Google App Engine Toolkit 2

Release 2.2.6.dev0

Jan 04, 2021

Contents:

gaetk2 is a modern approach to developing Python Code on Google App Engine. It is a reimplementation of *appengine-toolkit <https://github.com/mdornseif/appengine-toolkit>*. *appengine-toolkit* was a transfer of the techniques we used before in Django to the early Google App Engine Plattform. It was different time when it was developed - back then XML was still cool and REST was all the rage and App Engine was nearly feature free. Even webapp2 had not been developed.

gaetk2 is used in some big internal projects and tries to cover most of what an Web Application might need.

CHAPTER 1

Features

- Infrastructure for Continuous Integration and Continuous Delivery
- Sane Error Logging and Reporting with nice tracebacks during development. Including Error Reporting to Sentry. See *Error Handling Guide*.
- A configuration Framework in *config*
- A Simple, roubust framework for acceptance tests in *resttestlib*
- *gaetk2.forms.wtfbootstrap3()* to teach a WTForm bootstrap rendering.
- *gaetk2.helpers.check404()* to save boilerplate on loading datastore entries etc.
- Lot's of Template-Filters we use day to day in *jinja_filters*.
- Common conventions for ndb/datastore usage in *datastore*.
- Export of Datastore-Queries to XLS or CSV in *gaetk2.modelexporter*.
- An port of the Django Admin Site in `gaetk2.admin`.
- Day-do-day functionality in *tools*. Mostly meant for internal use but also available to you. Most noteworty:
  - *gaetk2.tools.datetools.convert_to_date()* might be the most used function in our whole codebase.
  - *gaetk2.tools.caching* provides cache decorators.
  - *gaetk2.tools.unicode* encode integers in base32, get rid of etc in strings.
  - *gaetk2.tools.structured_xls.XLSwriter* - `csv.writer` compatible interface to generate XLS-Files.

Todo:

- structured
- ids
- hujson2

- http
- auth0tools

1.1 GAETK2 - Concepts

gaetk is for deploying and developing appengine. We do not use local development servers very much. It also does all development and deploying to production in a single Google App Engine *application*. This means development and testing happens on live data. We are fine with that (see *Error Handling Guide* for details) but you may not.

We use very little of the backends/modules/services features of App Engine. See :*Services*.

1.1.1 App Engine Building Blocks

application A pice of software deployed under a specific Application ID on Google App Engine. The `application` field in your `app.yaml`.

version a deployment target within your *application*. There are specific *versions* for specific purposes. *production version*, *staging version*, a *tagged version* is for deployment and user traffic. A *development version* is for developer interaction.

production version is where *version* your users visit. Should be deployed with care and never without testing. Usually all the traffic of your external domain name like `application.example.com` goes here. Note that other App Engine Applications should prefer access under the `application.appspot.com` name to get Googles Inter-App Authentication. Code can check via *`gaetk2.config.is_production()`* if running on the production version.

staging version is the *version* for showcase A/B tests and internal training of upcoming stuff. Available under `staging-dot-application.appspot.com`.

tagged version like `v180228-cg89bd1-production`. A specific tagged *version* deployed for production testing. The usual approach is to deploy the production branch to a tagged version, run the test suite against it and then deploy the *production version*. This allows easy switching back to the second to last tagged version if there come up issues in the new *production version*. Available under names like `v180228-cg89bd1-production-dot-application.appspot.com`.

The name follows the pattern `v`, *date*, –, *git hash*, –, *branchname*.

development version like `dev-md`. Postfixed by the developers username. Meant for development and testing. Usually deployed with the local copy of a master or feature branch. Available under names like `dev-md-dot-application.appspot.com`. Also versions staring with `test` will be considered development. Code can check via `gaetk2.config.is_development()` if running on a development version.

Prior to pushing to *master* tests should be run against the deployed *development version*.

services

modules Generally where we are using App Engine Modules/Services we try to run the same codebase on all Modules/Services to keep deployment and versioning in under control. We mostly use them to fine tune latency and instance size. When the *production version* is deployed all services should be redeployed.

release number The string used for the *tagged version*. Also found in `gaetk2-release.txt` and available via *`gaetk2.config.get_release()`*

1.2 Error Handling Guide

td;dr: See *Installing Error Handling* how to just get started.

Generally when working on large distributed systems we have to life with lots of transient errors. So you have to be careful that your data is always in a consistent state. Making your application idempotent in most places is very helpful for that.

> **Crash Early.** A dead program normally does a lot less damage than a crippled one.
>
> —The Pragmatic Programmer

One approach which works well is to crash early and let the infrastructure handle the retries. For App Engine Applications this are usually taskqueues or by `defer()` which uses taskqueues behind the scenes. In other instances the retry may be initiated by the user and his web browser or by an external service (like Mail delivery).

This means there are certain types of errors we care only a about if they happen often (like timeouts) others we want to know about immediately (like Syntax Errors).

We strongly suggest to use an external log aggregation service. We use Sentry in the hosted variant provided by GetSentry. What convinced us to use that is that Armin Ronacher started working there. Armin created so many things we use every day so we thought sentry.io probably is great, too. Sentry also seems to be OpenSource in large parts so you might be able to run your own instance. We didn't try.

