setup with ckan.root_path param (#4140)
- Stable version URLs CKAN for documentation (#4209)
- Add Warning in docs sidebar (#4209)

8.48 v2.6.5 2018-03-15

Note: This version requires a database upgrade

- Activity Time stored in UTC (#2882)
- Migration script to adjust current activity timestamps to UTC
- Change CORS header keys and values to string instead of unicode (#3855)
- Fix cors header when all origins are allowed (#3898)
- Update SOLR schema.xml reference in Dockerfile
- Build local SOLR container by default
- Create datastore indexes only if they don't exist
- Properly close file responses
- Use javascript content-type for jsonp responses (#4022)
- Fix SOLR index delete_package implementation
- Add second half of DataStore set-permissions command (Docs)
- Return a 403 if not authorized on the search page (#4081)
- Add support for user/pass for Solr as ENV var
- Disallow solr local parameters
- Improve text view rendering
- Update Orgs/Groups logic for custom fields delete and update (#4094)

8.49 v2.6.4 2017-09-27

- Mail recepient header override (#3781)
- Skip url parsing in redirect (#3499)
- Support non root for fanstatic (#3618)

8.50 v2.6.3 2017-08-02

- Fix in organization / group form image URL field (#3661)
- Fix activity test to use utcnow (#3644)
- Changed required permission from 'update' to 'manage_group' (#3631)
- Catch invalid sort param exception (#3630)
- Choose direction of recreated package relationship depending on its type (#3626)
- Fix render_datetime for dates before year 1900 (#3611)
- Fix KeyError in 'package_create' (#3027)
- Allow slug preview to work with autocomplete fields (#2501)
- Fix filter results button not working for organization/group (#3620)

- Allow underscores in URL slug preview on create dataset (#3612)
- Create new resource view if resource format changed (#3515)
- Fixed escaping issues with *helpers.mail_to* and datapusher logs
- Autocomplete fields are more responsive - 300ms timeout instead of 1s (#3693)
- Fixed dataset count display for groups (#3711)
- Restrict access to form pages (#3684)

8.51 v2.6.2 2017-03-22

- Use fully qualified urls for reset emails (#3486)
- Fix edit_resource for resource with draft state (#3480)
- Tag fix for group/organization pages (#3460)
- Setting of datastore_active flag moved to separate function (#3481)

8.52 v2.6.1 2017-02-22

- Fix DataPusher being fired multiple times (#3245)
- Use the url_for() helper for datapusher URLs (#2866)
- Resource creation date use datetime.utcnow() (#3447)
- Fix locale error when using fix ckan.root_path
- *render_markdown* breaks links with ampersands
- Check group name and id during package creation
- Use utcnow() on dashboard_mark_activities_old (#3373)
- Fix encoding error on DataStore exception
- Datastore doesn't add site_url to resource created via API (#3189)
- Fix memberships after user deletion (#3265)
- Remove idle database connection (#3260)
- Fix package_owner_org_update action when called via the API (#2661)
- Fix French locale (#3327)
- Updated translations

8.53 v2.6.0 2016-11-02

Note: Starting from this version, CKAN requires at least Python 2.7 and Postgres 9.2

Note: This version requires a requirements upgrade on source installations

Note: This version requires a database upgrade

Note: This version does not require a Solr schema upgrade (You may want to upgrade the schema if you want to target Solr>=5, see #2914)

Major:

- Private datasets are now included in the default dataset search results (#3191)
- package_search API action now has an include_private parameter (#3191)

Minor:

- Make resource name default to file name (#1372)
- Customizable email templates (#1527)
- Change solrpy library to pysolr (#2352)
- Cache SQL query results (#2353)
- File Upload UX improvements (#2604)
- Helpers for multilingual fields (#2678)
- Improve Extension translation docs (#2783)
- Decouple configuration from Pylons (#3163)
- toolkit: add h, StopOnError, DefaultOrganizationForm (#2835)
- Remove Genshi support (#2833)
- Make resource URLs optional (#2844)
- Use 403 when actions are forbidden, not 401 (#2846)
- Upgrade requirements version (#3004, #3005)
- Add icons sources (#3048)
- Remove lib/dumper (#2879)
- ckan.__version__ available as template helper (#3103)
- Remove *site_url_nice* from app_globals (#3117)
- Remove *e.message* deprecation warning when running tests (#3121)
- Drop Python 2.6 support (#3126)
- Update Recline version (#3184)
- Refactor config/middleware.py to more closely match poc-flask-views (#3116)
- Creation of datasets sources with no organization specified (#3046)

Bug fixes:

- DataPusher called multiple times when creating a dataset (#2856)
- Default view is re-added when removed before DataStore upload is complete (#3011)
- "Data API" button disappears on resource page after empty update (#3012)

- Uncaught email exceptions on user invite (#3077)
- Resource view description is not rendered as Markdown (#3128)
- Fix broken html5lib dependency (#3180)
- ZH_cn translation formatter fix (#3238)
- Incorrect i18n-paths in extension's setup.cfg (#3275)
- Changing your user name produces an error and logs you out (#2394)
- Fix "Load more" functionality in the dashboard (#2346)
- Fix filters not working when embedding a resource view (#2657)
- Proper sanitation of header name on SlickGrid view (#2923)
- Fix unicode error when indexing field of type JSON (#2969)
- Fix group feeds returning no datasets (#2955)
- Replace MapQuest tiles in Recline with Stamen Terrain (#3162)
- Fix bulk operations not taking effect (#3199)
- Raise validation errors on group/org_member_create (#3108)
- Incorrect warnings when ckan.views.default_views is empty (#3093)
- Don't show deleted users/datasets on member_list (#3078)
- Fix Tag pagination widget styling (#2399)
- Fix package_owner_org_update standalone (#2661)
- Don't template fanstatic error pages (#2770)
- group_controller() on IGroupForm not in interface (#2771)
- Fix assert_true to test for message in response (#2802)
- Add user parameter to paster profile command (#2815)
- make context['user'] always username or None (#2817)
- remove some deprecated compatibility hacks (#2818)
- Param use_default_schema does not work on package_search (#2848)
- Sanitize offset when listing group activity (#2859)
- Incorrect 'download resource' hyperlink when a resource is unable to upload to datastore (#2873)
- Resolve datastore_delete erasing the database when filters was blank. (#2885)
- DomainObject.count() doesn't return count (#2919)
- Fix response code test failures (#2931)
- Fixed the url_for_* helpers when both SCRIPT_NAME and ckan.root_path are defined (#2936)
- Escape special characters in password while db loading (#2952)
- Fix redirect not working with non-root (#2968)
- Group pagination does not preserve sort order (#2981)
- Remove LazyJSONObject (#2983)
- Deleted users appear in sysadmin user lists (#2988)

- Server error at /organization if not authorized to list organizations (#2990)
- Slow page rendering when using lots of snippets (#3000)
- Only allow JSONP callbacks on GET requests (#3002)
- Attempting to access non-existing helpers should raise HelperException (#3041)
- Deprecate h.url, make it use h.url_for internally (#3055)
- Tests fail when LANG environment variable is set to German (#3060)
- Fix pagination style (CSS) (#3067)
- Login fails with 404 when using root_path (#3089)
- Resource view description is not rendered as Markdown (#3128)
- Clarify package_relationship_update documentation (#3132)
- *q* parameter in followee_list action has no effect (#3167)
- Zh cn translation formatter fix (#3238)
- Users are not removed in related tables if the main user entry is deleted (#3265)

API changes and deprecations:

- Replace *c.__version__* with new helper *h.ckan_version()* (#3103)

8.54 v2.5.9 2018-05-09

- Adding filter at resoruce preview doesn't work while site is setup with ckan.root_path param (#4140)
- Add Warning in docs sidebar (#4209)
- Point API docs to stable URL (#4209)

8.55 v2.5.8 2018-03-15

Note: This version requires a database upgrade

- Fix language switcher
- Activity Time stored in UTC (#2882)
- Migration script to adjust current activity timestamps to UTC
- Change CORS header keys and values to string instead of unicode (#3855)
- Fix cors header when all origins are allowed (#3898)
- Create datastore indexes only if they are not exist
- Use javascript content-type for jsonp responses (#4022)
- Fix SOLR index delete_package implementation
- Add second half of DataStore set-permissions command(Docs)
- Update SOLR client (pysolr -> solrpy)
- Return a 403 if not authorized on the search page (#4081)
- Add support for user/pass for Solr as ENV var

- Disallow solr local parameters
- Improve text view rendering
- Update Orgs/Groups logic for custom fields delete and update (#4094)

8.56 v2.5.7 2017-09-27

- Allow overriding email headers (#3781)
- Support non-root instances on fanstatic (#3618)
- Add missing close button on organization page (#3814)

8.57 v2.5.6 2017-08-02

- Fix in organization / group form image URL field (#3661)
- Fix activity test to use utcnow (#3644)
- Changed required permission from 'update' to 'manage_group' (#3631)
- Catch invalid sort param exception (#3630)
- Choose direction of recreated package relationship depending on its type (#3626)
- Fix render_datetime for dates before year 1900 (#3611)
- Fix KeyError in 'package_create' (#3027)
- Allow slug preview to work with autocomplete fields (#2501)
- Fix filter results button not working for organization/group (#3620)
- Allow underscores in URL slug preview on create dataset (#3612)
- Create new resource view if resource format changed (#3515)
- Fixed incorrect escaping in *mail_to* and datapusher's log
- Autocomplete fields are more responsive - 300ms timeout instead of 1s (#3693)
- Fixed dataset count display for groups (#3711)
- Restrict access to form pages (#3684)

8.58 v2.5.5 2017-03-22

- Use fully qualified urls for reset emails (#3486)
- Fix edit_resource for resource with draft state (#3480)
- Tag fix for group/organization pages (#3460)
- Setting of datastore_active flag moved to separate function (#3481)

8.59 v2.5.4 2017-02-22

- Fix DataPusher being fired multiple times (#3245)
- Use the url_for() helper for datapusher URLs (#2866)
- Resource creation date use datetime.utcnow() (#3447)
- Fix locale error when using fix ckan.root_path
- *render_markdown* breaks links with ampersands
- Check group name and id during package creation
- Use utcnow() on dashboard_mark_activities_old (#3373)
- Fix encoding error on DataStore exception
- Datastore doesn't add site_url to resource created via API (#3189)
- Fix memberships after user deletion (#3265)
- Remove idle database connection (#3260)
- Fix package_owner_org_update action when called via the API (#2661)

8.60 v2.5.3 2016-11-02

- DataPusher called multiple times when creating a dataset (#2856)
- Default view is re-added when removed before DataStore upload is complete (#3011)
- "Data API" button disappears on resource page after empty update (#3012)
- Uncaught email exceptions on user invite (#3077)
- Resource view description is not rendered as Markdown (#3128)
- Fix broken html5lib dependency (#3180)
- ZH_cn translation formatter fix (#3238)
- Incorrect i18n-paths in extension's setup.cfg (#3275)
- Changing your user name produces an error and logs you out (#2394)
- Fix "Load more" functionality in the dashboard (#2346)
- Fix filters not working when embedding a resource view (#2657)
- Proper sanitation of header name on SlickGrid view (#2923)
- Fix unicode error when indexing field of type JSON (#2969)
- Fix group feeds returning no datasets (#2955)
- Replace MapQuest tiles in Recline with Stamen Terrain (#3162)
- Fix bulk operations not taking effect (#3199)
- Raise validation errors on group/org_member_create (#3108)
- Incorrect warnings when ckan.views.default_views is empty (#3093)
- Don't show deleted users/datasets on member_list (#3078)

8.61 v2.5.2 2016-03-31

Bug fixes:

- Avoid submitting resources to the DataPusher multiple times (#2856)
- Use *resource.url* as raw_resource_url (#2873)
- Fix DomainObject.count() to return count (#2919)
- Prevent unicode/ascii conversion errors in DataStore
- Fix datastore_delete erasing the db when filters is blank (#2885)
- Avoid package_search exception when using use_default_schema (#2848)
- Encode EXPLAIN SQL before sending to datastore
- Use *ckan.site_url* to generate urls of resources (#2592)
- Fixed the url for the organization_item template

8.62 v2.5.1 2015-12-17

Note: This version requires a requirements upgrade on source installations

Note: This version requires a database upgrade

Note: This version does not require a Solr schema upgrade

Major:

- CKAN extension language translations integrated using ITranslations interface (#2461, #2643)
- Speed improvements for displaying a dataset (#2234), home page (#2554), searching (#2382, #2724) and API actions: package_show (#1078) and user_list (#2752).
- An interface to replace the file uploader, allowing integration with other cloud storage providers (IUploader interface) (#2510)

Minor:

- package_purge API action added (#1572)
- revision_list API action now has paging (#1431)
- Official Ubuntu 14.04 LTS support (#1651)
- Require/validate current password before allowing a password change (#1940)
- recline_map_view now recognizes GeoJSON fileds (#2387)
- Timezone setting (#2494)
- Updating a resource via upload now saves the last_modified value in the resource (#2519)
- DataPusher can be customized using the new IDataPusher interface (#2571)
- Exporting and importing users, with their passwords (if sysadmin) (#2647)

Bug fixes:

- Fix to allow uppercase letters in local part of email when sending user invitations (#2415)
- License pick-list changes would cause old values in datasets to be overwritten when edited (#2472)

- Schema was being passed to package_create_default_resource_views (#2484)
- Arabic translation format string issue (#2493)
- Error when deleting organizations (#2512)
- When DataPusher had an error storing a resource in Data Store, the resource data page gave an error (#2518)
- Data preview failed when it comes from a server that gives 403 error from a HEAD request (#2530)
- 'paster views create' failed for non-default dataset types (#2532)
- DataPusher didn't work for TSV files (#2553)
- DataPusher failed sometimes due to 'type mismatch' (#2581)
- IGroupForm wasn't allowing new groups (of type 'group') to use group_form (#2617, #2640)
- group_purge left behind a Member if it has a parent group/org (#2631)
- organization_purge left orphaned datasets still with owner_id (#2632)
- Fix Markdown rendering issue
- Return default error page on fanstatic errors
- Prevent authentication when using API callbacks

8.62.1 Changes and deprecations

- The old RDF templates to output a dataset in RDF/XML or N3 format have been removed. These can be now enabled using the `dcat` plugin on *ckanext-dcat*:

  https://github.com/ckan/ckanext-dcat#rdf-dcat-endpoints

- The library used to render markdown has been changed to python-markdown. This introduces both `python-markdown` and `bleach` as dependencies, as `bleach` is used to clean any HTML provided to the markdown processor.
- This is the last version of CKAN to support Postgresql 8.x, 9.0 and 9.1. The next minor version of CKAN will require Postgresql 9.2 or later.

8.63 v2.5.0 2015-12-17

Cancelled release

8.64 v2.4.9 2017-09-27

- Allow overriding email headers (#3781)
- Support non-root instances on fanstatic (#3618)
- Add missing close button on organization page (#3814)

8.65 v2.4.8 2017-08-02

- Fix in organization / group form image URL field (#3661)
- Fix activity test to use utcnow (#3644)
- Changed required permission from 'update' to 'manage_group' (#3631)
- Catch invalid sort param exception (#3630)
- Choose direction of recreated package relationship depending on its type (#3626)
- Fix render_datetime for dates before year 1900 (#3611)
- Fix KeyError in 'package_create' (#3027)
- Allow slug preview to work with autocomplete fields (#2501)
- Fix filter results button not working for organization/group (#3620)
- Allow underscores in URL slug preview on create dataset (#3612)
- Create new resource view if resource format changed (#3515)
- Fixed incorrect escaping in *mail_to*
- Autocomplete fields are more responsive - 300ms timeout instead of 1s (#3693)
- Fixed dataset count display for groups (#3711)
- Restrict access to form pages (#3684)

8.66 v2.4.7 2017-03-22

- Use fully qualified urls for reset emails (#3486)
- Fix edit_resource for resource with draft state (#3480)
- Tag fix for group/organization pages (#3460)
- Fix for package_search context (#3489)

8.67 v2.4.6 2017-02-22

- Use the url_for() helper for datapusher URLs (#2866)
- Resource creation date use datetime.utcnow() (#3447)
- Fix locale error when using fix ckan.root_path
- *render_markdown* breaks links with ampersands
- Check group name and id during package creation
- Use utcnow() on dashboard_mark_activities_old (#3373)
- Fix encoding error on DataStore exception
- Datastore doesn't add site_url to resource created via API (#3189)
- Fix memberships after user deletion (#3265)
- Remove idle database connection (#3260)

- Fix package_owner_org_update action when called via the API (#2661)

8.68 v2.4.5 2017-02-22

Cancelled release

8.69 v2.4.4 2016-11-02

- Changing your user name produces an error and logs you out (#2394)
- Fix "Load more" functionality in the dashboard (#2346)
- Fix filters not working when embedding a resource view (#2657)
- Proper sanitation of header name on SlickGrid view (#2923)
- Fix unicode error when indexing field of type JSON (#2969)
- Fix group feeds returning no datasets (#2955)
- Replace MapQuest tiles in Recline with Stamen Terrain (#3162)
- Fix bulk operations not taking effect (#3199)
- Raise validation errors on group/org_member_create (#3108)
- Incorrect warnings when ckan.views.default_views is empty (#3093)
- Don't show deleted users/datasets on member_list (#3078)

8.70 v2.4.3 2016-03-31

Bug fixes:

- Use *resource.url* as raw_resource_url (#2873)
- Fix DomainObject.count() to return count (#2919)
- Add offset param to organization_activity (#2640)
- Prevent unicode/ascii conversion errors in DataStore
- Fix datastore_delete erasing the db when filters is blank (#2885)
- Avoid package_search exception when using use_default_schema (#2848)
- resource_edit incorrectly setting action to new instead of edit
- Encode EXPLAIN SQL before sending to datastore
- Use *ckan.site_url* to generate urls of resources (#2592)
- Don't hide actual exception on paster commands

8.71 v2.4.2 2015-12-17

Note: This version requires a requirements upgrade on source installations

Bug fixes:

- Fix Markdown rendering issue
- Return default error page on fanstatic errors
- Prevent authentication when using API callbacks

8.72 v2.4.1 2015-09-02

Note: #2554 fixes a regression where `group_list` and `organization_list`
where returning extra additional fields by default, causing performance issues. This is now fixed, so the output for these actions no longer returns `users`, `extras`, etc. Also, on the homepage template the `c.groups` and `c.group_package_stuff` context variables are no longer available.

