ievv$_{o}pensourceDocumentation$

Release 5.13.0

ievv$_{o}pensource$

Oct 17, 2018

Contents

CHAPTER 1

Install

```
$ pip install ievv_opensource
```

CHAPTER 2

Development

2.1 Development guide

2.1.1 Install the requirements

Install the following:

1. Python
2. PIP
3. VirtualEnv
4. virtualenvwrapper

2.1.2 Install in a virtualenv

Create a virtualenv using Python 3 (an isolated Python environment):

```
$ mkvirtualenv -p /usr/local/bin/python3 ievv_opensource
```

Install the development requirements:

```
$ pip install -r requirements.txt
```

Note: Whenever you start a new shell where you need to use the virtualenv we created with `mkvirtualenv` above, you have to run:

```
$ workon ievv_opensource
```

2.1.3 Build the docs

Enable the virtualenv, and run:

```
$ ievv docs --build --open
```

2.1.4 Create a development database

Enable the virtualenv, and run:

```
$ ievv recreate_devdb
```

2.1.5 Running tests

To run the tests, we need to use a different settings file. We tell ievvtasks to do this using the `DJANGOENV` environent variable:

```
$ DJANGOENV=test python manage.py test
```

CHAPTER 3

Apps

3.1 *ievv_tagframework* — General purpose tagging framework

The intention of this module is to provide a re-usable tagging framework for Django apps.

3.1.1 Data model API

class ievv_opensource.ievv_tagframework.models.**Tag**(**args*, ***kwargs*)
Bases: django.db.models.base.Model

A single tag.

A tag has a unique name, and data models is added to a tag via *TaggedObject*.

taglabel
The label for the tag.

tagtype
The tagtype is a way for applications to group tags by type. No logic is assigned to this field by default, other than that is is db_indexed. The valid choices for the field is configured via the *IEVV_TAGFRAMEWORK_TAGTYPE_CHOICES* setting.

static get_tagtype_choices()
Returns choices for *tagtype*.

This is not set as choices on the field (because changing that would trigger a migration), but it should be used in any form displaying tagtype.

You configure the return value via the IEVV_TAGFRAMEWORK_TAGTYPE_CHOICES Django setting.

classmethod get_tagtype_valid_values()
Returns an iterator over the choices returned by *get_tagtype_choices()*, but only the values (not the labels) as a flat list.

clean()
Hook for doing any extra model-wide validation after clean() has been called on every field by

self.clean_fields. Any ValidationError raised by this method will not be associated with a particular field; it will have a special-case association with the field defined by NON_FIELD_ERRORS.

exception DoesNotExist
Bases: django.core.exceptions.ObjectDoesNotExist

exception MultipleObjectsReturned
Bases: django.core.exceptions.MultipleObjectsReturned

class ievv_opensource.ievv_tagframework.models.**TaggedObject**(**args*, ***kwargs*)
Bases: django.db.models.base.Model

Represents a many-to-many relationship between any data model object and a *Tag*.

tag
The *Tag*.

content_type
The ContentType of the tagged object.

object_id
The ID of the tagged object.

content_object
The GenericForeignKey using *content_type* and *object_id* to create a generic foreign key to the tagged object.

exception DoesNotExist
Bases: django.core.exceptions.ObjectDoesNotExist

exception MultipleObjectsReturned
Bases: django.core.exceptions.MultipleObjectsReturned

3.2 *ievv_batchframework* — Framework for batch/bulk tasks

The intention of this module is to make it easier to write code for background tasks and some kinds of bulk operations.

3.2.1 Configuration

Add the following to your INSTALLED_APPS-setting:

'ievv_opensource.ievv_batchframework.apps.BatchOperationAppConfig'

3.2.2 Batchregistry - the high level API

3.2.3 Recommended Celery setup

Install Redis

Redis is very easy to install and use, and it is one of the recommended broker and result backends for Celery, so we recommend that you use this when developing with Celery. You may want to use the Django database instead, but that leaves you with a setup that is further from a real production environment, and using Redis is very easy if you use *ievv devrun* as shown below.

On Mac OSX, you can install redis with Homebrew using:

```
$ brew install redis
```

and most linux systems have Redis in their package repository. For other systems, go to http://redis.io/, and follow their install guides.

Configure Celery

First, you have to create a Celery Application for your project. Create a file named `celery.py` within a module that you know is loaded when Django starts. The safest place is in the root of your project module. So if you have:

```
myproject/
    __init__.py
    myapp/
        __init__.py
        models.py
    mysettings/
        settings.py
```

You should add the celery configuration in `myproject/celery.py`. The rest of this guide will assume you put it at this location.

Put the following code in `myproject/celery.py`:

```
from __future__ import absolute_import
import os
from celery import Celery

# Ensure this matches your
os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'myproject.settings')

# The ``main``-argument is used as prefix for celery task names.
app = Celery(main='myproject')

# We put all the celery settings in out Django settings so we use
# this line to load Celery settings from Django settings.
# You could also add configuration for celery directly in this
# file using app.conf.update(...)
app.config_from_object('django.conf:settings')

# This debug task is only here to make it easier to verify that
# celery is working properly.
@app.task(bind=True)
def debug_add_task(self, a, b):
    print('Request: {0!r} - Running {} + {}, and returning the result.'.format(
        self.request, a, b))
    return a + b
```

And put the following code in `myproject/__init__.py`:

```
from __future__ import absolute_import

# This will make sure the Celery app is always imported when
# Django starts so that @shared_task will use this app.
from .celery import app as celery_app
```

Add the following to your Django settings:

```
# Celery settings
BROKER_URL = 'redis://localhost:6379'
CELERY_RESULT_BACKEND = 'redis://localhost:6379'
CELERY_ACCEPT_CONTENT = ['application/json']
CELERY_TASK_SERIALIZER = 'json'
CELERY_RESULT_SERIALIZER = 'json'
CELERY_TIMEZONE = 'Europe/Oslo'  # Change to your preferred timezone!
CELERY_IMPORTS = [
    'ievv_opensource.ievv_batchframework.celery_tasks',
]
CELERYD_TASK_LOG_FORMAT = '[%(asctime)s: %(levelname)s/%(processName)s] ' \
                          '[%(name)s] ' \
                          '[%(task_name)s(%(task_id)s)] ' \
                          '%(message)s'

# ievv_batchframework settings
IEVV_BATCHFRAMEWORK_CELERY_APP = 'myproject.celery_app'
```

Setup *ievv devrun — All your development servers in one command*, and add `ievvdevrun.runnables.redis_server.RunnableThread()` and ‘‘ to your `IEVVTASKS_DEVRUN_RUNNABLES`. You should end up with something like this:

```
IEVVTASKS_DEVRUN_RUNNABLES = {
    'default': ievvdevrun.config.RunnableThreadList(
        # ievvdevrun.runnables.dbdev_runserver.RunnableThread(),  # Uncomment if␣
↪using django_dbdev
        ievvdevrun.runnables.django_runserver.RunnableThread(),
        ievvdevrun.runnables.redis_server.RunnableThread(),
        ievvdevrun.runnables.celery_worker.RunnableThread(app='myproject'),
    ),
}
```

At this point, you should be able to run:

```
$ ievv devrun
```

to start the Django server, redis and the celery worker. To test that everything is working:

1. Take a look at the output from `ievv devrun`, and make sure that your `debug_add_task` (from celery.py) is listed as a task in the `[tasks]` list printed by the celery worker on startup. If it is not, this probably means you did not put the code in the `myproject/__init__.py` example above in a place that Django reads at startup. You may want to try to move it into the same module as your `settings.py` and restart `ievv devrun`.
2. Start up the django shell and run the `debug_add_task`:

   ```
   $ python manage.py shell
   >>> from myproject.celery import debug_add_task
   >>> result = debug_add_task.delay(10, 20)
   >>> result.wait()
   30
   ```

   If this works, Celery is configured correctly.

3.2.4 Developing with asyncronous actions

When developing with asyncronous tasks with the setup from the introduction guide above, you need to restart `ievv devrun` each time you change some code used by an asyncronous action. This means that if you add an Action or ActionGroup, or change any code used within an Action or ActionGroup, you have to stop `ievv devrun`, and start it again. We provide two options for avoiding this.

Option 1: Run all actions synchronously

This is great for unit tests, and for developing and debugging code in your `ievv_opensource.ievv_batchframework.batchregistry.Action.executable()` methods. To enable this, add the following to your settings:

```
IEVV_BATCHFRAMEWORK_ALWAYS_SYNCRONOUS = True
```

Option 2: Run celery worker manually

Restarting `ievv devrun` can take some time if you have lots of commands that have to stop and start again. You can save some time for each change if you remove/comment out the `ievvdevrun.runnables.celery_worker.RunnableThread` line from the `IEVVTASKS_DEVRUN_RUNNABLES` setting (restart `ievv devrun` after this change), and run Celery manually with the following command instead:

```
$ celery -A myproject worker -l debug
```

Now you can start and stop only the Celery worker instead of restarting `ievv devrun`.

3.2.5 Batchregistry API

exception ievv_opensource.ievv_batchframework.batchregistry.**ActionGroupSynchronousExecutio**
Bases: Exception

exception ievv_opensource.ievv_batchframework.batchregistry.**ActionError**(*error_data_dict*)
Bases: Exception

ievv_opensource.ievv_batchframework.batchregistry.**action_factory**(*baseclass*, *name*)
Factory for creating *Action* classes. This is simply a thin wrapper around `type` to dynamically create a subclass of the given `baseclass` with a different name.

There are two use cases for using this:

- Give re-usable action classes a better name.
- Use the same *Action* subclass multiple times in the same *ActionGroup*.

Both of these cases can also be solved with subclasses, and that is normally a better solution.

Parameters

- **baseclass** – The class to create a subclass of.
- **name** – The name of the subclass.

class ievv_opensource.ievv_batchframework.batchregistry.**Action**(***kwargs*)
Bases: object

An action is the subclass for code that can be executed as part of an *ActionGroup*.

You create a subclass of this class, and override `execute()` to implement an action, and you add your subclass to an *ActionGroup* to use your action class.

classmethod run(**kwargs)
Run the action - used internally.

Parameters

- **kwargs** – Kwargs for the action.
- **executed_by_celery** – Must be `True` if the action is executed by a celery task. This is required to configure the correct logger.

classmethod get_name()
Get the name of this action.

logger
Get the logger for this action.

execute()
Execute the action. Must be overridden in subclasses.

class ievv_opensource.ievv_batchframework.batchregistry.**ActionGroupExecutionInfo**(*actiongroup*, *mode*, *route_to_alias*, *actiongroupresult=None*, *batchoperation=None*)

Bases: `object`

Return value from *ActionGroup.run()*, with information about how the ActionGroup was executed.

actiongroup
The :class:`.ActionGroup` object that was executed.

mode
The mode the ActionGroup was executed with.

route_to_alias
The route_to_alias that was used to route/prioritize the execution of the ActionGroup.

is_asynchronous
Property that returns `True` if the ActionGroup was executed asynchronously.

actiongroupresult
Property for getting the `ActionGroupResult` if the ActionGroup was executed in synchronous mode.

Raises `AttributeError` – If `mode` is *ActionGroup.MODE_ASYNCHRONOUS*.

batchoperation
Property for getting the *ievv_opensource.ievv_batchframework.models.BatchOperation* object that was created if the ActionGroup was executed in asynchronous mode.

Raises `AttributeError` – If `mode` is *ActionGroup.MODE_ASYNCHRONOUS*.

class ievv_opensource.ievv_batchframework.batchregistry.**ActionGroup**(*name*, *actions=None*, *mode=None*, *route_to_alias=None*)

Bases: object

An ActionGroup is a list of *actions* that can be executed both synchronously and asynchronously.

Parameters

- **name** – The name of the ActionGroup.
- **actions** – A list of actions.
- **mode** – The default mode of the ActionGroup. Defaults to *MODE_ASYNCHRONOUS*. You will often want to determine this from the input (I.E.: Use asynchronous if sending more than 500 newsletters), and this can be done by extending this class and overriding *get_mode()*.
- **route_to_alias** – Defines where to route this ActionGroup when is is executed in asynchronous mode. Defaults to Registry.ROUTE_TO_ALIAS_DEFAULT. You can determine the route dynamically each time the ActionGroup is executed by overriding *get_route_to_alias()*.

MODE_ASYNCHRONOUS = 'asynchronous'
Constant for asynchronous (background/Celery) mode of execution.

MODE_SYNCHRONOUS = 'synchronous'
Constant for synchronous (blocking) mode of execution.

add_action(*action*)
Add a action.

Parameters **action** – A subclass of *Action* (not an object, but a class).

add_actions(*actions*)
Add actions.

Parameters **actions** – A list of *Action* subclasses (classes not actions).

get_mode(***kwargs*)
Get the mode to run the ActionGroup in. Must return one of *MODE_ASYNCHRONOUS* or *MODE_SYNCHRONOUS*.

The use-case for overriding this method is optimization. Lets say you have to re-index your blogposts in a search engine each time they are updated. If you update just a few blogpost, you may want to do that in synchronous mode, but if you update 500 blogposts, you will probably want to re-index in asynchronous mode (I.E. in Celery).

Parameters **kwargs** – The kwargs the user provided to *run()*.

get_route_to_alias(***kwargs*)
Define where to route this ActionGroup when is is executed in asynchronous mode.

This is the method you want to override to handle priority of your asynchronously executed ActionGroups.

Lets say you have a huge blog, with lots of traffic. After updating a blogpost, you need to do some heavy postprocessing (image optimization, video transcoding, etc.). If you update a newly posted blogpost this postprocessing should be placed in a high-priority queue, and if you update an old blogpost, this postprocessing should be placed in a low-priority queue. To achieve this, you simply need to create a subclass of ActionGroup, and override this method to return Registry.ROUTE_TO_ALIAS_HIGHPRIORITY for recently created blogpost, and Registry.ROUTE_TO_ALIAS_DEFAULT for old blogposts.

Parameters **kwargs** – The `kwargs` the user provided to *run_asynchronous()*.

Returns One of the route-to aliases added to the *Registry* using *Registry.add_route_to_alias()*. This will always include `Registry.ROUTE_TO_ALIAS_DEFAULT` and `Registry.ROUTE_TO_ALIAS_HIGHPRIORITY`.

Return type str

run_synchronous(**kwargs)

Run the ActionGroup in blocking/synchronous mode.

Parameters **kwargs** – Kwargs for *Action*.

get_batchoperation_options(**kwargs)

You can override this if you create a re-usable ActionGroup subclass that sets options for the *ievv_opensource.ievv_batchframework.models.BatchOperation* based on `kwargs`.

Called by *run_asynchronous()* to get the kwargs for *ievv_opensource.ievv_batchframework.models.BatchOperationManager.create_asynchronous()*.

If you override this, you should normally call `super()`, and update the kwargs returned by super.

Parameters **kwargs** – The `kwargs` the user provided to *run_asynchronous()*.

Returns Kwargs for *ievv_opensource.ievv_batchframework.models.BatchOperationManager.create_asynchronous()*

Return type dict

create_batchoperation(**kwargs)

Used by *run_asynchronous()* to create the *ievv_opensource.ievv_batchframework.models.BatchOperation* object.

You normally do not need to override this - override *get_batchoperation_options()* instead.

Warning: Overriding this may lead to breaking code if the inner workings of this framework is changed/optimized in the future.

Parameters **kwargs** – See *run_asynchronous()*.

Returns The created *ievv_opensource.ievv_batchframework.models.BatchOperation*.

Return type *BatchOperation*

run_asynchronous(**kwargs)

Parameters **kwargs** – Kwargs for *Action*.

run(**kwargs)

Runs one of *run_asynchronous()* and *run_synchronous()*. The method to run is determined by the return-value of *get_mode()*:

- If *get_mode()* returns *MODE_ASYNCHRONOUS*, *run_asynchronous()* is called.
- If *get_mode()* returns *MODE_SYNCHRONOUS*, *run_synchronous()* is called.

Parameters

- **context_object** – context_object for *ievv_opensource.ievv_batchframework.models.BatchOperation*.

- **started_by** – started_by for *ievv_opensource.ievv_batchframework.models.BatchOperation*.
- ****kwargs** – Kwargs for *Action*. Forwarded to *run_asynchronous()* and *run_synchronous()*.

class ievv_opensource.ievv_batchframework.batchregistry.**Registry**

Bases: *ievv_opensource.utils.singleton.Singleton*

The registry of *ActionGroup* objects.

add_route_to_alias(*route_to_alias*, *task_callable*)

Add a route-to alias.

Parameters

- **route_to_alias** (*str*) – The alias.
- **task_callable** (*func*) – The callable rq task.

add_actiongroup(*actiongroup*)

Add an *ActionGroup* to the registry.

Parameters **actiongroup** – The *ActionGroup* object to add.

remove_actiongroup(*actiongroup_name*)

Remove an *ActionGroup* from the registry.

Parameters **actiongroup_name** – The name of the actiongroup.

Raises KeyError – If no *ActionGroup* with the provided actiongroup_name exists in the registry.

get_actiongroup(*actiongroup_name*)

Get an *ActionGroup* object from the registry.

Parameters **actiongroup_name** – The name of the actiongroup.

Returns An *ActionGroup* object.

Return type *ActionGroup*

Raises KeyError – If no *ActionGroup* with the provided actiongroup_name exists in the registry.

run(*actiongroup_name*, ***kwargs*)

Shortcut for:

```
Registry.get_instance().get_actiongroup(actiongroup_name)
↪run(**kwargs)
```

See also:

get_actiongroup() and *ActionGroup.run()*.

3.2.6 The BatchOperation model

The BatchOperation model is at the hearth of ievv_batchframework. Each time you start a batch process, you create an object of *ievv_opensource.ievv_batchframework.models.BatchOperation* and use that to communicate the status, success/error data and other metadata about the batch operation.

Asynchronous operations

An asynchronous operation is the most common use case for the BatchOperation model. It is used to track a task that is handled (E.g.: completed) by some kind of asynchronous service such as a cron job or a Celery task.

Lets say you have want to send an email 15 minutes after a blog post has been created unless the user cancels the email sending within within 15 minutes. You would then need to:

- Create a BatchOperation object each time a blog post is created.
- Use some kind of batching service, like Celery, to poll for BatchOperation objects that asks it to send out email.
- Delete the BatchOperation if a user clicks "cancel" within 15 minutes of the creation timestamp.

The code for this would look something like this:

```
from ievv_opensource.ievv_batchframework.models import BatchOperation

myblogpost = Blog.objects.get(...)
BatchOperation.objects.create_asynchronous(
    context_object=myblogpost,
    operationtype='new-blogpost-email')
# code to send the batch operation to the batching service (like celery)
# with a 15 minute delay, or just a service that polls for
# BatchOperation.objects.filter(operationtype='new-blogpost-email',
#                               created_datetime__lt=timezone.now() -␣
↪timedelta(minutes=15))


# The batching service code
def my_batching_service(...):
    batchoperation = BatchOperation.objects.get(...)
    batchoperation.mark_as_running()
    # ... send out the emails ...
    batchoperation.finish()


# In the view for cancelling email sending
BatchOperation.objects\
    .filter(operationtype='new-blogpost-email',
            context_object=myblogpost)\
    .remove()
```

Synchronous operations

You may also want to use BatchOperation for synchronous operations. This is mainly useful for complex bulk create and bulk update operations.

Lets say you have Game objects with a one-to-many relationship to Player objects with a one-to-many relationship to Card objects. You want to start all players in a game with a card. How to you batch create all the players with a single card?

You can easily batch create players with `bulk_create`, but you can not batch create the cards because they require a Player. So you need to a way of retrieving the players you just batch created.

If you create a BatchOperation with *`context_object`* set to the Game, you will get a unique identifier for the operation (the id of the BatchOperation). Then you can set that identifier as an attribute on all the batch-created Player objects (preferrably as a foreign-key), and retrieve the batch created objects by filtering on the id of the BatchOperation.

After this, you can iterate through all the created Player objects, and create a list of Card objects for your batch create operation for the cards.

Example:

```
game = Game.objects.get(...)
batchoperation = BatchOperation.objects.create_synchronous(
    context_object=game)

players = []
for index in range(1000):
    player = Player(
        game=game,
        name='player{}'.format(index),
        batchoperation=batchoperation)
    players.append(player)
Player.objects.bulk_create(players)
created_players = Player.objects.filter(batchoperation=batchoperation)

cards = []
available_cards = [...]  # A list of available card IDs
for player in created_players:
    card = Card(
        player=player,
        cardid=random.choice(available_cards)
    )
    cards.append(card)
Card.objects.bulk_create(cards)
batchoperation.finish()
```

As you can see in the example above, instead of having to perform 2000 queries (one for each player, and one for each card), we now only need 5 queries no matter how many players we have (or a few more on database servers that can not bulk create 1000 items at a time).

3.2.7 Data model API

class ievv_opensource.ievv_batchframework.models.**BatchOperationManager**

Bases: `django.db.models.manager.Manager`

Manager for *BatchOperation*.

create_synchronous(*input_data=None*, ***kwargs*)

Create a synchronous *BatchOperation*.

An synchronous batch operation starts with *BatchOperation.status* set to *BatchOperation.STATUS_RUNNING* and `started_running_datetime` set just as if *BatchOperation.mark_as_running()* was called. So calling this would have the same result as calling *create_asynchronous()* and then calling *BatchOperation.mark_as_running()*, but this will just use one database query instead of two.

The *BatchOperation* is cleaned before it is saved.

Parameters

- **input_data** – The input data. A python object to set as the input data using the `BatchOperation.input_data()` property.
- ****kwargs** – Forwarded to the constructor for *BatchOperation*.

Returns The created BatchOperation object.

Return type *BatchOperation*

create_asynchronous(*input_data=None*, ***kwargs*)

Create an asynchronous *BatchOperation*. An asynchronous batch operation starts with *BatchOperation.status* set to *BatchOperation.STATUS_UNPROCESSED*.

The *BatchOperation* is cleaned before it is saved.

Parameters

- **input_data** – The input data. A python object to set as the input data using the `BatchOperation.input_data()` property.
- ****kwargs** – Forwarded to the constructor for *BatchOperation*.

Returns The created BatchOperation object.

Return type *BatchOperation*

class ievv_opensource.ievv_batchframework.models.**BatchOperation**(**args*, ***kwargs*)

Bases: `django.db.models.base.Model`

Defines a batch operation.

STATUS_UNPROCESSED = 'unprocessed'

One of the possible values for *status*. Defines the BatchOperation as uprocessed (not yet started). This only makes sense for background tasks. They will typically be created with the unprocessed status, and then set to *STATUS_RUNNING* when the batching service starts running the operation.

STATUS_RUNNING = 'running'

One of the possible values for *status*. Defines the BatchOperation as running (in progress).

STATUS_FINISHED = 'finished'

One of the possible values for *status*. Defines the BatchOperation as finished.

STATUS_CHOICES = [('unprocessed', 'unprocessed'), ('running', 'running'), ('finished',

Allowed values for *status*. Possible values are:

- *STATUS_UNPROCESSED*.
- *STATUS_RUNNING*.
- *STATUS_FINISHED*.

RESULT_NOT_AVAILABLE = 'not-available'

One of the possible values for *result*. This is used when we have no result yet (the operation is not finished).

RESULT_SUCCESSFUL = 'successful'

One of the possible values for *result*. Defines the BatchOperation as failed. This is set if the operation could not be completed because of an error. Any details about the result of the operation can be stored in *output_data_json*.