In gaetk2 we use three levels of error reporting: *high level* in application code, *medium level* in library code and *low level* in infrastructure code.

Error handling should be as simple as possible to avoid errors during error handling. Error Display to end users should be robust, plain, simple and without flashy design or wording.

1.2.1 High Level Error Handling

Most Errors will happen in request handlers. High Level Error Handling is happening in *`gaetk2.application.WSGIApplication()`* so all request handlers called via this application will get our error handling.

Note: Be also aware that exceptions are also used to communicate HTTP-Status-Codes throughout the systems. These we do not consider errors.

Also in parts of *webapp2* only `Exception` is caught. But some AppEngine Exceptions are derivered from `BaseException`.

Exceptions happening in Request-Handlers are caught in *`dispatch()`* and forwarded to *`handle_exception()`*. This is the place where you might implement your own exception handling or logging.

The exceptions are recaught in `gaetk2.application.WSGIApplication.__call__()` and forwarded to *`gaetk2.application.WSGIApplication.handle_exception()`*. There all the special stuff is happening. This is:

- *Handling of HTTPException*
- *Exception Classification*
- *Error-Page for the Client*
- **'Push Error Information to Log Aggregator'_**

Handling of HTTPException

gaetk2.exc.HTTPException are a clever concept introduced by webobj. They are used by request handlers to abort execution and set return status codes.

So instead of something like:

```
self.response.write('you are unauthenticated')
self.redirect('/login', permanent=False)
return
```

you can do something like this:

```
raise exc.HTTP301_Moved('you are unauthenticated', location='/login')
```

This makes control flow much more explicit. This functionality is implemented in `gaetk2.application.WSGIApplication.__call__()` where all instances of *HTTPException* and it's subclasses are just sent to the, client. *HTTPException* generates the necessary headers and body.

All other Exceptions are handled further down the line.

Note: In *gaetk1* / *gaetk_common* the same effect was reached via `make_app()` which set `app.error_handlers[500] = handle_500`. *gaetk2* integrates the functionality within *gaetk2.application.WSGIApplication*.

Exception Classification

Some Exceptions we usually just don't want to know about, like *gaetk2.exc.HTTP301_Moved*. Others we consider mere warnings which do not need actions of the admin or programmer like `google.appengine.api.datastore_errors.Timeout`.

webapp2 usually adds a status code 500 to all Python Exceptions. For finer grained logging we want to offer a bunch of different status code and also decide if we consider the event a *note* (e.g. Page Not Found) a *warning* (e.g. Timeout) or an *error* (e.g. Syntax Error).

This is happening in *gaetk2.application.WSGIApplication.classify_exception()* which you are encouraged extend to fit your needs.

Error-Page for the Client

If we are not running in production mode (see *is_production()*) extensive traceback information is sent to the client using the `cgitb` module. Be aware that this might expose server secrets!

If running in production mode a simple error page is generated from `templates/error/500.html` and sent to the client. Currently the file name is hardcoded.

Push Error Information to Log Aggregator (Sentry)

Optionally Log information can be sent to Sentry for log aggregation. This automatically happens when a Sentry DSN (see below) is configured. We do our best to add all kinds of useful information to the Sentry message.

Todo:

- Only send Traceback information to admins.

- Allow changing of 500.html template

1.2.2 Medium Level Error Handling

Errors occurring within the framework (e.g. during error handling or in code not based on *gaetk2.handlers.base.BasicHandler* are handled by a WSGI-Middleware. This is usually installed automatically if *gaetk2.config.is_production()* by importing *gaetk2.wsgi.webapp_add_wsgi_middleware()*.

Error-Handling will be a little less sophisticated than *High Level Error Handling*.

1.2.3 Low Level Error Handling

Some Errors we just can't handle via python code. Most notable syntax errors in low level modules and timeouts. But App Engine can display error pages for them.

For basic error handling add this to your app.yaml:

```
error_handlers:
- file: lib/appengine-toolkit2/templates/error/500.html
```

To get better error reporting we suggest you create a copy of *error/500.html* with some Javascript code to handle Javascript based front end error logging of the incident.

1.2.4 Frontend Error Handling

You want to log Javascript errors happening at the Client Side. Sentry and similar services offer that. gaetk2 allows easy integration.

1.2.5 Sentry Configuration

If you do not configure Sentry you loose a lot of the error handling functionality.

To setup Sentry, just create a Project at Sentry. There you can get your. Insert it into `appengine_config.py`:

```
GAETK2_SENTRY_DSN='https://15e...4ed:f10...passwd...2b2@app.getsentry.com/76098'
# for Client Side Javascript we obmit the Password
GAETK2_SENTRY_PUBLIC_DSN='https://15e...4ed@app.getsentry.com/76098'
```

This should be all you need. In the Default-Templates it will install raven-js and start logging frontend errors. This is be archived by *gaetk2.handlers.base.BasicHandler* and `templates/gaetk_base_bs4.html`.