Bug fixes:

- Fix dataset count in templates and show datasets on featured org/group (#2557)
- Fix autodetect for TSV resources (#2553)
- Improve character escaping in DataStore parameters
- Fix "paster db init" when celery is configured with a non-database backend
- Fix severe performance issues with groups and orgs listings (#2554)

8.73 v2.4.0 2015-07-22

Note: This version requires a database upgrade

Note: This version requires a Solr schema upgrade

Major:

- CKAN config can now be set from environment variables and via the API (#2429)

Minor:

- API calls now faster: `group_show`, `organization_show`, `user_show`, `package_show`, `vocabulary_show` & `tag_show` (#1886, #2206, #2207, #2376)
- Require/validate current password before allowing a password change (#1940)
- Added `organization_autocomplete` action (#2125)
- Default authorization no longer allows anyone to create datasets etc (#2164)
- `organization_list_for_user` now returns organizations in hierarchy if they exist for roles set in `ckan.auth.roles_that_cascade_to_sub_groups` (#2199)
- Improved accessibility (text based browsers) focused on the page header (#2258)
- Improved IGroupForm for better customizing groups and organization behaviour (#2354)
- Admin page can now be extended to have new tabs (#2351)

Bug fixes:

- Command line `paster user` failed for non-ascii characters (#1244)
- Memory leak fixed in datastore API (#1847)
- Modifying resource didn't update it's last updated timestamp (#1874)
- Datastore didn't update if you uploaded a new file of the same name as the existing file (#2147)
- Files with really long file were skipped by datapusher (#2057)
- Multi-lingual Solr schema is now updated so it works again (#2161)
- Resource views didn't display when embedded in another site (#2238)
- `resource_update` failed if you supplied a revision_id (#2340)
- Recline could not plot GeoJSON on a map (#2387)
- Dataset create form 404 error if you added a resource but left it blank (#2392)
- Editing a resource view for a file that was UTF-8 and had a BOM gave an error (#2401)
- Email invites had the email address changed to lower-case (#2415)
- Default resource views not created when using a custom dataset schema (#2421, #2482)
- If the licenses pick-list was customized to remove some, datasets with old values had them overwritten when edited (#2472)
- Recline views failed on some non-ascii characters (#2490)
- Resource proxy failed if HEAD responds with 403 (#2530)
- Resource views for non-default dataset types couldn't be created (#2532)

8.73.1 Changes and deprecations

- The default of allowing anyone to create datasets, groups and organizations has been changed to False. It is advised to ensure you set all of the *Authorization Settings* options explicitly in your CKAN config. (#2164)
- The `package_show` API call does not return the `tracking_summary`, keys in the dataset or resources by default any more.

  Any custom templates or users of this API call that use these values will need to pass: `include_tracking=True`.
- The legacy *tests* directory has moved to *tests/legacy*, the *new_tests* directory has moved to *tests* and the *new_authz.py* module has been renamed *authz.py*. Code that imports names from the old locations will continue to work in this release but will issue a deprecation warning. (#1753)
- `group_show` and `organization_show` API calls no longer return the datasets by default (#2206)

  Custom templates or users of this API call will need to pass `include_datasets=True` to include datasets in the response.
- The `vocabulary_show` and `tag_show` API calls no longer returns the `packages` key - i.e. datasets that use the vocabulary or tag. However `tag_show` now has an `include_datasets` option. (#1886)
- Config option `site_url` is now required - CKAN will not abort during start-up if it is not set. (#1976)

8.74 v2.3.5 2016-11-02

- Fix "Load more" functionality in the dashboard (#2346)
- Fix filters not working when embedding a resource view (#2657)
- Proper sanitation of header name on SlickGrid view (#2923)
- Fix unicode error when indexing field of type JSON (#2969)
- Fix group feeds returning no datasets (#2955)
- Replace MapQuest tiles in Recline with Stamen Terrain (#3162)
- Fix bulk operations not taking effect (#3199)
- Raise validation errors on group/org_member_create (#3108)
- Incorrect warnings when ckan.views.default_views is empty (#3093)
- Don't show deleted users/datasets on member_list (#3078)

8.75 v2.3.4 2016-03-31

Bug fixes:

- Use *resource.url* as raw_resource_url (#2873)
- Fix DomainObject.count() to return count (#2919)
- Prevent unicode/ascii conversion errors in DataStore
- Fix datastore_delete erasing the db when filters is blank (#2885)
- Avoid package_search exception when using use_default_schema (#2848)
- resource_edit incorrectly setting action to new instead of edit
- Use *ckan.site_url* to generate urls of resources (#2592)
- Don't hide actual exception on paster commands

8.76 v2.3.3 2015-12-17

Note: This version requires a requirements upgrade on source installations

Bug fixes:

- Fix Markdown rendering issue
- Return default error page on fanstatic errors
- Prevent authentication when using API callbacks

8.77 v2.3.2 2015-09-02

Bug fixes: * Fix autodetect for TSV resources (#2553) * Improve character escaping in DataStore parameters * Fix "paster db init" when celery is configured with a non-database backend

8.78 v2.3.1 2015-07-22

Bug fixes:

- Resource views won't display when embedded in another site (#2238)
- `resource_update` failed if you supplied a revision_id (#2340)
- Recline could not plot GeoJSON on a map (#2387)
- Dataset create form 404 error if you added a resource but left it blank (#2392)
- Editing a resource view for a file that was UTF-8 and had a BOM gave an error (#2401)
- Email invites had the email address changed to lower-case (#2415)
- Default resource views not created when using a custom dataset schema (#2421, #2482)
- If the licenses pick-list was customized to remove some, datasets with old values had them overwritten when edited (#2472)
- Recline views failed on some non-ascii characters (#2490)
- Resource views for non-default dataset types couldn't be created (#2532)

8.79 v2.3 2015-03-04

Note: This version requires a requirements upgrade on source installations

Note: This version requires a database upgrade

Note: This version requires a Solr schema upgrade

Note: This version requires a DataPusher upgrade on source installations. You should target DataPusher=>0.0.6 and upgrade its dependencies.

Major:

- Completely refactored resource data visualizations, allowing multiple persistent views of the same data an interface to manage and configure them. (#1251, #1851, #1852, #2204, #2205) Check the updated documentation to know more, and the "Changes and deprecations" section for migration details:

  http://docs.ckan.org/en/latest/maintaining/data-viewer.html
- Responsive design for the default theme, that allows nicer rendering across different devices (#1935)
- Improved DataStore filtering and full text search capabilities (#1792, #1830, #1838, #1815)
- Added new extension points to modify the DataStore behaviour (#1725)
- Simplified two-step dataset creation process (#1659)
- Ability for users to regenerate their own API keys (#1412)
- New `package_patch` action to allow individual fields dataset updates (#1416, #1679)

- Changes on the authentication mechanism to allow more secure setups (`httponly` and `secure` cookies, disable CORS, etc). (#2004. #2050, #2052 ...) See "Changes and deprecations" section for more details and "Troubleshooting" for migration instructions.
- Better support for custom dataset types (#1795, #2083)
- Extensions can combine free-form extras and `convert_to_extras` fields (#1894)
- Updated documentation theme, now clearer and responsive (#1845)

Minor:

- Adding custom fields tutorial (#790)
- Add metadata created and modified fields to the dataset page (#655)
- Improve IFacets plugin interface docstrings (#781)
- Remove help string from API calls (#1318)
- Add "datapusher submit" command to upload existing resources data (#1792)
- More template blocks to allow for easier extension maintenance (#1301)
- CKAN API - remove help string from standard calls (#1318)
- Hide activity by selected users on activity stream (#1330)
- Documentation and clarification about "CKAN Flavored Markdown" (#1332)
- Resource formats are now guessed automatically (#1350)
- New JavaScript modules tutorial (#1377)
- Allow overriding dataset, group, org validation (#1400)
- Remove ResourceGroups, show package_id on resources (#1407)
- Better errors for NAVL junk (#1418)
- DataPusher integration improvements (#1446)
- Allow people to create unowned datasets when they belong to an org (#1473)
- Add res_type to Solr schema (#1495)
- Separate data and metadata licenses on create dataset page (#1503)
- Allow CKAN (and paster) to find config from envvar (#1597)
- Added xlsx and tsv to the defaults for ckan.datapusher.formats. (#1644)
- Add resource extras to Solr search index (#1709)
- Prevent packages update in organization_update (#1711)
- Programatically log user in after registration (#1721)
- New plugin interfaces: IValidators.get_validators and IConverters.get_converters (#1841)
- Index resource name in Solr (#1905)
- Update search index after membership changes (#1917)
- resource_show: use package_show to get validated data (#1921)
- Serve placeholder images locally (#1951)
- Don't get all datasets when loading the org in the dataset page (#1978)
- Text file preview - lack of vertical scroll bar for long files (#1982)

- Changes to allow better use of custom group types in IGroupForm extensions (#1987)
- Remove moderated edits (#2006)
- package_create: allow sysadmins to set package ids (#2102)
- Enable a logged in user to move dataset to another organization (#2218)
- Move PDF views into a separate extension (#2270)
- Do not provide email configuration in default config file (#2273)
- Add custom DataStore SQLAlchemy properties (#2279)

Bug fixes:

- Set up stats extension as namespace plugin (#291)
- Fix visibility validator for datasets (#1188)
- Select boxes with autocomplete are clearing their placeholders (#1278)
- Default search ordering on organization home page is broken (#1368)
- related_list logic function throws a 503 without any parameters (#1384)
- Exception on group dictize due to 'with_capacity' on context (#1390)
- Wrong template on Add member page (#1392)
- Overflowing email address on user page (#1398)
- The reset password e-mail is using an incorrect translation string (#1409)
- You can't view a group when there is an IGroupForm (#1420)
- Disabling activity_streams borks editing groups and user (#1421)
- Use a more secure default for the repoze secret key (#1422)
- Duplicated Required Fields notice on Group form (#1426)
- UI language reset after account creation (#1429)
- num_followers and package_count not in default_group_schema (#1434)
- Fix extras deletion (#1449)
- Fix resource reordering (#1450)
- Datastore callback fails when browser url is different from site_url (#1451)
- sysadmins should not create datasets wihout org when config is set (#1453)
- Member Editing Fixes (#1454)
- Bulk editing broken on IE7 (#1455)
- Fix group deletion on IE7 (#1460)
- Organization ATOM feed is broken (#1463)
- Users can not delete a dataset that not belongs to an organization (#1471)
- Error during authorization in datapusher_hook (#1487)
- Wrong datapusher hook callback URL on non-root deployments (#1490)
- Wrong breadcrumbs on new dataset form and resource pages (#1491)
- Atom feed Content-Type returned as 'text/html' (#1504)

- Invite to organization causes Internal Server error (#1505)
- Dataset tags autocomplete doesn't work (#1512)
- Activity Stream from: Organization Error group not found (#1519)
- Improve password hashing algorithm (#1530)
- Can't download resources with geojson extension (#1534)
- All datasets for featured group/organization shown on home page (#1569)
- Able to list private datasets via the API (#1580)
- Don't lowercase the names of uploaded files (#1584)
- Show more facets only if there are more facts to show (#1612)
- resource_create should break when called without URL (#1641)
- Creating a DataStore resource with the package_id fails for a normal user (#1652)
- Fix package permission checks for create+update (#1664)
- bulk_process page for non-existent organization throws Exception (#1682)
- Catch NotFound error in resource_proxy (#1684)
- Fix int_validator (#1692)
- Current date indexed on empty "_date" fields (#1701)
- Possible to show a resource inside an arbitary dataset (#1707)
- Edit member page shows wrong fields (#1723)
- Insecure content warning when running Recline under SSL (#1729)
- Flash messages not displayed as part of page.html (#1743)
- package_show response includes solr rubbish when using ckan.cache_validated_datasets (#1764)
- "Add some resources" link shown to unauthorized users (#1766)
- email notifications via paster plugin post erroneously demands authentication (#1767)
- Inserting empty arrays in JSON type fields in datastore fails (#1776)
- Ordering a dataset listing loses the existing filters (#1791)
- Don't delete all cookies whose names start with "ckan" (#1793)
- Upgrade some major requirements (eg SQLAlchemy, Requests) (#1817, #1819)
- list of member roles disappears on add member page (#1873)
- Stats plugin should only show active datasets (#1936)
- Featured group on homepage not linking to group (#1996)
- –reload doesn't work on the 'paster serve' command (#2013)
- Can not override auth config options from tests (#2035)
- Fix `resource_create` authorization (#2037)
- package_search gives internal server error if page < 1 (#2042)
- Fix organization pagination (#2141)
- Resource extras can not be updated (#2158)

- package_show doesn't validate when a custom schema is used (#2175)
- Update jQuery minified version to match the unminified one (#1750)
- Fix exception during database upgrade (#2029)
- Fix resources disappearing on dataset upate (#1779)
- Fix activity stream queries performance on large instances (#2008)
- Only link to http, https and ftp resource urls (#2085)
- Avoid private and deleted datasets on stats plugin (#1936)
- Fix tags count and group links in stats extension (#1649)
- Make resource_create auth work against package_update (#2037)
- Fix DataStore permissions check on startup (#1374)
- Fix datastore docs link (#2044)
- Clean up field names before rendering the Recline table (#2319)
- Don't "normalize" resource URL in recline view (#2324)
- Don't assume resource format is there on text preview (#2320)
- And many, many more!

8.79.1 Changes and deprecations

- By convention, view plugin names now end with `_view` rather than `_preview` (eg `recline_view` rather than `recline_preview`). You will need to update them on the *ckan.plugins* setting.
- The way resource visualizations are created by default has changed. You might need to set the *ckan.views.default_views* configuration option and run a migration command on existing instances. Please refer to the migration guide for more details:

  http://docs.ckan.org/en/latest/maintaining/data-viewer.html#migrating-from-previous-ckan-versions
- The PDF Viewer extension has been moved to a separate extension: https://github.com/ckan/ckanext-pdfview. Please install it separately if you are using the `pdf_view` plugin (or the old `pdf_preview` one).
- The action API (v3) no longer returns the full help for the action on each request. It rather includes a link to a separate call to get the action help string.
- The `user_show` API call does not return the `datasets`, `num_followers` or `activity` keys by default any more.

  Any custom templates or users of this API call that use these values will need to specify parameters: `include_datasets` or `include_num_followers`.

  `activity` has been removed completely as it was actually a list of revisions, rather than the activity stream. If you want the actual activity stream for a user, call `user_activity_list` instead.
- The output of `resource_show` now contains a `package_id` key that links to the parent dataset.
- `helpers.get_action()` (or `h.get_action()` in templates) is deprecated.

  Since action functions raise exceptions and templates cannot catch exceptions, it's not a good idea to call action functions from templates.

  Instead, have your controller method call the action function and pass the result to your template using the `extra_vars` param of `render()`.

Alternatively you can wrap individual action functions in custom template helper functions that handle any exceptions appropriately, but this is likely to make your the logic in your templates more complex and templates are difficult to test and debug.

Note that logic.get_action() and toolkit.get_action() are *not* deprecated, core code and plugin code should still use `get_action()`.

- Cross-Origin Resource Sharing (CORS) support is no longer enabled by default. Previously, Access-Control-Allow-* response headers were added for all requests, with Access-Control-Allow-Origin set to the wildcard value `*`. To re-enable CORS, use the new `ckan.cors` configuration settings (*ckan.cors.origin_allow_all* and *ckan.cors.origin_whitelist*).
- The HttpOnly flag will be set on the authorization cookie by default. For enhanced security, we recommend using the HttpOnly flag, but this behaviour can be changed in the `Repoze.who` settings detailed in the Config File Options documentation (*who.httponly*).
- The OpenID login option has been removed and is no longer supported. See "Troubleshooting" if you are upgrading an existing CKAN instance as you may need to update your `who.ini` file.

8.79.2 Template changes

- Note to people with custom themes: If you've changed the `{% block secondary_content %}` in templates/package/search.html pay close attention as this pull request changes the structure of that template block a little.

  Also: There's a few more bootstrap classes (especially for grid layout) that are now going to be in the templates. Take a look if any of the following changes might effect your content blocks:

  https://github.com/ckan/ckan/pull/1935

8.79.3 Troubleshooting:

- Login does not work, for existing and new users.

  You need to update your existing `who.ini` file.

  - In the `[plugin:auth_tkt]` section, replace:

    ```
    use = ckan.config.middleware:ckan_auth_tkt_make_app
    ```

    with:

    ```
    use = ckan.lib.auth_tkt:make_plugin
    ```

  - In `[authenticators]`, add the `auth_tkt` plugin

  Also see the next point for OpenID related changes.