RESULT_FAILED = 'failed'

One of the possible values for *result*. Defines the BatchOperation as failed. This is set if the operation could not be completed because of an error. Any error message(s) should be stored in *output_data_json*.

RESULT_CHOICES = [('not-available', 'not available yet (processing not finished)'), ('

Allowed values for *result*. Possible values are:

- *RESULT_NOT_AVAILABLE*.
- *RESULT_SUCCESSFUL*.

- *RESULT_FAILED*.

started_by
The user that started this batch operation. Optional, but it is good metadata to add for debugging.

created_datetime
The datetime when this batch operation was created. Defaults to `timezone.now()`.

started_running_datetime
The datetime when this batch operation started running. This is not the same as *created_datetime*, this is the time when the operation started processing.

finished_datetime
The datetime when this batch operation was finished.

context_content_type
The content type for *context_object*.

context_object_id
The id field for *context_object*.

context_object
Generic foreign key that identifies the context this operation runs in. This is optional.

operationtype
The type of operation. This is application specific - you typically use this if you allow multiple different batch operations on the same *context_object*. This is not required, and defaults to empty string.

status
The status of the operation. The allowed values for this field is documented in *STATUS_CHOICES*. Defaults to *STATUS_UNPROCESSED*.

result
The result of the operation. The allowed values for this field is documented in *RESULT_CHOICES*. Defaults to *RESULT_NOT_AVAILABLE*.

input_data_json
Input data for the BatchOperation. You should not use this directly, use the *input_data* property instead.

output_data_json
Output data for the BatchOperation. You should not use this directly, use the *output_data* property instead.

input_data
Decode *BatchOperation.input_data_json* and return the result.

Return *None* if input_data_json is empty.

output_data
Decode *BatchOperation.output_data_json* and return the result.

Returns *None* if output_data_json is empty, or the decoded json data if the `output_data` is not empty.

Return type object

mark_as_running()
Mark the batch operation as running.

Sets the *status* to *STATUS_RUNNING*, *started_running_datetime* to the current datetime, clean and save.

finish (*failed=False*, *output_data=None*)
Mark the bulk operation as finished.

Sets *result* as documented in the `failed` parameter below. Sets *finished_datetime* to the current datetime. Sets *output_data_json* as documented in the `output_data` parameter below.

Parameters

- **failed** (*boolean*) – Set this to `False` to set *result* to *RESULT_FAILED*. The default is `True`, which means that *result* is set to *RESULT_SUCCESSFUL*
- **output_data** – The output data. A python object to set as the output data using the `BatchOperation.output_data()` property.

exception DoesNotExist
Bases: `django.core.exceptions.ObjectDoesNotExist`

exception MultipleObjectsReturned
Bases: `django.core.exceptions.MultipleObjectsReturned`

clean()
Hook for doing any extra model-wide validation after clean() has been called on every field by self.clean_fields. Any ValidationError raised by this method will not be associated with a particular field; it will have a special-case association with the field defined by NON_FIELD_ERRORS.

3.3 *ievv_elasticsearch* — Thin wrapper around pyelasticsearch

The `ievv_elasticsearch` module is designed to just make it a small bit easier to work with elasticsearch in Django than to just use pyelasticsearch.

3.3.1 Features

- Makes `pyelasticsearch` easy to use in Django.
- Very thin wrapper around `pyelasticsearch`. This means that you use the elasticsearch REST API almost directly with just some pythonic glue.
- Automatic indexing of data store data is decoupled from the search/get API.
- Automatic indexing of data store data works with any data store. Our examples use Django ORM, but you can just as easily use a NOSQL database like MongoDB or an XML database. The only difference is the code you provide to convert your data store data into ElasticSearch JSON compatible data structures.
- Small Django/python helpers that enhances the `pyelasticsearch` API.
- Shortcuts and solutions that makes unit testing easy.

Why not use Haystack?

Haystack is an awesome library for full text document search in a search engine independent manner. But if you want to make full use of ElasticSearch as more than just a document search index, and use it for analytics, document caching and a general purpose nosql data store, haystack just adds an extra layer of complexity that you have to work around. This is, of course, use-case dependent, and many use cases will probably be better served by a combination of `ievv_elasticsearch` and Haystack.

Note: If considering combining Haystack with `ievv_elasticsearch`, you should know that `ievv_elasticsearch` has loose coupling between index definition and querying. Indexe definitions in `ievv_elasticsearch` are only used to make it easy to sync the backend data store into an elasticseach index. If you define the search indexes in haystack, you can still use the *ievv_opensource.ievv_elasticsearch.search* API, you just ignore *ievv_opensource.ievv_elasticsearch.autoindex*.

3.3.2 Getting started

You only need the following to get started:

- ElasticSearch server.
- Configure `IEVV_ELASTICSEARCH_URL` with the URL of the server.

Then you can start using the *ievv_opensource.ievv_elasticsearch.search.Connection* API.

Setup for unit-testing and development

First, copy `not_for_deploy/elasticsearch.develop.yml` and `not_for_deploy/elasticsearch.unittest.yml` into your own project.

In your **test settings**, add:

```
IEVV_ELASTICSEARCH_TESTURL = 'http://localhost:9251'
IEVV_ELASTICSEARCH_TESTMODE = True
IEVV_ELASTICSEARCH_AUTOREFRESH_AFTER_INDEXING = True
```

In your **develop settings**, add:

```
IEVV_ELASTICSEARCH_URL = 'http://localhost:9252'
```

When this is configured, you can run elasticsearch with *ievv devrun — All your development servers in one command* if you add the following to *IEVVTASKS_DEVRUN_RUNNABLES*:

```
IEVVTASKS_DEVRUN_RUNNABLES = {
    'default': ievvdevrun.config.RunnableThreadList(
        # ...
        ievvdevrun.runnables.elasticsearch.RunnableThread(configpath='not_for_deploy/
↪elasticsearch.unittest.yml'),
        ievvdevrun.runnables.elasticsearch.RunnableThread(configpath='not_for_deploy/
↪elasticsearch.develop.yml'),
    ),
]
```

(the paths assumes you put the configfiles in the `not_for_deploy/` directory in your project).

3.3.3 Automatically update the search indexes

Unless you use ElasticSearch as the primary data source, you will most likely want an easy method of: 1. Update the search index when data in the data store changes. 2. Rebuild the search index from the data in the data store.

This is solved by:

1. Define a *ievv_opensource.ievv_elasticsearch.autoindex.AbstractIndex*. The `ievv_elasticsearch` convention is to put search indexes in `yourapp.elasticsearch_autoindexes`.
2. Register the index class in *ievv_opensource.ievv_elasticsearch.autoindex.Registry*.
3. Optionally override *ievv_opensource.ievv_elasticsearch.autoindex.AbstractIndex.register_index_update_triggers()*. and register triggers that react to data store changes and trigger re-indexing.

3.3.4 search API

ievv_opensource.ievv_elasticsearch.search.Connection	Singleton wrapper around `pyelasticsearch.ElasticSearch`.
ievv_opensource.ievv_elasticsearch.search.SearchResultWrapper	An efficient wrapper around the data returned by `pyelasticsearch.ElasticSearch.search()`.
ievv_opensource.ievv_elasticsearch.search.SearchResultItem	Wrapper around a single dictionary in the `hits.hits` list of the result returned by `pyelasticsearch.ElasticSearch.search()`.
ievv_opensource.ievv_elasticsearch.search.Paginator	Paginator for *ievv_opensource.ievv_elasticsearch.search.SearchResultWrapper*.

class ievv_opensource.ievv_elasticsearch.search.**IevvElasticSearch**(*urls='http://localhost'*, *timeout=60*, *max_retries=0*, *port=9200*, *username=None*, *password=None*, *ca_certs='/home/docs/checkouts/rea opensource/envs/stable/lib/python3.5 packages/certifi/cacert.pem'*, *client_cert=None*)

Bases: `pyelasticsearch.client.ElasticSearch`

Parameters

- **urls** – A URL or iterable of URLs of ES nodes. These can be full URLs with port numbers, like `http://elasticsearch.example.com:9200`, or you can pass the port separately using the `port` kwarg. To do HTTP basic authentication, you can use RFC-2617-style URLs like `http://someuser:somepassword@example.com:9200` or the separate `username` and `password` kwargs below.
- **timeout** – Number of seconds to wait for each request before raising Timeout
- **max_retries** – How many other servers to try, in series, after a request times out or a connection fails
- **username** – Authentication username to send via HTTP basic auth
- **password** – Password to use in HTTP basic auth. If a username and password are embedded in a URL, those are favored.

- **port** – The default port to connect on, for URLs that don't include an explicit port
- **ca_certs** – A path to a bundle of CA certificates to trust. The default is to use Mozilla's bundle, the same one used by Firefox.
- **client_cert** – A certificate to authenticate the client to the server

send_request(*method*, *path_components*, *body=''*, *query_params=None*)
Does exactly the same as the method from the superclass, but also prettyprints the request and response if the *IEVV_ELASTICSEARCH_PRETTYPRINT_ALL_REQUESTS* setting is `True`.

class ievv_opensource.ievv_elasticsearch.search.**Connection**
Bases: *ievv_opensource.utils.singleton.Singleton*

Singleton wrapper around `pyelasticsearch.ElasticSearch`.

We do not try to wrap everything, instead we use the pyelasticsearch API as it is, and add extra features that makes it easier to use with IEVV and Django. We provide shortcuts for the most commonly used methods of `pyelasticsearch.ElasticSearch`, some custom methods and we add some code to make unit testing easier.

Usage:

```
from ievv_opensource.ievv_elasticsearch import search

searchapi = search.Connection.get_instance()
searchapi.bulk_index(
    index='contacts',
    doc_type='person',
    docs=[{'name': 'Joe Tester'},
          {'name': 'Peter The Super Tester'}])
searchresult1 = searchapi.wrapped_search(query='name:joe OR name:freddy', index=
↪'contacts')
searchresult2 = searchapi.wrapped_search(query={
    'query': {
        'match': {
            'name': {
                'query': 'Joe'
            }
        }
    }
})
print(searchresult1.total)
for item in searchresult1:
    print(item.doc['name'])
```

elasticsearch
The `pyelasticsearch.ElasticSearch` object.

clear_all_data()
Clear all data from ElasticSearch. Perfect for unit tests.

Only allowed when the `IEVV_ELASTICSEARCH_TESTMODE`-setting is `True`.

Usage:

```
class MyTest(TestCase):
    def setUp(self):
        self.searchapi = search.Connection.get_instance()
        self.searchapi.clear_all_data()
```

index(**args*, ***kwargs*)
Wrapper around `pyelasticsearch.ElasticSearch.index()`.

Works exactly like the wrapped function, except that we provide some extra features that makes testing easier. When the `IEVV_ELASTICSEARCH_TESTMODE`-setting is `True`, we automatically run `pyelasticsearch.ElasticSearch.refresh()` before returning.

bulk_index(**args*, ***kwargs*)
Wrapper around `pyelasticsearch.ElasticSearch.bulk_index()`.

Works exactly like the wrapped function, except that we provide some extra features that makes testing easier. When the `IEVV_ELASTICSEARCH_TESTMODE`-setting is `True`, we automatically run `pyelasticsearch.ElasticSearch.refresh()` before returning.

bulk(**args*, ***kwargs*)
Wrapper around `pyelasticsearch.ElasticSearch.bulk()`.

Works exactly like the wrapped function, except that we provide some extra features that makes testing easier. When the `IEVV_ELASTICSEARCH_TESTMODE`-setting is `True`, we automatically run `pyelasticsearch.ElasticSearch.refresh()` before returning.

search(*query*, *prettyprint_query=False*, ***kwargs*)
Wrapper around `pyelasticsearch.ElasticSearch.search()`.

Works just like the wrapped function, except that `query` can also be an `elasticsearch_dsl.Search` object, and you can only specify arguments as kwargs (no positional arguments).

If `query` is an `elasticsearch_dsl.Search` object, we convert it to a dict with `query.to_dict` before forwaring it to the underling pyelasticsearch API.

Parameters

- **query** – A string, dict or `elasticsearch_dsl.Search` object.
- **prettyprint_query** – If this is `True`, we prettyprint the query before executing it. Good for debugging.

refresh(**args*, ***kwargs*)
Wrapper around `pyelasticsearch.ElasticSearch.refresh()`.

Works exactly like the wrapped function.

delete_index(**args*, ***kwargs*)
Wrapper around `pyelasticsearch.ElasticSearch.delete_index()`.

Works exactly like the wrapped function.

delete(**args*, ***kwargs*)
Wrapper around `pyelasticsearch.ElasticSearch.delete()`.

Works exactly like the wrapped function.

get(**args*, ***kwargs*)
Wrapper around `pyelasticsearch.ElasticSearch.get()`.

Works exactly like the wrapped function.

wrapped_get(**args*, ***kwargs*)
Just like *get()*, but we return a *SearchResultItem* instead of the raw search response.

get_or_none(**args*, ***kwargs*)
Works like *get()*, but instead of raising an exception, we return `None` if the requested object does not exist.

wrapped_get_or_none(**args*, ***kwargs*)
Works like *wrapped_get()*, but instead of raising an exception, we return `None` if the requested object does not exist.

wrapped_search(**args*, ***kwargs*)
Just like *search()*, but we return a *SearchResultWrapper* instead of the raw search response.

paginated_search(*query*, *page_number=0*, *page_size=100*, *resultitemwrapper=None*, ***kwargs*)
Performs a search much like *wrapped_search()*, but limit the size of the result to `page_size`, and start the results at `page_number * page_size`.

Parameters

- **page_number** – The page number to retrieve.
- **page_size** – The size of each page.
- **query** – A query dict for `pyelasticsearch.ElasticSearch.search()`. We add the `size` and `from` keys to this dict (calculated from `page_number` and `page_size`.
- **resultitemwrapper** – Forwarded to *Paginator*.
- **kwargs** – Forwarded to *wrapped_search()* alon with `query`.

Returns The search results wrapped in a *Paginator*.

Return type *Paginator*

search_all(***kwargs*)
Get all documents in the index. Nice for testing and debugging of small datasets. Useless in production.

`**kwargs` are forwarded to *search()*, but the query argument is added automatically.

wrapped_search_all(***kwargs*)
Just like *search_all()*, but wraps the results in a *SearchResultWrapper*.

class ievv_opensource.ievv_elasticsearch.search.**SearchResultItem**(*search_hit*)
Bases: `object`

Wrapper around a single dictionary in the `hits.hits` list of the result returned by `pyelasticsearch.ElasticSearch.search()`.

id
Returns the value of the `_id` key of the search hit.

index
Returns the value of the `_index` key of the search hit.

score
Returns the value of the `_score` key of the search hit.

source
Returns the value of the `_source` key of the search hit.

doc_type
Returns the value of the `_type` key of the search hit.

class ievv_opensource.ievv_elasticsearch.search.**SearchResultWrapper**(*searchresult*)
Bases: `object`

An efficient wrapper around the data returned by `pyelasticsearch.ElasticSearch.search()`.

Parameters **searchresult** – Data returned by `pyelasticsearch.ElasticSearch.search()`.

total
Returns the total number of hits.

retrieved_hits_count
Returns the number of retrieved hits.

first()
Shortcut for getting the first search result as a *SearchResultItem*.

class ievv_opensource.ievv_elasticsearch.search.**Paginator**(*searchresultwrapper*, *page_number*, *page_size=100*, *resultitemwrapper=None*)

Bases: object

Paginator for *ievv_opensource.ievv_elasticsearch.search.SearchResultWrapper*.

The paginator counts the first page as 0, the second as 1, and so on.

Parameters

- **searchresultwrapper** – A *ievv_opensource.ievv_elasticsearch.search.SearchResultWrapper*.
- **page_number** – The current page number.
- **page_size** – Number of items per page.
- **resultitemwrapper** – A class/callable that takes a single item in the `searchresultwrapper` and does something with it before returning it when iterating the search result. Defaults to just returning the item as returned from `searchresultwrapper.__iter__`.

total_items
Returns the number of items in total in all pages.

number_of_items_in_current_page
Returns the number of items in the current page.

get_page_startindex(*pagenumber*)
Get the start index of the given `pagenumber`.

get_current_page_startindex()
Get the start index of the current page.

page_has_content(*pagenumber*)
Check if the given `pagenumber` is within the total number of items in the given `searchresultwrapper`.

Returns A boolean.

current_page_has_content()
Check if current page is within the total number of items in the given `searchresultwrapper`.

Returns A boolean.

has_next()
Check if we have a next page. Checks if the start index of the next page is lower than `searchresultwrapper.total`.

Returns A boolean.

has_previous()
Check if we have a previous page. Checks if the start index of the previous page is larger than 0.

Returns A boolean.

3.3.5 autoindex API

ievv_opensource.ievv_elasticsearch.autoindex.AbstractIndex	Base class for describing a search index.
ievv_opensource.ievv_elasticsearch.autoindex.AbstractDocument	Base class for indexable documents for *AbstractIndex*.
ievv_opensource.ievv_elasticsearch.autoindex.AbstractDictDocument	Extends *AbstractDocument* to make it easy to put dicts in the database.
ievv_opensource.ievv_elasticsearch.autoindex.Registry	Registry of *AbstractIndex* objects.
ievv_opensource.ievv_elasticsearch.autoindex.MockableRegistry	A non-singleton version of *Registry*.

This module defines a *Registry* of objects that takes care of automatically updating the search index when we detect changes to the data in a data store. The data store can be anything you like (Django ORM, MongoDB, ...) - our examples use Django ORM.

This is completely decoupled from the *ievv_opensource.ievv_elasticsearch.search* API.

class ievv_opensource.ievv_elasticsearch.autoindex.**AbstractDocumentMeta**

Bases: `type`

Metaclass for *AbstractDocument*.

class ievv_opensource.ievv_elasticsearch.autoindex.**AbstractDocument**

Bases: `object`

Base class for indexable documents for *AbstractIndex*.

doc_type = None

The document type to store this as in the index.

index_name = None

The name of the index this document belongs to. This is set by *AbstractIndex* __init__ using *set_index_name_for_all_document_classes()*.

get_document()

Get document for the `doc` argument of `pyelasticsearch.ElasticSearch.index_op()`.

get_id()

Get the ID to use for the indexed document. Defaults to `None`, which means that a new document will be added to the index.

get_parent_id()

Get the parent-child mapping parent ID document to use for the indexed document. This should only be overridden if you have a parent specified

Defaults to `None`, which means that no `parent` will be sent during indexing operations.

get_index_op_kwargs()

Get kwargs for `pyelasticsearch.ElasticSearch.index_op()`.

You should not need to override this. Override *get_document()*, `get_meta()` and *doc_type*.

classmethod get_mapping_properties()

Get the mapping properties for custom mappings for this document type. You only need to specify those mappings you do not want elasticsearch to create automatically.

If you do not have any mappings, return `None` (or do not override).

Examples

Simple example:

```
class MyDocument(autoindex.AbstractDocument):
    @classmethod
    def get_mapping_properties(cls):
        return {
            'slug': {
                'type': 'string',
                'index': 'not_analyzed'
            },
            'author': {
                'username': {
                    'type': 'string',
                    'index': 'not_analyzed'
                }
            }
        }
```

classmethod get_mapping_parent_type()
Get the type of the parent document for parent-child mapping.

Lets say you have a Movie document, and want to create a parent-child relationship from the Category document with doc_type `category` to the Movie. In the Movie document class, you would have to:

- Override this method and return `"category"`.
- *get_parent_id()* and return the ID of the category.

class ievv_opensource.ievv_elasticsearch.autoindex.**AbstractDictDocument**(*document*, *id*)
Bases: *ievv_opensource.ievv_elasticsearch.autoindex.AbstractDocument*

Extends *AbstractDocument* to make it easy to put dicts in the database.

Parameters

- **document** – A dict that pyelasticsearch can convert to JSON.
- **id** – The ElasticSearch id of the document. Set to `None` to autocreate one.

get_document()
Get document for the `doc` argument of `pyelasticsearch.ElasticSearch.index_op()`.

get_id()
Get the ID to use for the indexed document. Defaults to `None`, which means that a new document will be added to the index.

class ievv_opensource.ievv_elasticsearch.autoindex.**AbstractIndex**
Bases: `object`

Base class for describing a search index.

To register an index:

1. Create a subclass of `AbstractIndex` and implement *iterate_all_documents()* and override *document_classes*.
2. Register the index with *Registry*.

Examples

Minimal implementation for indexing a Django Product model:

```
from ievv_opensource.ievv_elasticsearch import searchindex

class ProductDocument(searchindex.AbstractDictDocument):
    doc_type = 'product'

class ProductIndex(searchindex.AbstractIndex):
    name = 'products'
    document_classes = [
        ProductDocument
    ]

    def iterate_all_documents(self):
        for product in Product.objects.iterator():
            yield ProductDocument({
                'name': product.name,
                'price': product.price
            }, id=product.pk)
```

If you want a more general search index of sellable items, you could do something like this:

```
from ievv_opensource.ievv_elasticsearch import searchindex

class ProductDocument(searchindex.AbstractDictDocument):
    doc_type = 'product'

class ServiceDocument(searchindex.AbstractDictDocument):
    doc_type = 'service'

class SellableItemIndex(searchindex.AbstractIndex):
    name = 'sellableitems'

    def iterate_all_documents(self):
        for product in Product.objects.iterator():
            yield ProductDocument({
                'name': product.name,
                'price': product.price,
                'quantity': product.quantity
            }, id=product.pk)
        for service in Service.objects.iterator():
            yield ServiceDocument({
                'name': service.name,
                'price': service.price,
            }, id=service.pk)
```

You could also move the document creation into the index document classes like this:

```
class ProductDocument(searchindex.AbstractDictDocument):
    doc_type = 'product'

    def __init__(self, product):
        self.product = product

    def get_id(self):
```

(continues on next page)

(continued from previous page)

```
        return self.product.id

    def get_document(self):
        return {
            'name': self.product.name,
            'price': self.product.price,
            'quantity': self.product.quantity
        }

class SellableItemIndex(searchindex.AbstractIndex):
    # ... same as above

    def iterate_all_documents(self):
        for product in Product.objects.iterator():
            yield ProductDocument(product)
        # ...
```

name = None
The name of the index. Must be set in subclasses.

bulk_index_docs_per_chunk = 500
The number of docs to index per chunk when bulk updating the index.

bulk_index_bytes_per_chunk = 10000
The number of bytes to index per chunk when bulk updating the index.

document_classes = []
The *AbstractDocument* classes used in this index. Can also be overridden via *get_document_classes()*.

set_index_name_for_all_document_classes()
Called by __init__ to set the *AbstractDocument.index_name* of all documents in *document_classes*.

create()
Create the index and put any custom mappings.

You should not need to override this, instead you should override *get_document_classes()* (and *AbstractDocument.get_mapping_properties()*), and *get_settings()*.

get_settings()
Override this to provide settings for `pyelasticsearch.ElasticSearch.create_index()` (which is called by *create()*.

get_document_classes()
Returns an iterable of the *AbstractDocument* classes used in this index. Defaults to *document_classes*.

get_document_classes_for_mapping()
Get the document classes for mapping. You normally do not have to override this - it only return *get_document_classes()* reversed. It is reversed because parent-child mappings have to be created in the child before the parent mapping can be created, but you normally want to index parents before children.

create_mappings()
Create mappings.

You should not need to override this, but instead you should override *get_document_classes()* (and *AbstractDocument.get_mapping_properties()*).

iterate_all_documents()
Iterate over all documents returning documents that are ready to be added to the index.