1.2.6 Installing Error Handling

To install error handling, configure Sentry as shown above. Then add this to `appengine_config.py` to get *Medium Level Error Handling*:

```
# load gaetk2 bootstrap code without using `sys.path`
import imp
(fp, filename, data) = imp.find_module(
    'boot', ['./lib/appengine-toolkit2/gaetk2/'])
imp.load_module('gaetk_boot', fp, filename, data)
```

(continues on next page)

(continued from previous page)

```
# install middleware
from gaetk2.wsgi import webapp_add_wsgi_middleware
```

This will not only install Error Handling on production but also session handling etc. See *gaetk2.wsgi* for detailed documentation. The WSGI middleware now should catch all exceptions not being caught by our handlers or WSGI applications.

For *High Level Error Handling* just use *gaetk2.application.WSGIApplication*. For example in app.yaml add:

```
handlers:
    - url: /
      script: home.app
```

`home.py` should look like this:

```
from gaetk2.handlers import DefaultHandler
from gaetk2.application import WSGIApplication, Route

class HomeHandler(DefaultHandler):
    def get(self):
        self.return_text('it worked')

app = WSGIApplication([Route('/', handler=HomeHandler)])
```

And don't forget to add `GAETK2_SENTRY_DSN` to `appengine_config.py`!

1.2.7 Using Logging

If you followed the steps until here all Exceptions should go to Sentry. Also all logging with level `ERROR` or `CRITICAL` via the Python Standard `logging` module should go to Sentry. If there is an Exception Sentry will attach all previous log messages (also `DEBUG` and `WARNING` in the report.

To allow better filtering we strongly suggest that you don not do calls to `logging.error()` et. al. directly but instantiate a logger instance in each of your modules and use that:

```
logger = logging.getLogger(__name__)
logger.debug('happy coding away')
```

For structured Information you only need in case of an Exception or other event you can use `note()`:

```
from gaetk2.tools.sentry import sentry_client
sentry_client.note(
    'auth',
    message=u'Attaching Customer to Credential'
    data={'self.credential': self.credential,
          'userkunde': userkunde})
```

This functionality is based on raven.breadcrumbs functionality gut tries to pass objects in a more readable state to Sentry.

`WSGIHTTPException` can have a `comment` parameter in it's constructor. This is internal nformation meant for debugging purposes. If this is set we assume there are exceptional circumstances and record the exception to Sentry:

```
text = u'%s:%s tries to log in as inactive customer %r' % (
    userkunde.designator, self.credential, userkunde.to_dict())
raise exc.HTTP301_Moved(
    comment=text,
    location='/inaktiv.html?kunde={}&uid={}'.format(
        userkunde.designator, self.credential.uid))
```

Todo:

- Describe how to add front end logging via Sentry to Low Level Error Handling
- gaetk2.wsgi Documentation

Note:

- in gaetk2 the `debug` Parameter to `WSGIApplication` is not used for enabling reporting of trackbacks to the client. Instead it is used for configuring *`debug()`*

1.3 Authentication, Authorization & Access Control

Authentication is finding out who you are dealing with. *Authorization* is if the *authenticated* user allowed to do what he does. *Access Control* is the implementation of it all. All of it together is sometimes called AAA.

You can configure basic AAA in app.yaml.

See the **'templatetags-accesscontrol'_** Section in **'templatetags'_** Document for of to make your templates dependant on who is logged in.

1.3.1 Authentication

gaetk2 uses Datastore Backed *Credential* Entities to handle Authentication. Clients can use

- HTTP-Basic Auth (RfC 7617)
- Session based Authentication via Forms
- Google Appengine `google.appengine.api.users.GetCurrentUser()` Interface / Google Identity Platform
- Auth0
- HTTP-Bearer Auth (RfC 6750) with JSON Web Tokens (JWT, RfC 7519)

to provide authentication information. In addition gaetk2 can identifiy requests form certain infrastructure:

- App Engine Task-Queues (`X-AppEngine-QueueName`)
- Other App Engine Applications (`X-Appengine-Inbound-Appid`)
- Sentry

To activate Authentication, just inherit from `AuthenticationReaderMixin`. E.g.:

```
class DefaultHandler(BasicHandler, AuthenticationReaderMixin):
    pass
```

Per default `AuthenticationReaderMixin` just decodes Authentication Information provided by the browser on its own. But to log in you have to make the user to authenticate himself. While gaetk2 can use username and password the main usage scenario is login via a third Party (Auth0 or Google). gaetk2 currently supports Google Identity Platform and Auth0 as login providers. Google because to use App Engine you and your colleagues already use Google Sign-In. Auth0 because it is well designed, powerful, easy to use and has decent debugging support.

LoginGoogleHandler and *LoginAuth0Handler* redirect the user to the OpenID Connect process where Google or Auth0 Make sure the user is who he claims. The user is then redirected back to *GoogleOAuth2Callback* or *AuthOAuth2Callback* where the information sent by Google or Auth0 is decoded, verified and on first time a *Credential* entity is created in the database.

Todo:

- Explain usage

Currently users are identified by their E-Mail Address. This might be problematic if a user changes his address but is the easiest way to identify the same user across different identity platforms.

For every authenticated user the *uid* (E-Mail) of the *Credential* is safed in the session. You can assume that when *uid* exists in the session the user is authenticated.