- Exception on first load after upgrading from a previous CKAN version:

  ```
  ImportError: <module 'ckan.lib.authenticator' from '/usr/lib/ckan/default/src/ckan/
  ↪ckan/lib/authenticator.py'> has no 'OpenIDAuthenticator' attribute
  ```

  or:

  ```
  ImportError: No module named openid
  ```

There are OpenID related configuration options in your `who.ini` file which are no longer supported.

This file is generally located in `/etc/ckan/default/who.ini` but its location may vary if you used a custom deployment.

The options that you need to remove are:

- The whole `[plugin:openid]` section
- In `[general]`, replace:

```
challenge_decider = repoze.who.plugins.openid.classifiers:openid_challenge_
↪decider
```

  with:

```
challenge_decider = repoze.who.classifiers:default_challenge_decider
```

- In `[identifiers]`, remove `openid`
- In `[authenticators]`, remove `ckan.lib.authenticator:OpenIDAuthenticator`
- In `[challengers]`, remove `openid`

This is a diff with the whole changes:

https://github.com/ckan/ckan/pull/2058/files#diff-2

Also see the previous point for other `who.ini` changes.

8.80 v2.2.4 2015-12-17

Note: This version requires a requirements upgrade on source installations

Bug fixes:

- Fix Markdown rendering issue
- Return default error page on fanstatic errors
- Prevent authentication when using API callbacks

8.81 v2.2.3 2015-07-22

Bug fixes:

- Allow uppercase emails on user invites (#2415)
- Fix broken boolean validator (#2443)
- Fix auth check in resources_list.html (#2037)
- Key error on resource proxy (#2425)
- Ignore revision_id passed to resources (#2340)
- Add reset for reset_key on successful password change (#2379)

8.82 v2.2.2 2015-03-04

Bug fixes:

- Update jQuery minified version to match the unminified one (#1750)
- Fix exception during database upgrade (#2029)
- Fix resources disappearing on dataset upate (#1779)
- Fix activity stream queries performance on large instances (#2008)
- Only link to http, https and ftp resource urls (#2085)
- Avoid private and deleted datasets on stats plugin (#1936)
- Fix tags count and group links in stats extension (#1649)
- Make resource_create auth work against package_update (#2037)
- Fix DataStore permissions check on startup (#1374)
- Fix datastore docs link (#2044)
- Fix resource extras getting lost on resource update (#2158)
- Clean up field names before rendering the Recline table (#2319)
- Don't "normalize" resource URL in recline view (#2324)
- Don't assume resource format is there on text preview (#2320)

8.83 v2.2.1 2014-10-15

Bug fixes:

- Organization image_url is not displayed in the dataset view. (#1934)
- list of member roles disappears on add member page if you enter a user that doesn't exist (#1873)
- group/organization_member_create do not return a value. (#1878)
- i18n: Close a tag in French translation in Markdown syntax link (#1919)
- organization_list_for_user() fixes (#1918)
- Don't show private datasets to group members (#1902)
- Incorrect link in Organization snippet on dataset page (#1882)
- Prevent reading system tables on DataStore SQL search (#1871)
- Ensure that the DataStore is running on legacy mode when using PostgreSQL < 9.x (#1879)
- Select2 in the Tags field is broken(#1864)
- Edit user encoding error (#1436)
- Able to list private datasets via the API (#1580)
- Insecure content warning when running Recline under SSL (#1729)
- Add quotes to package ID in Solr query in _bulk_update_dataset to prevent Solr errors with custom dataset IDs. (#1853)
- Ordering a dataset listing loses the existing filters (#1791)

- Inserting empty arrays in JSON type fields in datastore fails (#1776)
- email notifications via paster plugin post erroneously demands authentication (#1767)
- "Add some resources" link shown to unauthorized users (#1766)
- Current date indexed on empty "*_date" fields (#1701)
- Edit member page shows wrong fields (#1723)
- programatically log user in after registration (#1721)
- Dataset tags autocomplete doesn't work (#1512)
- Deleted Users bug (#1668)
- UX problem with previous and next during dataset creation (#1598)
- Catch NotFound error in resources page (#1685)
- _tracking page should only respond to POST (#1683)
- bulk_process page for non-existent organization throws Exception (#1682)
- Fix package permission checks for create+update (#1664)
- Creating a DataStore resource with the package_id fails for a normal user (#1652)
- Trailing whitespace in resource URLs not stripped (#1634)
- Move the closing div inside the block (#1620)
- Fix open redirect (#1419)
- Show more facets only if there are more facts to show (#1612)
- Fix breakage in package groups page (#1594)
- Fix broken links in RSS feed (#1589)
- Activity Stream from: Organization Error group not found (#1519)
- DataPusher and harvester collision (#1500)
- Can't download resources with geojson extension (#1534)
- Oversized Forgot Password button and field (#1508)
- Invite to organization causes Internal Server error (#1505)

8.84 v2.2 2014-02-04

Note: This version does not require a requirements upgrade on source installations

Note: This version requires a database upgrade

Note: This version requires a Solr schema upgrade (The Solr schema file has been renamed, the schema file from the previous release is compatible with this version, but users are encouraged to point to the new one, see "API changes and deprecations")

Major:

- Brand new automatic importer of tabular data to the DataStore, the DataPusher. This is much more robust and simple to deploy and maintain than its predecesor (ckanext-datastorer). Whole new UI for re-importing data to the DataStore and view the import logs (#932, #938, #940, #981, #1196, #1200 . . .)

- Completely revamped file uploads that allow closer integration with resources and the DataStore, as well as making easir to integrate file uploads in other features. For example users can now upload images for organizations and groups. See "API changes and deprecations" if you are using the current FileStore. (#1273, #1173 …)
- UI and API endpoints for resource reordering (#1277)
- Backend support for organization hierarchy, allowing parent and children organizations. Frontend needs to be implemented in extensions (#1038)
- User invitations: it is now possible to create new users with just their email address. An invite email is sent to them, allowing to change their user name and password (#1178)
- Disable user registration with a configuration option (#1226)
- Great effort in improving documentation, specially for customizing CKAN, with a complete tutorial for writing extensions and customizing the theme. User and sysadmin guides have also been moved to the main documentation (#943, #847, #1253)

Minor:

- Homepage modules to allow predefined layouts (#1126)
- Ability to delete users (#1163)
- Dedicated dataset groups page for displaying and managing them (#1102)
- Implement organization_purge and group_purge action functions (#707)
- Improve package_show performance (#1078)
- Support internationalization of rendered dates and times (#1041)
- Improve plugin load handling (#549)
- Authorization function auditing for action functions (#1060)
- Improve datetime rendering (#518)
- New SQL indexes to improve performance (#1164)
- Changes in requirements management (#1149)
- Add offset/limit to package_list action (#1179)
- Document all available configuraton options (#848)
- Make CKAN sqlalchemy 0.8.4 compatible (#1427)
- UI labelling and cleanup (#1030)
- Better UX for empty groups/orgs (#1094)
- Improve performance of group_dictize when the group has a lot of packages (#1208)
- Hide __extras from extras on package_show (#1218)
- "Clear all" link within each facet block is unnecessary (#1263)
- Term translations of organizations (#1274)
- '–reset-db' option for when running tests (#1304)

Bug fixes:

- Fix plugins load/unload issues (#547)
- Improve performance when new_activities not needed (#1013)

- Resource preview breaks when CSV headers include percent sign (#1067)
- Package index not rebuilt when resources deleted (#1081)
- Don't accept invalid URLs in resource proxy (#1106)
- UI language reset after account creation (#1429)
- Catch non-integer facet limits (#1118)
- Error when deleting custom tags (#1114)
- Organization images do not display on Organization user dashboard page (#1127)
- Can not reactivate a deleted dataset from the UI (#607)
- Non-existent user profile should give error (#1068)
- Recaptcha not working in CKAN 2.0 (jinja templates) (#1070)
- Groups and organizations can be visited with interchangeable URLs (#1180)
- Dataset Source (url) and Version fields missing (#1187)
- Fix problems with private / public datasets and organizations (#1188)
- group_show should never return private data (#1191)
- When editing a dataset, the organization field is not set (#1199)
- Fix resource_delete action (#1216)
- Fix trash purge action redirect broken for CKAN instances not at / (#1217)
- Title edit for existing dataset changes the URL (#1232)
- 'facet.limit' in package_search wrongly handled (#1237)
- h.SI_number_span doesn't close correctly (#1238)
- CkanVersionException wrongly raised (#1241)
- (group|organization)_member_create only accepts username (and not id) (#1243)
- package_create uses the wrong parameter for organization (#1257)
- ValueError for non-int limit and offset query params (#1258)
- Visibility field value not kept if there are errors on the form (#1265)
- package_list should not return private datasets (#1295)
- Fix 404 on organization activity stream and about page (#1298)
- Fix placeholder images broken on non-root locations (#1309)
- "Add Dataset" button shown on org pages when not authorized (#1348)
- Fix exception when visiting organization history page (#1359)
- Fix search ordering on organization home page (#1368)
- datastore_search_sql failing for some anonymous users (#1373)
- related_list logic function throws a 503 without any parameters (#1384)
- Disabling activity_streams borks editing groups and user (#1421)
- Member Editing Fixes (#1454)
- Bulk editing broken in IE7 (#1455)

- Fix group deletion in IE7 (#1460)
- And many, many more!

API changes and deprecations:

- The Solr schema file is now always named `schema.xml` regardless of the CKAN version. Old schema files have been kept for backwards compatibility but users are encouraged to point to the new unified one (#1314)
- The FileStore and file uploads have been completely refactored and simplified to only support local storage backend. The links from previous versions of the FileStore to hosted files will still work, but there is a command available to migrate the files to new Filestore. See this page for more details: http://docs.ckan.org/en/latest/filestore.html#filestore-21-to-22-migration
- By default, the authorization for any action defined from an extension will require a logged in user, otherwise a `ckan.logic.NotAuthorized` exception will be raised. If an action function allows anonymous access (eg search, show status, etc) the `auth_allow_anonymous_access` decorator (available on the plugins toolkit) must be used (#1210)
- `package_search` now returns results with custom schemas applied like `package_show`, a `use_default_schema` parameter was added to request the old behaviour, this change may affect customized search result templates (#1255)
- The `ckan.api_url` configuration option has been completely removed and it can no longer be used (#960)
- The `edit` and `after_update` methods of IPackageController plugins are now called when updating a resource using the web frontend or the resource_update API action (#1052)
- Dataset moderation has been deprecated, and the code will probably be removed in later CKAN versions (#1139)
- Some front end libraries have been updated, this may affect existing custom themes: Bootstrap 2.0.3 > 2.3.2, Font Awesome 3.0.2 > 3.2.1, jQuery 1.7.2 > 1.10.2 (#1082)
- SQLite is officially no longer supported as the tests backend

Troubleshooting:

- Exception on startup after upgrading from a previous CKAN version:

```
AttributeError: 'instancemethod' object has no attribute 'auth_audit_exempt'
```

  Make sure that you are not loading a 2.1-only plugin (eg `datapusher-ext`) and update all the plugin in your configuration file to the latest stable version.

- Exception on startup after upgrading from a previous CKAN version:

```
File "/usr/lib/ckan/default/src/ckan/ckan/lib/dictization/model_dictize.py",␣
↪line 330, in package_dictize
    result_dict['metadata_modified'] = pkg.metadata_modified.isoformat()
AttributeError: 'NoneType' object has no attribute 'isoformat'
```

  One of the database changes on this version is the addition of a `metadata_modified` field in the package table, that was filled during the DB migration process. If you have previously migrated the database and revert to an older CKAN version the migration process may have failed at this step, leaving the fields empty. Also make sure to restart running processes like harvesters after the update to make sure they use the new code base.

8.85 v2.1.6 2015-12-17

Note: This version requires a requirements upgrade on source installations

Bug fixes:

- Fix Markdown rendering issue
- Return default error page on fanstatic errors
- Prevent authentication when using API callbacks

8.86 v2.1.5 2015-07-22

Bug fixes:

- Fix broken boolean validator (#2443)
- Key error on resource proxy (#2425)
- Ignore revision_id passed to resources (#2340)
- Add reset for reset_key on successful password change (#2379)

8.87 v2.1.4 2015-03-04

Bug fixes:

- Only link to http, https and ftp resource urls (#2085)
- Avoid private and deleted datasets on stats plugin (#1936)
- Fix tags count and group links in stats extension (#1649)
- Make resource_create auth work against package_update (#2037)
- Fix DataStore permissions check on startup (#1374)
- Fix datastore docs link (#2044)
- Fix resource extras getting lost on resource update (#2158)
- Clean up field names before rendering the Recline table (#2319)
- Don't "normalize" resource URL in recline view (#2324)
- Don't assume resource format is there on text preview (#2320)

8.88 v2.1.3 2014-10-15

Bug fixes:

- Organization image_url is not displayed in the dataset view. (#1934)
- i18n: Close a tag in French translation in Markdown syntax link (#1919)
- organization_list_for_user() fixes (#1918)
- Incorrect link in Organization snippet on dataset page (#1882)

- Prevent reading system tables on DataStore SQL search (#1871)
- Ensure that the DataStore is running on legacy mode when using PostgreSQL < 9.x (#1879)
- Edit user encoding error (#1436)
- Able to list private datasets via the API (#1580)
- Insecure content warning when running Recline under SSL (#1729)
- Add quotes to package ID in Solr query in _bulk_update_dataset to prevent Solr errors with custom dataset IDs. (#1853)
- Ordering a dataset listing loses the existing filters (#1791)
- Inserting empty arrays in JSON type fields in datastore fails (#1776)
- programatically log user in after registration (#1721)
- Deleted Users bug (#1668)
- Catch NotFound error in resources page (#1685)
- bulk_process page for non-existent organization throws Exception (#1682)
- Default search ordering on organization home page is broken (#1368)
- Term translations of organizations (#1274)
- Preview fails on private datastore resources (#1221)
- Strip whitespace from title in model dictize (#1228)

8.89 v2.1.2 2014-02-04

Bug fixes:

- Fix context for group/about setup_template_variables (#1433)
- Call setup_template_variables in group/org read, about and bulk_process (#1281)
- Remove repeated sort code in package_search (#1461)
- Ensure that check_access is called on activity_create (#1421)
- Fix visibility validator (#1188)
- Remove p.toolkit.auth_allow_anonymous_access as it is not available on 2.1.x (#1373)
- Add organization_revision_list to avoid exception on org history page (#1359)
- Fix activity and about organization pages (#1298)
- Show 404 instead of login page on user not found (#1068)
- Don't show Add Dataset button on org pages unless authorized (#1348)
- Fix datastore_search_sql authorization function (#1373)
- Fix extras deletion (#1449)
- Better word breaking on long words (#1398)
- Fix activity and about organization pages (#1298)
- Remove limit of number of arguments passed to `user add` command.
- Fix related_list logic function (#1384)

- Avoid UnicodeEncodeError on feeds when params contains non ascii characters

8.90 v2.1.1 2013-11-8

Bug fixes:

- Fix errors on preview on non-root locations (#960)
- Fix place-holder images on non-root locations (#1309)
- Don't accept invalid URLs in resource proxy (#1106)
- Make sure came_from url is local (#1039)
- Fix logout redirect in non-root locations (#1025)
- Wrong auth checks for sysadmins on package_create (#1184)
- Don't return private datasets on package_list (#1295)
- Stop tracking failing when no lang/encoding headers (#1192)
- Fix for paster db clean command getting frozen
- Fix organization not set when editing a dataset (#1199)
- Fix PDF previews (#1194)
- Fix preview failing on private datastore resources (#1221)

8.91 v2.1 2013-08-13

Note: This version requires a requirements upgrade on source installations

Note: This version requires a database upgrade

Note: This version does not require a Solr schema upgrade

Note: The `json_preview` plugin has been renamed to `text_preview` (see #266). If you are upgrading CKAN from a previous version you need to change the plugin name on your CKAN config file after upgrading to avoid a PluginNotFound exception.

Major:

- Bulk updates of datasets within organizations (delete, make public/private) (#278)
- Organizations and Groups search (#303)
- Generic text preview extension for JSON, XML and plain text files (#226)
- Improve consistency of the Action API (#473)
- IAuthenticator interface for plugging into authorization platforms (Work in progress) (#1007)
- New clearer dashboard with more information easier to access (#626)
- New `rebuild_fast` command to speed up reindex using multiple cores (#700)
- Complete restructure of the documentation, with updated sections on installation, upgrading, release process, etc and guidelines on how to write new documentation (#769 and multiple others)

Minor:

- Add group members page to templates (#844)
- Show search facets on organization page (#776)
- Changed default sort ordering (#869)
- More consistent display of buttons across pages (#890)
- History page ported to new templates (#368)
- More blocks to templates to allow furhter customization (#688)
- Improve imports from lib.helpers (#262)
- Add support for callback parameter on Action API (#414)
- Create site_user at startup (#952)
- Add warning before deleting an organization (#803)
- Remove flags from language selector (#822)
- Hide the Data API button when datastore is disabled (#752)
- Pin all requirements and separate minimal requirements in a separate file (#491, #1149)
- Better preview plugin selection (#1002)
- Add new functions to the plugins toolkit (#1015)
- Improve ExampleIDatasetFormPlugin (#2750)
- Extend h.sorted_extras() to do substitutions and auto clean keys (#440)
- Separate default database for development and testing (#517)
- More descriptive Solr exceptions when indexing (#674)
- Validate datastore input through schemas (#905)

Bug fixes:

- Fix 500 on password reset (#264)
- Fix exception when indexing a wrong date on a _date field (#267)
- Fix datastore permissions issues (#652)
- Placeholder images are not linked with h.url_for_static (#948)
- Explore dropdown menu is hidden behind other resources in IE (#915)
- Buttons interrupt file uploading (#902)
- Fix resource proxy encoding errors (#896)
- Enable streaming in resource proxy (#989)
- Fix cache_dir and beaker paths on deployment.ini_tmpl (#888)
- Fix multiple issues on create dataset form on IE (#881)
- Fix internal server error when adding member (#869)
- Fix license faceting (#853)
- Fix exception in dashboard (#830)
- Fix Google Analytics integration (#827)

- Fix ValueError when resource size is not an integer (#1009)
- Catch NotFound on new resource when package does not exist (#1010)
- Fix Celery configuration to allow overriding from config (#1027)
- came_from after login is validated to not redidirect to another site (#1039)
- And many, many more!