Returns An iterable of *AbstractDocument*.

iterate_important_documents()
Just like *iterate_all_documents()*, but just yield the most important documents in case of a complete search index wipeout/rebuild.

This is typically the newest and most important documents in the database.

Defaults to returning an empty list.

index_items(*index_documents*)
Index the given index_documents.

Iterates over the given `index_documents`, and send documents to *ievv_opensource.ievv_elasticsearch.search.Connection.bulk()* in batches of `IEVV_ELASTICSEARCH_INDEX_BATCH_SIZE` index_documents.

Parameters **index_documents** – An iterable of *AbstractDocument*.

register_index_update_triggers()
Override this to register behaviors that trigger updates to the index. This is typically something like this:

- Register one or more post_save signals that updates the index in realtime (be very careful with this since it can easily become a bottleneck).
- Register one or more post_save signals that updates the index via a Celery job or some other background queue.

Does nothing by default, so it is up to you to override it if you want to register any triggers.

delete_index()
Delete this index.

classmethod get_instance()
Get an instance of this class.

Use this instead of instanciating the class directly.

rebuild_index()
Rebuild this index completely.

Very useful when writing tests, but probably a bit less than optimal in production code/batch tasks unless you have a really small index. In production you should most likely want to create a management command to rebuild the index with the most recent/most important documents beeing indexed first.

class ievv_opensource.ievv_elasticsearch.autoindex.**Registry**
Bases: *ievv_opensource.utils.singleton.Singleton*

Registry of *AbstractIndex* objects.

Examples

First, define an index (see *AbstractIndex*).

Register the searchindex with the searchindex registry via an AppConfig for your Django app:

```
from django.apps import AppConfig
from ievv_opensource.ievv_elasticsearch import searchindex
```

(continues on next page)

(continued from previous page)

```
from myapp import elasticsearch_indexes


class MyAppConfig(AppConfig):
    name = 'myapp'

    def ready(self):
        searchindex.Registry.get_instance().add(elasticsearch_indexes.
↪SellableItemIndex)
```

add(*searchindex_class*)
Add the given `searchindex_class` to the registry.

get(*indexname*)
Get the index named `indexname`.

Returns: An *AbstractIndex* or None if no index matching the given `indexname` is found.

get_indexnames()
Get a view with the names of all indexes.

class ievv_opensource.ievv_elasticsearch.autoindex.**MockableRegistry**
Bases: *ievv_opensource.ievv_elasticsearch.autoindex.Registry*

A non-singleton version of *Registry*. For tests.

Typical usage in a test:

```
class MockSearchIndex(searchindex.AbstractIndex):
    name = 'myindex'
    # ...

mockregistry = searchindex.MockableRegistry()
mockregistry.add(searchindex.MockSearchIndex())

with mock.patch('ievv_opensource.ievv_elasticsearch.searchindex.Registry.get_
↪instance',
                lambda: mockregistry):
    pass  # ... your code here ...
```

3.3.6 jsondecode API

Utilities for decoding the JSON returned by ElasticSearch.

ievv_opensource.ievv_elasticsearch.jsondecode.**datetime**(*datetimestring*)
Convert a datetime object from ElasticSearch (iso format) into a timezone-aware `datetime.datetime` object.

3.3.7 viewhelpers API

ievv_opensource.ievv_elasticsearch.viewhelpers.searchview.ViewMixin	Makes it a bit easier to search with paging.
ievv_opensource.ievv_elasticsearch.viewhelpers.searchview.View	A Django TemplateView with *ViewMixin*.

Continued on next page

Table 3 – continued from previous page

ievv_opensource.ievv_elasticsearch.viewhelpers.searchview.SortMixin	Mixin class for sort-keyword in the querystring based sort (E.g.: `?o=name`).
ievv_opensource.ievv_elasticsearch.viewhelpers.searchview.SearchMixin	Mixin class that makes it slightly easier to add search via a querystring attribute (defaults to `?s=<search_string>`).

class ievv_opensource.ievv_elasticsearch.viewhelpers.searchview.**ViewMixin**

Bases: `object`

Makes it a bit easier to search with paging.

Examples

Minimal example:

```
class MyView(TemplateView, searchview.ViewMixin):
    template_name = 'myapp/mytemplate.html'

    def get_search_query(self):
        return {
            'match_all': {}
        }

    def get_context_data(self, **kwargs):
        context = super().get_context_data(**kwargs)
        context['searchpaginator'] = self.get_paginator()
```

A more full featured example:

```
class MyView(TemplateView, searchview.ViewMixin):
    template_name = 'myapp/mytemplate.html'
    page_size = 30
    paging_querystring_attribute = 'page'

    def get_search_query(self):
        return {
            'match_all': {}
        }

    def get_search_sort(self):
        return {'name': {'order': 'asc'}}

    def get_context_data(self, **kwargs):
        context = super().get_context_data(**kwargs)
        context['searchpaginator'] = self.get_paginator()
```

You should normally just need to override:

- *get_search_query()*.
- *get_search_sort()* (optional).
- *get_page_size()* or *page_size* (optional).
- *get_resultitemwrapper()* (optional).
- *get_paging_querystring_attribute()* or *paging_querystring_attribute* (optional).

And call *get_paginator()* to retrieve results that you can iterate and ask for pagination information.

page_size = 100
The number of items per page. Defaults to `100`. See *get_page_size()*.

search_index = None
See *get_search_index()*. Defaults to `None`.

search_doc_type = None
See *get_search_doc_type()*. Defaults to `None`.

paging_querystring_attribute = 'p'
The querystring attribute to use for paging. Defaults to `p`. Used by *get_paging_querystring_attribute()*.

get_page_size()
Get the page size.

Defaults to returning *page_size*.

get_paging_querystring_attribute()
The querystring attribute to use for paging. Defaults to *paging_querystring_attribute*.

get_current_page_number()
Get the current page number from `request.GET[self.get_paging_querystring_attribute()]`.

You can override this if you want to get the page number some other way.

get_search_index()
Get the search index name.

Defaults to *search_index*.

get_search_doc_type()
Get the document types to search. Can be a single document type or an iterable of document types.

Defaults to *search_doc_type*.

get_search_query()
Get the query attribute of the elasticsearch query.

You MUST override this.

While *get_search_full_query()* returns something like:

```
{
    'query': {
        'match_all': {}
    },
    'size': 20,
    'from': 40
}
```

This method should only return the value of the `query` key:

```
{
    'match_all': {}
}
```

get_search_sort()
Get the sort dict for the search query.

While *get_search_full_query()* returns something like:

```
{
    'query': {
        'match_all': {}
    },
    'sort': {'name': {'order': 'asc'}},
    'size': 20,
    'from': 40
}
```

This method should only return the value of the `sort` key:

```
{'name': {'order': 'asc'}}
```

Defaults to `None`, which means no sorting is performed.

get_search_full_query()
Builds the full ElasticSearch query dict including paging.

You should normally not override this directly. Override *get_search_query()* and *get_search_sort()* instead.

get_paginated_search_kwargs()
Get the kwargs for `ievv_opensource.ievv_elasticsearch.search.Connection#paginated_search()`.

get_resultitemwrapper()
See the `resultitemwrapper` argument for *ievv_opensource.ievv_elasticsearch.search.Paginator*.

get_paginator()
Performs the search and wraps it in a *ievv_opensource.ievv_elasticsearch.search.Paginator*.

Raises

- `django.http.response.Http404` if the search does not match
- any items. You will typically catch this exception and
- show a message in a normal search setting.

class ievv_opensource.ievv_elasticsearch.viewhelpers.searchview.**View**(***kwargs*)
Bases: `django.views.generic.base.TemplateView`, *ievv_opensource.ievv_elasticsearch.viewhelpers.searchview.ViewMixin*

A Django TemplateView with *ViewMixin*.

For usage example, see *ViewMixin* (just inherit from this class instead of TemplateView and ViewMixin)

Constructor. Called in the URLconf; can contain helpful extra keyword arguments, and other things.

class ievv_opensource.ievv_elasticsearch.viewhelpers.searchview.**SortMixin**
Bases: `object`

Mixin class for sort-keyword in the querystring based sort (E.g.: `?o=name`).

This MUST be mixed in before *View* (or *ViewMixin*), since it overrides *ViewMixin.get_search_sort()*.

Examples

Simple example of a map where `?o=name` sorts by name ascending and `?o=created` sorts by created datetime descending:

```
class MySortView(searchview.SortMixin, searchview.View):
    default_sort_keyword = 'name'

    sort_map = {
        'name': {'name': {'order': 'asc'}},
        'created': {'created_datetime': {'order': 'desc'}},
    }

    def get_search_query(self):
        return {
            'match_all': {}
        }

    def get_context_data(self, **kwargs):
        context = super().get_context_data(**kwargs)
        context['searchpaginator'] = self.get_paginator()
```

The default if `?o` is not specified will be to sort by name ascending.

You should normally just need to override:

- *get_sort_map()* or *sort_map*
- *get_default_sort_keyword()* or *default_sort_keyword*.

If you do not get the sort keyword from the querystring, you also need to override *get_sort_keyword()*.

If you do not want to use `o` as the querystring attribute for sort keywords, you need to override *get_sort_querystring_attribute()* or *sort_querystring_attribute*.

sort_querystring_attribute = 'o'
The querystring attribute to use for sort. Used by *get_sort_querystring_attribute()*. Defaults to `o` (for ordering). We use `o` instead of `s` to avoid collision with *SearchMixin*.

default_sort_keyword = 'default'
The default sort keyword to use when ordering is not specified in the querystring. Defaults to `"default"`.

sort_map = {}
See *get_sort_map()*.

get_sort_querystring_attribute()
The querystring attribute to use for sort. Defaults to *sort_querystring_attribute*.

get_default_sort_keyword()
Get the default sort keyword to use when ordering is not specified in the querystring. Defaults to *default_sort_keyword*.

get_sort_keyword()
Get the sort keyword. Defaults to getting it from the querystring argument defined by *get_sort_querystring_attribute()*, and falling back to *get_default_sort_keyword()*.

get_sort_map()
Get a mapping object that maps keywords to sort dicts compatible with elasticsearch. Defaults to *sort_map*.

get_search_sort_by_keyword(*keyword*)
Get the elasticsearch sort dict by looking up the given `keyword` in *get_sort_map()*.

If the given `keyword` is not in *get_sort_map()*, we fall back on *get_default_sort_keyword()*.

get_search_sort()
Overrides *ViewMixin.get_search_sort()* and gets the value of the sort dict by via *get_search_sort_by_keyword()* with *get_sort_keyword()* as the keyword argument.

This means that you should not override this method, but instead override:

- *get_sort_map()* (or *sort_map*)
- *get_default_sort_keyword()* (or *default_sort_keyword*)

class ievv_opensource.ievv_elasticsearch.viewhelpers.searchview.**SearchMixin**
Bases: object

Mixin class that makes it slightly easier to add search via a querystring attribute (defaults to `?s=<search_string>`).

This is perfect for views that use ElasticSearch for traditional search where a user types in a string, and we wish to search some fields for that string.

This MUST be mixed in before *View* (or *ViewMixin*), since it overrides *ViewMixin.get_search_query()*.

Can safely be used with *SortMixin*. The order of *SortMixin* and *SearchMixin* does not matter, but they must both be mixed in before *View* (or *ViewMixin*).

Examples

Minimal example:

```
class MySearchView(searchview.SearchMixin, searchview.View):
    search_query_fields = ['name', 'email']

    def get_context_data(self, **kwargs):
        context = super().get_context_data(**kwargs)
        context['searchpaginator'] = self.get_paginator()
```

search_querystring_attribute = 's'
The querystring attribute to use for sort. Used by *get_sort_querystring_attribute()*. Defaults to `s`.

search_query_fields = []
List of fields for *get_search_query_fields()*. Defaults to empty list, so you need to set this, or override *get_search_query_fields()*.

get_search_querystring_attribute()
The querystring attribute to use for search. Defaults to *search_querystring_attribute*.

clean_search_string(*search_string*)
Can be overridden to clean/modify the search_string.

Does nothing by default.

Used by *get_search_string()*.

get_search_string()
Get the search string.

We get the search string from the querystring, and clean it with *clean_search_string()* before returning it.

get_search_query_fields()
Get the fields for the `multi_match` query performed by *get_search_query_with_search_string()*.

Defaults to *search_query_fields*

get_search_query_with_search_string(*search_string*)
Called by *get_search_query()* when *get_search_string()* returns something.

Defaults to a `multi_matach` query with the fields returned by *get_search_query_fields()*.

You will not need to override this for simple cases, but for more complex queries with boosting, filtering, etc. you will most likely have to override this.

get_search_query_without_search_string()
Called by *get_search_query()* when *get_search_string()* returns empty string or `None`.

Defaults to a `match_all` query.

get_search_query()
Overrides *ViewMixin.get_search_query()* and splits the logic into two separate states:

1. We have a search string, call *get_search_query_with_search_string()*.
2. We do not have a search string, call *get_search_query_without_search_string()*.

You should not need to override this method, but instead override:

- `get_search_fields()` or perhaps *get_search_query_with_search_string()* (for advanced cases).
- Perhaps *get_search_query_without_search_string()*.

3.4 *ievv_customsql* — Framework for adding custom SQL to Django apps

The intention of this module is to make it easier to add and update custom SQL in a Django app. We do this by providing a registry of all custom SQL, and a management command to add and update custom SQL and any refresh data that is maitained by triggers.

3.4.1 Configuration

Add the following to your `INSTALLED_APPS`-setting:

```
'ievv_opensource.ievv_customsql'
```

3.4.2 Add custom SQL to your app

Create the class containing your custom SQL

First you need to create a subclass of *AbstractCustomSql*.

Lets say you have a Person model with name and description. You want to maintain a fulltext search vector to efficiently search the person model. You would then create a subclass of AbstractCustomSql that looks something like this:

```
from ievv_opensource.ievv_customsql import customsql_registry


class PersonCustomSql(customsql_registry.AbstractCustomSql):
    def initialize(self):
        self.execute_sql("""

            -- Add search_vector column to the Person model
            ALTER TABLE myapp_person DROP COLUMN IF EXISTS search_vector;
            ALTER TABLE myapp_person ADD COLUMN search_vector tsvector;

            -- Function used to create the search_vector value both in the trigger,
            -- and in the UPDATE statement (in recreate_data()).
            CREATE OR REPLACE FUNCTION myapp_person_get_search_vector_value(param_
↪table myapp_person)
            RETURNS tsvector AS $$
            BEGIN
                RETURN setweight(to_tsvector(param_table.name), 'A') ||
                    setweight(to_tsvector(param_table.description), 'C');
            END
            $$ LANGUAGE plpgsql;

            -- Trigger function called on insert or update to keep the search_vector␣
↪column
            -- in sync.
            CREATE OR REPLACE FUNCTION myapp_person_set_search_vector() RETURNS␣
↪trigger AS $$
            BEGIN
                NEW.search_vector := myapp_person_get_search_vector_value(NEW);
              return NEW;
            END
            $$ LANGUAGE plpgsql;

            DROP TRIGGER IF EXISTS myapp_person_set_search_vector_trigger ON myapp_
↪person;
            CREATE TRIGGER myapp_person_set_search_vector_trigger BEFORE INSERT OR␣
↪UPDATE
                ON myapp_person FOR EACH ROW
                EXECUTE PROCEDURE myapp_person_set_search_vector();
        """)

    def recreate_data(self):
        self.execute_sql("""
            UPDATE myapp_person SET
                search_vector = myapp_person_get_search_vector_value(myapp_person);
        """)
```

You can put this code anywhere in your app, but the recommended location is to put it in a file named `customsql.py` in the root of your app.

Add your custom SQL to the registry

Next, you need to register your PersonCustomSql class in the registry. Create an AppConfig for your app with the following code:

```
from django.apps import AppConfig

from ievv_opensource.ievv_customsql import customsql_registry
from myproject.myapp.customsqldemo.customsql import PersonCustomSql


class CustomSqlDemoAppConfig(AppConfig):
    name = 'myproject.myapp'
    verbose_name = "My APP"

    def ready(self):
        registry = customsql_registry.Registry.get_instance()
        registry.add('myapp', PersonCustomSql)
```

Using your custom SQL

During development and as part of production releases, you use the `ievvtasks_customsql` command to update your custom SQL. Run the following to execute both:

- *initialize()*
- *recreate_data()*

for all the custom SQL classes in the registry:

```
$ python manage.py ievvtasks_customsql -i -r
```

Since this is an ievvtasks command, you can also run it as:

```
$ ievv customsql -i -r
```

Writing tests using your custom SQL

The custom SQL is not added automatically, so you need to use it explicitly in your tests. You have three choices:

1. Call `PersonCustomSql().initialize()` in your setUp() method, or in your test method(s). You will probably also want to call `PersonCustomSql().recreate_data()` when required. This is **normally the recommented method**, since it provides the largest amount of control. See *AbstractCustomSql.initialize()* and *AbstractCustomSql.recreate_data()* for more info.
2. Call `ievv_customsql.Registry.get_instance().run_all_in_app('myapp')`. This may be useful to test views and other code that require all the custom SQL in your app. See *Registry.run_all_in_app()* for more info.
3. Call `ievv_customsql.Registry.get_instance().run_all()`. This is **not recommended** because it runs SQL from ALL apps in `INSTALLED_APPS`. See See *Registry.run_all()* for more info.

Example of using option (1) to create a TestCase:

```
class TestPersonCustomSql(test.TestCase):
    def test_add_person_and_search(self):
        PersonCustomSql().initialize()
        jack = mommy.make('myapp.Person', name='Jack The Man', description='Also␣
↪called john by some.')
        mommy.make('myapp.Person', name='NoMatch Man')
        john = mommy.make('myapp.Person', name='John Peterson', description='Hello␣
↪world')

        tsquery = 'john'
        queryset = Person.objects.extra(
            select={
                'rank': 'ts_rank_cd(search_vector, to_tsquery(%s))',
            },
            select_params=[tsquery],
            where=['search_vector @@ to_tsquery(%s)'],
            params=[tsquery],
            order_by=['-rank']
        )
        self.assertEqual([john, jack], list(queryset))
```

Demo

See `ievv_opensource/demo/customsqldemo/` for a full demo of everything explained above.

3.4.3 API

AbstractCustomSql

class ievv_opensource.ievv_customsql.customsql_registry.**AbstractCustomSql**(*appname=None*)

Bases: `object`

Defines custom SQL that can be executed by the `ievv_customsql` framework.

You typically override *`initialize()`* and use *`execute_sql()`* to add triggers and functions, and override *`recreate_data()`* to rebuild the data maintained by the triggers, but many other use-cases are also possible.

Parameters **appname** – Not required - it is added automatically by *`Registry`*, and used by `__str__` `()` for easier debugging / prettier output.

execute_sql(*sql*)

Execute the provided SQL.

Parameters **sql** (*`str`*) – String of SQL.

get_sql_from_file(*path*)

Get SQL from a file as a string.

Parameters **path** (*`str`*) – A path relative to a directory named the same as the module where this class is located without the filename extension and suffixed with `_sqlcode`.

So if the file with this class is in `path/to/customsql.py`, path will be relative to `path/to/customsqlcode/`.

execute_sql_from_file(*path*)

Execute SQL from the provided file.

Parameters **path** – See *`get_sql_from_file()`*.

execute_sql_from_files(*paths*)
Shortcut for calling *execute_sql_from_file()* with multiple files.

Calls *execute_sql_from_file()* once for each file in `paths`.

Parameters **paths** (*list*) – A list of paths. See *get_sql_from_file()* for the format of each path.

execute_sql_from_template_file(*path*, *context_data=None*)
Execute SQL from the provided Django template file.

The SQL is retrieved from the file, then processed as a Django template with the provided `context_data`, and the result is executed using *execute_sql()*.

Parameters

- **path** – See *get_sql_from_file()*.
- **context_data** (*dict*) – Template context data. Can be `None`.

execute_sql_from_template_files(*paths*, *context_data=None*)
Shortcut for calling *execute_sql_from_template_file()* with multiple files.

Calls *execute_sql_from_template_file()* once for each file in `paths`.

Parameters

- **paths** (*list*) – A list of paths. See *get_sql_from_file()* for the format of each path.
- **context_data** (*dict*) – Forwarded to *execute_sql_from_template_file()*.

initialize()
Code to initialize the custom SQL.

You should create triggers, functions, columns, indexes, etc. in this method, using *execute_sql()*, or using plain Django code.

Make sure to write everything in a manner that updates or creates everything in a self-contained manner. This method is called both for the first initialization, and to update code after updates/changes.

Must be overridden in subclasses.

clear()
Revert *initialize()* and remove any data created by the triggers.

Drop/delete triggers, functions, columns, indexes, etc. created in *initialize()*.

recreate_data()
Recreate all data that any triggers created in *initialize()* would normally keep in sync automatically.

Can not be used unless *initialize()* has already be run (at some point). This restriction is here to make it possible to create SQL functions in *initialize()* that this method uses to recreate the data. Without this restriction, code-reuse between *initialize()* and this function would be very difficult.

run()
Run both *initialize()* and *recreate_data()*.

Registry

class ievv_opensource.ievv_customsql.customsql_registry.**Registry**
Bases: *ievv_opensource.utils.singleton.Singleton*

Registry of *AbstractCustomSql* objects.

Examples

First, define a subclass of *AbstractCustomSql*.

Register the custom SQL class with the registry via an AppConfig for your Django app:

```
from django.apps import AppConfig
from ievv_opensource.ievv_customsql import customsql_registry
from myapp import customsql

class MyAppConfig(AppConfig):
    name = 'myapp'

    def ready(self):
        customsql_registry.Registry.get_instance().add(customsql.MyCustomSql)
```

See `ievv_opensource/demo/customsql/apps.py` for a complete demo.

add(*appname*, *customsql_class*)
Add the given `customsql_class` to the registry.

Parameters

- **appname** – The django appname where the `customsql_class` belongs.
- **customsql_class** – A subclass of *AbstractCustomSql*.

iter_appnames()
Returns an iterator over all the appnames in the registry. Each item in the iterator is an appname (a string).

iter_customsql_in_app(*appname*)
Iterate over all *AbstractCustomSql* subclasses registered in the provided appname. The yielded values are objects of the classes initialized with no arguments.

run_all_in_app(*appname*)
Loops through all the *AbstractCustomSql* classes registered in the registry with the provided `appname`, and call *AbstractCustomSql.run()* for each of them.

run_all()
Loops through all the *AbstractCustomSql* classes in the registry, and call *AbstractCustomSql.run()* for each of them.

MockableRegistry

class ievv_opensource.ievv_customsql.customsql_registry.**MockableRegistry**
Bases: *ievv_opensource.ievv_customsql.customsql_registry.Registry*

A non-singleton version of *Registry*. For tests.

Typical usage in a test:

```
from ievv_opensource.ievv_customsql import customsql_registry

class MockCustomSql(customsql_registry.AbstractCustomSql):
    # ...

mockregistry = customsql_registry.MockableRegistry()
mockregistry.add(MockCustomSql())
```

(continues on next page)

(continued from previous page)

```
with mock.patch('ievv_opensource.ievv_customsql.customsql_registry.Registry.get_
↪instance',
                lambda: mockregistry):
    pass  # ... your code here ...
```

3.5 *ievv_jsbase* — Javascript library with general purpose functionality

3.5.1 Javascript API docs

The API docs for the javascript library is available here: IEVV jsbase API.