Configure Auth0

Create a new Client **'at the Auth0 Dashboard <https://manage.auth0.com/'_**. Should be "Regular Web Applications - Traditional web app (with refresh).". Note the "Domain", "Client ID" and "Client Secret" and put them into `appengine_config.py`:

```
GAETK2_AUTH0_DOMAIN='exam...ple.eu.auth0.com'
GAETK2_AUTH0_CLIENT_ID='QJ...um'
GAETK2_AUTH0_CLIENT_SECRET='mnttt-k0...supersecret'
```

Now you have to list all allowed URLs where your App may live - even for testing - in "Allowed Callback URLs".

Configure Google

Create at https://console.cloud.google.com/apis/credentials

Authenticating Sentry

If you use Sentry for Log Aggregation and Error Reporting (See sentry-configuration.) then the Sentry Server will try to fetch certain resources like source maps from your App. Sentry uses a bilateral token to authenticate these calles. If you set `GAETK2_SENTRY_SECURITY_TOKEN` in `appengine_config` to the same value than in the Sentry Web Page Settings section all calls from the Sentry Sertver will be authenticated automatically with a `uid` of `X-Sentry-Token@auth.gaetk2.23.nu`.

How JWTs work in gaetk2

`/gaetk2/auth/getjwt.txt` can be requested to get a JWT. To access `getjwt.txt` you have to be already authenticated by other means. The JWT will be returnesd as a plain text string. See jwt.io for more information on how JWTs are constucted.

The token obtained this way can be used to authenticate to oter parts of the gaetk2 app. This is done doing HTTP-Requests with an Authorisation-Header:

```
Authorization: bearer <your token>
```

The tokens provided by `/gaetk2/auth/getjwt.txt` are only calid for a limited time.

`AuthenticationReaderMixin` can load credentials from the tokens provided by `/gaetk2/auth/getjwt.txt`. It also can load credentials based on data provided by Auth0. More documentation is needed.

1.3.2 Authorisation

Currently gaetk2 assumes each user which is authenticated is also authorized. Needs work.

1.4 Frontend Guidelines

See `gaetk2.handers.base.BasicHandler` for generic Template Variables etc. See the **'templatetags'_** Document for filters available in your Jinja2 Templates.

Frontends are assumed to be generated using Jinja2 and Bootstrap 4. All displayed content is based in `templates/gaetk_base_bs4.html`

The usual approach is to generate one Template inherited from `gaetk_base_bs4.html` for your app where you set defaults an then inherit in all your actual templates from that and only overwrite `maincontent`.

So for example your `base_myapp.html` looks like this:

```
{% extends "gaetk_base_bs4.html" %}
{% block header %}
 My Cool Navbar
{% endblock %}
{% block secondarycontent %}
<div class="card" style="width: 18rem;">
<div class="card-body">
  <p class="card-text">navigation, current news</p>
</div>
</div>
{% endblock secondarycontent %}
```

The individual templates then just inherit from `base_myapp.html`:

```
{% extends "base_bs4.html" %}
{% block maincontent %}
Here are our most recent offers:
...
{% endblock maincontent %}
```

The main structure of the layout look like this:

Available blocks to overwrite:

- `maincontent` - where the content of your app lives. `<h1>{{title}}</h1>` is displayed above it. (The *<h1>* and title can be overwritten with `{% block title %}`)
- `secondarycontent` - sidebar style content to the right.
- `header` - usually filled with the auto-generated navbar. To hide it, use `{% block header %}{% endblock header %}`.
- `footer` - below `maincontent` and `secondarycontent`.

- page basically overwrites header, <h1>{{title}}</h1>, maincontent and secondarycontent leaving only footer.

1.4.1 Breadcrumbs

If you add something like this to your template Variables:

```
breadcrumbs = [('Market', '/'), (kundennr, '#'), (u'Aufträge', '#')]
```

There will be a list of breadcrumbs rendered above the Title.

1.4.2 Snippets

Snippets are gaetk2's stab at simple CMS functionality. You still write hardcoded HTML-Templates. But inside you can insert parts editable by your staff in the browser without the need to update the application.

This happens by adding show_snippet template tags:

```
{{ show_snippet('welcome') }}
```

When the resulting page is rendered there will be no text because the snippet has no content so far. But there should be an edit icons.

If you click on it you will be redirected to an editing page where you can change the Snippet. You can also provide a default text to be used for initial snippet content:

```
{{ show_snippet('welcome', 'Welcome to our Site!') }}
```

Todo:

- insert show_snippet into template contest to make it usable
- remove pagedown_bootstrap and replace it with something usable

---

1.4.3 Progressive enhancements

Todo:

- gaetkenhance-confirm, table
- ChiqView
- Breadcrumbs with hooks

1.4.4 Best Practices

No Tables for Definition Lists

Don't use Tables for non tabular Data. dl-horizontal (Bootstrap 3) is way to go. In Bootstrap 4 the Markup is somewhat convoluted:

```
<dl class="row">
  <dt class="col-3">AuftragsNr</dt>
  <dd class="col-9">{{ a.auftragsnr }}</dd>

  <dt class="col-3">Auftragsdatum / Status</dt>
  <dd class="col-9">{{ a.eingegangen_am|dateformat }} / {{ a.nicestatus }}</dd>
</dl>
```

Table Styling

Tagles we usually style with `class="table table-striped table-sm"`. For large rows like Product Listing with Images we use `class="table table-hover"`.