Deprecated and removed:

- The `json_preview` plugin has been replaced by a new `text_preview` one. Please update your config files if using it. (#226)

Known issues:

- Under certain authorization setups the frontend for the groups functionality may not work as expected (See #1176 #1175).

8.92 v2.0.8 2015-12-17

Note: This version requires a requirements upgrade on source installations

Bug fixes:

- Fix Markdown rendering issue
- Return default error page on fanstatic errors
- Prevent authentication when using API callbacks

8.93 v2.0.7 2015-07-22

Bug fixes:

- Fix broken boolean validator (#2443)
- Key error on resource proxy (#2425)
- Ignore revision_id passed to resources (#2340)
- Add reset for reset_key on successful password change (#2379)

8.94 v2.0.6 2015-03-04

Bug fixes:

- Only link to http, https and ftp resource urls (#2085)
- Avoid private and deleted datasets on stats plugin (#1936)
- Fix tags count and group links in stats extension (#1649)
- Make resource_create auth work against package_update (#2037)
- Fix datastore docs link (#2044)
- Fix resource extras getting lost on resource update (#2158)

- Clean up field names before rendering the Recline table (#2319)
- Don't "normalize" resource URL in recline view (#2324)
- Don't assume resource format is there on text preview (#2320)

8.95 v2.0.5 2014-10-15

Bug fixes:

- organization_list_for_user() fixes (#1918)
- Incorrect link in Organization snippet on dataset page (#1882)
- Prevent reading system tables on DataStore SQL search (#1871)
- Ensure that the DataStore is running on legacy mode when using PostgreSQL < 9.x (#1879)
- Current date indexed on empty "*_date" fields (#1701)
- Able to list private datasets via the API (#1580)
- Insecure content warning when running Recline under SSL (#1729)
- Inserting empty arrays in JSON type fields in datastore fails (#1776)
- Deleted Users bug (#1668)

8.96 v2.0.4 2014-02-04

Bug fixes:

- Fix extras deletion (#1449)
- Better word breaking on long words (#1398)
- Fix activity and about organization pages (#1298)
- Show 404 instead of login page on user not found (#1068)
- Remove limit of number of arguments passed to `user add` command.
- Fix related_list logic function (#1384)

8.97 v2.0.3 2013-11-8

Bug fixes:

- Fix errors on preview on non-root locations (#960)
- Don't accept invalid URLs in resource proxy (#1106)
- Make sure came_from url is local (#1039)
- Fix logout redirect in non-root locations (#1025)
- Don't return private datasets on package_list (#1295)
- Stop tracking failing when no lang/encoding headers (#1192)

- Fix for paster db clean command getting frozen

8.98 v2.0.2 2013-08-13

Bug fixes:

- Fix markdown in group descriptions (#303)
- Fix resource proxy encoding errors (#896)
- Fix datastore exception on first run (#907)
- Enable streaming in resource proxy (#989)
- Fix in user search (#1024)
- Fix Celery configuration to allow overriding from config (#1027)
- Undefined function on organizations controller (#1036)
- Fix license not translated in orgs/groups (#1040)
- Fix link to documentation from the footer (#1062)
- Fix missing close breadcrumb tag in org templates (#1071)
- Fix recently_changed_packages_activity_stream function (#1159)
- Fix Recline map sidebar not showing in IE 7-8 (#1133)

8.99 v2.0.1 2013-06-11

Bug fixes:

- Use IDatasetForm schema for resource_update (#897)
- Fixes for CKAN being run on a non-root URL (#948, #913)
- Fix resource edit errors losing info (#580)
- Fix Czech translation (#900)
- Allow JSON filters for datastore_search on GET requests (#917)
- Install vdm from the Python Package Index (#764)
- Allow extra parameters on Solr queries (#739)
- Create site user at startup if it does not exist (#952)
- Fix modal popups positioning (#828)
- Fix wrong redirect on dataset form on IE (#963)

8.100 v2.0 2013-05-10

Note: Starting on v2.0, issue numbers with four digits refer to the old ticketing system at http://trac.ckan.org and the ones with three digits refer to GitHub issues. For example:

- #3020 is http://trac.ckan.org/ticket/3020
- #271 is https://github.com/ckan/ckan/issues/271

Some GitHub issues URLs will redirect to GitHub pull request pages.

Note: v2.0 is a huge release so the changes listed here are just the highlights. Bug fixes are not listed.

Note: This version requires a requirements upgrade on source installations

Note: This version requires a database upgrade

Note: This version requires a Solr schema upgrade

Organizations based authorization (see *Organizations and authorization*):

CKAN's new "organizations" feature replaces the old authorization system with a new one based on publisher organizations. It replaces the "Publisher Profile and Workflow" feature from CKAN 1.X, any instances relying on it will need to be updated.

- New organization-based authorization and organization of datasets
- Supports private datasets
- Publisher workflow
- New authorization ini file options

New frontend (see *Theming guide*):

CKAN's frontend has been completely redesigned, inside and out. There is a new default theme and the template engine has moved from Genshi to Jinja2. Any custom templates using Genshi will need to be updated, although there is a `ckan.legacy_templates` setting to aid in the migration.

- Block-based template inheritance
- Custom jinja tags: {% ckan_extends %}, {% snippet %} and {% url_for %} (#2502, #2503)
- CSS "primer" page for theme developers
- We're now using LESS for CSS
- Scalable font icons (#2563)
- Social sharing buttons (google plus, facebook, twitter) (this replaces the ckanext-social extension)
- Three-stage dataset creation form (#2501)
- New *paster front-end-build* command does everything needed to build the frontend for a production CKAN site (runs *paster less* to compile the css files, *paster minify* to minify the css and js files, etc.)

Plugins & Extensions:

- New plugins toolkit provides a stable set of utility and helper functions for CKAN plugins to depend on.
- The IDatasetForm plugin interface has been redesigned (note: this breaks backwards-compatibility with existing IDatasetForm plugins) (#649)
- Many IDatasetForm bugs were fixed

- New example extensions in core, and better documentation for the relevant plugin interfaces: example_itemplatehelpers (#447), example_idatasetform (#2750), hopefully more to come in 2.1!
- New IFacets interface that allows to modify the facets shown on various pages. (#400)
- The get_action() function now automatically adds 'model' and 'session' to the context dict (this saves on boiler-plate code, and means plugins don't have to import ckan.model in order to call get_action()) (#172)

Activity Streams, Following & User Dashboard:

- New visual design for activity streams (#2941)
- Group activity streams now include activities for changes to any of the group's datasets (#1664)
- Group activity streams now appear on group pages (previously they could only be retrieved via the api)
- Dataset activity streams now appear on dataset pages (previously they could only be retrieved via the api) (#3024)
- Users can now follow groups (previously you could only follow users or datasets) (#3005)
- Activity streams and following are also supported for organizations (#505)
- When you're logged into CKAN, you now get a notifications count in the top-right corner of the site, telling you how many new notifications you have on your dashboard. Clicking on the count takes you to your dashboard page to view your notifications. (#3009)
- Optionally, you can also receive notifications by email when you have new activities on your dashboard (#1635)
- Infinite scrolling of activity streams (if you scroll to the bottom of a an activity stream, CKAN will automatically load more activities) (#3018)
- Redesigned user dashboard (#3028):
  - New dropdown-menu enables you to filter you dashboard activity stream to show only activities from a particular user, dataset, group or organization that you're following
  - New sidebar shows previews and unfollow buttons (when the activity stream is filtered)
- New *ckan.activity_streams_enabled* config file setting allows you to disable the generation of activity streams (#654)

Data Preview:

- PDF files preview (#2203)
- JSON files preview
- HTML pages preview (in an iframe) (#2888)
- New plugin extension point that allows plugins to add custom data previews for different data types (#2961)
- Improved Recline Data Explorer previews (CSV files, Excel files..)
- Plain text files preview

API:

- The Action API is now CKAN's default API, and the API documentation has been rewritten (#357)

Other highlights:

- CKAN now has continuous integration testing at https://travis-ci.org/ckan/ckan/
- Dataset pages now have <link rel="alternate" type="application/rdf+xml" links in the HTML headers, allows linked-data tools to find CKAN's RDF rendering of a dataset's metadata (#413)

- CKAN's DataStore and Data API have been rewritten, and now use PostgreSQL instead of elasticsearch, so there's no need to install elasticsearch anymore (this feature was also back-ported to CKAN 1.8) (#2733)
- New Config page for sysadmins (/ckan-admin/config) enables sysadmins to set the site title, tag line, logo, the intro text shown on the front page, the about text shown on the /about page, select a theme, and add custom CSS (#2302, #2781)
- New *paster color* command for creating color schemes
- Fanstatic integration (#2371):
  - CKAN now uses Fanstatic to specify required static resource files (js, css..) for web pages
  - Enables each page to only include the static files that it needs, reducing page loads
  - Enables CKAN to use bundled and minified static files, further reducing page loads
  - CKAN's new *paster minify* command is used to create minified js and css files (#2950) (also see *paster front-end-build*)
- CKAN will now recognise common file format strings such as "application/json", "JSON", ".json" and "json" as a single file type "json" (#2416)
- CKAN now supports internalization of strings in javascript files, the new *paster trans* command is used to pull translatable strings out of javascript files (#2774, #2750)
- convert_to/from_extras have been fixed to not add quotes around strings (#2930)
- Updated CKAN coding standards (#3020) and CONTRIBUTING.rst file
- Built-in page view counting and 'popular' badges on datasets and resources There's also a paster command to export the tracking data to a csv file (#195)
- Updated CKAN Coding Standards and new CONTRIBUTING.rst file
- You can now change the sort ordering of datasets on the dataset search page

Deprecated and removed:

- The IGenshiStreamFilter plugin interface is deprecated (#271), use the ITemplateHelpers plugin interface instead
- The Model, Search and Util APIs are deprecated, use the Action API instead
- Removed restrict_template_vars config setting (#2257)
- Removed deprecated facet_title() template helper function, use get_facet_title() instead (#2257)
- Removed deprecated am_authorized() template helper function, use check_access() instead (#2257)
- Removed deprecated datetime_to_datestr() template helper function (#2257)

8.101 v1.8.2 2013-08-13

Bug fixes:

- Fix for using harvesters with organization setup
- Refactor for user update logic
- Tweak resources visibility query

8.102 v1.8.1 2013-05-10

Bug fixes:

- Fixed possible XSS vulnerability on html input (#703)
- Fix unicode template 500 error (#808)
- Fix error on related controller

8.103 v1.8 2012-10-19

Note: This version requires a requirements upgrade on source installations

Note: This version requires a database upgrade

Note: This version does not require a Solr schema upgrade

Major

- New 'follow' feature that allows logged in users to follow other users or datasets (#2304)
- New user dashboard that shows an activity stream of all the datasets and users you are following. Thanks to Sven R. Kunze for his work on this (#2305)
- New version of the Datastore. It has been completely rewritten to use PostgreSQL as backend, it is more stable and fast and supports SQL queries (#2733)
- Clean up and simplifyng of CKAN's dependencies and source install instructions. Ubuntu 12.04 is now supported for source installs (#2428,#2592)
- Big speed improvements when indexing datasets (#2788)
- New action API reference docs, which individually document each function and its arguments and return values (#2345)
- Updated translations, added Japanese and Korean translations

Minor

- Add source install upgrade docs (#2757)
- Mark more strings for translation (#2770)
- Allow sort ordering of dataset listings on group pages (#2842)
- Reenable simple search option (#2844)
- Editing organization removes all datasets (#2845)
- Accessibility enhancements on templates

Bug fixes

- Fix for relative url being used when doing file upload to local storage
- Various fixes on IGroupFrom (#2750)
- Fix group dataset sort (#2722)
- Fix adding existing datasets to organizations (#2843)
- Fix 500 error in related controller (#2856)
- Fix for non-open licenses appearing open

- Editing organization removes all datasets (#2845)

API changes and deprecation:

- Template helper functions are now restricted by default. By default only those helper functions listed in lib.helpers.__allowed_functions__ are available to templates. The full functions can still be made available by setting *ckan.restrict_template_vars = false* in your ini file. Only restricted functions will be allowed in future versions of CKAN.
- Deprecated functions related to the old faceting data structure have been removed: *helpers.py:facet_items()*, *facets.html:facet_sidebar()*, *facets.html:facet_list_items()*. Internal use of the old facets datastructure (attached to the context, *c.facets*) has been superseded by use of the improved facet data structure, *c.search_facets*. The old data structure is still available on *c.facets*, but is deprecated, and will be removed in future versions. (#2313)

8.104 v1.7.4 2013-08-13

Bug fixes:

- Refactor for user update logic
- Tweak resources visibility query

8.105 v1.7.3 2013-05-10

Bug fixes:

- Fixed possible XSS vulnerability on html input (#703)

8.106 v1.7.2 2012-10-19

Minor:

- Documentation enhancements regarding file uploads

Bug fixes:

- Fixes for lincences i18n
- Remove sensitive data from user dict (#2784)
- Fix bug in feeds controller (#2869)
- Show dataset author and maintainer names even if they have no emails
- Fix URLs for some Amazon buckets
- Other minor fixes

8.107 v1.7.1 2012-06-20

Minor:

- Documentation enhancements regarding install and extensions (#2505)
- Home page and search results speed improvements (#2402,#2403)
- I18n: Added Greek translation and updated other ones (#2506)

Bug fixes:

- UI fixes (#2507)
- Fixes for i18n login and logout issues (#2497)
- Date on add/edit resource breaks if offset is specified (#2383)
- Fix in organizations read page (#2509)
- Add synchronous_search plugin to deployment.ini template (#2521)
- Inconsistent language on license dropdown (#2575)
- Fix bug in translating lists in multilingual plugin
- Group autocomplete doesn't work with multiple words (#2373)
- Other minor fixes

8.108 v1.7 2012-05-09

Major:

- Updated SOLR schema (#2327). Note: This will require and update of the SOLR schema file and a reindex.
- Support for Organization based workflow, with membership determinig access permissions to datasets (#1669,#2255)
- Related items such as visualizations, applications or ideas can now be added to datasets (#2204)
- Restricted vocabularies for tags, allowing grouping related tags together (#1698)
- Internal analytics that track number of views and downloads for datasets and resources (#2251)
- Consolidated multilingual features in an included extension (#1821,#1820)
- Atom feeds for publishers, tags and search results (#1593,#2277)
- RDF dump paster command (#2303)
- Better integration with the DataStore, based on ElasticSearch, with nice helper docs (#1797)
- Updated the Recline data viewer with new features such as better graphs and a map view (#2236,#2283)
- Improved and redesigned documentation (#2226,#2245,#2248)

Minor:

- Groups can have an image associated (#2275)
- Basic resource validation (#1711)
- Ability to search without accents for accented words (#906)
- Weight queries so that title is more important than rest of body (#1826)

- Enhancements in the dataset and resource forms (#1506)
- OpenID can now be disabled (#1830)
- API and forms use same validation (#1792)
- More robust bulk search indexing, with options to ignore exceptions and just refresh (#1616i,#2232)
- Modify where the language code is placed in URLs (#2261)
- Simplified licenses list (#1359)
- Add extension point for dataset view (#1741)

Bug fixes:

- Catch exceptions on the QA archiver (#1809)
- Error when changing language when CKAN is mounted in URL (#1804)
- Naming of a new package/group can clash with a route (#1742)
- Can't delete all of a package's resources over REST API (#2266)
- Group edit form didn't allow adding multiple datasets at once (#2292)
- Fix layout bugs in IE 7 (#1788)
- Bug with Portugese translation and Javascript (#2318)
- Fix broken parse_rfc_2822 helper function (#2314)

8.109 v1.6 2012-02-24

Major:

- Resources now have their own pages, as well as showing in the Dataset (#1445, #1449)
- Group pages enhanced, including in-group search (#1521)
- User pages enhanced with lists of datasets (#1396) and recent activity (#1515)
- Dataset view page decluttered (#1450)
- Tags not restricted to just letters and dashes (#1453)
- Stats Extension and Storage Extension moved into core CKAN (#1576, #1608)
- Ability to mounting CKAN at a sub-URL (#1401, #1659)
- 5 Stars of Openness ratings show by resources, if ckanext-qa is installed (#1583)
- Recline Data Explorer (for previewing and plotting data) improved and v2 moved into core CKAN (#1602, #1630)

Minor:

- 'About' page rewritten and easily customisable in the config (#1626)
- Gravatar picture displayed next to My Account link (#1528)
- 'Delete' button for datasets (#1425)
- Relationships API more RESTful, validated and documented (#1695)
- User name displayed when logged in (#1529)
- Database dumps now exclude deleted packages (#1623)