3.6 *ievv_developemail* — Develop mail backend that lets you view mails in django admin

3.6.1 Setup

Add it to `INSTALLED_APPS` setting, and set the `EMAIL_BACKEND` setting (typically only in develop settings):

```
INSTALLED_APPS = [
    # ...
    'ievv_opensource.ievv_developemail',
]

EMAIL_BACKEND = 'ievv_opensource.ievv_developemail.email_backend.DevelopEmailBackend'
```

Migrate:

```
$ python manage.py migrate
```

You should now get new emails both logged to the terminal, and added as a DevelopEmail object in the database which you can browse in Django admin.

3.6.2 How it works

We have a custom email backend that sends emails to the DevelopEmail database model (and to log).

3.6.3 Management script for sending test emails

We provide the `ievv_developemail_send_testmail` management script for sending test emails. It can be useful just to check that emails are sent, but also for testing `cradmin_email` styling etc.

In its simplest form:

```
$ python manage.py ievv_developemail_send_testmail --to "test@example.com"
```

The same, but with an HTML message:

```
$ python manage.py ievv_developemail_send_testmail --to "test@example.com" --html
```

With a custom message instead of the default lorem ipsum message:

```
$ python manage.py ievv_developemail_send_testmail --to "test@example.com" --html --
↪message-html "Dette er <em>en test lizzm</em>"
```

Send using django_cradmin.apps.cradmin_email.emailutils.AbstractEmail:

```
$ python manage.py ievv_developemail_send_testmail --to "test@example.com" --html --
↪use-cradmin-email
.. or with custom message ..
$ python manage.py ievv_developemail_send_testmail --to "test@example.com" --html  --
↪use-cradmin-email --message-html "Dette er <em>en test lizzm</em>"
```

From email can be set too! Defaults to the DEFAULT_FROM_EMAIL setting:

```
$ python manage.py ievv_developemail_send_testmail --to "test@example.com" --from
↪"from@example.com"
```

3.7 *ievv_model_mommy_extras* — Add support for more fields types to model-mommy

Model-mommy does not support all the fields in Django. To remidy this, we provide the `ievv_opensource.ievv_model_mommy_extras`.

For now it only adds support for:

```
- django.contrib.postgres.fields.ranges.DateTimeRangeField
- django.contrib.postgres.fields.ranges.DateRangeField
```

3.7.1 Setup

To use this, you only need to add it to your `INSTALLED_APPS` setting:

```
INSTALLED_APPS = [
    # Other apps ...
    'ievv_opensource.ievv_model_mommy_extras'
]
```

Note: You only need this for testing, so if you have split out your test settings, you should only add it to `INSTALLED_APPS` there.

3.8 *ievv_restframework_helpers* — Helpers for working with Django rest framework

3.8.1 FullCleanModelSerializer

class ievv_opensource.ievv_restframework_helpers.full_clean_model_serializer.**FullCleanModel**

Bases: `rest_framework.serializers.ModelSerializer`

A `rest_framework.serializers.ModelSerializer` subclass that calls full_clean() on the model instance before saving.

clean_model_instance(*instance*)
Clean the model instance (full_clean()) and ensures any `django.core.exceptions.ValidationError` raised is re-raised as a `rest_framework.exceptions.ValidationError`.

create(*validated_data*)
We have a bit of extra checking around this in order to provide descriptive messages when something goes wrong, but this method is essentially just:

return ExampleModel.objects.create(**validated_data)

If there are many to many fields present on the instance then they cannot be set until the model is instantiated, in which case the implementation is like so:

example_relationship = validated_data.pop('example_relationship') instance = ExampleModel.objects.create(**validated_data) instance.example_relationship = example_relationship return instance

The default implementation also does not handle nested relationships. If you want to support writable nested relationships you'll need to write an explicit *.create()* method.

3.9 *ievv_sms* — SMS sending - multiple backends supported

3.9.1 Setup

Add it to your `INSTALLED_APPS` setting:

```
INSTALLED_APPS = [
    # Other apps ...
    'ievv_opensource.ievv_sms'
]
```

For development, you probably also want to use the *ievv_opensource.ievv_sms.backends.debugprint.Backend* backend. You configure that with the following setting:

```
IEVV_SMS_DEFAULT_BACKEND_ID = 'debugprint'
```

3.9.2 Sending SMS

Send an SMS using the default backend with:

```
from ievv_opensource.ievv_sms.sms_registry import send_sms
send_sms(phone_number='12345678', message='This is a test')
```

Send using another backend using the `backend_id` argument:

```
from ievv_opensource.ievv_sms.sms_registry import send_sms
send_sms(phone_number='12345678', message='This is a test',
         backend_id='some_backend_id')
```

See *ievv_opensource.ievv_sms.sms_registry.send_sms()* for more details.

3.9.3 Creating a custom backend

See the example in *AbstractSmsBackend*.

3.9.4 Specifying the default backend

Just set the backend ID (see *get_backend_id()*) of a backend in the *IEVV_SMS_DEFAULT_BACKEND_ID* setting.

3.9.5 Core API

send_sms()

ievv_opensource.ievv_sms.sms_registry.**send_sms**(*phone_number*, *message*, *backend_id=None*, ***kwargs*)

Send SMS message.

Just a shortcut for *Registry.send()* (`Registry.get_instance().send(...)`).

Parameters

- **phone_number** (*str*) – The phone number to send the message to.
- **message** (*str*) – The message to send.
- **backend_id** (*str*) – The ID of the backend to use for sending. If this is `None`, we use the default backend (see *get_default_backend_id()*).
- ****kwargs** – Extra kwargs for the *AbstractSmsBackend* constructor.

Returns An instance of a subclass of *AbstractSmsBackend*.

Return type *AbstractSmsBackend*

AbstractSmsBackend

class ievv_opensource.ievv_sms.sms_registry.**AbstractSmsBackend**(*phone_number*, *message*, ***kwargs*)

Bases: `object`

Base class for SMS backends.

An instance of this class is created each time an SMS is created.

This means that you can store temporary information related to building the SMS on `self`.

Example (simple print SMS backend):

```
class PrintBackend(sms_registry.AbstractSmsBackend):
    @classmethod
    def get_backend_id(cls):
        return 'print'

    def send(self):
        print(
            'Phone number: {phone_number}. '
            'Message: {message}'.format(
                phone_number=self.cleaned_phone_number,
                message=self.cleaned_message
            )
        )
```

To use the PrintBackend, add it to the registry via an AppConfig for your Django app:

```
from django.apps import AppConfig

class MyAppConfig(AppConfig):
    name = 'myapp'

    def ready(self):
        from ievv_opensource.ievv_sms import sms_registry
        from myapp import sms_backends
        sms_registry.Registry.get_instance().add(sms_backends.PrintBackend)
```

Now you can use the backend to send an SMS:

```
from ievv_opensource.ievv_sms import sms_registry
sms_registry.Registry.get_instance().send(
    phone_number='12345678',
    message='This is a test',
    backend_id='print')
```

You can also set the backend as the default backend for SMS sending by adding `IEVV_SMS_DEFAULT_BACKEND_ID = 'print'` to your django settings. With this setting you can call send() without the `backend_id` argument, and the SMS will be sent with the print backend.

All the arguments are forwarded from *Registry.send()* / *Registry.make_backend_instance()*.

Parameters

- **phone_number** (*str*) – The phone number to send the message to.
- **message** (*str*) – The message to send.
- ****kwargs** – Extra kwargs. Both for future proofing, and to make it possible for backends to support extra kwargs.

classmethod get_backend_id()

The ID this backend will get get in the *Registry* singleton.

Defaults to the full python path for the class.

classmethod validate_backend_setup()

Validate backend setup (required settings, etc).

Raises `SmsBackendSetupError` – When the setup validation fails.

clean_phone_number(*phone_number*)
Clean/validate the phone number.

By default this does nothing. It is here for backends that need to format phone numbers in a specific way.

Parameters **phone_number** (*str*) – The phone number to clean.

Raises django.core.exceptions.ValidationError – If validation of the phone number fails.

Returns The cleaned phone number.

Return type str

clean_message(*message*)
Clean/validate the message.

By default this does nothing. It is here for backends that need to format or validate the message in a specific way.

Parameters **message** (*str*) – The message to clean.

Raises django.core.exceptions.ValidationError – If validation of the message fails.

Returns The cleaned message.

Return type str

clean()
Clean the phone number, message, and kwargs.

Calls *clean_phone_number()* and *clean_message()*.

If you need to clean extra kwargs, you should override this method, but make sure you call `super().clean()`.

Raises django.core.exceptions.ValidationError – If validation fails.

send()
Send the message.

Must be overridden in subclasses.

Should send `self.cleaned_message` to `self.cleaned_phone_number`.

Registry

class ievv_opensource.ievv_sms.sms_registry.**Registry**
Bases: *ievv_opensource.utils.singleton.Singleton*

Registry of *AbstractSmsBackend* objects.

add(*backend_class*)
Add the given `backend_class` to the registry.

Parameters **backend_class** – A subclass of *AbstractSmsBackend*.

remove_by_backend_id(*backend_id*)
Remove the backend class with the provided `backend_id` from the registry.

remove(*backend_class*)
Remove the provided backend class from the registry.

iter_backend_classes()
Returns an iterator over all backend classes in the registry.

get_default_backend_id()
Get the default backend ID.

Retrieved from the *IEVV_SMS_DEFAULT_BACKEND_ID* setting.

Defaults to `debugprint` if the setting is not defined, or if it boolean False (None, empty string, ...).

get_backend_class_by_id(*backend_id*)
Get backend class by ID.

Parameters **backend_id** (*str*) – The backend ID. If this is `None`, we use the default backend (see *get_default_backend_id()*)

make_backend_instance(*phone_number*, *message*, *backend_id=None*, ***kwargs*)
Make a backend instance. Does not send the message.

Parameters

- **phone_number** (*str*) – The phone number to send the message to.
- **message** (*str*) – The message to send.
- **backend_id** (*str*) – The ID of the backend to use for sending. If this is `None`, we use the default backend (see *get_default_backend_id()*).
- ****kwargs** – Extra kwargs for the *AbstractSmsBackend* constructor.

Returns An instance of a subclass of *AbstractSmsBackend*.

Return type *AbstractSmsBackend*

send(*phone_number*, *message*, *backend_id=None*, ***kwargs*)
Send an SMS message.

Shortcut for `make_backend_instance(...).send()`.

See *make_backend_instance()*.

Parameters

- **phone_number** – See *make_backend_instance()*.
- **message** – See *make_backend_instance()*.
- **backend_id** – See *make_backend_instance()*.
- ****kwargs** – See *make_backend_instance()*.

Raises `django.core.exceptions.ValidationError` – If validation of the phone number, message or kwargs fails.

Returns An instance of a subclass of *AbstractSmsBackend*.

Return type *AbstractSmsBackend*

3.9.6 Backends

Debug/develop backends

class ievv_opensource.ievv_sms.backends.debugprint.**Backend**(*phone_number*, *message*, ***kwargs*)
Bases: *ievv_opensource.ievv_sms.sms_registry.AbstractSmsBackend*

Backend that just prints the phone number and message. Does not send an SMS.

Useful for development.

The backend_id for this backend is `debugprint`.

All the arguments are forwarded from *Registry.send()* / *Registry.make_backend_instance()*.

Parameters

- **phone_number** (*str*) – The phone number to send the message to.
- **message** (*str*) – The message to send.
- ****kwargs** – Extra kwargs. Both for future proofing, and to make it possible for backends to support extra kwargs.

classmethod get_backend_id()
The ID this backend will get get in the *Registry* singleton.

Defaults to the full python path for the class.

send()
Send the message.

Must be overridden in subclasses.

Should send `self.cleaned_message` to `self.cleaned_phone_number`.

class ievv_opensource.ievv_sms.backends.debugprint.**Latin1Backend**(*phone_number*, *message*, ***kwargs*)
Bases: *ievv_opensource.ievv_sms.backends.debugprint.Backend*

Just like *Backend*, but this backend crashes if you try to send any characters outside the latin1 charset.

The backend_id for this backend is `debugprint_latin1`.

Very useful when the targeted production backend only supports latin1.

All the arguments are forwarded from *Registry.send()* / *Registry.make_backend_instance()*.

Parameters

- **phone_number** (*str*) – The phone number to send the message to.
- **message** (*str*) – The message to send.
- ****kwargs** – Extra kwargs. Both for future proofing, and to make it possible for backends to support extra kwargs.

classmethod get_backend_id()
The ID this backend will get get in the *Registry* singleton.

Defaults to the full python path for the class.

clean_message(*message*)
Clean/validate the message.

By default this does nothing. It is here for backends that need to format or validate the message in a specific way.

Parameters **message** (*str*) – The message to clean.

Raises `django.core.exceptions.ValidationError` – If validation of the message fails.

Returns The cleaned message.

Return type str

PSWIN backend

class ievv_opensource.ievv_sms.backends.pswin.**Backend**(*phone_number*, *message*, *pswin_sender=None*, *pswin_username=None*, *pswin_password=None*, *pswin_default_country_code=None*, ***kwargs*)

Bases: *ievv_opensource.ievv_sms.sms_registry.AbstractSmsBackend*

A pswin (https://pswin.com) backend.

To use this backend, you should set the following Django settings:

- `PSWIN_USERNAME`: The pswin username (USER).
- `PSWIN_PASSWORD`: The pswin password (PWD).
- `PSWIN_SENDER`: The pswin sender (SND).
- `PSWIN_DEFAULT_COUNTRY_CODE`: The country code to use if received phone numbers do not start with `+<code>` or `00<code>`. E.g.: 47 for norway.

If you do not set these settings, you must include them as kwargs each time you send a message, and this makes it hard to switch SMS sending provider.

The backend_id for this backend is `pswin`.

classmethod get_backend_id()
The ID this backend will get get in the *Registry* singleton.

Defaults to the full python path for the class.

clean_message(*message*)
Clean/validate the message.

By default this does nothing. It is here for backends that need to format or validate the message in a specific way.

Parameters **message** (*str*) – The message to clean.

Raises `django.core.exceptions.ValidationError` – If validation of the message fails.

Returns The cleaned message.

Return type str

clean_phone_number(*phone_number*)
Clean/validate the phone number.

By default this does nothing. It is here for backends that need to format phone numbers in a specific way.

Parameters **phone_number** (*str*) – The phone number to clean.

Raises `django.core.exceptions.ValidationError` – If validation of the phone number fails.

Returns The cleaned phone number.

Return type str

send()
Send the message using pswin.

DebugSmsMessage backend

class ievv_opensource.ievv_sms.backends.debug_dbstore.**Backend**(*phone_number*, *message*, ***kwargs*)

Bases: *ievv_opensource.ievv_sms.sms_registry.AbstractSmsBackend*

Backend that creates DebugSmsMessage object

Useful for development.

The backend_id for this backend is `debug_dbstore`.

All the arguments are forwarded from *Registry.send()* / *Registry.make_backend_instance()*.

Parameters

- **phone_number** (*str*) – The phone number to send the message to.
- **message** (*str*) – The message to send.
- ****kwargs** – Extra kwargs. Both for future proofing, and to make it possible for backends to support extra kwargs.

classmethod get_backend_id()
The ID this backend will get get in the *Registry* singleton.

Defaults to the full python path for the class.

send()
Send the message.

Must be overridden in subclasses.

Should send `self.cleaned_message` to `self.cleaned_phone_number`.

CHAPTER 4

Settings

4.1 Settings

4.1.1 ievvtasks_dump_db_as_json

IEVVTASKS_DUMPDATA_DIRECTORY

The directory where we put dumps created by the `ievvtasks_dump_db_as_json` management command. Typically, you put something like this in your develop settings:

```
THIS_DIR = os.path.dirname(__file__)

IEVVTASKS_DUMPDATA_DIRECTORY = os.path.join(THIS_DIR, 'dumps')
```

IEVVTASKS_DUMPDATA_ADD_EXCLUDES

Use this setting to add models and apps to exclude from the dumped json. We exclude:

- contenttypes
- auth.Permission
- sessions.Session

By default, and we exclude `thumbnail.KVStore` by default if `sorl.thumbnail` is in installed apps, and the `THUMBNAIL_KVSTORE` setting is configured to use the database (`sorl.thumbnail.kvstores.cached_db_kvstore.KVStore`).

Example:

```
IEVVTASKS_DUMPDATA_ADD_EXCLUDES = [
    'auth.Group',
    'myapp.MyModel',
]
```

IEVVTASKS_DUMPDATA_EXCLUDES

If you do not want to get the default excludes, you can use this instead of *IEVVTASKS_DUMPDATA_ADD_EXCLUDES* to specify exactly what to exclude.

4.1.2 ievvtasks_makemessages

IEVVTASKS_MAKEMESSAGES_LANGUAGE_CODES

The languages to build translations for. Example:

```
IEVVTASKS_MAKEMESSAGES_LANGUAGE_CODES = ['en', 'nb']
```

IEVVTASKS_MAKEMESSAGES_IGNORE

The patterns to ignore when making translations. Defaults to:

```
IEVVTASKS_MAKEMESSAGES_IGNORE = [
    'static/*'
]
```

IEVVTASKS_MAKEMESSAGES_DIRECTORIES

Directories to run makemessages and compilemessages in. Defaults to a list with the current working directory as the only item. The use case for this setting is if you have your translation files split over multiple apps or directories. Then you can use this setting to specify the parent directories of all your `locale` directories.

Lets say you have this structure:

```
myproject/
    usersapp/
        locale/
    untranslatedapp/
    themeapp/
        locale/
```

You can then configure `ievv makemessages` and `ievv compilemessages` to build the translations in `myproject.usersapp` and `myproject.themeapp` with the following setting:

```
IEVVTASKS_MAKEMESSAGES_DIRECTORIES = [
    'myproject/usersapp',
    'myproject/themeapp',
]
```

Just adding strings to `IEVVTASKS_MAKEMESSAGES_DIRECTORIES` is just a shortcut. You can add dicts instead:

```
IEVVTASKS_MAKEMESSAGES_DIRECTORIES = [
    {
        'directory': 'myproject/usersapp',
    },
    {
        'directory': 'myproject/themeapp',
        'python': True,  # Build python translations
```

(continues on next page)

(continued from previous page)

```
        'javascript': True,  # Build javascript translations
        # 'javascript_ignore': ['something/*'],  # Override IEVVTASKS_MAKEMESSAGES_
↪JAVASCRIPT_IGNORE for the directory
        # 'python_ignore': ['something/*'],  # Override IEVVTASKS_MAKEMESSAGES_IGNORE␣
↪for the directory
    }
]
```

IEVVTASKS_MAKEMESSAGES_BUILD_JAVASCRIPT_TRANSLATIONS

Set this to `True` if you want to built translations for javascript code. Defaults to `False`.

IEVVTASKS_MAKEMESSAGES_JAVASCRIPT_IGNORE

The patterns to ignore when making javascript translations. Defaults to:

```
IEVVTASKS_MAKEMESSAGES_JAVASCRIPT_IGNORE = [
    'node_modules/*',
    'bower_components/*',
    'not_for_deploy/*',
]
```

IEVVTASKS_MAKEMESSAGES_PRE_MANAGEMENT_COMMANDS

Iterable of managemement commands to run before running makemessages. Example:

```
IEVVTASKS_MAKEMESSAGES_PRE_MANAGEMENT_COMMANDS = [
    {
        'name': 'ievvtasks_buildstatic',
        'options': {
            'includegroups': ['i18n']
        }
    }
]
```

Defaults to empty list.

The items in the iterable can be one of:

- A string with the name of a management command (for commands without any arguments or options).
- A dict with `name`, `args`, and `options` keys. The `name` key is required, but `args` and `options` are optional. `args` and `options` is just forwarded to `django.core.management.call_command`.

IEVVTASKS_MAKEMESSAGES_EXTENSIONS

Extensions to look for strings marked for translations in normal python/django code (for the `django` –domain for makemessages).

Defaults to `['py', 'html', 'txt']`.

IEVVTASKS_MAKEMESSAGES_JAVASCRIPT_EXTENSIONS

Extensions to look for strings marked for translations in javascript code (for the `djangojs` –domain for makemessages).

Defaults to `['js']`.

4.1.3 ievvtasks_docs

IEVVTASKS_DOCS_DIRECTORY

The directory where your sphinx docs resides (the directory where you have your sphinx `conf.py`). Defaults to `not_for_deploy/docs/`.

IEVVTASKS_DOCS_BUILD_DIRECTORY

The directory where your sphinx docs should be built. Defaults to `not_for_deploy/docs/_build`.

4.1.4 ievvtasks_recreate_devdb

IEVVTASKS_RECREATE_DEVDB_POST_MANAGEMENT_COMMANDS

Iterable of managemement commands to after creating/restoring and migrating the database in `ievv recreate_devdb`. Example:

```
IEVVTASKS_RECREATE_DEVDB_POST_MANAGEMENT_COMMANDS = [
    {
        'name': 'createsuperuser',
        'args': ['test@example.com'],
        'options': {'verbosity': 3}
    },
    'ievvtasks_set_all_passwords_to_test',
]
```

The items in the iterable can be one of:

- A string with the name of a management command (for commands without any arguments or options).
- A dict with `name`, `args`, and `options` keys. The `name` key is required, but `args` and `options` are optional. `args` and `options` is just forwarded to `django.core.management.call_command`.

4.1.5 ievv_tagframework

IEVV_TAGFRAMEWORK_TAGTYPE_CHOICES

The legal values for *ievv_opensource.ievv_tagframework.models.Tag.tagtype*.

Example:

```
IEVV_TAGFRAMEWORK_TAGTYPE_CHOICES = [
    ('', ''),
    ('mytype', 'My tag type'),
]
```

4.1.6 ievv devrun

IEVVTASKS_DEVRUN_RUNNABLES

Dict mapping `ievv devrun` target names to *ievv_opensource.utils.ievvdevrun.config.RunnableThreadList* objects. Must contain the `"default"` key.

Documented in *ievv devrun — All your development servers in one command*.

4.1.7 ievv_elasticsearch

IEVV_ELASTICSEARCH_URL

The URL of the elasticsearch instance.

IEVV_ELASTICSEARCH_TESTURL

The URL where we run elasticsearch for UnitTests. We provide a config file in `not_for_deploy/elasticsearch.unittest.yml` used with:

```
$ elasticsearch --config=path/to/elasticsearch.unittest.yml
```

to configure elasticsearch in a manner suitable for Unit testing as long as this setting is set to:

```
IEVV_ELASTICSEARCH_TESTURL = 'http://localhost:9251'
```

IEVV_ELASTICSEARCH_TESTMODE

Set this to True to make ElasticSearch behave in a manner that makes writing Unit tests a bit easier:

- Automatically refresh the indexes after any index update.
- Use `IEVV_ELASTICSEARCH_TESTURL` instead of `IEVV_ELASTICSEARCH_URL`.

Add the following to you test settings to enable testmode:

```
IEVV_ELASTICSEARCH_TESTMODE = True
```

IEVV_ELASTICSEARCH_PRETTYPRINT_ALL_SEARCH_QUERIES

Set this to True to prettyprint all ElasticSearch search queries. Defaults to `False`. Good for debugging.

IEVV_ELASTICSEARCH_PRETTYPRINT_ALL_REQUESTS

Set this to True to prettyprint all ElasticSearch requests (both input and output). Defaults to `False`. Good for debugging.

IEVV_ELASTICSEARCH_AUTOREFRESH_AFTER_INDEXING

Automatically refresh after indexing with meth:*ievv_opensource.ievv_elasticsearch.searchindex.AbstractIndex.index_items*. Useful for unit tests, but not much else.