1.5 Backup and Replication Guide

The general flow ist that you do a Managed Export of your datastore entities to Google Cloud Storage. Than load that data into Google BigQuery via a load job and do all further exporting and analysis from there.

This replaces *gaetk_replication <https://github.com/hudora/gaetk_replication>* which was able export to MySQL and JSON on S3 directly although unreliably.

Following Parameters in `gaetk2_config.py` (see *gaetk2.config*) define the behaviour of managed export and loding into BigQuery.

`GAETK2_BACKUP_BUCKET` defines where in Cloud Storage the backup should be saved. Defaults to `google.appengine.api.app_identity.get_default_gcs_bucket_name()`.

`GAETK2_BACKUP_QUEUE` defines the TaskQueue to use for backup. Defaults to `default`.

`GAETK2_BIGQUERY_PROJECT` is the BigQuery Project to load data in to. If not set, no data loading will happen.

`GAETK2_BIGQUERY_DATASET` is the dataset to use for the load job. If not set, `google.appengine.api.app_identity.get_default_gcs_bucket_name()` is used.

To use the functionality, you have to add two handlers to `cron.yaml`:

```
cron:
- description: Scheduled Backup and Source for BigQuery
  url: /gaetk2/backup/
  schedule:  every day 03:01
  timezone: Europe/Berlin
- description: Backup loading into BigQuery
  url: /gaetk2/load_into_bigquery
  schedule:  every day 05:01
  timezone: Europe/Berlin
```

See *gaetk2.views.backup.BackupHandler* and `gaetk2.views.load_into_bigquery.BqReplication` for the actual implementation.

1.6 Deployment, CI & CD

1.6.1 Outline

For AppEngine Python Standard Environment you have to provide a full Setup including all (most) libraries you want to use. So *building* consists of pulling in all needed python libraries. If you use any complex Javascript building also includes running webpack to construct the needed javascript bundles.

Checking means runing code analysys tools to find bugs and ensure coding standards are adhered to.

Testing includes running *unit tests* for Python and Javascript to check single components whithout any test to live data and APIs. *Acceptance testing* we run against a complete app installed on a special App Engine *version* but with access to all life data.

Deployment is the installation of the App on Google App Engine. Be it on a Development or Production Version.

Releasing includes automated *Checking* and *Testing* of software and preparing it for *Deployment* in Production.

Production deployment is done via an blue/green schema where you deploy to an inactive version and then migrate traffic from the active version to the newly deployed version. In case of issues you can quickly migrate the traffic back. Google App engine is very well suited to this approach.

All this steps are meant to be run inside Docker Containers to ensure repeatable and stable infrastructure. Services like Circle CI or Travis CI can provide automation for this steps.

All the semi-automated and automated steps are handled via doit, a Python based *make* alternative.

1.6.2 Docker Primer

You might think of the Docker container engine as a light weight VM system. It downloads the containers (think "VM") it needs automatically from the internet and can give you shell access to the container.

The mdornseif/gae image is well suited to build, test and deploy for and to Google App Engine Python Standard Environment.

If you have Docker installed, it is easy to get a shell inside the container ready for building:

```
docker run --rm -ti mdornseif/gae bash
```

Docker containers are destroyed after each run so do not save anything important in them. To keep data permanently store it on the host system. To exchange data between the container and your host computer you can mount a directory via *-v* (mount):

```
docker run  --rm -ti -v "$(pwd)":/hostdir mdornseif/gae bash
ls /hostdir
touch /hostdir/test.txt
```

When you want to checkout something from inside the docker container you need SSH keys. It is somewhat difficult to that.

> **Warning:** The way described here is inherently insecure. Do only use it unless you are the only user on the host and the host does only run trusted processes. Also only run a single trusted container.

If you have your SSH-Key for accessing github in `~/.ssh/id_github_key` you can pass it into the container via this command:

```
docker run --rm --env CHECKOUT_KEY="`cat ~/.ssh/id_github_key`" -ti mdornseif/gae bash
```

This will make the key available inside the image under `~/.ssh/id_rsa` where git/ssh should pick it up automatically. You can put additional variables into the file `env.list` and use it via `--env-file env.list`. OR just add them to the command line via `--env NAME=valiue`.

Most usuable is `CIRCLE_REPOSITORY_URL` where you can provide the repository to be checked out on start. `CIRCLE_BRANCH` selects the branch to check out. The usual Setup would be something like this:

```
echo "CIRCLE_REPOSITORY_URL=git@github.com:myUser/myProj.git" > env.list
echo "CIRCLE_BRANCH=testing" >> env.list
docker run --rm --env-file env.list --env CHECKOUT_KEY="`cat ~/.ssh/id_github_key`" -
↪ti mdornseif/gae bash
```

1.6.3 Repository Structure

It is assumed that you work based the lines of the GitHub Flow (See also here.