- Dataset/Tag name length now limited to 100 characters in API (#1473)
- 'Status' API call now includes installed extensions (#1488)
- Command-line interface for list/read/deleting datasets (#1499)
- Slug API calls tidied up and documented (#1500)
- Users nagged to add email address if missing from their account (#1413)
- Model and API for Users to become Members of a Group in a certain Capacity (#1531, #1477)
- Extension interface to adjust search queries, indexing and results (#1547, #1738)
- API for changing permissions (#1688)

Bug fixes:

- Group deletion didn't work (#1536)
- metadata_created used to return an entirely wrong date (#1546)
- Unicode characters in field-specific API search queries caused exception (since CKAN 1.5) (#1798)
- Sometimes task_status errors weren't being recorded (#1483)
- Registering or Logging in failed silently when already logged in (#1799)
- Deleted packages were browseable by administrators and appeared in dumps (#1283, #1623)
- Facicon was a broken link unless corrected in config file (#1627)
- Dataset search showed last result of each page out of order (#1683)
- 'Simple search' mode showed 0 packages on home page (#1709)
- Occasionally, 'My Account' shows when user is not logged in (#1513)
- Could not change language when on a tag page that had accented characters or dataset creation page (#1783, #1791)
- Editing package via API deleted its relationships (#1786)

8.110 v1.5.1 2012-01-04

Major:

- Background tasks (#1363, #1371, #1408)
- Fix for security issue affecting CKAN v1.5 (#1585)

Minor:

- Language support now excellent for European languages: en de fr it es no sv pl ru pt cs sr ca
- **Web UI improvements:**
  - Resource editing refreshed
  - Group editing refreshed
  - Indication that group creation requires logging-in (#1004)
  - Users' pictures displayed using Gravatar (#1409)
  - 'Welcome' banner shown to new users (#1378)
  - Group package list now ordered alphabetically (#1502)

- Allow managing a dataset's groups also via package entity API (#1381)
- Dataset listings in API standardised (#1490)
- Search ordering by modification and creation date (#191)
- Account creation disallowed with Open ID (create account in CKAN first) (#1386)
- User name can be modified (#1386)
- Email address required for registration (for password reset) (#1319)
- Atom feeds hidden for now
- New config options to ease CSS insertion into the template (#1380)
- Removed ETag browser cache headers (#1422)
- CKAN version number and admin contact in new 'status_show' API (#1087)
- Upgrade SQLAlchemy to 0.7.3 (compatible with Postgres up to 9.1) (#1433)
- SOLR schema is now versioned (#1498)

Bug fixes:

- Group ordering on main page was alphabetical but should be by size (since 1.5) (#1487)
- Package could get added multiple times to same Group, distorting Group size (#1484)
- Search index corruption when multiple CKAN instances on a server all storing the same object (#1430)
- Dataset property metadata_created had wrong value (since v1.3) (#1546)
- Tag browsing showed tags for deleted datasets (#920)
- User name change field validation error (#1470)
- You couldn't edit a user with a unicode email address (#1479)
- Package search API results missed the extra fields (#1455)
- OpenID registration disablement explained better (#1532)
- Data upload (with ckanext-storage) failed if spaces in the filename (#1518)
- Resource download count fixed (integration with ckanext-googleanalytics) (#1451)
- Multiple CKANs with same dataset IDs on the same SOLR core would conflict (#1462)

8.111 v1.5 2011-11-07

Deprecated due to security issue #1585

Major:

- **New visual theme (#1108)**
  - Package & Resource edit overhaul (#1294/#1348/#1351/#1368/#1296)
  - JS and CSS reorganization (#1282, #1349, #1380)
- Apache Solr used for search in core instead of Postgres (#1275, #1361, #1365)
- Authorization system now embedded in the logic layer (#1253)
- Captcha added for user registration (#1307, #1431)

- UI language translations refreshed (#1292, #1350, #1418)
- Action API improved with docs now (#1315, #1302, #1371)

Minor:

- Cross-Origin Resource Sharing (CORS) support (#1271)
- Strings to translate into other languages tidied up (#1249)
- Resource format autocomplete (#816)
- Database disconnection gives better error message (#1290)
- Log-in cookie is preserved between sessions (#78)
- Extensions can access formalchemy forms (#1301)
- 'Dataset' is the new name for 'Package' (#1293)
- Resource standard fields added: type, format, size (#1324)
- Listing users speeded up (#1268)
- Basic data preview functionality moved to core from QA extension (#1357)
- Admin Extension merged into core CKAN (#1264)
- URLs in the Notes field are automatically linked (#1320)
- Disallow OpenID for account creation (but can be linked to accounts) (#1386)
- Tag name now validated for max length (#1418)

Bug fixes:

- Purging of revisions didn't work (since 1.4.3) (#1258)
- Search indexing wasn't working for SOLR (since 1.4.3) (#1256)
- Configuration errors were being ignored (since always) (#1172)
- Flash messages were temporarily held-back when using proxy cache (since 1.3.2) (#1321)
- On login, user told 'welcome back' even if he's just registered (#1194)
- Various minor exceptions cropped up (mostly since 1.4.3) (#1334, #1346)
- Extra field couldn't be set to original value when key deleted (#1356)
- JSONP callback parameter didn't work for the Action API (since 1.4.3) (#1437)
- The same tag could be added to a package multiple times (#1331)

8.112 v1.4.3.1 2011-09-30

Minor:

- Added files to allow debian packaging of CKAN
- Added Catalan translation

Bug fixes:

- Incorrect Group creation form parameter caused exception (#1347)
- Incorrect AuthGroup creation form parameter caused exception (#1346)

8.113 v1.4.3 2011-09-13

Major:

- Action API (API v3) (beta version) provides powerful RPC-style API to CKAN data (#1335)
- Documentation overhaul (#1142, #1192)

Minor:

- Viewing of a package at a given date (as well as revision) with improved UI (#1236)
- Extensions can now add functions to the logic layer (#1211)
- Refactor all remaining database code out of the controllers and into the logic layer (#1229)
- Any OpenID log-in errors that occur are now displayed (#1228)
- 'url' field added to search index (e9214)
- Speed up tag reading (98d72)
- Cope with new WebOb version 1 (#1267)
- Avoid exceptions caused by bots hitting error page directly (#1176)
- Too minor to mention: #1234,

Bug fixes:

- Re-adding tags to a package failed (since 1.4.1 in Web UI, 1.4 in API) (#1239)
- Modified revisions retrieved over API caused exception (since 1.4.2) (#1310)
- Whichever language you changed to, it announced "Language set to: English" (since 1.3.1) (#1082)
- Incompatibilities with Python 2.5 (since 1.3.4.1 and maybe earlier) (#1325)
- You could create an authorization group without a name, causing exceptions displaying it (#1323)
- Revision list wasn't showing deleted packages (b21f4)
- User editing error conditions handled badly (#1265)

8.114 v1.4.2 2011-08-05

Major:

- Packages revisions can be marked as 'moderated' (#1141, #1147)
- Password reset facility (#1186/#1198)

Minor:

- Viewing of a package at any revision (#1236)
- API POSTs can be of Content-Type "application/json" as alternative to existing "application/x-www-form-urlencoded" (#1206)
- Caching of static files (#1223)

Bug fixes:

- When you removed last row of resource table, you could't add it again - since 1.0 (#1215)
- Adding a tag to package that had it previously didn't work - since 1.4.1 in UI and 1.4.0 in API (#1239)

- Search index was not updated if you added a package to a group - since 1.1 (#1140)
- Exception if you had any Groups and migrated between CKAN v1.0.2 to v1.2 (migration 29) - since v1.0.2 (#1205)
- API Package edit requests returned the Package in a different format to usual - since 1.4 (#1214)
- API error responses were not all JSON format and didn't have correct Content-Type (#1214)
- API package delete doesn't require a Content-Length header (#1214)

8.115 v1.4.1 2011-06-27

Major:

- Refactor Web interface to use logic layer rather than model objects directly. Forms now defined in navl schema and designed in HTML template. Forms use of Formalchemy is deprecated. (#1078)

Minor:

- Links in user-supplied text made less attractive to spammers (nofollow) #1181
- Package change notifications - remove duplicates (#1149)
- Metadata dump linked to (#1169)
- Refactor authorization code to be common across Package, Group and Authorization Group (#1074)

Bug fixes

- Duplicate authorization roles were difficult to delete (#1083)

8.116 v1.4 2011-05-19

Major:

- Authorization forms now in grid format (#1074)
- Links to RDF, N3 and Turtle metadata formats provided by semantic.ckan.net (#1088)
- Refactor internal logic to all use packages in one format - a dictionary (#1046)
- A new button for administrators to change revisions to/from a deleted state (#1076)

Minor:

- Etags caching can now be disabled in config (#840)
- Command-line tool to check search index covers all packages (#1073)
- Command-line tool to load/dump postgres database (#1067)

Bug fixes:

- Visitor can't create packages on new CKAN install - since v1.3.3 (#1090)
- OpenID user pages couldn't be accessed - since v1.3.2 (#1056)
- Default site_url configured to ckan.net, so pages obtains CSS from ckan.net- since v1.3 (#1085)

8.117 v1.3.3 2011-04-08

Major:

- Authorization checks added to editing Groups and PackageRelationships (#1052)
- API: Added package revision history (#1012, #1071)

Minor:

- API can take auth credentials from cookie (#1001)
- Theming: Ability to set custom favicon (#1051)
- Importer code moved out into ckanext-importlib repo (#1042)
- API: Group can be referred to by ID (in addition to name) (#1045)
- Command line tool: rights listing can now be filtered (#1072)

Bug fixes:

- SITE_READ role setting couldn't be overridden by sysadmins (#1044)
- Default 'reader' role too permissive (#1066)
- Resource ordering went wrong when editing and adding at same time (#1054)
- GET followed by PUTting a package stored an incorrect license value (#662)
- Sibling package relationships were shown for deleted packages (#664)
- Tags were displayed when they only apply to deleted packages (#920)
- API: 'Last modified' time was localised - now UTC (#1068)

8.118 v1.3.2 2011-03-15

Major:

- User list in the Web interface (#1010)
- CKAN packaged as .deb for install on Ubuntu
- Resources can have extra fields (although not in web interface yet) (#826)
- CSW Harvesting - numerous of fixes & improvements. Ready for deployment. (#738 etc)
- Language switcher (82002)

Minor:

- Wordpress integration refactored as a Middleware plugin (#1013)
- Unauthorized actions lead to a flash message (#366)
- Resources Groups to group Resources in Packages (#956)
- Plugin interface for authorization (#1011)
- Database migrations tested better and corrected (#805, #998)
- Government form moved out into ckanext-dgu repo (#1018)
- Command-line user authorization tools extended (#1038, #1026)

- Default user roles read from config file (#1039)

Bug fixes:

- Mounting of filesystem (affected versions since 1.0.1) (#1040)
- Resubmitting a package via the API (affected versions since 0.6?) (#662)
- Open redirect (affected v1.3) (#1026)

8.119 v1.3 2011-02-18

http://ckan.org/milestone/ckan-v1.3

Highlights of changes:

- **Package edit form improved:**
  - field instructions (#679)
  - name autofilled from title (#778)
- Group-based access control - Authorization Groups (#647)
- Metadata harvest job management (#739, #884, #771)
- CSW harvesting now uses owslib (#885)
- Package creation authorization is configurable (#648)
- Read-only maintenance mode (#777)
- Stats page (#832) and importer (#950) moved out into CKAN extensions

Minor:

- site_title and site_description config variables (#974)
- Package creation/edit timestamps (#806)
- Caching configuration centralised (#828)
- Command-line tools - sysadmin management (#782)
- Group now versioned (#231)

8.120 v1.2 2010-11-25

http://ckan.org/milestone/ckan-v1.2

Highlights of changes:

- Package edit form: attach package to groups (#652) & revealable help
- Form API - Package/Harvester Create/New (#545)
- Authorization extended: user groups (#647) and creation of packages (#648)
- Plug-in interface classes (#741)
- WordPress twentyten compatible theming (#797)
- Caching support (ETag) (#693)
- Harvesting GEMINI2 metadata records from OGC CSW servers (#566)

Minor:

- New API key header (#466)
- Group metadata now revisioned (#231)

8.121 v1.1 2010-08-10

http://ckan.org/milestone/v1.1

Highlights of changes:

- Changes to the database cause notifications via AMQP for clients (#325)
- Pluggable search engines (#317), including SOLR (#353)
- API is versioned and packages & groups can be referred to by invariant ID (#313)
- Resource search in API (#336)
- Visual theming of CKAN now easy (#340, #320)
- Greater integration with external Web UIs (#335, #347, #348)
- Plug-ins can be configured to handle web requests from specified URIs and insert HTML into pages.

Minor:

- Search engine optimisations e.g. alphabetical browsing (#350)
- CSV and JSON dumps improved (#315)

8.122 v1.0.2 2010-08-27

- Bugfix: API returns error when creating package (#432)

8.123 v1.0.1 2010-06-23

Functionality:

- API: Revision search 'since id' and revision model in API
- API: Basic API versioning - packages specified by ID (#313)
- Pluggable search - initial hooks
- Customisable templates (#340) and external UI hooks (#335)

Bugfixes:

- Revision primary key lost in migrating data (#311)
- Local authority license correction in migration (#319)
- I18n formatting issues
- Web i/f searches foreign characters (#319)
- Data importer timezone issue (#330)

8.124 v1.0 2010-05-11

CKAN comes of age, having been used successfully in several deployments around the world. 56 tickets covered in this release. See: http://ckan.org/milestone/v1.0

Highlights of changes:

- Package edit form: new pluggable architecture for custom forms (#281, #286)
- Package revisions: diffs now include tag, license and resource changes (#303)
- Web interface: visual overhaul (#182, #206, #214-#227, #260) including a tag cloud (#89)
- i18n: completion in Web UI - now covers package edit form (#248)
- API extended: revisions (#251, #265), feeds per package (#266)
- Developer documentation expanded (#289, #290)
- Performance improved and CKAN stress-tested (#201)
- Package relationships (Read-Write in API, Read-Only in Web UI) (#253-257)
- Statistics page (#184)
- Group edit: add multiple packages at once (#295)
- Package view: RDF and JSON formatted metadata linked to from package page (#247)

Bugfixes:

- Resources were losing their history (#292)
- Extra fields now work with spaces in the name (#278, #280) and international characters (#288)
- Updating resources in the REST API (#293)

Infrastructural:

- Licenses: now uses external License Service ('licenses' Python module)
- Changesets introduced to support distributed revisioning of CKAN data - see doc/distributed.rst for more information.

8.125 v0.11 2010-01-25

Our biggest release so far (55 tickets) with lots of new features and improvements. This release also saw a major new production deployment with the CKAN software powering http://data.gov.uk/ which had its public launch on Jan 21st!

For a full listing of tickets see: <http://ckan.org/milestone/v0.11>. Main highlights:

- Package Resource object (multiple download urls per package): each package can have multiple 'resources' (urls) with each resource having additional metadata such as format, description and hash (#88, #89, #229)
- "Full-text" searching of packages (#187)
- Semantic web integration: RDFization of all data plus integration with an online RDF store (e.g. for http://www.ckan.net/ at http://semantic.ckan.net/ or Talis store) (#90 #163)
- Package ratings (#77 #194)
- i18n: we now have translations into German and French with deployments at http://de.ckan.net/ and http://fr.ckan.net/ (#202)
- Package diffs available in package history (#173)

- Minor:
  - Package undelete (#21, #126)
  - Automated CKAN deployment via Fabric (#213)
  - Listings are sorted alphabetically (#195)
  - Add extras to rest api and to ckanclient (#158 #166)
- Infrastructural:
  - Change to UUIDs for revisions and all domain objects
  - Improved search performance and better pagination
  - Significantly improved performance in API and WUI via judicious caching

8.126 v0.10 2009-09-30

- Switch to repoze.who for authentication (#64)
- Explicit User object and improved user account UI with recent edits etc (#111, #66, #67)
- Generic Attributes for Packages (#43)
- Use sqlalchemy-migrate to handle db/model upgrades (#94)
- "Groups" of packages (#105, #110, #130, #121, #123, #131)
- Package search in the REST API (#108)
- Full role-based access control for Packages and Groups (#93, #116, #114, #115, #117, #122, #120)
- New CKAN logo (#72)
- Infrastructural:
  - Upgrade to Pylons 0.9.7 (#71)
  - Convert to use formalchemy for all forms (#76)
  - Use paginate in webhelpers (#118)
- Minor:
  - Add author and maintainer attributes to package (#91)
  - Change package state in the WUI (delete and undelete) (#126)
  - Ensure non-active packages don't show up (#119)
  - Change tags to contain any character (other than space) (#62)
  - Add Is It Open links to package pages (#74)

8.127 v0.9 2009-07-31

- (DM!) Add version attribute for package
- Fix purge to use new version of vdm (0.4)
- Link to changed packages when listing revision
- Show most recently registered or updated packages on front page
- Bookmarklet to enable easy package registration on CKAN
- Usability improvements (package search and creation on front page)
- Use external list of licenses from license repository
- Convert from py.test to nosetests

8.128 v0.8 2009-04-10

- View information about package history (ticket:53)
- Basic datapkg integration (ticket:57)
- Show information about package openness using icons (ticket:56)
- One-stage package create/registration (r437)
- Reinstate package attribute validation (r437)
- Upgrade to vdm 0.4

8.129 v0.7 2008-10-31

- Convert to use SQLAlchemy and vdm v0.3 (v. major)
- Atom/RSS feed for Recent Changes
- Package search via name and title
- Tag lists show number of associated packages

8.130 v0.6 2008-07-08

- Autocompletion (+ suggestion) of tags when adding tags to a package.
- Paginated lists for packages, tags, and revisions.
- RESTful machine API for package access, update, listing and creation.
- API Keys for users who wish to modify information via the REST API.
- Update to vdm v0.2 (SQLObject) which fixes ordering of lists.
- Better immunity to SQL injection attacks.