You **should not** add this to your test settings, but use it in your tests where appropriate like this:

```
class MyTestCase(TestCase):
    def test_something(self):
        with self.settings(IEVV_ELASTICSEARCH_AUTOREFRESH_AFTER_INDEXING=False):
            # test something here
```

IEVV_ELASTICSEARCH_DO_NOT_REGISTER_INDEX_UPDATE_TRIGGERS

Do not register index update triggers on Django startup? Defaults to `False`. Mostly useful during development.

IEVV_ELASTICSEARCH_MAJOR_VERSION

The major version of elasticsearch you are using. Defaults to `1`, but we also support `2`.

4.1.8 ievv_elasticsearch2

IEVV_ELASTICSEARCH2_CONNECTION_ALIASES

Setup elasticsearch connections (almost exactly like setting up Django databases). Example:

```
IEVV_ELASTICSEARCH2_CONNECTION_ALIASES = {
    'default': {
        'host': '127.0.0.1',
        'port': '9251'
    },
    'theother': {
        'host': '127.0.0.1',
        'port': '9254'
    }
}
```

The inner dict (the one with host and port) are kwargs for `elasticsearch.client.Elasticsearch`. These configurations are all registered with `elasticsearch_dsl.connections.Connections`. This means that you can call `elasticsearch_dsl.connections.connections.get_connection(alias=<alias>)` to get an `elasticsearch.client.Elasticsearch` object with the configured connection settings:

```
from elasticsearch_dsl.connections import connections

default_elasticsearch = connections.get_connection()  # defaults to alias="default"
theother_elasticsearch = connections.get_connection(alias="theother")
```

Elasticsearch-dsl also uses `elasticsearch_dsl.connections.Connections`, so this means that you can use these aliases with `elasticsearch_dsl.search.Search.using()` and `ievv_opensource.ievv_elasticsearch2.search.Search.using()`:

```
from ievv_opensource import ievv_elasticsearch2

result = ievv_elasticsearch2.Search()\
    .query('match', name='Peter')\
    .using('theother')
    .execute()
```

Note: The `IEVV_ELASTICSEARCH2_CONNECTION_ALIASES` setting only works if you add `ievv_elasticsearch2` to `INSTALLED_APPS` with the AppConfig:

```
INSTALLED_APPS = [
    # .. Other apps ...
    "ievv_opensource.ievv_elasticsearch2.apps.IevvEsAppConfig",
]
```

IEVV_ELASTICSEARCH2_DEBUGTRANSPORT_PRETTYPRINT_ALL_REQUESTS

If this is `True`, it makes `ievv_opensource.ievv_elasticsearch2.transport.debug.DebugTransport` prettyprint all requests performed by any `elasticsearch.client.Elasticsearch` object it is configured as the `transport_class` for.

This does not work unless you use `ievv_opensource.ievv_elasticsearch2.transport.debug.DebugTransport` as the transport class. This is easiest to achieve by just adding it to your *IEVV_ELASTICSEARCH2_CONNECTION_ALIASES* setting:

```
IEVV_ELASTICSEARCH2_CONNECTION_ALIASES = {
    'default': {
        'host': '<somehost>',
        'port': '<someport>',
        'transport_class': 'ievv_opensource.ievv_elasticsearch2.transport.debug.
↪DebugTransport'
    }
}
```

You may want to set this to `True` to just see all elasticsearch requests and responses, but you can also use this in tests to debug just some requests:

```
class MyTest(TestCase):
    def test_something(self):
        # ... some code here ...
        with self.settings(IEVV_ELASTICSEARCH2_DEBUGTRANSPORT_PRETTYPRINT_ALL_
↪REQUESTS=True):
            # ... some elasticsearch queries here ...
        # ... more code here ...
```

IEVV_ELASTICSEARCH2_TESTMODE

Set this to True to make ElasticSearch behave in a manner that makes writing Unit tests a bit easier.

Add the following to you test settings to enable testmode:

```
IEVV_ELASTICSEARCH2_TESTMODE = True
```

4.1.9 ievv_batchframework

IEVV_BATCHFRAMEWORK_ALWAYS_SYNCRONOUS

If this is `True`, all actions will be executed syncronously. Read more about this in ievv_batchframework_develop_asyncronous.

4.1.10 ievv_sms

IEVV_SMS_DEFAULT_BACKEND_ID

The default backend ID to use for SMS sending. Example:

```
IEVV_SMS_DEFAULT_BACKEND_ID = 'debugprint'
```

4.1.11 utils

IEVV_SLUGIFY_CHARACTER_REPLACE_MAP

Custom character replacement map for the `ievv_slugify` function

IEVV_COLORIZE_USE_COLORS

Colorize output from *ievv_opensource.utils.ievv_colorize.colorize()*? Defaults to `True`.

IEVV_VALID_REDIRECT_URL_REGEX

Valid redirect URLs for *utils.validate_redirect_url — Validate redirect urls*.

Defaults to `^/.*$`, which means that only urls without domain is allowed by default.

Example for only allowing redirect urls that does not contain a domain, or redirect urls within the example.com domain:

```
IEVV_VALID_REDIRECT_URL_REGEX = r'^(https?://example\.com/|/).*$'
```

Warning: Do not use `^https://example\.com.*$` (no / after the domain). This can potentially lead to attacks using subdomains. `^https://example\.com.*$` would allow `example.com.spamdomain.io/something`, but `^https://example\.com/.*$` would not allow this.

CHAPTER 5

The ievv command

5.1 The `ievv` command

The `ievv` command does two things:

1. It avoids having to write `python manange.py appressotaks_something` and lets you write `ievv something` istead.
2. It provides commands that are not management commands, such as the commands for building docs and creating new projects.

When we add the command for initializing a new project, the ievv command will typically be installed globally instead of as a requirement of each project.

You find the source code for the command in `ievv_opensource/ievvtasks_common/cli.py`.

Some of the commands has required settings. See *Settings*.

5.2 *ievv buildstatic* — A static file builder (less, coffee, . . .)

The `ievv buildstatic` command is a fairly full featured general purpose static asset builder that solves the same general tasks as Grunt and Gulp, but in a very Django friendly and pythonic way.

You extend the system with object oriented python programming, and you configure the system using python classes.

5.2.1 Getting started

First of all, make sure you have the following in your `INSTALLED_APPS` setting:

```
'ievv_opensource.ievvtasks_common',
```

`ievv buildstatic` assumes you have the sources for your static files in the `staticsources/<appname>/` directory within each Django app.

For this example, we will assume your Django app is named `myapp`, and that it is located in `myproject/myapp/`.

First you need to create the staticsources directory:

```
$ mkdir -p myproject/myapp/staticsource/myapp/
```

Setup building of LESS files

To kick things off, we will configure building of LESS sources into CSS. Create `myproject/myapp/staticsource/myapp/styles/theme.less`, and add some styles:

```
.myclass {
    color: red;
}
```

Now we need to add configurations to `settings.py` for building this less file. Add the following Django setting:

```
from ievv_opensource.utils import ievvbuildstatic
IEVVTASKS_BUILDSTATIC_APPS = ievvbuildstatic.config.Apps(
    ievvbuildstatic.config.App(
        appname='myapp',
        version='1.0.0',
        plugins=[
            ievvbuildstatic.lessbuild.Plugin(sourcefile='theme.less'),
        ]
    ),
)
```

Now run the following command in your terminal:

```
$ ievv buildstatic
```

This should create `myproject/myapp/static/myapp/1.0.0/styles/theme.css`.

Add media files

You will probably need some media files for your styles (fonts, images, ect.). First, put an image in `myproject/myapp/staticsource/myapp/media/images/`, then add the following to the `plugins` list in the `IEVVTASKS_BUILDSTATIC_APPS` Django setting:

```
ievvbuildstatic.mediacopy.Plugin()
```

Run:

```
$ ievv buildstatic
```

This should add your image to `myproject/myapp/static/myapp/1.0.0/media/images/`.

Watch for changes

Re-running `ievv buildstatic` each time you make changes is tedious. You can watch for changes using:

```
$ ievv buildstatic --watch
```

5.2.2 Advanced topics

Using multiple apps

Using multiple apps is easy. You just add another *ievv_opensource.utils.ievvbuildstatic.config.App* to the IEVVTASKS_BUILDSTATIC_APPS setting:

```
from ievv_opensource.utils import ievvbuildstatic
IEVVTASKS_BUILDSTATIC_APPS = ievvbuildstatic.config.Apps(
    ievvbuildstatic.config.App(
        appname='myapp',
        version='1.0.0',
        plugins=[
            ievvbuildstatic.lessbuild.Plugin(sourcefile='theme.less'),
        ]
    ),
    ievvbuildstatic.config.App(
        appname='anotherapp',
        version='2.3.4',
        plugins=[
            ievvbuildstatic.lessbuild.Plugin(sourcefile='theme.less'),
        ]
    ),
)
```

5.2.3 NPM packages and ievv buildstatic

How ievv buildstatic interracts with package.json

Many of the ievv buildstatic plugins install their own npm packages, so they will modify `package.json` if needed. Most plugins do not specify a specific version of a package, but they will not override your versions if you specify them in `package.json`.

Yarn or NPM?

ievv buildstatic uses `yarn` by default, but you can configure it to use `npm` with the following settings:

```
IEVVTASKS_BUILDSTATIC_APPS = ievvbuildstatic.config.Apps(
    ievvbuildstatic.config.App(
        appname='myapp',
        version='1.0.0',
        installers_config={
            'npm': {
                'installer_class': ievvbuildstatic.installers.npm.NpmInstaller
            }
        },
        plugins=[
            # ...
        ]
```

(continues on next page)

(continued from previous page)

```
    )
)
```

5.2.4 Working with npm packages guide

Note: You can create your `package.json` manually, using `npm init`/`yarn init`. If you do not create a package.json, ievv buildstatic will make one for you if you use any plugins that require npm packages.

You work with npm/yarn just like you would for any javascript project. The package.json file must be in:

```
<django appname>/staticsources/<django appname>/package.json
```

So you should be in `<django appname>/staticsources/<django appname>/` when running npm or yarn commands.

Example

1. Create the `staticsources/myapp/` directory in your django app. Replace myapp with your django app name.
2. Create `staticsources/myapp/scripts/javascript/app.js`.
3. Configure ievv buildstatic with the following in your django settings:

```
IEVVTASKS_BUILDSTATIC_APPS = ievvbuildstatic.config.Apps(
    ievvbuildstatic.config.App(
        appname='myapp',
        version='1.0.0',
        plugins=[
            ievvbuildstatic.browserify_jsbuild.Plugin(
                sourcefile='app.js',
                destinationfile='app.js',
            ),
        ]
    )
)
```

4. Run `ievv buildstatic` to build this app. This will create a `package.json` file.
5. Lets add momentjs as a dependency:

```
$ cd /path/to/myapp/staticsources/myapp/
$ yarn add momentjs
```

6. ... and so on ...

5.2.5 Plugins

Overview

pluginbase.Plugin([group])	Base class for all plugins in ievvbuildstatic.
cssbuildbaseplugin.AbstractPlugin([lint, ...])	Base class for builders that produce CSS.
sassbuild.Plugin(sourcefile[, sourcefolder, ...])	SASS build plugin — builds .scss files into css, and supports watching for changes.
lessbuild.Plugin(sourcefile[, sourcefolder, ...])	LESS build plugin — builds .less files into css, and supports watching for changes.
mediacopy.Plugin([sourcefolder, ...])	Media copy plugin — copies media files from staticsources into the static directory, and supports watching for file changes.
bowerinstall.Plugin(packages, **kwargs)	Bower install plugin — installs bower packages.
npminstall.Plugin(packages, **kwargs)	NPM install plugin — installs NPM packages.
browserify_jsbuild.Plugin(sourcefile, ...[, ...])	Browserify javascript bundler plugin.
browserify_babelbuild.Plugin([...])	Browserify javascript babel build plugin.
browserify_reactjsbuild.Plugin([...])	Browserify javascript babel build plugin.
autosetup_jsdoc.Plugin([group])	Autosetup jsdoc config and npm script.
autosetup_esdoc.Plugin([title])	Autosetup esdoc config and npm script.
npmrun.Plugin(script[, script_args])	Run a script from package.json (I.E.: npm run <something>.
npmrun_jsbuild.Plugin([extra_import_paths])	Webpack builder plugin.
run_jstests.Plugin(**kwargs)	Run javascript tests by running npm test if the package.json has a "test" entry in "scripts".

Details

class ievv_opensource.utils.ievvbuildstatic.pluginbase.**Plugin**(*group=None*)

Bases: *ievv_opensource.utils.logmixin.LogMixin*, ievv_opensource.utils.ievvbuildstatic.options_mixin.OptionsMixin

Base class for all plugins in ievvbuildstatic.

Parameters **group** – The group of the plugin. Defaults to *default_group*. If this is None, the plugin can not be skipped when building an app.

You can use any value for groups, but these groups are convenient because the ievv buildstatic command has command line options that makes it easier to skip them:

- "css": Should be used for plugins that build css. Skipped when running ievv buildstatic with --skip-css.
- "js": Should be used for plugins that build javascript. Skipped when running ievv buildstatic with --skip-js.
- "jstest": Should be used for plugins that run automatic tests for javascript code. Should also be used for tests in languages that compile to javascript such as TypeScript and CoffeeScript. Skipped when running ievv buildstatic with --skip-jstests or --skip-js.
- "slow-jstest": Should be used instead of "jstest" for very slow tests. Skipped when running ievv buildstatic with --skip-slow-jstests, --skip-jstests or --skip-js.

name = None

The name of the plugin.

default_group = None
The default group of the plugin The default value for the `group` kwarg for `__init__` ().

install()
Install any packages required for this plugin.

Should use `ievv_opensource.utils.ievvbuild.config.App.get_installer()`.

Examples

Install an npm package:

```
def install(self):
    self.app.get_installer('npm').install(
        'somepackage')
    self.app.get_installer('npm').install(
        'otherpackage', version='~1.0.0')
```

post_install()
Called just like install, but after all apps has finished installing.

run()
Run the plugin. Put the code executed by the plugin each time files change here.

watch()
Configure watching for this plugin.

You normally do not override this method, instead you override *get_watch_folders()* and *get_watch_regexes()*.

Returns

A **ievv_opensource.utils.ievvbuildstatic.watcher.WatchConfig** object if you want to watch for changes in this plugin, or `None` if you do not want to watch for changes.

Return type *WatchdogWatchConfig*

get_watch_regexes()
Get the regex used when watching for changes to files.

Defaults to a regex matching any files.

get_watch_folders()
Get folders to watch for changes when using `ievv buildstatic --watch`.

Defaults to an empty list, which means that no watching thread is started for the plugin.

The folder paths must be absolute, so in most cases you should use `self.app.get_source_path()` (see `ievv_opensource.utils.ievvbuildstatic.config.App#get_source_path()`) to turn user provided relative folder names into absolute paths.

get_logger_name()
Get the name of the logger.

make_temporary_build_directory()
Make a temporary directory that you can use for building something.

Returns The absolute path of the new directory.

Return type str

register_temporary_file_or_directory(*absolute_path*)
Register a temporary file or directory.

This will automatically be cleaned after run() is finished (even if it crashes) if the app uses `keep_temporary_files=False` (the default).

exception ievv_opensource.utils.ievvbuildstatic.cssbuildbaseplugin.**CssBuildException**
Bases: `Exception`

Raised when *AbstractPlugin.build_css()* fails.

class ievv_opensource.utils.ievvbuildstatic.cssbuildbaseplugin.**AbstractPlugin**(*lint=True*, *lintrules=None*, *lintrules_overrides=*{}, *autoprefix=True*, *browserslist='> 5%'*, ***kwargs*)
Bases: *ievv_opensource.utils.ievvbuildstatic.pluginbase.Plugin*, *ievv_opensource.utils.shellcommandmixin.ShellCommandMixin*

Base class for builders that produce CSS.

Parameters

- **lint** (*bool*) – Lint the styles? Defaults to `True`.
- **lintrules** (*dict*) – Rules for the linter. See https://github.com/stylelint/stylelint. Defaults to:

```
{
    "block-no-empty": None,
    "color-no-invalid-hex": True,
    "comment-empty-line-before": ["always", {
        "ignore": ["stylelint-commands", "after-comment"],
    }],
    "declaration-colon-space-after": "always",
    "indentation": 4,
    "max-empty-lines": 2
}
```

- **lintrules_overrides** (*dict*) – Overrides for `lintrules`. Use this if you just want to add new rules or override some of the existing rules.
- **autoprefix** (*bool*) – Run https://github.com/postcss/autoprefixer on the css? Defaults to `True`.
- **browserslist** (*str*) – A string with defining supported browsers for https://github.com/ai/browserslist. Used by the autoprefix and cssnano commands.
- ****kwargs** – Kwargs for *ievv_opensource.utils.ievvbuildstatic.pluginbase.Plugin*.

install()
Install any packages required for this plugin.

Should use `ievv_opensource.utils.ievvbuild.config.App.get_installer()`.

Examples

Install an npm package:

```
def install(self):
    self.app.get_installer('npm').install(
        'somepackage')
    self.app.get_installer('npm').install(
        'otherpackage', version='~1.0.0')
```

build_css(*temporary_directory*)
Override this method and implement the code to build the css.

get_destinationfile_path()
Override this method and return the absolute path of the destination/output css file.

get_all_source_file_paths()
Used for utilities like the linter to get a list of all source files.

You must override this in subclasses.

run()
Run the plugin. Put the code executed by the plugin each time files change here.

class ievv_opensource.utils.ievvbuildstatic.sassbuild.**Plugin**(*sourcefile*, *sourcefolder='styles'*, *destinationfolder=None*, *other_sourcefolders=None*, *sass_include_paths=None*, *sass_variables=None*, ***kwargs*)

Bases: *ievv_opensource.utils.ievvbuildstatic.cssbuildbaseplugin.AbstractPlugin*

SASS build plugin — builds .scss files into css, and supports watching for changes.

By default, we assume `sassc` is available on PATH, but you can override the path to the sassc executable by setting the `IEVVTASKS_BUILDSTATIC_SASSC_EXECUTABLE` environment variable.

Examples

Very simple example where the source file is in `demoapp/staticsources/styles/theme.scss`:

```
IEVVTASKS_BUILDSTATIC_APPS = ievvbuildstatic.config.Apps(
    ievvbuildstatic.config.App(
        appname='demoapp',
        version='1.0.0',
        plugins=[
            ievvbuildstatic.sassbuild.Plugin(sourcefile='theme.scss'),
        ]
    )
)
```

A more complex example that builds a django-cradmin theme where sources are split in multiple directories, and the bower install directory is on the scss path (the example also uses *ievv_opensource.utils.ievvbuildstatic.bowerinstall.Plugin*):

```
IEVVTASKS_BUILDSTATIC_APPS = ievvbuildstatic.config.Apps(
    ievvbuildstatic.config.App(
        appname='demoapp',
        version='1.0.0',
        plugins=[
            ievvbuildstatic.bowerinstall.Plugin(
                packages={
                    'bootstrap': '~3.1.1'
                }
            ),
            ievvbuildstatic.sassbuild.Plugin(
                sourcefolder='styles/cradmin_theme_demoapp',
                sourcefile='theme.scss',
                other_sourcefolders=[
                    'styles/cradmin_base',
                    'styles/cradmin_theme_default',
                ],
                sass_include_paths=[
                    'bower_components',
                ]
            )
        ]
    )
)
```

Parameters

- **sourcefile** – Main source file (the one including all other scss files) relative to `sourcefolder`.
- **sourcefolder** – The folder where `sourcefile` is located relative to the source folder of the `App`. You can specify a folder in another app using a *ievv_opensource.utils.ievvbuildstatic.filepath.SourcePath* object, but this is normally not recommended.
- **sass_include_paths** – Less include paths as a list. Paths are relative to the source folder of the `App`. You can specify folders in other apps using *ievv_opensource.utils.ievvbuildstatic.filepath.SourcePath* objects
- ****kwargs** – Kwargs for *ievv_opensource.utils.ievvbuildstatic.cssbuildbaseplugin.AbstractPlugin*.

install()

Install any packages required for this plugin.

Should use `ievv_opensource.utils.ievvbuild.config.App.get_installer()`.

Examples

Install an npm package:

```
def install(self):
    self.app.get_installer('npm').install(
```

(continues on next page)

(continued from previous page)

```
        'somepackage')
    self.app.get_installer('npm').install(
        'otherpackage', version='~1.0.0')
```

get_destinationfile_path()
Override this method and return the absolute path of the destination/output css file.

build_css(*temporary_directory*)
Override this method and implement the code to build the css.

get_watch_folders()
We only watch the folder where the scss sources are located, so this returns the absolute path of the `sourcefolder`.

get_watch_regexes()
Get the regex used when watching for changes to files.

Defaults to a regex matching any files.

get_all_source_file_paths()
Used for utilities like the linter to get a list of all source files.

You must override this in subclasses.

class ievv_opensource.utils.ievvbuildstatic.lessbuild.**Plugin**(*sourcefile*, *sourcefolder='styles'*, *other_sourcefolders=None*, *less_include_paths=None*, ***kwargs*)

Bases: *ievv_opensource.utils.ievvbuildstatic.pluginbase.Plugin*, *ievv_opensource.utils.shellcommandmixin.ShellCommandMixin*

LESS build plugin — builds .less files into css, and supports watching for changes.

Examples

Very simple example where the source file is in `demoapp/staticsources/styles/theme.less`:

```
IEVVTASKS_BUILDSTATIC_APPS = ievvbuildstatic.config.Apps(
    ievvbuildstatic.config.App(
        appname='demoapp',
        version='1.0.0',
        plugins=[
            ievvbuildstatic.lessbuild.Plugin(sourcefile='theme.less'),
        ]
    )
)
```

A more complex example that builds a django-cradmin theme where sources are split in multiple directories, and the bower install directory is on the less path (the example also uses *ievv_opensource.utils.ievvbuildstatic.bowerinstall.Plugin*):

```
IEVVTASKS_BUILDSTATIC_APPS = ievvbuildstatic.config.Apps(
    ievvbuildstatic.config.App(
        appname='demoapp',
        version='1.0.0',
        plugins=[
```

(continues on next page)

(continued from previous page)

```
            ievvbuildstatic.bowerinstall.Plugin(
                packages={
                    'bootstrap': '~3.1.1'
                }
            ),
            ievvbuildstatic.lessbuild.Plugin(
                sourcefolder='styles/cradmin_theme_demoapp',
                sourcefile='theme.less',
                other_sourcefolders=[
                    'styles/cradmin_base',
                    'styles/cradmin_theme_default',
                ],
                less_include_paths=[
                    'bower_components',
                ]
            )
        ]
    )
)
```

Parameters

- **sourcefile** – Main source file (the one including all other less files) relative to `sourcefolder`.
- **sourcefolder** – The folder where `sourcefile` is located relative to the source folder of the `App`.
- **less_include_paths** – Less include paths as a list. Paths are relative to the source folder of the `App`.
- ****kwargs** – Kwargs for *ievv_opensource.utils.ievvbuildstatic.pluginbase.Plugin*.

install()

Install any packages required for this plugin.

Should use `ievv_opensource.utils.ievvbuild.config.App.get_installer()`.

Examples

Install an npm package:

```
def install(self):
    self.app.get_installer('npm').install(
        'somepackage')
    self.app.get_installer('npm').install(
        'otherpackage', version='~1.0.0')
```

run()

Run the plugin. Put the code executed by the plugin each time files change here.

get_watch_folders()

We only watch the folder where the less sources are located, so this returns the absolute path of the `sourcefolder`.

get_watch_regexes()
Get the regex used when watching for changes to files.