- Anything in the `master` branch should be ready for production
- To work on something new, create a descriptively named branch
- regularly push your york to the server to profit from automated testing
- For help or feedback, or the branch is ready for merging, open a pull request
- Once `master` has something significant new, a release should follod imediately.

There ist a `staging` branch for reviewing features which are not ready for production. This is our way to get arround using [feature flags](https://featureflags.io).

The `hotfix` branch is for getting around usually processes in emergencies.

In addition there is a `release` branch which is meant for a final Acceptance-Check and to ensure certain steps like writing a change log and informing the user base is done.

So the branches with special meaning are:: * master - where your stable code lives, automatically deployed to http://master-dot-yourapp.appspot.com * release - where your production code lives, automatically deployed to http://release-dot-yourapp.appspot.com * staging - testing of certain features http://staging-dot-yourapp.appspot.com * hotfix - Experiments used in production http://hotfix-dot-yourapp.appspot.com

1.6.4 Checks

Todo: docker run –env-file env.list –env CHECKOUT_KEY="*cat yourkey*" -ti mdornseif/gae doit check

If you want to run a somewhat less strict code analysis, use ‘‘doit CICHECK‘.

1.6.5 Unit Tests

1.6.6 CI - Continues Integration

If you have a docker based CI system this works very well with the gaetk2 deployment strategy. For example a Circle CI configuration would look like this:

```
version: 2
defaults: &defaults
  working_directory: ~/repo/
  docker:
    # - image: circleci/python:2.7.15-node-browsers
    - image: mdornseif/gae:stable
jobs:
  build:
    <<: *defaults
    steps:
      - checkout:
          path: ~/repo
      - run: doit submodules
      - run: doit BUILD
      - run: doit CICHECK CITEST
  deploy:
    <<: *defaults
    steps:
      - checkout:
          path: ~/repo
      - run: doit submodules
      - run: doit BUILD
      # see https://circleci.com/docs/2.0/google-auth/
      # https://circleci.com/docs/1.0/deploy-google-app-engine/
      # add key at https://circleci.com/gh/hudora/huWaWi/edit#env-vars
      - run: echo $GCLOUD_SERVICE_KEY | base64 --decode --ignore-garbage > ${HOME}/
↪gcloud-service-key.json > ~/gcloud-service-key.json
      - run: gcloud auth activate-service-account --key-file ${HOME}/gcloud-service-
↪key.json
      - deploy: gcloud -q app deploy ./app.yaml --project=huwawi2 --version=$CIRCLE_
↪BRANCH --no-promote
  test_acceptance:
    <<: *defaults
    steps:
      - checkout:
          path: ~/repo
      - run: doit submodules
      - run: doit BUILD
      - run: doit CITEST_ACCEPTANCE

workflows:
  version: 2
  build-and-deploy:
    jobs:
      - build
      - deploy:
          requires:
            - build
          filters:
            branches:
              only:
                - staging
                - hotfix
                - master
                - release
      - test_acceptance:
          requires:
```

(continues on next page)

(continued from previous page)

```
            - build
            - deploy
          filters:
            branches:
              only:
                - staging
                - hotfix
                - master
                - release
```

That's all.

1.6.7 Automated Deployments

Create a Service Account at https://console.cloud.google.com/iam-admin/serviceaccounts/project?project=huwawi2 Permissions needed are *App Engine -> App Engine Deployer* and *Storage -> Storag Object Admin*. (See http://filez.foxel.org/2d1Q2W0y2E33). Download the Key as JSON, Pass it throu base64 and add it as Circle CI environment variable *GCLOUD_SERVICE_KEY* at https://circleci.com/gh/hudora/huWaWi/edit#env-vars

Also set `GAE_PROJECT`.

1.7 HOWTO - Guides

A collection of examples and best practices.

1.7.1 30x Redirect

Usually you just *raise* an 30x Exception like this:

```
from gaetk2.handlers import DefaultHandler
from gaetk2.application import WSGIApplication, Route
from gaetk2 import exc

class ExampleHandler(DefaultHandler):
    def get(self):
        raise exc.HTTP302_Found(location='/bar')

app = WSGIApplication([Route('/foo', ExampleHandler)])
```

1.7.2 404 Not Found

Like a *30x Redirect* you just raise `HTTP404_NotFound`:

```
class ExampleHandler(DefaultHandler):
    def get(self, customernumber):
        obj = Customer.get_by_id(customernumber)
        if not obj:
            raise exc.HTTP404_NotFound('')
        self.return_text('found')
```

But this common case can be handled much more elegant with *gaetk2.helpers.check404()*:

```
from gaetk2.helpers import check404

class ExampleHandler(DefaultHandler):
    def get(self, customernumber):
        obj = check404(Customer.get_by_id(customernumber))
        self.return_text('found')
```

This will *raise* `HTTP404_NotFound` whenever `obj` evaluates to `False`.

Todo:

- How to implement nice error pages

1.8 Migrating from appengine-toolkit 1 to Version2

Some suggestions on moving from Appengine Toolkit Version 1 (gaetk) to GAETK2. Obviously you need to add gaetk2 to your source tree:

```
git submodule add https://github.com/mdornseif/appengine-toolkit2.git lib/appengine-
↪toolkit2
```

First get all the *Error Handling Guide* goodness from GAETK2.