8.131 v0.5 2008-01-22

- Purging of a Revision and associated changes from cli and wui (ticket:37)
- Make data available in machine-usable form via sql dump (ticket:38)
- Upgrade to Pylons 0.9.6.* and deploy (ticket:41)
- List and search tags (ticket:33)
- (bugfix) Manage reserved html characters in urls (ticket:40)
- New spam management utilities including (partial) blacklist support

8.132 v0.4 2007-07-04

- Preview support when editing a package (ticket:36).
- Correctly list IP address of of not logged in users (ticket:35).
- Improve read action for revision to list details of changed items (r179).
- Sort out deployment using modpython.

8.133 v0.3 2007-04-12

- System now in a suitable state for production deployment as a beta
- Domain model versioning via the vdm package (currently released separately)
- Basic Recent Changes listing log messages
- User authentication (login/logout) via open ID
- License page
- Myriad of small fixes and improvements

8.134 v0.2 2007-02

- Complete rewrite of ckan to use pylons web framework
- Support for full CRUD on packages and tags
- No support for users (authentication)
- No versioning of domain model objects

8.135 v0.1 2006-05

NB: not an official release

- Almost functional system with support for persons, packages
- Tag support only half-functional (tags are per package not global)
- Limited release and file support

PYTHON MODULE INDEX

C

ckan.lib.helpers, 370
ckan.lib.navl.validators, 297
ckan.logic.action.create, 195
ckan.logic.action.delete, 212
ckan.logic.action.get, 176
ckan.logic.action.patch, 211
ckan.logic.action.update, 204
ckan.logic.converters, 305
ckan.logic.validators, 300
ckan.plugins.interfaces, 255
ckan.plugins.toolkit, 282
ckan.tests.controllers, 465
ckan.tests.factories, 446
ckan.tests.helpers, 449
ckan.tests.lib, 466
ckan.tests.logic.action, 462
ckan.tests.logic.auth, 462
ckan.tests.logic.test_schema, 465
ckan.tests.model, 466
ckan.tests.plugins, 466
ckan.tests.pytest_ckan.fixtures, 455
ckanext.datastore.backend, 90
ckanext.datastore.logic.action, 81

INDEX

Symbols

_()
built-in function, 369

A

abort() (*ckan.plugins.interfaces.IAuthenticator method*), 275
action_succeeded (*in module ckan.lib.signals*), 313
actions (*built-in class*), 369
add_extra_fields() (*ckan.plugins.interfaces.IApiToken method*), 279
add_url_param() (*in module ckan.lib.helpers*), 378
after_dataset_create() (*ckan.plugins.interfaces.IPackageController method*), 259
after_dataset_delete() (*ckan.plugins.interfaces.IPackageController method*), 259
after_dataset_search() (*ckan.plugins.interfaces.IPackageController method*), 260
after_dataset_show() (*ckan.plugins.interfaces.IPackageController method*), 259
after_dataset_update() (*ckan.plugins.interfaces.IPackageController method*), 259
after_load() (*ckan.plugins.interfaces.IPluginObserver method*), 260
after_resource_create() (*ckan.plugins.interfaces.IResourceController method*), 269
after_resource_delete() (*ckan.plugins.interfaces.IResourceController method*), 270
after_resource_update() (*ckan.plugins.interfaces.IResourceController method*), 269
after_unload() (*ckan.plugins.interfaces.IPluginObserver method*), 260
all_jobs() (*ckan.tests.helpers.RQTestBase method*), 452
am_following_dataset() (*in module ckan.logic.action.get*), 191
am_following_group() (*in module ckan.logic.action.get*), 191
am_following_user() (*in module ckan.logic.action.get*), 191
api_create() (*ckan.tests.factories.CKANFactory class method*), 448
api_token_create() (*in module ckan.logic.action.create*), 204
api_token_list() (*in module ckan.logic.action.get*), 194
api_token_revoke() (*in module ckan.logic.action.delete*), 216
APIToken (*class in ckan.tests.factories*), 449
APITokenFactory (*class in ckan.tests.pytest_ckan.fixtures*), 455
app() (*in module ckan.tests.pytest_ckan.fixtures*), 456
app_globals (*built-in variable*), 368
as_list() (*in module ckan.logic.converters*), 306
assert_log() (*ckan.tests.helpers.RecordingLogHandler method*), 454
authenticate() (*ckan.plugins.interfaces.IAuthenticator method*), 275

B

base (*ckan.plugins.toolkit.ckan.plugins.toolkit attribute*), 285
before_dataset_index() (*ckan.plugins.interfaces.IPackageController method*), 260
before_dataset_search() (*ckan.plugins.interfaces.IPackageController method*), 260
before_dataset_view() (*ckan.plugins.interfaces.IPackageController method*), 260
before_fork() (*ckan.plugins.interfaces.IForkObserver method*), 277
before_load() (*ckan.plugins.interfaces.IPluginObserver method*), 260
before_resource_create()

(*ckan.plugins.interfaces.IResourceController method*), 269
before_resource_delete() (*ckan.plugins.interfaces.IResourceController method*), 270
before_resource_show() (*ckan.plugins.interfaces.IResourceController method*), 270
before_resource_update() (*ckan.plugins.interfaces.IResourceController method*), 269
before_unload() (*ckan.plugins.interfaces.IPluginObserver method*), 260
before_view() (*ckan.plugins.interfaces.IGroupController method*), 258
before_view() (*ckan.plugins.interfaces.IOrganizationController method*), 259
before_view() (*ckan.plugins.interfaces.ITagController method*), 272
blanket (*ckan.plugins.toolkit.ckan.plugins.toolkit attribute*), 285
boolean_validator() (*in module ckan.logic.validators*), 300
both_not_empty() (*in module ckan.lib.navl.validators*), 298
build_nav() (*in module ckan.lib.helpers*), 374
build_nav_icon() (*in module ckan.lib.helpers*), 374
build_nav_main() (*in module ckan.lib.helpers*), 373
built-in function
    _(), 369
    N_(), 369
    ungettext(), 369
bulk_update_delete() (*in module ckan.logic.action.update*), 210
bulk_update_private() (*in module ckan.logic.action.update*), 210
bulk_update_public() (*in module ckan.logic.action.update*), 210

C

c (*built-in variable*), 368
c (*ckan.plugins.toolkit.ckan.plugins.toolkit attribute*), 287
call_action() (*in module ckan.tests.helpers*), 450
call_auth() (*in module ckan.tests.helpers*), 450
can_update_owner_org() (*in module ckan.lib.helpers*), 382
can_view() (*ckan.plugins.interfaces.IResourceView method*), 268
chained_helper() (*in module ckan.lib.helpers*), 371
change_config() (*in module ckan.tests.helpers*), 453
changed_config() (*in module ckan.tests.helpers*), 453
check_access() (*in module ckan.lib.helpers*), 376
check_ckan_version() (*in module ckan.lib.helpers*), 382
check_config_permission() (*in module ckan.lib.helpers*), 381
ckan (*ckan.plugins.toolkit.ckan.plugins.toolkit attribute*), 289
ckan.lib.helpers
    module, 370
ckan.lib.navl.validators
    module, 297
ckan.logic.action.create
    module, 195
ckan.logic.action.delete
    module, 212
ckan.logic.action.get
    module, 176
ckan.logic.action.patch
    module, 211
ckan.logic.action.update
    module, 204
ckan.logic.converters
    module, 305
ckan.logic.validators
    module, 300
ckan.plugins.interfaces
    module, 255
ckan.plugins.toolkit
    module, 282
ckan.plugins.toolkit._() (*in module ckan.plugins.toolkit*), 283
ckan.plugins.toolkit.abort() (*in module ckan.plugins.toolkit*), 283
ckan.plugins.toolkit.add_public_directory() (*in module ckan.plugins.toolkit*), 283
ckan.plugins.toolkit.add_resource() (*in module ckan.plugins.toolkit*), 284
ckan.plugins.toolkit.add_template_directory() (*in module ckan.plugins.toolkit*), 284
ckan.plugins.toolkit.asbool() (*in module ckan.plugins.toolkit*), 284
ckan.plugins.toolkit.asint() (*in module ckan.plugins.toolkit*), 284
ckan.plugins.toolkit.aslist() (*in module ckan.plugins.toolkit*), 284
ckan.plugins.toolkit.auth_allow_anonymous_access() (*in module ckan.plugins.toolkit*), 284
ckan.plugins.toolkit.auth_disallow_anonymous_access() (*in module ckan.plugins.toolkit*), 284
ckan.plugins.toolkit.auth_sysadmins_check() (*in module ckan.plugins.toolkit*), 284
ckan.plugins.toolkit.BaseModel (*class in ckan.plugins.toolkit*), 282
ckan.plugins.toolkit.chained_action() (*in module ckan.plugins.toolkit*), 287
ckan.plugins.toolkit.chained_auth_function() (*in module ckan.plugins.toolkit*), 287

ckan.plugins.toolkit.chained_helper() (*in module ckan.plugins.toolkit*), 288
ckan.plugins.toolkit.check_access() (*in module ckan.plugins.toolkit*), 288
ckan.plugins.toolkit.check_ckan_version() (*in module ckan.plugins.toolkit*), 289
ckan.plugins.toolkit.CkanVersionException (*class in ckan.plugins.toolkit*), 282
ckan.plugins.toolkit.DefaultDatasetForm (*class in ckan.plugins.toolkit*), 282
ckan.plugins.toolkit.DefaultGroupForm (*class in ckan.plugins.toolkit*), 282
ckan.plugins.toolkit.DefaultOrganizationForm (*class in ckan.plugins.toolkit*), 282
ckan.plugins.toolkit.enqueue_job() (*in module ckan.plugins.toolkit*), 289
ckan.plugins.toolkit.error_shout() (*in module ckan.plugins.toolkit*), 290
ckan.plugins.toolkit.fresh_context() (*in module ckan.plugins.toolkit*), 290
ckan.plugins.toolkit.get_action() (*in module ckan.plugins.toolkit*), 290
ckan.plugins.toolkit.get_converter() (*in module ckan.plugins.toolkit*), 291
ckan.plugins.toolkit.get_endpoint() (*in module ckan.plugins.toolkit*), 291
ckan.plugins.toolkit.get_or_bust() (*in module ckan.plugins.toolkit*), 291
ckan.plugins.toolkit.get_validator() (*in module ckan.plugins.toolkit*), 292
ckan.plugins.toolkit.HelperError (*class in ckan.plugins.toolkit*), 283
ckan.plugins.toolkit.Invalid (*class in ckan.plugins.toolkit*), 283
ckan.plugins.toolkit.literal (*class in ckan.plugins.toolkit*), 292
ckan.plugins.toolkit.login_user() (*in module ckan.plugins.toolkit*), 292
ckan.plugins.toolkit.logout_user() (*in module ckan.plugins.toolkit*), 292
ckan.plugins.toolkit.mail_recipient() (*in module ckan.plugins.toolkit*), 292
ckan.plugins.toolkit.mail_user() (*in module ckan.plugins.toolkit*), 293
ckan.plugins.toolkit.navl_validate() (*in module ckan.plugins.toolkit*), 293
ckan.plugins.toolkit.NotAuthorized (*class in ckan.plugins.toolkit*), 283
ckan.plugins.toolkit.ObjectNotFound (*class in ckan.plugins.toolkit*), 283
ckan.plugins.toolkit.redirect_to() (*in module ckan.plugins.toolkit*), 293
ckan.plugins.toolkit.render() (*in module ckan.plugins.toolkit*), 294
ckan.plugins.toolkit.render_snippet() (*in module ckan.plugins.toolkit*), 294
ckan.plugins.toolkit.requires_ckan_version() (*in module ckan.plugins.toolkit*), 294
ckan.plugins.toolkit.side_effect_free() (*in module ckan.plugins.toolkit*), 294
ckan.plugins.toolkit.StopOnError (*class in ckan.plugins.toolkit*), 283
ckan.plugins.toolkit.ungettext() (*in module ckan.plugins.toolkit*), 295
ckan.plugins.toolkit.UnknownValidator (*class in ckan.plugins.toolkit*), 283
ckan.plugins.toolkit.url_for() (*in module ckan.plugins.toolkit*), 295
ckan.plugins.toolkit.ValidationError (*class in ckan.plugins.toolkit*), 283
ckan.tests.controllers
    module, 465
ckan.tests.factories
    module, 446
ckan.tests.helpers
    module, 449
ckan.tests.lib
    module, 466
ckan.tests.logic.action
    module, 462
ckan.tests.logic.auth
    module, 462
ckan.tests.logic.test_schema
    module, 465
ckan.tests.model
    module, 466
ckan.tests.plugins
    module, 466
ckan.tests.pytest_ckan.fixtures
    module, 455
ckan_config() (*in module ckan.tests.pytest_ckan.fixtures*), 455
ckan_version() (*in module ckan.lib.helpers*), 373
CKANCliRunner (*class in ckan.tests.helpers*), 451
ckanext.datastore.backend
    module, 90
ckanext.datastore.logic.action
    module, 81
CKANFactory (*class in ckan.tests.factories*), 448
CKANOptions (*class in ckan.tests.factories*), 448
CKANResponse (*class in ckan.tests.helpers*), 451
CKANTestApp (*class in ckan.tests.helpers*), 451
CKANTestClient (*class in ckan.tests.helpers*), 451
clean_db() (*in module ckan.tests.pytest_ckan.fixtures*), 457
clean_format() (*in module ckan.logic.validators*), 304
clean_html() (*in module ckan.lib.helpers*), 382

clean_index() (*in module ckan.tests.pytest_ckan.fixtures*), 458
clean_redis() (*in module ckan.tests.pytest_ckan.fixtures*), 457
clear() (*ckan.tests.helpers.RecordingLogHandler method*), 454
cli() (*in module ckan.tests.pytest_ckan.fixtures*), 456
collect_prefix_validate() (*in module ckan.logic.validators*), 304
config (*ckan.plugins.toolkit.ckan.plugins.toolkit attribute*), 289
config_option_list() (*in module ckan.logic.action.get*), 194
config_option_show() (*in module ckan.logic.action.get*), 193
config_option_update() (*in module ckan.logic.action.update*), 210
configure() (*ckan.plugins.interfaces.IConfigurable method*), 260
configure() (*ckanext.datastore.backend.DatastoreBackend method*), 91
configured_default() (*in module ckan.lib.navl.validators*), 299
convert_from_extras() (*in module ckan.logic.converters*), 305
convert_from_tags() (*in module ckan.logic.converters*), 305
convert_group_name_or_id_to_id() (*in module ckan.logic.converters*), 306
convert_int() (*in module ckan.lib.navl.validators*), 299
convert_package_name_or_id_to_id() (*in module ckan.logic.converters*), 305
convert_to_dict() (*in module ckan.lib.helpers*), 378
convert_to_extras() (*in module ckan.logic.converters*), 305
convert_to_json_if_string() (*in module ckan.logic.converters*), 306
convert_to_list_if_string() (*in module ckan.logic.converters*), 306
convert_to_tags() (*in module ckan.logic.converters*), 305
convert_user_name_or_id_to_id() (*in module ckan.logic.converters*), 305
core_helper() (*in module ckan.lib.helpers*), 370
create() (*ckan.plugins.interfaces.IGroupController method*), 258
create() (*ckan.plugins.interfaces.IOrganizationController method*), 259
create() (*ckan.plugins.interfaces.IPackageController method*), 259
create() (*ckanext.datastore.backend.DatastoreBackend method*), 91
create_api_token_schema() (*ckan.plugins.interfaces.IApiToken method*), 277
create_function() (*ckanext.datastore.backend.DatastoreBackend method*), 93
create_package_schema() (*ckan.plugins.interfaces.IDatasetForm method*), 263
create_with_upload() (*in module ckan.tests.pytest_ckan.fixtures*), 459
csrf_input() (*in module ckan.lib.helpers*), 382
current_package_list_with_resources() (*in module ckan.logic.action.get*), 176
current_url() (*in module ckan.lib.helpers*), 373
current_user (*ckan.plugins.toolkit.ckan.plugins.toolkit attribute*), 289