Defaults to a regex matching any files.

class ievv_opensource.utils.ievvbuildstatic.mediacopy.**Plugin**(*sourcefolder='media'*, *destinationfolder=None*, ***kwargs*)

Bases: *ievv_opensource.utils.ievvbuildstatic.pluginbase.Plugin*

Media copy plugin — copies media files from staticsources into the static directory, and supports watching for file changes.

Examples

Copy all files in `demoapp/staticsources/demoapp/media/` into `demoapp/static/1.0.0/media/`:

```
IEVVTASKS_BUILDSTATIC_APPS = ievvbuildstatic.config.Apps(
    ievvbuildstatic.config.App(
        appname='demoapp',
        version='1.0.0',
        plugins=[
            ievvbuildstatic.mediacopy.Plugin(),
        ]
    )
)
```

Using a different source folder. This will copy all files in `demoapp/staticsources/demoapp/assets/` into `demoapp/static/1.0.0/assets/`:

```
IEVVTASKS_BUILDSTATIC_APPS = ievvbuildstatic.config.Apps(
    ievvbuildstatic.config.App(
        appname='demoapp',
        version='1.0.0',
        plugins=[
            ievvbuildstatic.mediacopy.Plugin(sourcefolder="assets"),
        ]
    )
)
```

Parameters

- **sourcefolder** – The folder where media files is located relative to the source folder of the `App`. You can specify a folder in another app using a *ievv_opensource.utils.ievvbuildstatic.filepath.SourcePath* object.
- ****kwargs** – Kwargs for *ievv_opensource.utils.ievvbuildstatic.pluginbase.Plugin*.

run()
Run the plugin. Put the code executed by the plugin each time files change here.

get_watch_folders()
We only watch the folder where the less sources are located, so this returns the absolute path of the `sourcefolder`.

class ievv_opensource.utils.ievvbuildstatic.bowerinstall.**Plugin**(*packages*, ***kwargs*)

Bases: *ievv_opensource.utils.ievvbuildstatic.pluginbase.Plugin*, *ievv_opensource.utils.shellcommandmixin.ShellCommandMixin*

Bower install plugin — installs bower packages.

The packages are installed when `ievv buildstatic` starts up. The plugin creates a `bower.json` file, and runs `bower install` using the created `bower.json`-file.

You will most likely want to add `bower.json` and `bower_components` to your VCS ignore file.

Examples

Install bootstrap 3.1.1 and angularjs 1.4.1:

```
IEVVTASKS_BUILDSTATIC_APPS = ievvbuildstatic.config.Apps(
    ievvbuildstatic.config.App(
        appname='demoapp',
        version='1.0.0',
        plugins=[
            ievvbuildstatic.bowerinstall.Plugin(
                packages={
                    'bootstrap': '~3.1.1',
                    'angular': '~1.4.1'
                }
            ),
        ]
    )
)
```

install()

Install any packages required for this plugin.

Should use `ievv_opensource.utils.ievvbuild.config.App.get_installer()`.

Examples

Install an npm package:

```
def install(self):
    self.app.get_installer('npm').install(
        'somepackage')
    self.app.get_installer('npm').install(
        'otherpackage', version='~1.0.0')
```

run()

Run the plugin. Put the code executed by the plugin each time files change here.

class ievv_opensource.utils.ievvbuildstatic.npminstall.**Plugin**(*packages*, ***kwargs*)

Bases: *ievv_opensource.utils.ievvbuildstatic.pluginbase.Plugin*, *ievv_opensource.utils.shellcommandmixin.ShellCommandMixin*

NPM install plugin — installs NPM packages.

The packages are installed when `ievv buildstatic` starts up.

Examples

Install uniq and momentjs packages:

```
IEVVTASKS_BUILDSTATIC_APPS = ievvbuildstatic.config.Apps(
    ievvbuildstatic.config.App(
        appname='demoapp',
        version='1.0.0',
        plugins=[
            ievvbuildstatic.npminstall.Plugin(
                packages={
                    'uniq': None,
                    'momentjs': None
                }
            ),
        ]
    )
)
```

You can specify a version instead of `None` if you do not want to install the latest version.

install()

Install any packages required for this plugin.

Should use `ievv_opensource.utils.ievvbuild.config.App.get_installer()`.

Examples

Install an npm package:

```
def install(self):
    self.app.get_installer('npm').install(
        'somepackage')
    self.app.get_installer('npm').install(
        'otherpackage', version='~1.0.0')
```

class ievv_opensource.utils.ievvbuildstatic.browserify_jsbuild.**Plugin**(*sourcefile*, *destination-file*, *source-folder='scripts/javascript'*, *destination-folder='scripts'*, *extra_watchfolders=None*, *extra_import_paths=None*, *sourcemap=True*, ***kwargs*)

Bases: *ievv_opensource.utils.ievvbuildstatic.pluginbase.Plugin*, *ievv_opensource.utils.shellcommandmixin.ShellCommandMixin*

Browserify javascript bundler plugin.

Examples

Simple example:

```
IEVVTASKS_BUILDSTATIC_APPS = ievvbuildstatic.config.Apps(
    ievvbuildstatic.config.App(
        appname='demoapp',
        version='1.0.0',
        plugins=[
            ievvbuildstatic.browserify_jsbuild.Plugin(
                sourcefile='app.js',
                destinationfile='app.js',
            ),
        ]
    )
)
```

Custom source folder example:

```
IEVVTASKS_BUILDSTATIC_APPS = ievvbuildstatic.config.Apps(
    ievvbuildstatic.config.App(
        appname='demoapp',
        version='1.0.0',
        plugins=[
            ievvbuildstatic.browserify_jsbuild.Plugin(
                sourcefolder=os.path.join('scripts', 'javascript', 'api'),
                sourcefile='api.js',
                destinationfile='api.js',
            ),
        ]
    )
)
```

Parameters

- **sourcefile** – The source file relative to `sourcefolder`.
- **destinationfile** – Path to destination file relative to `destinationfolder`.
- **sourcefolder** – The folder where `sourcefiles` is located relative to the source folder of the `App`. Defaults to `scripts/javascript`.
- **destinationfolder** – The folder where `destinationfile` is located relative to the destination folder of the `App`. Defaults to `scripts/`.
- **extra_watchfolders** – List of extra folders to watch for changes. Relative to the source folder of the `App`.
- ****kwargs** – Kwargs for *ievv_opensource.utils.ievvbuildstatic.pluginbase.Plugin*.

install()

Installs the `browserify` NPM package.

The package is installed with no version specified, so you probably want to freeze the version using the *ievv_opensource.utils.ievvbuildstatic.npminstall.Plugin* plugin.

get_browserify_extra_args()
Get extra browserify args.

Override this in subclasses to add transforms and such.

make_browserify_args()
Make browserify args list.

Adds the following in the listed order:

- `-d` if the `sourcemap` kwarg is `True`.
- the source file.
- `-d <destination file path>`.
- whatever *get_browserify_extra_args()* returns.

Should normally not be extended. Extend *get_browserify_extra_args()* instead.

post_run()
Called at the start of run(), after the initial command start logmessage, but before any other code is executed.

Does nothing by default, but subclasses can hook in configuration code here if they need to generate config files, perform validation of file structure, etc.

run()
Run the plugin. Put the code executed by the plugin each time files change here.

get_watch_folders()
We only watch the folder where the javascript is located, so this returns the absolute path of the `sourcefolder`.

get_watch_extensions()
Returns a list of the extensions to watch for in watch mode.

Defaults to `['js']`.

Unless you have complex needs, it is probably easier to override this than overriding *get_watch_regexes()*.

get_watch_regexes()
Get the regex used when watching for changes to files.

Defaults to a regex matching any files.

class ievv_opensource.utils.ievvbuildstatic.browserify_babelbuild.**Plugin**(*echmascript_version='es2*
au-
tocre-
ate_babelrc=True,
***kwargs*)

Bases: *ievv_opensource.utils.ievvbuildstatic.browserify_jsbuild.Plugin*

Browserify javascript babel build plugin.

Examples

Simple example:

```
IEVVTASKS_BUILDSTATIC_APPS = ievvbuildstatic.config.Apps(
    ievvbuildstatic.config.App(
        appname='demoapp',
        version='1.0.0',
        plugins=[
            ievvbuildstatic.browserify_babelbuild.Plugin(
                sourcefile='app.js',
                destinationfile='app.js',
            ),
        ]
    )
)
```

Custom source folder example:

```
IEVVTASKS_BUILDSTATIC_APPS = ievvbuildstatic.config.Apps(
    ievvbuildstatic.config.App(
        appname='demoapp',
        version='1.0.0',
        plugins=[
            ievvbuildstatic.browserify_babelbuild.Plugin(
                sourcefolder=os.path.join('scripts', 'javascript', 'api'),
                sourcefile='api.js',
                destinationfile='api.js',
            ),
        ]
    )
)
```

Parameters

- **echmascript_version** – The echmascript version to use. Defaults to `"es2015"`.
- **autocreate_babelrc** (*bool*) –
- ****kwargs** – Kwargs for *ievv_opensource.utils.ievvbuildstatic.browserify_jsbuild.Plugin*.

install()

Installs the `babelify` and `babel-preset-<echmascript_version kwarg>` NPM packages in addition to the packages installed by *ievv_opensource.utils.ievvbuildstatic.browserify_jsbuild.Plugin.install()*.

The packages are installed with no version specified, so you probably want to freeze the versions using the *ievv_opensource.utils.ievvbuildstatic.npminstall.Plugin* plugin.

get_babel_presets()

Get a list of babelify presets.

This is the presets that go into `<HERE>` in `babelify -t [ babelify --presets [ <HERE> ] ]`.

Defaults to `["<the echmascript_version kwarg>"]`.

get_browserify_extra_args()

Get extra browserify args.

Override this in subclasses to add transforms and such.

post_run()
Called at the start of run(), after the initial command start logmessage, but before any other code is executed.

Does nothing by default, but subclasses can hook in configuration code here if they need to generate config files, perform validation of file structure, etc.

class ievv_opensource.utils.ievvbuildstatic.browserify_reactjsbuild.**Plugin**(*echmascript_version='es2015'*, *autocreate_babelrc=True*, ***kwargs*)

Bases: *ievv_opensource.utils.ievvbuildstatic.browserify_babelbuild.Plugin*

Browserify javascript babel build plugin.

Examples

Simple example:

```
IEVVTASKS_BUILDSTATIC_APPS = ievvbuildstatic.config.Apps(
    ievvbuildstatic.config.App(
        appname='demoapp',
        version='1.0.0',
        plugins=[
            ievvbuildstatic.browserify_reactjsbuild.Plugin(
                sourcefile='app.js',
                destinationfile='app.js',
            ),
        ]
    )
)
```

Custom source folder example:

```
IEVVTASKS_BUILDSTATIC_APPS = ievvbuildstatic.config.Apps(
    ievvbuildstatic.config.App(
        appname='demoapp',
        version='1.0.0',
        plugins=[
            ievvbuildstatic.browserify_reactjsbuild.Plugin(
                sourcefolder=os.path.join('scripts', 'javascript', 'api'),
                sourcefile='api.js',
                destinationfile='api.js',
            ),
        ]
    )
)
```

Parameters

- **echmascript_version** – The echmascript version to use. Defaults to `"es2015"`.
- **autocreate_babelrc** (*bool*) –
- ****kwargs** – Kwargs for *ievv_opensource.utils.ievvbuildstatic.browserify_jsbuild.Plugin*.

install()
Installs the `babel-preset-react` NPM package in addition to the packages installed by *ievv_opensource.utils.ievvbuildstatic.browserify_babelbuild.Plugin.install()*.

The packages are installed with no version specified, so you probably want to freeze the versions using the *ievv_opensource.utils.ievvbuildstatic.npminstall.Plugin* plugin.

get_babel_presets()
Adds `react` to the babelify presets added by *ievv_opensource.utils.ievvbuildstatic.browserify_babelbuild.Plugin.get_babel_presets()*.

get_watch_extensions()
Returns a list of the extensions to watch for in watch mode.

Defaults to `['js']`.

Unless you have complex needs, it is probably easier to override this than overriding *get_watch_regexes()*.

class ievv_opensource.utils.ievvbuildstatic.autosetup_jsdoc.**Plugin**(*group=None*)
Bases: *ievv_opensource.utils.ievvbuildstatic.pluginbase.Plugin*, *ievv_opensource.utils.shellcommandmixin.ShellCommandMixin*

Autosetup jsdoc config and npm script.

Examples

It is really simple to use:

```
IEVVTASKS_BUILDSTATIC_APPS = ievvbuildstatic.config.Apps(
    ievvbuildstatic.config.App(
        appname='demoapp',
        version='1.0.0',
        plugins=[
            ievvbuildstatic.autosetup_jsdoc.Plugin(),
        ]
    )
)
```

If you need to adjust the config, simply setup your own `build-docs` script in package.json instead of using this plugin.

Parameters **group** – The group of the plugin. Defaults to *default_group*. If this is `None`, the plugin can not be skipped when building an app.

You can use any value for groups, but these groups are convenient because the `ievv buildstatic` command has command line options that makes it easier to skip them:

- `"css"`: Should be used for plugins that build css. Skipped when running `ievv buildstatic` with `--skip-css`.
- `"js"`: Should be used for plugins that build javascript. Skipped when running `ievv buildstatic` with `--skip-js`.
- `"jstest"`: Should be used for plugins that run automatic tests for javascript code. Should also be used for tests in languages that compile to javascript such as TypeScript and CoffeeScript. Skipped when running `ievv buildstatic` with `--skip-jstests` or `--skip-js`.

- `"slow-jstest"`: Should be used instead of `"jstest"` for very slow tests. Skipped when running `ievv buildstatic` with `--skip-slow-jstests`, `--skip-jstests` or `--skip-js`.

install()
Install any packages required for this plugin.

Should use `ievv_opensource.utils.ievvbuild.config.App.get_installer()`.

Examples

Install an npm package:

```
def install(self):
    self.app.get_installer('npm').install(
        'somepackage')
    self.app.get_installer('npm').install(
        'otherpackage', version='~1.0.0')
```

run()
Run the plugin. Put the code executed by the plugin each time files change here.

class ievv_opensource.utils.ievvbuildstatic.autosetup_esdoc.**Plugin**(*title=None*, ***kwargs*)
Bases: *ievv_opensource.utils.ievvbuildstatic.pluginbase.Plugin*, *ievv_opensource.utils.shellcommandmixin.ShellCommandMixin*

Autosetup esdoc config and npm script.

The plugin creates the esdoc.json config file automatically. It also creates a directory named `esdoc` and a file named `index.md` within that directory. That file is the frontpage of your docs, and you should edit it and provide some information about the app/library.

If you want to add a manual "page" to your docs, you do that by adding files and/or directories to the `esdoc/manual/` directory (relative to the source directory - the directory with package.json). Esdoc has a specific set of manual sections that you can add:

- asset
- overview
- installation
- usage
- tutorial
- configuration
- example
- faq
- changelog

This plugin automatically adds these to the `esdoc.json` with the following rules:

- Does `esdoc/manual/<section>.md` exist? Add that as the first entry in the `manual.<section>` list.
- Does `esdoc/manual/<section>/` exist? Add all .md files in that directory to the `manual.<section>` list in sorted order.

Examples

It is really simple to use:

```
IEVVTASKS_BUILDSTATIC_APPS = ievvbuildstatic.config.Apps(
    ievvbuildstatic.config.App(
        appname='demoapp',
        version='1.0.0',
        plugins=[
            ievvbuildstatic.autosetup_esdoc.Plugin(),
        ]
    )
)
```

Lets add a tutorial! First create `esdoc/manual/tutorial.md` and run ievv buildstatic. That file should end up in `esdoc.json` and if you build the docs you should get a "manual" link at the top of the manual. If your tutorial grows, simply create the `esdoc/manual/tutorial/` directory and add markdown files to it. The files are sorted by filename and the output in the docs is still a single file. The `esdoc/manual/tutorial.md` file will still end up first if it is present. You can choose if you want to use the `esdoc/manual/tutorial.md`, the `esdoc/manual/tutorial/` directory or both. The tutorial example works for all the manual sections documented above.

Parameters

- **title** (*str*) – The title of the docs. Falls back to the app name if not specified.
- ****kwargs** – Kwargs for *ievv_opensource.utils.ievvbuildstatic.pluginbase.Plugin*.

install()
Installs the `esdoc` and `esdoc-importpath-plugin` npm packages as dev dependencies.

run()
Run the plugin. Put the code executed by the plugin each time files change here.

class ievv_opensource.utils.ievvbuildstatic.npmrun.**Plugin**(*script*, *script_args=None*, ***kwargs*)

Bases: *ievv_opensource.utils.ievvbuildstatic.pluginbase.Plugin*, *ievv_opensource.utils.shellcommandmixin.ShellCommandMixin*

Run a script from package.json (I.E.: `npm run <something>`.

Examples

Lets say you have the following in your package.json:

```
{
  "scripts": {
    "myscript": "echo 'hello'"
  }
}
```

You can then make this script run with:

```
IEVVTASKS_BUILDSTATIC_APPS = ievvbuildstatic.config.Apps(
    ievvbuildstatic.config.App(
```

(continues on next page)

(continued from previous page)

```
        appname='demoapp',
        version='1.0.0',
        plugins=[
            ievvbuildstatic.npmrun.Plugin(script='myscript'),
        ]
    )
)
```

Parameters

- **script** (*str*) – The name of the key for the script in the `scripts` object in `package.json`.
- **script_args** (*list*) – Arguments for the script as a list of strings.
- ****kwargs** – Kwargs for *ievv_opensource.utils.ievvbuildstatic.pluginbase.Plugin*.

run()

Run the plugin. Put the code executed by the plugin each time files change here.

class ievv_opensource.utils.ievvbuildstatic.npmrun_jsbuild.**Plugin**(*extra_import_paths=None*, ***kwargs*)

Bases: *ievv_opensource.utils.ievvbuildstatic.pluginbase.Plugin*, *ievv_opensource.utils.shellcommandmixin.ShellCommandMixin*

Webpack builder plugin.

Examples

Simple example:

```
IEVVTASKS_BUILDSTATIC_APPS = ievvbuildstatic.config.Apps(
    ievvbuildstatic.config.App(
        appname='demoapp',
        version='1.0.0',
        plugins=[
            ievvbuildstatic.npmrun_jsbuild.Plugin(),
        ]
    )
)
```

Webpack example:

```
Install webpack:

$ yarn add webpack

Add the following to your package.json:

    {
        ...
        "scripts": {
            ...
            "jsbuild": "webpack --config webpack.config.js",
            "jsbuild-production": "webpack --config webpack.config.js -p"
```

(continues on next page)

(continued from previous page)

```
            ...
        }
        ...
    }

Create a webpack.config.js with something like this:

    let path = require('path');

    const isProduction = process.env.IEVV_BUILDSTATIC_MODE == 'production';
    const appconfig = require("./ievv_buildstatic.appconfig.json");
    console.log(isProduction);
    console.log(appconfig);

    let webpackConfig = {
        entry: './scripts/javascript/ievv_jsbase/ievv_jsbase_core.js',
        output: {
            filename: 'ievv_jsbase_core.js',
            path: path.resolve(appconfig.destinationfolder, 'scripts')
        },
        module: {
            loaders: [
                {
                    test: /.jsx?$/,
                    loader: 'babel-loader',
                    // exclude: /node_modules/
                    include: [
                        path.resolve(__dirname, "scripts/javascript/ievv_jsbase"),
                    ]
                }
            ]
        }
    };

    if(isProduction) {
        webpackConfig.devtool = 'source-map';
    } else {
        webpackConfig.devtool = 'cheap-module-eval-source-map';
        webpackConfig.output.pathinfo = true;
    }

    module.exports = webpackConfig;
```

install()

Install any packages required for this plugin.

Should use `ievv_opensource.utils.ievvbuild.config.App.get_installer()`.

Examples

Install an npm package:

```
def install(self):
    self.app.get_installer('npm').install(
        'somepackage')
```

(continues on next page)

(continued from previous page)

```
    self.app.get_installer('npm').install(
        'otherpackage', version='~1.0.0')
```

run()
Run the plugin. Put the code executed by the plugin each time files change here.

watch()
Configure watching for this plugin.

You normally do not override this method, instead you override *get_watch_folders()* and *get_watch_regexes()*.

Returns

A **ievv_opensource.utils.ievvbuildstatic.watcher.WatchConfig** object if you want to watch for changes in this plugin, or `None` if you do not want to watch for changes.

Return type *WatchdogWatchConfig*

class ievv_opensource.utils.ievvbuildstatic.run_jstests.**Plugin**(***kwargs*)
Bases: *ievv_opensource.utils.ievvbuildstatic.pluginbase.Plugin*, *ievv_opensource.utils.shellcommandmixin.ShellCommandMixin*

Run javascript tests by running `npm test` if the `package.json` has a `"test"` entry in `"scripts"`.

Examples

Lets say you have the following in your package.json:

```
{
  "scripts": {
    "test": "jest"
  }
}
```

You can then make `jest` run at startup (not on watch change) with:

```
IEVVTASKS_BUILDSTATIC_APPS = ievvbuildstatic.config.Apps(
    ievvbuildstatic.config.App(
        appname='demoapp',
        version='1.0.0',
        plugins=[
            ievvbuildstatic.run_jstests.Plugin(),
        ]
    )
)
```

If you have a NPM script named `"test-debug"`, that will be run instead of `"test"` when loglevel is DEBUG. This means that if you add something like this to your package.json:

```
{
  "scripts": {
    "test": "jest",
    "test-debug": "jest --debug"
  }
}
```

and run `ievv buildstatic --debug`, `jest --debug` would be run instead of `jest`.

Parameters ****kwargs** – Kwargs for *ievv_opensource.utils.ievvbuildstatic.pluginbase.Plugin*.

log_shell_command_stderr(*line*)
Called by *run_shell_command()* each time the shell command outputs anything to stderr.

run()
Run the plugin. Put the code executed by the plugin each time files change here.

5.2.6 Apps and App

Overview

App(appname, version, plugins[, ...])	Configures how `ievv buildstatic` should build the static files for a Django app.
Apps(*apps, **kwargs)	Basically a list around *App* objects.

Details

class ievv_opensource.utils.ievvbuildstatic.config.**App**(*appname*, *version*, *plugins*, *sourcefolder='staticsources'*, *destinationfolder='static'*, *keep_temporary_files=False*, *installers_config=None*, *docbuilder_classes=None*, *default_skipgroups=None*, *default_includegroups=None*)

Bases: *ievv_opensource.utils.logmixin.LogMixin*

Configures how `ievv buildstatic` should build the static files for a Django app.

Parameters

- **appname** – Django app label (I.E.: `myproject.myapp`).
- **plugins** – Zero or more `ievv_opensource.utils.ievvbuild.pluginbase.Plugin` objects.
- **sourcefolder** – The folder relative to the app root folder where static sources (I.E.: less, coffescript, ... sources) are located. Defaults to `staticsources`.

add_plugin(*plugin*)
Add a *ievv_opensource.utils.ievvbuildstatic.lessbuild.Plugin*.

run(*skipgroups=None*, *includegroups=None*)
Run *ievv_opensource.utils.ievvbuildstatic.pluginbase.Plugin.run()* for all plugins within the app.

install(*skipgroups=None*, *includegroups=None*)
Run *ievv_opensource.utils.ievvbuildstatic.pluginbase.Plugin.install()* for all plugins within the app.

get_app_config()
Get the AppConfig for the Django app.

get_appfolder()
Get the absolute path to the Django app root folder.

get_app_path(*apprelative_path*)
Returns the path to the directory joined with the given `apprelative_path`.

get_source_path(**path*)
Returns the absolute path to a folder within the source folder of this app or another app.