Just ensure that you import the right WSGI Application:

```
from gaetk2.application import WSGIApplication
....
application = WSGIApplication([ ...
```

Often you might have to replace *make_app* by *WSGIApplication*.

With that you did the most important change. GAETK1 and GAETK2 get along quite well so you might leave it at that for a moment.

1.8.1 Change configuration-files

In `app.yaml` make sure `lib/appengine-toolkit2/include.yaml` is included and `jinja2` is not included via Google (we need jinja 2.10, Google provides 2.6):

```
includes:
- lib/appengine-toolkit2/include.yaml
...
libraries:
- name: ssl
  version: latest
- name: pycrypto
  version: "latest"
- name: numpy
  version: "1.6.1"
- name: PIL
  version: latest
```

Usually you can remove the `skip_files` section, because `lib/appengine-toolkit2/include.yaml` should contain all the necessary exclusions.

Your `requirements.txt` should end with `-r lib/appengine-toolkit2/requirements-lib.txt`.

At the top of your `appengine_configuration.py` include this:

```
# load gaetk2 bootstrap code without using `sys.path`
import imp
(fp, filename, data) = imp.find_module('boot', ['./lib/appengine-toolkit2/gaetk2/'])
imp.load_module('gaetk_boot', fp, filename, data)
```

This will set up paths as needed. To get error- and session-handling and add the following lines at the end of `appengine_config.py`.

from gaetk2.wsgi import webapp_add_wsgi_middleware # pylint: disable=W0611

Various configuration needs to be done in `gaetk2_config.py`. Try `grep GAETK2_ >> gaetk2_config.py`. Minimal contents would be:

```
GAETK2_SECRET='13f221234567890fae123-c0decafe'
GAETK2_TEMPLATE_DIRS=['./templates', './lib/CentralServices/templates']
```

1.8.2 Backup and BigQuery Loading

Remove */gaetk_replication/bigquery/cron* and */gaetk/backup/* from `cron.yaml` and add instead:

```
cron:
- description: Scheduled Backup and Source for BigQuery
  url: /gaetk2/backup/
  schedule:  every day 03:01
  timezone: Europe/Berlin
- description: Backup loading into BigQuery
  url: /gaetk2/load_into_bigquery
  schedule:  every day 05:01
  timezone: Europe/Berlin
```

Be sure to include the handlers in `app.yaml`:

```
includes:
- lib/appengine-toolkit2/include.yaml
```

Add configuration to `gaetk2_config.py`:

```
GAETK2_BACKUP_BUCKET = 'my-backups-eu-nearline'
GAETK2_BACKUP_QUEUE = 'backup'
GAETK2_BIGQUERY_PROJECT = 'myproject'
GAETK2_BIGQUERY_DATASET = 'mydataset'
```

Then check if you can remove *gaetk_replication*. See *Backup and Replication Guide* for further Information on how it all is supposed to play together.

1.8.3 Replace Imports

Replace this:

```
from gaetk.helpers import check404
from google.appengine.ext.deferred import defer
from gaetk.infrastructure import taskqueue_add_multi
from gaetk.infrastructure import query_iterator
from gaetk.tools import slugify
from huTools import hujson2
from huTools.unicode import deUmlaut
from huTools import cache
from gaetk.tools import hd_cache
from huTools.calendar.tools import date_trunc
from huTools.calendar.formats import convert_to_date, convert_to_datetime
from gaetk import configuration
```

With this:

```
from gaetk2.helpers import check404
from gaetk2.taskqueue import defer
from gaetk2.taskqueue import taskqueue_add_multi
from gaetk2.datastore import query_iterator
from gaetk2.tools.unicode import slugify
from gaetk2.tools import hujson2
from gaetk2.tools.unicode import de_umlaut
from gaetk2.tools.caching import lru_cache, lru_cache_memcache
from gaetk2.tools.caching import lru_cache, lru_cache_memcache
from gaetk2.tools.datetools import date_trunc
from gaetk2.tools.datetools import convert_to_date, convert_to_datetime
from gaetk2 import config as configuration
```

s/import gaetk.handler/from gaetk2 import exc/ /raise gaetk.handler.HTTP/raise exc.HTTP/

1.8.4 Use a local logger

At the top of each module create a local logger instance:

```
logger = logging.getLogger(__name__)
```

Then replace calls to `logging.info()` et. al. with calls to `logger.info()` et. al.

1.8.5 Change your views / handlers

Todo:

- if you used the *get_impl()* pattern to wrap your handler functions, you don't need that anymore. The often used *read_basedata()* can be moved into *method_preperation_hook()*.
- Replace *self.is_admin()* with *self.is_staff()* (or *self.is_sysadmin()*).
- **attrencode to xmlattr:** `<meta property="og:price:amount" content="{{ preis|euroword|attrencode }}" />` to `<meta property="og:price:amount" {{ {'content': preis|euroword}|xmlattr }} />`
- `authchecker` to `authorisation_hook`

Replace *default_template_vars()* with *build_context()* - no *super()* calls necessary anymore.