D

Dataset (*class in ckan.tests.factories*), 448
dataset_display_name() (*in module ckan.lib.helpers*), 377
dataset_facets() (*ckan.plugins.interfaces.IFacets method*), 273
dataset_followee_count() (*in module ckan.logic.action.get*), 192
dataset_followee_list() (*in module ckan.logic.action.get*), 192
dataset_follower_count() (*in module ckan.logic.action.get*), 190
dataset_follower_list() (*in module ckan.logic.action.get*), 190
dataset_link() (*in module ckan.lib.helpers*), 377
dataset_purge() (*in module ckan.logic.action.delete*), 213
datasets_with_no_organization_cannot_be_private() (*in module ckan.logic.validators*), 303
datastore_create() (*in module ckanext.datastore.logic.action*), 81
datastore_delete (*in module ckan.lib.signals*), 314
datastore_delete() (*in module ckanext.datastore.logic.action*), 84
datastore_function_create() (*in module ckanext.datastore.logic.action*), 87
datastore_function_delete() (*in module ckanext.datastore.logic.action*), 87
datastore_info() (*in module ckanext.datastore.logic.action*), 83
datastore_records_delete() (*in module ckanext.datastore.logic.action*), 84
datastore_run_triggers() (*in module ckanext.datastore.logic.action*), 82
datastore_search() (*in module ckanext.datastore.logic.action*), 85

datastore_search_sql() (*in module ckanext.datastore.logic.action*), 86
datastore_upsert (*in module ckan.lib.signals*), 313
datastore_upsert() (*in module ckanext.datastore.logic.action*), 83
DatastoreBackend (*class in ckanext.datastore.backend*), 91
DatastoreException, 91
date_str_to_datetime() (*in module ckan.lib.helpers*), 377
datetime_from_timestamp_validator() (*in module ckan.logic.validators*), 300
db_to_form_schema() (*ckan.plugins.interfaces.IGroupForm method*), 271
debug_inspect() (*in module ckan.lib.helpers*), 379
declare_config_options() (*ckan.plugins.interfaces.IConfigDeclaration method*), 261
decode_api_token() (*ckan.plugins.interfaces.IApiToken method*), 278
decode_view_request_filters() (*in module ckan.lib.helpers*), 382
default() (*in module ckan.lib.navl.validators*), 299
default_group_type() (*in module ckan.lib.helpers*), 374
default_package_type() (*in module ckan.lib.helpers*), 374
delete() (*ckan.plugins.interfaces.IGroupController method*), 258
delete() (*ckan.plugins.interfaces.IOrganizationController method*), 259
delete() (*ckan.plugins.interfaces.IPackageController method*), 259
delete() (*ckanext.datastore.backend.DatastoreBackend method*), 92
dict_list_reduce() (*in module ckan.lib.helpers*), 376
dict_only() (*in module ckan.logic.validators*), 304
drop_function() (*ckanext.datastore.backend.DatastoreBackend method*), 93
dump_json() (*in module ckan.lib.helpers*), 378
duplicate_extras_key() (*in module ckan.logic.validators*), 302

E

edit() (*ckan.plugins.interfaces.IGroupController method*), 258
edit() (*ckan.plugins.interfaces.IOrganizationController method*), 259
edit() (*ckan.plugins.interfaces.IPackageController method*), 259
edit_template() (*ckan.plugins.interfaces.IDatasetForm method*), 265
edit_template() (*ckan.plugins.interfaces.IGroupForm method*), 271
email_is_unique() (*in module ckan.logic.validators*), 304
email_validator() (*in module ckan.logic.validators*), 304
emit() (*ckan.tests.helpers.RecordingLogHandler method*), 454
empty() (*in module ckan.lib.navl.validators*), 298
empty_if_not_sysadmin() (*in module ckan.logic.validators*), 304
encode_api_token() (*ckan.plugins.interfaces.IApiToken method*), 278
enqueue() (*ckan.tests.helpers.RQTestBase method*), 452
escape() (*ckan.lib.helpers.literal class method*), 370
escape_js() (*in module ckan.lib.helpers*), 379
extra_key_not_in_root_schema() (*in module ckan.logic.validators*), 304
extras_unicode_convert() (*in module ckan.logic.converters*), 305
extras_valid_json() (*in module ckan.logic.validators*), 305

F

facets() (*in module ckan.lib.helpers*), 381
failed_login (*in module ckan.lib.signals*), 313
FakeFileStorage (*class in ckan.tests.pytest_ckan.fixtures*), 459
FakeSMTP (*class in ckan.tests.helpers*), 454
featured_group_org() (*in module ckan.lib.helpers*), 381
filter_fields_and_values_exist_and_are_valid() (*in module ckan.logic.validators*), 304
filter_fields_and_values_should_have_same_length() (*in module ckan.logic.validators*), 304
flash_error() (*in module ckan.lib.helpers*), 373
flash_notice() (*in module ckan.lib.helpers*), 373
flash_success() (*in module ckan.lib.helpers*), 373
follow_button() (*in module ckan.lib.helpers*), 378
follow_count() (*in module ckan.lib.helpers*), 378
follow_dataset() (*in module ckan.logic.action.create*), 203
follow_group() (*in module ckan.logic.action.create*), 203
follow_user() (*in module ckan.logic.action.create*), 202
followee_count() (*in module ckan.logic.action.get*), 191
followee_list() (*in module ckan.logic.action.get*), 192
form_template() (*ckan.plugins.interfaces.IResourceView method*), 268
form_to_db_schema() (*ckan.plugins.interfaces.IGroupForm method*),

271
format_autocomplete() (*in module ckan.logic.action.get*), 184
format_resource_items() (*in module ckan.lib.helpers*), 380
free_tags_only() (*in module ckan.logic.converters*), 305
full_current_url() (*in module ckan.lib.helpers*), 373
FunctionalRQTestBase (*class in ckan.tests.helpers*), 452
FunctionalTestBase (*class in ckan.tests.helpers*), 452

G

g (*built-in variable*), 368
g (*ckan.plugins.toolkit.ckan.plugins.toolkit attribute*), 290
get_actions() (*ckan.plugins.interfaces.IActions method*), 262
get_active_backend() (*ckanext.datastore.backend.DatastoreBackend class method*), 91
get_all_ids() (*ckanext.datastore.backend.DatastoreBackend method*), 93
get_all_resources_ids_in_datastore() (*in module ckanext.datastore.backend*), 90
get_allowed_view_types() (*in module ckan.lib.helpers*), 380
get_auth_functions() (*ckan.plugins.interfaces.IAuthFunctions method*), 256
get_blueprint() (*ckan.plugins.interfaces.IBlueprint method*), 277
get_collaborators() (*in module ckan.lib.helpers*), 382
get_commands() (*ckan.plugins.interfaces.IClick method*), 279
get_dataset_labels() (*ckan.plugins.interfaces.IPermissionLabels method*), 277
get_display_timezone() (*in module ckan.lib.helpers*), 376
get_facet_items_dict() (*in module ckan.lib.helpers*), 375
get_featured_groups() (*in module ckan.lib.helpers*), 381
get_featured_organizations() (*in module ckan.lib.helpers*), 381
get_feed_class() (*ckan.plugins.interfaces.IFeed method*), 258
get_flashed_messages() (*in module ckan.lib.helpers*), 373
get_helpers() (*ckan.plugins.interfaces.ITemplateHelpers method*), 272
get_item_additional_fields() (*ckan.plugins.interfaces.IFeed method*), 258
get_organization() (*in module ckan.lib.helpers*), 381
get_page_number() (*in module ckan.lib.helpers*), 376
get_param_int() (*in module ckan.lib.helpers*), 375
get_pkg_dict_extra() (*in module ckan.lib.helpers*), 379
get_request_param() (*in module ckan.lib.helpers*), 380
get_resource_uploader() (*ckan.plugins.interfaces.IUploader method*), 276
get_rtl_theme() (*in module ckan.lib.helpers*), 373
get_signal_subscriptions() (*ckan.plugins.interfaces.ISignal method*), 280
get_site_protocol_and_host() (*in module ckan.lib.helpers*), 371
get_site_user() (*in module ckan.logic.action.get*), 189
get_translated() (*in module ckan.lib.helpers*), 381
get_uploader() (*ckan.plugins.interfaces.IUploader method*), 275
get_user_dataset_labels() (*ckan.plugins.interfaces.IPermissionLabels method*), 277
get_validators() (*ckan.plugins.interfaces.IValidators method*), 266
gravatar() (*in module ckan.lib.helpers*), 376
Group (*class in ckan.tests.factories*), 448
group_autocomplete() (*in module ckan.logic.action.get*), 184
group_controller() (*ckan.plugins.interfaces.IGroupForm method*), 271
group_create() (*in module ckan.logic.action.create*), 199
group_delete() (*in module ckan.logic.action.delete*), 214
group_facets() (*ckan.plugins.interfaces.IFacets method*), 273
group_followee_count() (*in module ckan.logic.action.get*), 192
group_followee_list() (*in module ckan.logic.action.get*), 193
group_follower_count() (*in module ckan.logic.action.get*), 190
group_follower_list() (*in module ckan.logic.action.get*), 190
group_form() (*ckan.plugins.interfaces.IGroupForm method*), 271
group_id_exists() (*in module ckan.logic.validators*), 301
group_id_or_name_exists() (*in module ckan.logic.validators*), 301
group_link() (*in module ckan.lib.helpers*), 378
group_list() (*in module ckan.logic.action.get*), 178

group_list_authz() (*in module ckan.logic.action.get*), 179
group_member_create() (*in module ckan.logic.action.create*), 203
group_member_delete() (*in module ckan.logic.action.delete*), 215
group_name_to_title() (*in module ckan.lib.helpers*), 376
group_name_validator() (*in module ckan.logic.validators*), 302
group_package_show() (*in module ckan.logic.action.get*), 183
group_patch() (*in module ckan.logic.action.patch*), 212
group_purge() (*in module ckan.logic.action.delete*), 214
group_show() (*in module ckan.logic.action.get*), 182
group_types() (*ckan.plugins.interfaces.IGroupForm method*), 271
group_update() (*in module ckan.logic.action.update*), 207
GroupFactory (*class in ckan.tests.pytest_ckan.fixtures*), 455
groups_available() (*in module ckan.lib.helpers*), 379

H

h (*built-in variable*), 368
h (*ckan.plugins.toolkit.ckan.plugins.toolkit attribute*), 292
has_more_facets() (*in module ckan.lib.helpers*), 375
help_show() (*in module ckan.logic.action.get*), 193
HelperAttributeDict (*class in ckan.lib.helpers*), 370
history_template() (*ckan.plugins.interfaces.IDatasetForm method*), 265
history_template() (*ckan.plugins.interfaces.IGroupForm method*), 271
html_auto_link() (*in module ckan.lib.helpers*), 380
humanize_entity_type() (*in module ckan.lib.helpers*), 374

I

i18n_directory() (*ckan.plugins.interfaces.ITranslation method*), 275
i18n_domain() (*ckan.plugins.interfaces.ITranslation method*), 275
i18n_locales() (*ckan.plugins.interfaces.ITranslation method*), 275
IActions (*class in ckan.plugins.interfaces*), 262
IApiToken (*class in ckan.plugins.interfaces*), 277
IAuthenticator (*class in ckan.plugins.interfaces*), 274
IAuthFunctions (*class in ckan.plugins.interfaces*), 256
IBlueprint (*class in ckan.plugins.interfaces*), 277
IClick (*class in ckan.plugins.interfaces*), 279
IConfigDeclaration (*class in ckan.plugins.interfaces*), 261
IConfigurable (*class in ckan.plugins.interfaces*), 260
IConfigurer (*class in ckan.plugins.interfaces*), 261
IDataDictionaryForm (*class in ckanext.datastore.interfaces*), 314
IDatasetForm (*class in ckan.plugins.interfaces*), 262
identify() (*ckan.plugins.interfaces.IAuthenticator method*), 275
IDomainObjectModification (*class in ckan.plugins.interfaces*), 257
if_empty_guess_format() (*in module ckan.logic.validators*), 304
if_empty_same_as() (*in module ckan.lib.navl.validators*), 298
IFacets (*class in ckan.plugins.interfaces*), 273
IFeed (*class in ckan.plugins.interfaces*), 257
IForkObserver (*class in ckan.plugins.interfaces*), 277
ignore() (*in module ckan.lib.navl.validators*), 298
ignore_empty() (*in module ckan.lib.navl.validators*), 299
ignore_missing() (*in module ckan.lib.navl.validators*), 299
ignore_not_group_admin() (*in module ckan.logic.validators*), 302
ignore_not_package_admin() (*in module ckan.logic.validators*), 302
ignore_not_sysadmin() (*in module ckan.logic.validators*), 302
IGroupController (*class in ckan.plugins.interfaces*), 258
IGroupForm (*class in ckan.plugins.interfaces*), 270
IMiddleware (*class in ckan.plugins.interfaces*), 255
implemented_by() (*ckan.plugins.interfaces.Interface class method*), 255
implements() (*in module ckan.plugins*), 255
index_template() (*ckan.plugins.interfaces.IGroupForm method*), 271
info() (*ckan.plugins.interfaces.IResourceView method*), 266
int_validator() (*in module ckan.logic.validators*), 300
Interface (*class in ckan.plugins.interfaces*), 255
invoke() (*ckan.tests.helpers.CKANCliRunner method*), 451
IOrganizationController (*class in ckan.plugins.interfaces*), 259
IPackageController (*class in ckan.plugins.interfaces*), 259
IPermissionLabels (*class in ckan.plugins.interfaces*), 277
IPluginObserver (*class in ckan.plugins.interfaces*), 260
IResourceController (*class in ckan.plugins.interfaces*), 269
IResourceUrlChange (*class in ckan.plugins.interfaces*), 262

IResourceView (*class in ckan.plugins.interfaces*), 266
is_fallback() (*ckan.plugins.interfaces.IDatasetForm method*), 263
is_fallback() (*ckan.plugins.interfaces.IGroupForm method*), 271
is_positive_integer() (*in module ckan.logic.validators*), 300
is_rtl_language() (*in module ckan.lib.helpers*), 373
is_url() (*in module ckan.lib.helpers*), 372
ISignal (*class in ckan.plugins.interfaces*), 280
isodate() (*in module ckan.logic.validators*), 301
ITagController (*class in ckan.plugins.interfaces*), 272
ITemplateHelpers (*class in ckan.plugins.interfaces*), 272
ITranslation (*class in ckan.plugins.interfaces*), 275
IUploader (*class in ckan.plugins.interfaces*), 275
IValidators (*class in ckan.plugins.interfaces*), 266

J

job_cancel() (*in module ckan.logic.action.delete*), 216
job_clear() (*in module ckan.logic.action.delete*), 216
job_list() (*in module ckan.logic.action.get*), 194
job_show() (*in module ckan.logic.action.get*), 194
json_list_or_string() (*in module ckan.logic.converters*), 306
json_object() (*in module ckan.logic.validators*), 305
json_or_string() (*in module ckan.logic.converters*), 306

K

keep_extras() (*in module ckan.lib.navl.validators*), 297

L

lang() (*in module ckan.lib.helpers*), 373
lang_native_name() (*in module ckan.lib.helpers*), 373
license_list() (*in module ckan.logic.action.get*), 180
license_options() (*in module ckan.lib.helpers*), 381
limit_to_configured_maximum() (*in module ckan.lib.navl.validators*), 300
link_to() (*in module ckan.lib.helpers*), 373
linked_user() (*in module ckan.lib.helpers*), 376
list_dict_filter() (*in module ckan.lib.helpers*), 381
list_of_strings() (*in module ckan.logic.validators*), 303
literal (*class in ckan.lib.helpers*), 370
load_plugin_helpers() (*in module ckan.lib.helpers*), 382
login() (*ckan.plugins.interfaces.IAuthenticator method*), 275
logout() (*ckan.plugins.interfaces.IAuthenticator method*), 275

M

mail_server() (*in module ckan.tests.pytest_ckan.fixtures*), 459
mail_to() (*in module ckan.lib.helpers*), 382
make_app() (*in module ckan.tests.pytest_ckan.fixtures*), 456
make_error_log_middleware() (*ckan.plugins.interfaces.IMiddleware method*), 256
make_login_url() (*in module ckan.lib.helpers*), 382
make_middleware() (*ckan.plugins.interfaces.IMiddleware method*), 255
map_pylons_to_flask_route_name() (*in module ckan.lib.helpers*), 374
markdown_extract() (*in module ckan.lib.helpers*), 376
member_count() (*in module ckan.lib.helpers*), 379
member_create() (*in module ckan.logic.action.create*), 198
member_delete() (*in module ckan.logic.action.delete*), 213
member_list() (*in module ckan.logic.action.get*), 177
member_roles_list() (*in module ckan.logic.action.get*), 193
migrate_db_for() (*in module ckan.tests.pytest_ckan.fixtures*), 458
missing (*ckan.plugins.toolkit.ckan.plugins.toolkit attribute*), 293
MockUser (*class in ckan.tests.factories*), 449
model() (*ckan.tests.factories.CKANFactory class method*), 448
module
    ckan.lib.helpers, 370
    ckan.lib.navl.validators, 297
    ckan.logic.action.create, 195
    ckan.logic.action.delete, 212
    ckan.logic.action.get, 176
    ckan.logic.action.patch, 211
    ckan.logic.action.update, 204
    ckan.logic.converters, 305
    ckan.logic.validators, 300
    ckan.plugins.interfaces, 255
    ckan.plugins.toolkit, 282
    ckan.tests.controllers, 465
    ckan.tests.factories, 446
    ckan.tests.helpers, 449
    ckan.tests.lib, 466
    ckan.tests.logic.action, 462
    ckan.tests.logic.auth, 462
    ckan.tests.logic.test_schema, 465
    ckan.tests.model, 466
    ckan.tests.plugins, 466
    ckan.tests.pytest_ckan.fixtures, 455
    ckanext.datastore.backend, 90
    ckanext.datastore.logic.action, 81

N

N_()
built-in function, 369
name_validator() (*in module ckan.logic.validators*), 301
natural_number_validator() (*in module ckan.logic.validators*), 300
nav_link() (*in module ckan.lib.helpers*), 373
new_template() (*ckan.plugins.interfaces.IDatasetForm method*), 264
new_template() (*ckan.plugins.interfaces.IGroupForm method*), 271
no_loops_in_hierarchy() (*in module ckan.logic.validators*), 304
non_clean_db() (*in module ckan.tests.pytest_ckan.fixtures*), 459
not_empty() (*in module ckan.lib.navl.validators*), 297
not_missing() (*in module ckan.lib.navl.validators*), 297
notify() (*ckan.plugins.interfaces.IDomainObjectModification method*), 257
notify() (*ckan.plugins.interfaces.IResourceUrlChange method*), 262
notify_after_commit() (*ckan.plugins.interfaces.IDomainObjectModification method*), 257