Examples

Get the source path for a coffeescript file:

```
self.get_source_path('mylib', 'app.coffee')
```

Getting the path of a source file within another app using a *ievv_opensource.utils.ievvbuildstatic.filepath.SourcePath* object (a subclass of *ievv_opensource.utils.ievvbuildstatic.filepath.FilePathInterface*) as the path:

```
self.get_source_path(
    ievvbuildstatic.filepath.SourcePath('myotherapp', 'scripts', 'typescript',
↪ 'app.ts'))
```

Parameters ***path** – Zero or more strings to specify a path relative to the source folder of this app - same format as `os.path.join()`. A single *ievv_opensource.utils.ievvbuildstatic.filepath.FilePathInterface* object to specify an absolute path.

get_destination_path(**path*, ***kwargs*)
Returns the absolute path to a folder within the destination folder of this app or another app.

Examples

Get the destination path for a coffeescript file - extension is changed from `.coffee` to `.js`:

```
self.get_destination_path('mylib', 'app.coffee', new_extension='.js')
```

Getting the path of a destination file within another app using a *ievv_opensource.utils.ievvbuildstatic.filepath.SourcePath* object (a subclass of *ievv_opensource.utils.ievvbuildstatic.filepath.FilePathInterface*) as the path:

```
self.get_destination_path(
    ievvbuildstatic.filepath.DestinationPath(
        'myotherapp', '1.1.0', 'scripts', 'typescript', 'app.ts'),
    new_extension='.js')
```

Parameters

- **path** – Path relative to the source folder. Same format as `os.path.join()`. A single *ievv_opensource.utils.ievvbuildstatic.filepath.FilePathInterface* object to specify an absolute path.
- **new_extension** – A new extension to give the destination path. See example below.

watch(*skipgroups=None*, *includegroups=None*)
Start a watcher thread for each plugin.

get_installer(*alias*)
Get an instance of the installer configured with the provided `alias`.

Parameters **alias** – A subclass of .

Returns

An instance of the requested installer.

Return type *ievv_opensource.utils.ievvbuildstatic.installers.base.AbstractInstaller*

get_logger_name()
Get the name of the logger.

make_temporary_build_directory(*name*)
Make a temporary directory that you can use for building something.

Returns The absolute path of the new directory.

Return type str

class ievv_opensource.utils.ievvbuildstatic.config.**Apps**(**apps*, ***kwargs*)
Bases: *ievv_opensource.utils.logmixin.LogMixin*

Basically a list around *App* objects.

Parameters **apps** – *App* objects to add initially. Uses *add_app()* to add the apps.

add_app(*app*)
Add an *App*.

get_app(*appname*)
Get app by appname.

install(*appnames=None*, *skipgroups=None*, *includegroups=None*)
Run *ievv_opensource.utils.ievvbuildstatic.pluginbase.Plugin.install()* for all plugins within all *apps*.

iterapps(*appnames=None*)
Get an interator over the apps.

run(*appnames=None*, *skipgroups=None*, *includegroups=None*)
Run *ievv_opensource.utils.ievvbuildstatic.pluginbase.Plugin.run()* for all plugins within all *apps*.

watch(*appnames=None*, *skipgroups=None*, *includegroups=None*)
Start watcher threads for all folders that at least one *plugin* within any of the *apps* has configured to be watched for changes.

Blocks until `CTRL-c` is pressed.

get_logger_name()
Get the name of the logger.

5.2.7 Utils

class ievv_opensource.utils.ievvbuildstatic.utils.**RegexFileList**(*include_patterns=None*, *exclude_patterns=None*)
Bases: `object`

A list of regexes for matching files.

Examples

Get all javascript files in a directory:

```
from ievv_opensource.utils import ievvbuildstatic

filelist = ievvbuildstatic.utils.RegexFileList(include_patterns=['^.*?\.js$'])
filelist.get_files_as_list('/path/to/javascript/sources/')
```

Exclude some files:

```
filelist = ievvbuildstatic.utils.RegexFileList(
    include_patterns=['^.*\.js$'],
    exclude_patterns=['^.*\.spec\.js$']
)
filelist.get_files_as_list('/path/to/javascript/sources/')
```

class ievv_opensource.utils.ievvbuildstatic.filepath.**FilePathInterface**
Bases: object

Base interface for file path objects.

We provide subclasses if this interface with different use cases:

- ***SourcePath*: Use this to specify a source file or folder** in the sources directory of a *ievv_opensource.utils.ievvbuildstatic.config.App*.
- ***DestinationPath*: Use this to specify a destination file or folder** in the sources directory of a *ievv_opensource.utils.ievvbuildstatic.config.App*.
- ***AbsoluteFilePath*: Use this to specify a file or folder that is** not organized using the ievvbuildstatic directory layout.

abspath
Property that returns the absolute path to the file (or directory).

Must be overridden in subclasses.

class ievv_opensource.utils.ievvbuildstatic.filepath.**AbsoluteFilePath**(**path*)
Bases: *ievv_opensource.utils.ievvbuildstatic.filepath.FilePathInterface*

Absolute file path.

Works just like `os.path.join()`, and we just assume that you provide a path that is absolute.

Parameters ***path** – One or more strings that make up an absolute path when joined with `os.path.join(*path)`.

Returns:

abspath
Property that returns the absolute path to the file (or directory).

Must be overridden in subclasses.

class ievv_opensource.utils.ievvbuildstatic.filepath.**AbstractDjangoAppPath**
Bases: *ievv_opensource.utils.ievvbuildstatic.filepath.FilePathInterface*

Abstract base class for file paths within a Django app.

abspath
Property that returns the absolute path to the file (or directory).

Must be overridden in subclasses.

class ievv_opensource.utils.ievvbuildstatic.filepath.**SourcePath**(*appname*, **path*)
Bases: *ievv_opensource.utils.ievvbuildstatic.filepath.AbstractDjangoAppPath*

A path to a file or directory within the source directory of a *ievv_opensource.utils.ievvbuildstatic.config.App*.

Assumes that the sourcefolder of the App is `"staticsources"` (the default).

Parameters

- **appname** – The name of the Django app.
- ***path** – The path relative to the source directory of the *ievv_opensource.utils.ievvbuildstatic.config.App* identified by `appname`. Same format as `os.path.join()`.

abspath
Property that returns the absolute path to the file (or directory).

Must be overridden in subclasses.

class ievv_opensource.utils.ievvbuildstatic.filepath.**DestinationPath**(*appname*, *version*, **path*)
Bases: *ievv_opensource.utils.ievvbuildstatic.filepath.AbstractDjangoAppPath*

A path to a file or directory within the destination directory of a *ievv_opensource.utils.ievvbuildstatic.config.App*.

Assumes that the destinationfolder of the App is `"static"` (the default).

Parameters

- **appname** – The name of the Django app.
- **version** – The version of the app.
- ***path** – The path relative to the destination directory of the *ievv_opensource.utils.ievvbuildstatic.config.App* identified by `appname`. Same format as `os.path.join()`.

abspath
Property that returns the absolute path to the file (or directory).

Must be overridden in subclasses.

class ievv_opensource.utils.ievvbuildstatic.filepath.**AppPath**(*appname*, **path*)
Bases: *ievv_opensource.utils.ievvbuildstatic.filepath.AbstractDjangoAppPath*

A path to a file or directory within the root directory of a *ievv_opensource.utils.ievvbuildstatic.config.App*.

Parameters

- **appname** – The name of the Django app.
- ***path** – The path relative to the root directory of the *ievv_opensource.utils.ievvbuildstatic.config.App* identified by `appname`. Same format as `os.path.join()`.

abspath
Property that returns the absolute path to the file (or directory).

Must be overridden in subclasses.

5.2.8 Installers

Overview

base.AbstractInstaller(app)	Base class for installers.
npm.NpmInstaller(*args, **kwargs)	NPM installer.
yarn.YarnInstaller(*args, **kwargs)	Yarn installer.

Details

class ievv_opensource.utils.ievvbuildstatic.installers.base.**AbstractInstaller**(*app*)
Bases: *ievv_opensource.utils.logmixin.LogMixin*, *ievv_opensource.utils.shellcommandmixin.ShellCommandMixin*, ievv_opensource.utils.ievvbuildstatic.options_mixin.OptionsMixin

Base class for installers.

Each installer defines most of their own API, the only thing they have in common is a reference to their *ievv_opensource.utils.ievvbuildstatic.config.App*, a *name* and the methods defined in *ievv_opensource.utils.logmixin.LogMixin* and ievv_opensource.utils.ievvbuildstatic.shellcommand.ShellCommandMixin.

Plugins instantiate installers using *ievv_opensource.utils.ievvbuildstatic.config.App.get_installer()*.

Parameters **app** – The *ievv_opensource.utils.ievvbuildstatic.config.App* where this installer object was instantiated using *ievv_opensource.utils.ievvbuildstatic.config.App.get_installer()*.

name = None
The name of the installer.

get_logger_name()
Get the name of the logger.

exception ievv_opensource.utils.ievvbuildstatic.installers.npm.**NpmInstallerError**
Bases: Exception

exception ievv_opensource.utils.ievvbuildstatic.installers.npm.**PackageJsonDoesNotExist**
Bases: *ievv_opensource.utils.ievvbuildstatic.installers.npm.NpmInstallerError*

class ievv_opensource.utils.ievvbuildstatic.installers.npm.**NpmInstaller**(**args*, ***kwargs*)
Bases: ievv_opensource.utils.ievvbuildstatic.installers.abstract_npm_installer.AbstractNpmInstaller

NPM installer.

log_shell_command_stderr(*line*)
Called by *run_shell_command()* each time the shell command outputs anything to stderr.

run_packagejson_script(*script*, *args=None*)
Run a script in the scripts section of the package.json.

Parameters

- **script** – The npm script to run.
- **args** (*list*) – List of arguments.

exception ievv_opensource.utils.ievvbuildstatic.installers.yarn.**NpmInstallerError**
Bases: Exception

exception ievv_opensource.utils.ievvbuildstatic.installers.yarn.**PackageJsonDoesNotExist**
Bases: *ievv_opensource.utils.ievvbuildstatic.installers.yarn.NpmInstallerError*

class ievv_opensource.utils.ievvbuildstatic.installers.yarn.**YarnInstaller**(**args*, ***kwargs*)
Bases: ievv_opensource.utils.ievvbuildstatic.installers.abstract_npm_installer.AbstractNpmInstaller

Yarn installer.

log_shell_command_stdout(*line*)
Called by *run_shell_command()* each time the shell command outputs anything to stdout.

log_shell_command_stderr(*line*)
Called by *run_shell_command()* each time the shell command outputs anything to stderr.

run_packagejson_script(*script*, *args=None*)
Run a script in the scripts section of the package.json.

Parameters

- **script** – The npm script to run.
- **args** (*list*) – List of arguments.

5.2.9 Low level API

class ievv_opensource.utils.ievvbuildstatic.watcher.**WatchdogWatchConfig**(*watchfolders*, *watchregexes*, *plugin*)
Bases: object

Used by plugins to configure watching.

See *ievv_opensource.utils.ievvbuildstatic.pluginbase.Plugin.watch()*.

Parameters

- **watchfolders** – List of folders to watch.
- **watchregexes** – List of regexes to watch.
- **plugin** – A *ievv_opensource.utils.ievvbuildstatic.pluginbase.Plugin* object.

class ievv_opensource.utils.ievvbuildstatic.watcher.**ProcessWatchConfig**(*plugin*)
Bases: object

Parameters **plugin** – A *ievv_opensource.utils.ievvbuildstatic.pluginbase.Plugin* object.

class ievv_opensource.utils.ievvbuildstatic.watcher.**WatchConfigPool**

Bases: *ievv_opensource.utils.logmixin.LogMixin*

get_logger_name()

Get the name of the logger.

class ievv_opensource.utils.ievvbuildstatic.watcher.**EventHandler**(**args*, ***kwargs*)

Bases: watchdog.events.RegexMatchingEventHandler

Event handler for watchdog — this is used by each watcher thread to react to changes in the filesystem.

This is instantiated by *ievv_opensource.utils.ievvbuildstatic.pluginbase.Plugin.watch()* to watch for changes in files matching *ievv_opensource.utils.ievvbuildstatic.pluginbase.Plugin.get_watch_regexes()* in the folders specified in *ievv_opensource.utils.ievvbuildstatic.pluginbase.Plugin.get_watch_folders()*.

on_any_event(*event*)

Catch-all event handler.

Parameters **event** (FileSystemEvent) – The event object representing the file system event.

5.3 *ievv devrun* — All your development servers in one command

The ievv devrun command makes it easy to run (start/stop) all your development servers with a single command. It uses multithreading, background processes to run all your servers in a single blocking process that stops all the processes when CTRL-C is hit.

5.3.1 Getting started

First of all, make sure you have the following in your INSTALLED_APPS setting:

```
'ievv_opensource.ievvtasks_common',
'ievv_opensource.ievvtasks_development',
```

Next, you need to configure what to run when you run ievv devrun. You do this with the IEVVTASKS_DEVRUN_RUNNABLES-setting.

For the first example, we will use ievv devrun to just run the Django development server, just like python manage.py runserver. Add the following you your Django settings:

```
IEVVTASKS_DEVRUN_RUNNABLES = {
    'default': ievvdevrun.config.RunnableThreadList(
        ievvdevrun.runnables.django_runserver.RunnableThread()
    )
}
```

With this configured, you can run:

```
$ ievv devrun
```

to start the Django development. Hit CTRL-C to stop the server.

Using Django dbdev

For this example, we will setup *Django runserver* and *Django dbdev database server*:

```
IEVVTASKS_DEVRUN_RUNNABLES = {
    'default': ievvdevrun.config.RunnableThreadList(
        ievvdevrun.runnables.dbdev_runserver.RunnableThread(),
        ievvdevrun.runnables.django_runserver.RunnableThread()
    )
}
```

With this configured, you can run:

```
$ ievv devrun
```

to start both the Django development server and your django_dbdev database. Hit `CTRL-C` to stop both the servers.

Multiple run configurations

You may already have guessed that you can add multiple configurations since we only add a `default`-key to `IEVVTASKS_DEVRUN_RUNNABLES`. To add multiple configurations, just add another key to the dict. For this example, we will add an `design` key that also runs `ievv buildstatic --watch`:

```
IEVVTASKS_DEVRUN_RUNNABLES = {
    'default': ievvdevrun.config.RunnableThreadList(
        # ... same as above ...
    ),
    'design': ievvdevrun.config.RunnableThreadList(
        ievvdevrun.runnables.dbdev_runserver.RunnableThread(),
        ievvdevrun.runnables.django_runserver.RunnableThread(),
        ievvdevrun.runnables.ievv_buildstatic.RunnableThread(),
    )
}
```

To run the `design`-set of runnables, use:

```
$ ievv devrun -n design
```

5.3.2 Adding ievv devrun as PyCharm run config

If you use PyCharm, you can do the following to add `ievv devrun` as a run target:

- Select `Run -> Edit configurations ...`.
- **Select `+ -> Python`.**
  - Give it a name (E.g.: `ievv devrun default`).
  - Check *Single instance*.
  - Check *Share* if you want to share it with your co-workers.
  - Select your `manage.py` as the script.
  - Set `ievvtasks_devrun -n default` as *Script parameters*.

If you want to create a target for the `design` config shown above, you just create another PyCharm run target with `ievvtasks_devrun -n design`.

5.3.3 Custom runnables

We bundle a fairly limited set of runnables, but adding one is really easy. Check out the docs for:

- *ievv_opensource.utils.ievvdevrun.runnables.base.ShellCommandRunnableThread*
- *ievv_opensource.utils.ievvdevrun.runnables.base.AbstractRunnableThread*

5.3.4 Bundled runnables

Overview

base.AbstractRunnableThread([name])	Abstract class for a thread that runs something for the `ievvdevrun` framework.
base.ShellCommandRunnableThread([name, ...])	Makes it very easy to implement *AbstractRunnableThread* for system/shell commands.
django_runserver.RunnableThread([host, port])	Django runserver runnable thread.
dbdev_runserver.RunnableThread([name, ...])	Django-DBdev run database runnable thread.
elasticsearch.RunnableThread(configpath[, ...])	Elasticsearch runnable thread.
redis_server.RunnableThread([port, config_path])	redis-server runnable thread.

Details

class ievv_opensource.utils.ievvdevrun.runnables.base.**AbstractRunnableThread**(*name=None*)

Bases: `threading.Thread`, *ievv_opensource.utils.logmixin.LogMixin*

Abstract class for a thread that runs something for the `ievvdevrun` framework.

You have to override the `run()`-method (refer to the docs for `threading.Thread.run`), and the *stop()* method.

Parameters **name** – An optional name for the logger. See *get_logger_name()*.

log_startup()

Called automatically on startup with `"Starting <name>"` message.

log_successful_stop(*message=''*)

Call this in *stop()* to show a message after successfully stopping the runnable.

Parameters **message** – Optional extra message to show.

start()

Start the thread's activity.

It must be called at most once per thread object. It arranges for the object's run() method to be invoked in a separate thread of control.

This method will raise a RuntimeError if called more than once on the same thread object.

get_logger_name()

Get the logger name. Defaults to the `name`-parameter and falls back on the module name of the class.

stop()
Must stop the code running in the thread (the code in the run method).

class ievv_opensource.utils.ievvdevrun.runnables.base.**ShellCommandRunnableThread**(*name=None*, *command_config=*, *autorestart_on_c*

Bases: *ievv_opensource.utils.ievvdevrun.runnables.base.AbstractRunnableThread*, *ievv_opensource.utils.shellcommandmixin.ShellCommandMixin*

Makes it very easy to implement *AbstractRunnableThread* for system/shell commands.

Examples

Implement Django runserver:

```
class DjangoRunserverRunnableThread(base.ShellCommandRunnableThread):
    def get_logger_name(self):
        return 'Django development server'

    def get_command_config(self):
        return {
            # We use sys.executable instead of "python" to ensure we run the same
↪python executable
            # as we are using to start the process.
            'executable': sys.executable,
            'args': ['manage.py', 'runserver']
        }
```

As you can see, we just need specify the name and the command we want to run. You can even do this without creating a class by sending the command config as a parameter to this class in your django settings:

```
IEVVTASKS_DEVRUN_RUNNABLES = {
    'default': ievvdevrun.config.RunnableThreadList(
        ievvdevrun.runnables.base.ShellCommandRunnableThread(
            name='Django development server',
            command_config={
                'executable': sys.executable,
                'args': ['manage.py', 'runserver']
            }
        ),
    )
}
```

Making the command automatically restart when it crashes:

```
class DjangoRunserverRunnableThread(base.ShellCommandRunnableThread):
    default_autorestart_on_crash = True
    # ... same code as the example above ...
```

You can also autorestart by sending `autorestart_on_crash=True` as a parameter for the class.

Parameters

- **name** – Optional name for the runnable. Defaults to the module name.
- **command_config** – Same format as the return value from *get_command_config()*.

- **autorestart_on_crash** – Set this to `True` if you want to automatically restart the process if it crashes.

default_autorestart_on_crash = False
The default value for the `autorestart_on_crash` parameter. You can override this in subclasses if it is more natural to automatically restart on crash by default.

get_command_config()
Get the config for the shell/system command as a dict with the following keys:

- `executable`: The name of the executable (E.g.: `python`, `sqlite`, ...).
- `args`: Arguments for the executable as a list.
- **kwargs: Keyword arguments for the executable as a dict.** `my_option: "myvalue"` is automatically translated to `--my-option="myvalue"`.

run()
Method representing the thread's activity.

You may override this method in a subclass. The standard run() method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the args and kwargs arguments, respectively.

stop()
Must stop the code running in the thread (the code in the run method).

class ievv_opensource.utils.ievvdevrun.runnables.django_runserver.**RunnableThread**(*host='127.0.0*
port='8000')

Bases: *ievv_opensource.utils.ievvdevrun.runnables.base.ShellCommandRunnableThread*

Django runserver runnable thread.

Examples

You can just add it to your Django development settings with:

```
IEVVTASKS_DEVRUN_RUNNABLES = {
    'default': ievvdevrun.config.RunnableThreadList(
        ievvdevrun.runnables.django_runserver.RunnableThread()
    )
}
```

And you can make it not restart on crash with:

```
IEVVTASKS_DEVRUN_RUNNABLES = {
    'default': ievvdevrun.config.RunnableThreadList(
        ievvdevrun.runnables.django_runserver.RunnableThread(
            autorestart_on_crash=False)
    )
}
```

Parameters

- **host** – The host to run the Django server on. Defaults to `"127.0.0.1"`.
- **port** – The port to run the Django server on. Defaults to `"8000"`.

get_logger_name()
Get the logger name. Defaults to the `name`-parameter and falls back on the module name of the class.

get_command_config()
Get the config for the shell/system command as a dict with the following keys:

- `executable`: The name of the executable (E.g.: `python`, `sqlite`, ...).
- `args`: Arguments for the executable as a list.
- **kwargs: Keyword arguments for the executable as a dict. `my_option: "myvalue"` is automatically translated to `--my-option="myvalue"`.

class ievv_opensource.utils.ievvdevrun.runnables.dbdev_runserver.**RunnableThread**(*name=None*, *command_config=N*, *autorestart_on_cr*

Bases: *ievv_opensource.utils.ievvdevrun.runnables.base.ShellCommandRunnableThread*

Django-DBdev run database runnable thread.

Examples

You can just add it to your Django development settings with:

```
IEVVTASKS_DEVRUN_RUNNABLES = {
    'default': ievvdevrun.config.RunnableThreadList(
        ievvdevrun.runnables.dbdev_runserver.RunnableThread()
    )
}
```

Parameters

- **name** – Optional name for the runnable. Defaults to the module name.
- **command_config** – Same format as the return value from *get_command_config()*.
- **autorestart_on_crash** – Set this to `True` if you want to automatically restart the process if it crashes.

get_logger_name()
Get the logger name. Defaults to the `name`-parameter and falls back on the module name of the class.

get_command_config()
Get the config for the shell/system command as a dict with the following keys:

- `executable`: The name of the executable (E.g.: `python`, `sqlite`, ...).
- `args`: Arguments for the executable as a list.
- **kwargs: Keyword arguments for the executable as a dict. `my_option: "myvalue"` is automatically translated to `--my-option="myvalue"`.

start()
Start the thread's activity.

It must be called at most once per thread object. It arranges for the object's run() method to be invoked in a separate thread of control.

This method will raise a RuntimeError if called more than once on the same thread object.

run()
Method representing the thread's activity.

You may override this method in a subclass. The standard run() method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the args and kwargs arguments, respectively.

stop()
Must stop the code running in the thread (the code in the run method).

class ievv_opensource.utils.ievvdevrun.runnables.elasticsearch.**RunnableThread**(*configpath*, *elasticsearch_executable*, ***kwargs*)

Bases: *ievv_opensource.utils.ievvdevrun.runnables.base.ShellCommandRunnableThread*

Elasticsearch runnable thread.

Examples

You can just add it to your Django development settings with:

```
IEVVTASKS_DEVRUN_RUNNABLES = {
    'default': ievvdevrun.config.RunnableThreadList(
        ievvdevrun.runnables.elasticsearch.RunnableThread(
            configpath='not_for_deploy/elasticsearch.develop.yml')
    )
}
```

get_logger_name()
Get the logger name. Defaults to the `name`-parameter and falls back on the module name of the class.

get_command_config()
Get the config for the shell/system command as a dict with the following keys:

- `executable`: The name of the executable (E.g.: `python`, `sqlite`, ...).
- `args`: Arguments for the executable as a list.
- **`kwargs`: Keyword arguments for the executable as a dict. `my_option`: `"myvalue"`** is automatically translated to `--my-option="myvalue"`.

class ievv_opensource.utils.ievvdevrun.runnables.redis_server.**RunnableThread**(*port='6379'*, *config_path=None*)

Bases: *ievv_opensource.utils.ievvdevrun.runnables.base.ShellCommandRunnableThread*

redis-server runnable thread.

Examples

You can just add it to your Django development settings with:

```
IEVVTASKS_DEVRUN_RUNNABLES = {
    'default': ievvdevrun.config.RunnableThreadList(
        ievvdevrun.runnables.redis_server.RunnableThread()
    )
}
```

Or if you want Redis to run with your custom `redis.conf` file:

```
IEVVTASKS_DEVRUN_RUNNABLES = {
    'default': ievvdevrun.config.RunnableThreadList(
        ievvdevrun.runnables.redis_server.RunnableThread(config_path=/path/to/
↪config/redis.conf)
    )
}
```

And you can make it not restart on crash with:

```
IEVVTASKS_DEVRUN_RUNNABLES = {
    'default': ievvdevrun.config.RunnableThreadList(
        ievvdevrun.runnables.redis_server.RunnableThread(
            autorestart_on_crash=False)
    )
}
```

Parameters

- **port** – The port to run the Redis server on. Defaults to `"6379"`.
- **config_path** – Path to the redis.conf file. Defaults to `None`.

get_logger_name()
Get the logger name. Defaults to the `name`-parameter and falls back on the module name of the class.

get_command_config()
Get the config for the shell/system command as a dict with the following keys:

- `executable`: The name of the executable (E.g.: `python`, `sqlite`, ...).
- `args`: Arguments for the executable as a list.
- **kwargs: Keyword arguments for the executable as a dict. `my_option: "myvalue"` is automatically translated to `--my-option="myvalue"`.

5.3.5 Low level API

class ievv_opensource.utils.ievvdevrun.config.**RunnableThreadList**(**runnablethreads*)
Bases: `object`

List of *AbstractRunnableThread* objects.

You use this with the *IEVVTASKS_DEVRUN_RUNNABLES* setting to define what to run with `ievv devrun`.

Parameters **runnablethreads** – *AbstractRunnableThread* objects to add to the list.

append(*runnablethread*)
Append a *AbstractRunnableThread*.

Parameters **runnablethread** – A *AbstractRunnableThread* object.

start()
Start all the runnable threads and block until `SIGTERM` or `KeyboardInterrupt`.

CHAPTER 6

Utilities

6.1 *virtualenvutils* — Utilities for virtualenvs

ievv_opensource.utils.virtualenvutils.**is_in_virtualenv**()
Returns `True` if we are in a virtualenv.

ievv_opensource.utils.virtualenvutils.**get_virtualenv_directory**()
Get the root directory of the current virtualenv.

Raises OSError if not in a virtualenv.

ievv_opensource.utils.virtualenvutils.**add_virtualenv_bin_directory_to_path**()
Add *get_virtualenv_directory()* to `os.environ['PATH']`.

Why do we need this? This is used to work around limitations in how certain IDE's implement virtualenv support. They may add the virtualenv to PYTHONPATH, but to to PATH.

6.2 *utils.desktopnotifications* — Very simple desktop notification system

6.3 *utils.logmixin* — Colorized logging

class ievv_opensource.utils.logmixin.**Logger**(*name*, *level=None*)
Bases: `object`

Logger class used by *LogMixin*.

Parameters **name** – The name of the logger.

classmethod **get_instance**(*name*, *level=None*, *command_error_message=None*)
Get an instance of the logger by the given name.

Parameters **name** – The name of the logger.

stdout(*line*)
Use this to redirecting sys.stdout when running shell commands.

stderr(*line*)
Use this to redirecting sys.stderr when running shell commands.

info(*message*)
Log an info message.

success(*message*)
Log a success message.

warning(*message*)
Log a warning message.

error(*message*)
Log a warning message.

debug(*message*)
Log a debug message.

command_start(*message*)
Log the start of a command. This should be used in the beginning of each *ievv_opensource.utils.ievvbuildstatic.pluginbase.Plugin.run()*.

command_error(*message*)
Log failing end of a command. This should be used in *ievv_opensource.utils.ievvbuildstatic.pluginbase.Plugin.run()* when the task fails.

command_success(*message*)
Log successful end of a command. This should be used in *ievv_opensource.utils.ievvbuildstatic.pluginbase.Plugin.run()* when the task succeeds.

class ievv_opensource.utils.logmixin.**LogMixin**
Bases: object

Mixin class that takes care of logging for all the classes in the `ievvbuildstatic` package.

Subclasses must override `get_logger_name()`, and use `get_logger()`.

get_logger_name()
Get the name of the logger.

get_logger()
Get an instance of `Logger()` with *get_logger_name()* as the logger name.

6.4 *utils.shellcommandmixin* — Simplifies shell commands

exception ievv_opensource.utils.shellcommandmixin.**ShellCommandError**
Bases: Exception

Raised when `LogMixin.run_shell_command()` fails.

class ievv_opensource.utils.shellcommandmixin.**ShellCommandMixin**
Bases: object

Shell command mixin - for classes that need to run shell commands.

Requires *LogMixin*.

log_shell_command_stdout(*line*)
Called by *run_shell_command()* each time the shell command outputs anything to stdout.

log_shell_command_stderr(*line*)
Called by *run_shell_command()* each time the shell command outputs anything to stderr.

run_shell_command(*executable*, *args=None*, *kwargs=None*, *_cwd=None*, *_out=None*, *_err=None*, *_env=None*)
Run a shell command.

Parameters

- **executable** – The name or path of the executable.
- **args** – List of arguments for the `sh.Command` object.
- **kwargs** – Dict of keyword arguments for the `sh.Command` object.

Raises *ShellCommandError* – When the command fails. See *ShellCommandError*.

kill_process(*pid*)
Kill the system process with the given `pid`, and all its child processes.

> **Warning:** You should normally use *terminate_process()* instead of this method since that normally gives the process the chance to cleanup.

Parameters **pid** – The process ID of the process you want to kill.

Returns A list of all the killed processes.

terminate_process(*pid*)
Terminate the system process with the given `pid`, and all its child processes.

Parameters **pid** – The process ID of the process you want to terminate.

Returns A list of all the terminated processes.

6.5 *utils.singleton* — Singleton

class ievv_opensource.utils.singleton.**Singleton**
Bases: `object`

Implements the singleton pattern.

Example

Create a singleton class:

```
class MySingleton(Singleton):
    def __init__(self):
        super().__init__()
        self.value = 0

    def add(self):
        self.value += 1
```

Use the singleton:

```
MySingleton.get_instance().add()
MySingleton.get_instance().add()
print(MySingleton.get_instance().value)
```

Ensures there is only one instance created. Make sure to use super() in subclasses.

classmethod get_instance()
Get an instance of the singleton.

6.6 *utils.class_registry_singleton* — Framework for swappable classes

6.6.1 What is this for?

If you are creating a library where you need to enable apps using the library to replace or add some classes with injection. There are two main use-cases:

1. You have a choice field, and you want to bind the choices to values backed by classes (for validation, etc.), AND you want apps using the library to be able to add more choices and/or replace the default choices.
2. You have some classes, such as adapters working with varying user models, and you need to be able to allow apps to inject their own implementations.

See `ievv_opensource.utils.class_registry_singleton.ClassRegistrySingleton` examples.

6.6.2 API docs

6.7 *utils.text* — Text utils for simple text transformations

6.7.1 Settings for character replacement map

If you want to control the character replacements. You may add a custom map in your settings

In your **settings**, add:

```
IEVV_SLUGIFY_CHARACTER_REPLACE_MAP = {'<character>': '<replacement_character>'}
```

Where you replace your characters matching your need

6.8 *utils.ievv_colorize* — Colorized output for terminal stdout/stderr

ievv_opensource.utils.ievv_colorize.**COLOR_RED = 'red'**
Red color constant for `ievv_colorize()`.

ievv_opensource.utils.ievv_colorize.**COLOR_BLUE = 'blue'**
Blue color constant for `ievv_colorize()`.

ievv_opensource.utils.ievv_colorize.**COLOR_YELLOW = 'yellow'**
Yellow color constant for `ievv_colorize()`.

ievv_opensource.utils.ievv_colorize.**COLOR_GREY = 'grey'**
Grey color constant for `ievv_colorize()`.

ievv_opensource.utils.ievv_colorize.**COLOR_GREEN = 'green'**
Green color constant for `ievv_colorize()`.

ievv_opensource.utils.ievv_colorize.**colorize**(*text*, *color*, *bold=False*)
Colorize a string for stdout/stderr.

Colors are only applied if *IEVV_COLORIZE_USE_COLORS* is `True` or not defined (so it defaults to `True`).

Examples

Print blue text:

```
from ievv_opensource.utils import ievv_colorize

print(ievv_colorize('Test', color=ievv_colorize.COLOR_BLUE))
```

Print bold red text:

```
print(ievv_colorize('Test', color=ievv_colorize.COLOR_RED, bold=True))
```

Parameters

- **text** – The text (string) to colorize.
- **color** – The color to use. Should be one of:
  - *COLOR_RED*
  - *COLOR_BLUE*
  - *COLOR_YELLOW*
  - *COLOR_GREY*
  - *COLOR_GREEN*
  - `None` (no color)
- **bold** – Set this to `True` to use bold font.

6.9 *utils.choices_with_meta* — Object oriented choices for choice fields

6.10 *utils.validate_redirect_url* — Validate redirect urls

A very small set of utilities for validating a redirect URL. You will typically use this to secure URLs that you get as input from an insecure source, such as a `?next=<anything>` argument for a login view.

6.10.1 Configure

The only configuration is via the *IEVV_VALID_REDIRECT_URL_REGEX*, where you configure the valid redirect URLs.

6.10.2 Typical use case

Lets say you have a login view that supports `?next=<some url or path>`. This could lead to attacks for mining personal information if someone shares a link where the next URL points to an external domain. What you want is probably to allow redirects within your domain. To achieve this, you only need to set the *IEVV_VALID_REDIRECT_URL_REGEX* setting to match your domain and paths without a domain, and do something like this in your login view:

```
from ievv_opensource.utils import validate_redirect_url

class MyLoginView(...):
    # ... other code ...

    def get_success_url(self):
        nexturl = self.request.GET.get('next')
        if nexturl:
            validate_redirect_url.validate_url(nexturl)
            return nexturl
        else:
            # return some default
```

This will raise a ValidationError if the validation fails. You may cach this exception if you want to handle this with something other than a crash message in your server log.

6.10.3 Functions

ievv_opensource.utils.validate_redirect_url.**is_valid_url**(*url*)

Returns `True` if the provided URL matches the *IEVV_VALID_REDIRECT_URL_REGEX* regex.

Parameters **url** (*str*) – An URL. Can just be a path too (E.g.: all of these work `http://example.com`, `/test`, `http://example.com/test`.

ievv_opensource.utils.validate_redirect_url.**validate_url**(*url*)

Validate the provided url against the regex in the *IEVV_VALID_REDIRECT_URL_REGEX* setting.

Parameters **url** (*str*) – An URL. Can just be a path too (E.g.: all of these work `http://example.com`, `/test`, `http://example.com/test`.

Raises `django.core.exceptions.ValidationError` – If the `url` is not valid.

6.11 *utils.testhelpers* — Testing utilities

6.11.1 Testing Django migrations

Warning: You can not test migrations if you have a `MIGRATION_MODULES` setting that disabled migrations. So make sure you remove that setting if you have it in your test settings.

Guide

Lets say you have the following model:

```
class Node(models.Model):
    name = models.CharField(max_length=255)
```

You have an initial migration, and you have created a migration named `0002_suffix_name_with_stuff` which looks like this:

```
suffix = ' STUFF'

def add_stuff_to_all_node_names(apps, schema_editor):
    Node = apps.get_model('myapp', 'Node')
    for node in Node.objects.all():
        node.name = '{}{}'.format(node.name, suffix)
        node.save()

def reverse_add_stuff_to_all_node_names(apps, schema_editor):
    Node = apps.get_model('myapp', 'Node')
    for node in Node.objects.all():
        if node.name.endswith(suffix):
            node.name = node.name[:-len(suffix)]
            node.save()

class Migration(migrations.Migration):
    dependencies = [
        ('myapp', '0001_initial'),
    ]

    operations = [
        migrations.RunPython(add_stuff_to_all_node_names, reverse_code=reverse_add_
↪stuff_to_all_node_names),
    ]
```

Note: You can not test migrations that can not be reversed, so you **MUST** write reversible migrations if you want to be able to test them. Think of this as a good thing - it forces you to write reversible migrations.

To test this, you can write a test case like this:

```
from ievv_opensource.utils.testhelpers import testmigrations

class TestSomeMigrations(testmigrations.MigrationTestCase):
    app_label = 'myapp'
    migrate_from = '0001_initial'
    migrate_to = '0002_suffix_name_with_stuff'

    def test_migrate_works(self):

        # Add some data to the model using the ``apps_before`` model state
        Node = self.apps_before.get_model('myapp', 'Node')
        node1_id = Node.objects.create(
            name='Node1'
        ).id
        node2_id = Node.objects.create(
            name='Node2'
        ).id

        # Migrate (run the 0002_suffix_name_with_stuff migration)
```

(continues on next page)

(continued from previous page)

```
        self.migrate()

        # Test using the ``apps_after`` model state.
        Node = self.apps_after.get_model('myapp', 'Node')
        self.assertEqual(Node.objects.get(id=node1_id).name, 'Node1 STUFF')
        self.assertEqual(Node.objects.get(id=node2_id).name, 'Node2 STUFF')

    def test_reverse_migrate_works(self):

        # First, we migrate to get to a state where we can reverse the migration
        self.migrate()

        # Add some data to the model using the ``apps_after`` model state
        Node = self.apps_after.get_model('myapp', 'Node')
        node1_id = Node.objects.create(
            name='Node1 STUFF'
        ).id
        node2_id = Node.objects.create(
            name='Node2 STUFF'
        ).id

        # Reverse the migration
        self.reverse_migrate()

        # Test using the ``apps_before`` model state.
        Node = self.apps_before.get_model('myapp', 'Node')
        self.assertEqual(Node.objects.get(id=node1_id).name, 'Node1')
        self.assertEqual(Node.objects.get(id=node2_id).name, 'Node2')
```

The MigrationTestCase class

class ievv_opensource.utils.testhelpers.testmigrations.**MigrationTestCase**(*methodName='runTest'*)

Bases: `django.test.testcases.TransactionTestCase`

Test case for a Django database migration.

Example:

```
class TestSomeMigrations(MigrationTestCase):
    migrate_from = '0002_previous_migration'
    migrate_to = '0003_migration_being_tested'

    def test_is_selected_is_flipped(self):
        MyModel = self.apps_before.get_model('myapp', 'MyModel')
        MyModel.objects.create(
            name='Test1',
            is_selected=True
        )
        MyModel.objects.create(
            name='Test2',
            is_selected=False
        )
        MyModel.objects.create(
            name='Test3',
            is_selected=True
        )
```

(continues on next page)

(continued from previous page)

```
        self.migrate()

        MyModel = self.apps_after.get_model('myapp', 'MyModel')
        self.assertEqual(MyModel.objects.filter(is_selected=True).count, 1)
        self.assertEqual(MyModel.objects.filter(is_selected=False).count, 2)
```

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

app_label = None
The django app_label for the app you are migrating. This is the same app_label as you use with `python manage.py makemigrations <app_label>` to create the migration.

migrate_dependencies = None
Dependencies. A list of `(app_label, migration_name)` tuples.

migrate_from_dependencies = None
Same as *migrate_dependencies*, but ONLY for *migrate_from*.

migrate_to_dependencies = None
Same as *migrate_dependencies*, but ONLY for *migrate_from*.

migrate_from = None
The name of the migration to migrate from. Can be the full name, or just the number (I.E.: `0002` or `0002_something`.

migrate_to = None
The name of the migration to migrate to. Can be the full name, or just the number (I.E.: `0003` or `0003_something`.

classmethod setUpClass()
Hook method for setting up class fixture before running tests in the class.

setUp()
Perform required setup.

If you override `setUp()`, you must call `super().setUp()`!

apps_before
Get an `apps` object just like the first argument to a Django data migration at the state before migration has been run.

Only available **before** *migrate()* has been called, or after *reverse_migrate()* has been called.

apps_after
Get an `apps` object just like the first argument to a Django data migration at the state after migration has been run, and not available after *reverse_migrate()* has been called (unless *migrate()* is called again).

Only available **after** *migrate()* has been called.

migrate()
Migrate the database from *migrate_from* to *migrate_to*.

reverse_migrate()
Migrate the database from *migrate_to* to *migrate_from*.

You must call *migrate()* before calling this.

get_migrate_command_kwargs()
Get kwargs for the `migrate` management command.

The defaults are sane, by you may want to override this and change the `verbosity` argument for debugging purposes.

6.12 *utils.validation_error_util* — Util for working with ValidationError

class ievv_opensource.utils.validation_error_util.**ValidationErrorUtil**(*validation_error*)
Bases: `object`

A wrapper around `django.core.exceptions.ValidationError` that provides useful extra functionality.

Examples

Get a serializable dict:

```
ValidationErrorUtil(some_validation_error).as_serializable_dict()
```

Convert ValidationError to `rest_framework.exceptions.ValidationError`:

```
ValidationErrorUtil(some_validation_error).as_drf_validation_error()
```

as_dict()
Get the ValidationError as a dict mapping fieldname to a list of ValidationError object.

Works no matter how the ValidationError was created. If the ValidationError was created with `ValidationError('message')` or `ValidationError(['message1', 'message2'])` the values will end up in the `__all__` key of the dict.

as_list()
Get the ValidationError as a list of `ValidationError` objects.

Works even if the ValidationError was created with a dict as input.

as_serializable_list()
Get the ValidationError as a serializable list (a flat list with all the error messages).

as_serializable_dict()
Get the ValidationError as a serializable dict - a dict mapping fieldname to a list of error messages. If the ValidationError was not created with a dict as input, the error messages will be added to the `__all__` key.

as_drf_validation_error()
Convert the ValidationError to a `rest_framework.exceptions.ValidationError`.

6.13 *utils.datetime_format* — Utilities for formatting datetimes

ievv_opensource.utils.datetime_format.**format_datetime_in_timezone**(*datetime_object*, *datetime_format='DATETIME_FORMA*
timezone='UTC')

Format a datetime object in a timezone.

Parameters

- **datetime_object** (*datetime.datetime*) – Datetime object to format.
- **datetime_format** (*str*) – A Django datetime formatting string name, such as "DATETIME_FORMAT", "SHORT_DATETIME_FORMAT", "DATE_FORMAT", ...
- **timezone** (*str*) – Defaults to settings.TIME_ZONE. The datetime is converted to this timezone. So if you use UTC in the database, and want to present another timezone, this will convert it correctly.

Returns The formatted datetime.

Return type str

6.14 *utils.model_field_choices* — Utils for validating model choice fields

6.15 *utils.progress_print_iterator* — Util for printing progress for long running scripts

CHAPTER 7

Releasenotes

7.1 ievv_opensource 1.1.0 releasenotes

7.1.1 What is new?

- **Django 1.10b1 support.**
  - Minumum Django version is still 1.8.
- Minimum version of django-cradmin updated to 1.1.1.

7.1.2 Breaking changes

No breaking changes.

7.2 ievv_opensource 5.0.0 releasenotes

7.2.1 What is new?

- Make `ievv docs` not run `ievv buildstatic` by default. Have to use `--buildstatic-docs` to get the old default behavior.
- `ievv_developemail`: New module that provides a develop email backend that makes all sent emails browsable through Django admin. See *ievv_developemail — Develop mail backend that lets you view mails in django admin* for more details.

7.2.2 Breaking changes

Removed all traces of celery. Should not break anything since the only code using celery was `ievv_batchframework`, and that has been updated to using django-rq a few releases ago.

7.3 ievv_opensource 5.0.1 releasenotes

7.3.1 What is new?

- Bugfix in `ievv_developemail`. Did not handle unicode correctly when parsing the emails.

7.4 ievv_opensource 5.1.0 releasenotes

7.4.1 What is new?

- New module for writing tests for Django database migrations. See *Testing Django migrations*.

7.5 ievv_opensource 5.10.0 releasenotes

7.5.1 New features

- New `utils.datetime_format` module - see *utils.datetime_format — Utilities for formatting datetimes*.

7.6 ievv_opensource 5.11.0 releasenotes

7.6.1 New features

- New `utils.model_field_choices` module - see *utils.model_field_choices — Utils for validating model choice fields*.

7.7 ievv_opensource 5.12.0 releasenotes

7.7.1 New features

- New `utils.class_registry_singleton` module - see *utils.class_registry_singleton — Framework for swappable classes*.
- New `utils.progress_print_iterator` module - see *utils.progress_print_iterator — Util for printing progress for long running scripts*.

7.8 ievv_opensource 5.13.0 releasenotes

7.8.1 New features

- Better handling of unicode in pswin backend for ievv_sms.

7.9 ievv_opensource 5.2.0 releasenotes

7.9.1 What is new?

- Extend the `iterchoices()`, `iter_as_django_choices_short()` and `iter_as_django_choices_long()` methods of `ievv_opensource.utils.choices_with_meta.ChoicesWithMeta` with support for optional extra choices.

7.10 ievv_opensource 5.2.1 releasenotes

7.10.1 What is new?

- ievv_developemail support for python 2.x.

7.11 ievv_opensource 5.2.2 releasenotes

7.11.1 Notes

This is a quickfix for release 5.2.2.

Bug-fixes

It did not break anything for python 3.x, but python 2.x was not fully supported.

7.12 ievv_opensource 5.3.0 releasenotes

7.12.1 New features

- Add *ievv_opensource.utils.validation_error_util.ValidationErrorUtil*.

7.13 ievv_opensource 5.4.0 releasenotes

7.13.1 New features

- Add *ievv_model_mommy_extras — Add support for more fields types to model-mommy*.

7.14 ievv_opensource 5.5.0 releasenotes

7.14.1 New features

- Add *ievv_restframework_helpers — Helpers for working with Django rest framework*.

7.15 ievv_opensource 5.6.0 releasenotes

7.15.1 New features

- Add support for per directory config for *ievv makemessages*.

7.16 ievv_opensource 5.7.0 releasenotes

7.16.1 New features

- Add *ievv_sms — SMS sending - multiple backends supported*.

7.17 ievv_opensource 5.8.0 releasenotes

7.17.1 New features

- Add new `debug_dbstore` backend to *ievv_sms — SMS sending - multiple backends supported*.

7.18 ievv_opensource 5.9.0 releasenotes

7.18.1 New features

- New ievvtask - `ievv make_source_dist`.

CHAPTER 8

Indices and tables

- genindex
- modindex
- search

Python Module Index

i

Index

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

V

W

Y