O

one_of() (*in module ckan.logic.validators*), 305
open() (*ckan.tests.helpers.CKANTestClient method*), 451
Organization (*class in ckan.tests.factories*), 448
organization_autocomplete() (*in module ckan.logic.action.get*), 185
organization_create() (*in module ckan.logic.action.create*), 200
organization_delete() (*in module ckan.logic.action.delete*), 214
organization_facets() (*ckan.plugins.interfaces.IFacets method*), 274
organization_followee_count() (*in module ckan.logic.action.get*), 192
organization_followee_list() (*in module ckan.logic.action.get*), 193
organization_follower_count() (*in module ckan.logic.action.get*), 190
organization_follower_list() (*in module ckan.logic.action.get*), 191
organization_link() (*in module ckan.lib.helpers*), 378
organization_list() (*in module ckan.logic.action.get*), 178
organization_list_for_user() (*in module ckan.logic.action.get*), 180
organization_member_create() (*in module ckan.logic.action.create*), 203
organization_member_delete() (*in module ckan.logic.action.delete*), 215
organization_patch() (*in module ckan.logic.action.patch*), 212
organization_purge() (*in module ckan.logic.action.delete*), 214
organization_show() (*in module ckan.logic.action.get*), 182
organization_update() (*in module ckan.logic.action.update*), 208
OrganizationFactory (*class in ckan.tests.pytest_ckan.fixtures*), 455
organizations_available() (*in module ckan.lib.helpers*), 379
owner_org_validator() (*in module ckan.logic.validators*), 300

P

package_autocomplete() (*in module ckan.logic.action.get*), 184
package_collaborator_create() (*in module ckan.logic.action.create*), 199
package_collaborator_delete() (*in module ckan.logic.action.delete*), 214
package_collaborator_list() (*in module ckan.logic.action.get*), 177
package_collaborator_list_for_user() (*in module ckan.logic.action.get*), 177
package_create() (*in module ckan.logic.action.create*), 195
package_create_default_resource_views() (*in module ckan.logic.action.create*), 198
package_delete() (*in module ckan.logic.action.delete*), 212
package_form() (*ckan.plugins.interfaces.IDatasetForm method*), 265
package_id_does_not_exist() (*in module ckan.logic.validators*), 301
package_id_exists() (*in module ckan.logic.validators*), 301
package_id_not_changed() (*in module ckan.logic.validators*), 300
package_id_or_name_exists() (*in module ckan.logic.validators*), 301
package_list() (*in module ckan.logic.action.get*), 176
package_name_exists() (*in module ckan.logic.validators*), 301
package_name_validator() (*in module ckan.logic.validators*), 302
package_owner_org_update() (*in module ckan.logic.action.update*), 210

package_patch() (*in module ckan.logic.action.patch*), 211
package_relationship_create() (*in module ckan.logic.action.create*), 198
package_relationship_delete() (*in module ckan.logic.action.delete*), 213
package_relationship_update() (*in module ckan.logic.action.update*), 207
package_relationships_list() (*in module ckan.logic.action.get*), 181
package_resource_reorder() (*in module ckan.logic.action.update*), 207
package_revise() (*in module ckan.logic.action.update*), 205
package_search() (*in module ckan.logic.action.get*), 185
package_show() (*in module ckan.logic.action.get*), 181
package_types() (*ckan.plugins.interfaces.IDatasetForm method*), 263
package_update() (*in module ckan.logic.action.update*), 205
package_version_validator() (*in module ckan.logic.validators*), 302
PackageFactory (*class in ckan.tests.pytest_ckan.fixtures*), 455
pager_url() (*in module ckan.lib.helpers*), 376
parse_rfc_2822_date() (*in module ckan.lib.helpers*), 377
perform_password_reset (*in module ckan.lib.signals*), 313
Plugin (*class in ckan.plugins*), 255
postprocess_api_token() (*ckan.plugins.interfaces.IApiToken method*), 279
prepare_dataset_blueprint() (*ckan.plugins.interfaces.IDatasetForm method*), 266
prepare_group_blueprint() (*ckan.plugins.interfaces.IGroupForm method*), 272
prepare_resource_blueprint() (*ckan.plugins.interfaces.IDatasetForm method*), 266
preprocess_api_token() (*ckan.plugins.interfaces.IApiToken method*), 278
provided_by() (*ckan.plugins.interfaces.Interface class method*), 255

R

read() (*ckan.plugins.interfaces.IGroupController method*), 258
read() (*ckan.plugins.interfaces.IOrganizationController method*), 259
read() (*ckan.plugins.interfaces.IPackageController method*), 259
read_template() (*ckan.plugins.interfaces.IDatasetForm method*), 264
read_template() (*ckan.plugins.interfaces.IGroupForm method*), 271
recorded_logs() (*in module ckan.tests.helpers*), 453
RecordingLogHandler (*class in ckan.tests.helpers*), 454
redirect_to() (*in module ckan.lib.helpers*), 371
register_backends() (*ckanext.datastore.backend.DatastoreBackend class method*), 91
register_blueprint (*in module ckan.lib.signals*), 313
remove_linebreaks() (*in module ckan.lib.helpers*), 381
remove_url_param() (*in module ckan.lib.helpers*), 379
remove_whitespace() (*in module ckan.logic.converters*), 306
render_datetime() (*in module ckan.lib.helpers*), 377
render_markdown() (*in module ckan.lib.helpers*), 380
rendered_resource_view() (*in module ckan.lib.helpers*), 380
request (*built-in variable*), 368
request (*ckan.plugins.toolkit.ckan.plugins.toolkit attribute*), 294
request_finished (*in module ckan.lib.signals*), 313
request_password_reset (*in module ckan.lib.signals*), 313
request_started (*in module ckan.lib.signals*), 313
reset_db() (*in module ckan.tests.helpers*), 449
reset_db() (*in module ckan.tests.pytest_ckan.fixtures*), 456
reset_db_once() (*in module ckan.tests.pytest_ckan.fixtures*), 459
reset_index() (*in module ckan.tests.pytest_ckan.fixtures*), 456
reset_redis() (*in module ckan.tests.pytest_ckan.fixtures*), 456
Resource (*class in ckan.tests.factories*), 448
resource_create() (*in module ckan.logic.action.create*), 196
resource_create_default_resource_views() (*in module ckan.logic.action.create*), 197
resource_delete() (*in module ckan.logic.action.delete*), 213
resource_display_name() (*in module ckan.lib.helpers*), 378
resource_download (*in module ckan.lib.signals*), 313
resource_exists() (*ckanext.datastore.backend.DatastoreBackend method*), 92
resource_fields() (*ckanext.datastore.backend.DatastoreBackend method*), 93

resource_form() (*ckan.plugins.interfaces.IDatasetForm method*), 265
resource_formats() (*in module ckan.lib.helpers*), 381
resource_formats_default_file() (*in module ckan.lib.helpers*), 381
resource_id_does_not_exist() (*in module ckan.logic.validators*), 301
resource_id_exists() (*in module ckan.logic.validators*), 301
resource_id_from_alias() (*ckanext.datastore.backend.DatastoreBackend method*), 93
resource_id_validator() (*in module ckan.logic.validators*), 301
resource_info() (*ckanext.datastore.backend.DatastoreBackend method*), 93
resource_link() (*in module ckan.lib.helpers*), 378
resource_patch() (*in module ckan.logic.action.patch*), 212
resource_search() (*in module ckan.logic.action.get*), 187
resource_show() (*in module ckan.logic.action.get*), 181
resource_template() (*ckan.plugins.interfaces.IDatasetForm method*), 265
resource_update() (*in module ckan.logic.action.update*), 204
resource_view_clear() (*in module ckan.logic.action.delete*), 213
resource_view_create() (*in module ckan.logic.action.create*), 197
resource_view_delete() (*in module ckan.logic.action.delete*), 213
resource_view_display_preview() (*in module ckan.lib.helpers*), 380
resource_view_full_page() (*in module ckan.lib.helpers*), 380
resource_view_get_fields() (*in module ckan.lib.helpers*), 380
resource_view_icon() (*in module ckan.lib.helpers*), 380
resource_view_is_filterable() (*in module ckan.lib.helpers*), 380
resource_view_is_iframed() (*in module ckan.lib.helpers*), 380
resource_view_list() (*in module ckan.logic.action.get*), 182
resource_view_reorder() (*in module ckan.logic.action.update*), 205
resource_view_show() (*in module ckan.logic.action.get*), 182
resource_view_update() (*in module ckan.logic.action.update*), 204
ResourceFactory (*class in ckan.tests.pytest_ckan.fixtures*), 455
ResourceView (*class in ckan.tests.factories*), 448
ResourceViewFactory (*class in ckan.tests.pytest_ckan.fixtures*), 455
response (*built-in variable*), 368
role_exists() (*in module ckan.logic.validators*), 303
roles_translated() (*in module ckan.lib.helpers*), 379
RQTestBase (*class in ckan.tests.helpers*), 452

S

sanitize_id() (*in module ckan.lib.helpers*), 382
sanitize_url() (*in module ckan.lib.helpers*), 376
search() (*ckanext.datastore.backend.DatastoreBackend method*), 92
search_sql() (*ckanext.datastore.backend.DatastoreBackend method*), 92
search_template() (*ckan.plugins.interfaces.IDatasetForm method*), 265
sendmail() (*ckan.tests.helpers.FakeSMTP method*), 454
session (*built-in variable*), 369
set_active_backend() (*ckanext.datastore.backend.DatastoreBackend class method*), 91
set_datastore_active_flag() (*in module ckanext.datastore.logic.action*), 87
setup() (*ckan.tests.helpers.FunctionalRQTestBase method*), 452
setup() (*ckan.tests.helpers.FunctionalTestBase method*), 452
setup() (*ckan.tests.helpers.RQTestBase method*), 452
setup_template_variables() (*ckan.plugins.interfaces.IDatasetForm method*), 264
setup_template_variables() (*ckan.plugins.interfaces.IGroupForm method*), 271
setup_template_variables() (*ckan.plugins.interfaces.IResourceView method*), 268
show_package_schema() (*ckan.plugins.interfaces.IDatasetForm method*), 264
SI_number_span() (*in module ckan.lib.helpers*), 381
signals (*ckan.plugins.toolkit.ckan.plugins.toolkit attribute*), 295
SingletonPlugin (*class in ckan.plugins*), 255
snippet() (*in module ckan.lib.helpers*), 378
sorted_extras() (*in module ckan.lib.helpers*), 375
status_show() (*in module ckan.logic.action.get*), 189
strip_value() (*in module ckan.logic.validators*), 304
strxfrm() (*in module ckan.lib.helpers*), 373
Sysadmin (*class in ckan.tests.factories*), 448

SysadminFactory (*class in ckan.tests.pytest_ckan.fixtures*), 455
SysadminWithToken (*class in ckan.tests.factories*), 449
SystemInfo (*class in ckan.tests.factories*), 449
SystemInfoFactory (*class in ckan.tests.pytest_ckan.fixtures*), 455

T

Tag (*class in ckan.tests.factories*), 449
tag_autocomplete() (*in module ckan.logic.action.get*), 188
tag_create() (*in module ckan.logic.action.create*), 202
tag_delete() (*in module ckan.logic.action.delete*), 215
tag_in_vocabulary_validator() (*in module ckan.logic.validators*), 303
tag_length_validator() (*in module ckan.logic.validators*), 302
tag_link() (*in module ckan.lib.helpers*), 378
tag_list() (*in module ckan.logic.action.get*), 180
tag_name_validator() (*in module ckan.logic.validators*), 302
tag_not_in_vocabulary() (*in module ckan.logic.validators*), 303
tag_not_uppercase() (*in module ckan.logic.validators*), 302
tag_search() (*in module ckan.logic.action.get*), 188
tag_show() (*in module ckan.logic.action.get*), 183
tag_string_convert() (*in module ckan.logic.validators*), 302
TagFactory (*class in ckan.tests.pytest_ckan.fixtures*), 455
task_status_delete() (*in module ckan.logic.action.delete*), 215
task_status_show() (*in module ckan.logic.action.get*), 188
task_status_update() (*in module ckan.logic.action.update*), 208
task_status_update_many() (*in module ckan.logic.action.update*), 209
term_translation_show() (*in module ckan.logic.action.get*), 189
term_translation_update() (*in module ckan.logic.action.update*), 209
term_translation_update_many() (*in module ckan.logic.action.update*), 209
test_request_context() (*in module ckan.tests.pytest_ckan.fixtures*), 459
time_ago_from_timestamp() (*in module ckan.lib.helpers*), 377
tmpl_context (*built-in variable*), 368
translator (*built-in variable*), 369
truncate() (*in module ckan.lib.helpers*), 376

U

unfollow_dataset() (*in module ckan.logic.action.delete*), 215
unfollow_group() (*in module ckan.logic.action.delete*), 216
unfollow_user() (*in module ckan.logic.action.delete*), 215
ungettext() built-in function, 369
unicode_only() (*in module ckan.lib.navl.validators*), 300
unicode_safe() (*in module ckan.lib.navl.validators*), 300
unified_resource_format() (*in module ckan.lib.helpers*), 381
update_config() (*ckan.plugins.interfaces.IConfigurer method*), 261
update_config_schema() (*ckan.plugins.interfaces.IConfigurer method*), 261
update_datastore_create_schema() (*ckanext.datastore.interfaces.IDataDictionaryForm method*), 314
update_datastore_info_field() (*ckanext.datastore.interfaces.IDataDictionaryForm method*), 314
update_package_schema() (*ckan.plugins.interfaces.IDatasetForm method*), 263
uploads_enabled() (*in module ckan.lib.helpers*), 381
upsert() (*ckanext.datastore.backend.DatastoreBackend method*), 91
url_for() (*in module ckan.lib.helpers*), 372
url_for_static() (*in module ckan.lib.helpers*), 372
url_for_static_or_external() (*in module ckan.lib.helpers*), 372
url_is_local() (*in module ckan.lib.helpers*), 372
url_validator() (*in module ckan.logic.validators*), 303
User (*class in ckan.tests.factories*), 448
user_about_validator() (*in module ckan.logic.validators*), 303
user_autocomplete() (*in module ckan.logic.action.get*), 184
user_both_passwords_entered() (*in module ckan.logic.validators*), 302
user_create() (*in module ckan.logic.action.create*), 201
user_created (*in module ckan.lib.signals*), 313
user_delete() (*in module ckan.logic.action.delete*), 212
user_followee_count() (*in module ckan.logic.action.get*), 191
user_followee_list() (*in module

ckan.logic.action.get), 192
user_follower_count() (*in module ckan.logic.action.get*), 190
user_follower_list() (*in module ckan.logic.action.get*), 190
user_id_exists() (*in module ckan.logic.validators*), 301
user_id_or_name_exists() (*in module ckan.logic.validators*), 301
user_image() (*in module ckan.lib.helpers*), 376
user_in_org_or_group() (*in module ckan.lib.helpers*), 379
user_invite() (*in module ckan.logic.action.create*), 201
user_list() (*in module ckan.logic.action.get*), 181
user_logged_in (*in module ckan.lib.signals*), 313
user_logged_out (*in module ckan.lib.signals*), 313
user_name_exists() (*in module ckan.logic.validators*), 303
user_name_validator() (*in module ckan.logic.validators*), 302
user_password_not_empty() (*in module ckan.logic.validators*), 303
user_password_validator() (*in module ckan.logic.validators*), 303
user_passwords_match() (*in module ckan.logic.validators*), 303
user_patch() (*in module ckan.logic.action.patch*), 212
user_show() (*in module ckan.logic.action.get*), 183
user_update() (*in module ckan.logic.action.update*), 208
UserFactory (*class in ckan.tests.pytest_ckan.fixtures*), 455
UserWithToken (*class in ckan.tests.factories*), 449

V

validate() (*ckan.plugins.interfaces.IDatasetForm method*), 265
validate() (*ckan.plugins.interfaces.IGroupForm method*), 271
view_resource_url() (*in module ckan.lib.helpers*), 380
view_template() (*ckan.plugins.interfaces.IResourceView method*), 268
Vocabulary (*class in ckan.tests.factories*), 448
vocabulary_create() (*in module ckan.logic.action.create*), 202
vocabulary_delete() (*in module ckan.logic.action.delete*), 215
vocabulary_id_exists() (*in module ckan.logic.validators*), 303
vocabulary_id_not_changed() (*in module ckan.logic.validators*), 303
vocabulary_list() (*in module ckan.logic.action.get*), 189
vocabulary_name_validator() (*in module ckan.logic.validators*), 303
vocabulary_show() (*in module ckan.logic.action.get*), 189
vocabulary_update() (*in module ckan.logic.action.update*), 210
VocabularyFactory (*class in ckan.tests.pytest_ckan.fixtures*), 455

W

with_extended_cli() (*in module ckan.tests.pytest_ckan.fixtures*), 459
with_plugins() (*in module ckan.tests.pytest_ckan.fixtures*), 458
with_request_context() (*in module ckan.tests.pytest_ckan.fixtures*), 459
with_test_worker() (*in module ckan.tests.pytest_ckan.fixtures*), 459