This:

```
def default_template_vars(self, uservalues):
    u"""Default variablen für Breadcrumbs etc."""
    myvalues = dict()
    myvalues.update(super(AbstractAuiHandler, self).default_template_vars(uservalues))
    myvalues.update(
        navsection='artikel',
        artikelnavsection=getattr(self, 'artikelnavsection', ''),
    )
    # stellt sicher, dass die Werte aus `uservalues` Vorrang haben
    myvalues.update(uservalues)
    return myvalues
```

Becomes that:

```
def build_context(self, uservalues):
    u"""Add Messages to context."""
    myvalues = dict(
        navsection='artikel',
        artikelnavsection=getattr(self, 'artikelnavsection', '')
    )
    myvalues.update(uservalues)
    return myvalues
```

Authentication has changed significanty. *authchecker()* now handled by *pre_authentication_hook()*, *authentication_hook* and *authorisation_hook()*.

This:

```
def authchecker(self, method, *args, **kwargs):
    """Sicherstellen, das Sources diese Seiten nicht anschauen dürfen."""
    super(MasterdataHomepage, self).authchecker(method, *args, **kwargs)
    if self.credential.get_typ() == 'source':
        raise exc.HTTP403_Forbidden('Dies ist ein reiner Kundenbereich')
```

Becomes that:

```
def authorisation_hook(self, method_name, *args, **kwargs):
    u"""Sicherstellen, dass nur kunden diese seite sehen düfen."""
    if self.credential.get_typ() == 'source':
        raise exc.HTTP403_Forbidden('Dies ist ein reiner Kundenbereich')
```

See filters-gaetk1 on how to handle Templates.

1.8.6 Templates

Todo:

- Autoescaping

1.8.7 Migrate to Bootstrap 4

See Migrating to v4 for general guidelines. See *Frontend Guidelines* for the desired results.

Usually you want to use `{% extends "gaetk_base_bs4.html" %}`.

Breadcrubs are now implemented by gaetk. See breadcrumbs.

Takeaways:

```
* ``.pull-left`` and ``.pull-right`` become ``.float-left`` and ``.float-right``.
* ``.btn-default`` becomes ``.btn-secondary``
* ``.label`` becomes ``.badge`` and ``.label-default`` becomes ``.badge-secondary``.
```

1.9 Build Commands & Deployment

gaetk2 based Application still use old school Makefiles as their main interface for comand line building, testing and deploying. But because make is too complex for young people to understand we use Python helpers in the background and let grunt and webpack do some of the work. This is not optimal and slowish.

This is not implemented so far in gaetk2. It has to be ported over from gaetk1.

1.9.1 Commands

doit openlogs open App Engine logfiles in Browser

doit deploy installs the current checkout as a developer specific version and opens it in the browser

doit build builds assets (Javascript, CSS) and other dependencies for development

doit mergeproduction process to merge *master* into *production*

doit check TBD

doit staging_deploy TBD

testing_deploy TBD

testing_test TBD

doit production_clean_checkout TBD

doit production_build like *doit built* but produces minified, optimized versions

doit production_deploy TBD

Parameter -a, –always-execute execute even if dependencies are up to date

Parameter -v ARG, –verbosity=ARG 0-2

Parameter -s, –single Execute only specified tasks ignoring their dependencies

doit doit info -s <task> Show on what the task depends

1.10 gaetk2 package

1.10.1 Subpackages

gaetk2.config module

Framework Configuration

gaetk2 expects it information to be found in `gaetk2_config.py`. Minimal content is:

```
GAETK2_SECRET='*some random value*'
```

`GAETK2_SECRET` is used for Session generation etc. Try something like `(dd if=/dev/urandom count=1000 ) | shasum` to get a nice value for *GAETK2_SECRET*.

Todo: Document other configuration values.

Runtime Information

The functions *get_environment()*, *get_release()* and *get_revision()* allow the caller to find out about the deployment.

Runtime Configuration

get_config() and *set_config()* allow you to set datastore backed configration values. These are saved via `gaetk_Configuration`. NDB caching applies so keep in mind that changing the values in the datastore via the Google App Engine Admin Console does not update this cache.

Todo: Document the view to change runtime configuration values.

Module contents

`gaetk2.config.`**`get_release`**(**args*, ***kwds*)
Get the *tagged version* of the current deployment.

> Which usually means the first line `gaetk2-release.txt`. E.g. `v180228-cg89bd1-production-dot-application.appspot.com`.

Results are cached locally (maxsize=1 ttl=43200)

`gaetk2.config.`**`get_version`**()
Do not use this.

`gaetk2.config.`**`get_revision`**(**args*, ***kwds*)
Get the git SHA1 revision of the current deployment.

> Get the first line of `gaetk2-revision.txt`. E.g. `14006259d78fa918054f774d20480b52e38c4707`.

Results are cached locally (maxsize=1 ttl=43200)

`gaetk2.config.`**`get_userversion`**(**args*, ***kwds*)
Return the User-Visible Version (eg *2018.11.3*).

Results are cached locally (maxsize=1 ttl=43200)

`gaetk2.config.`**`get_productiondomain`**(**args*, ***kwds*)

gaetk2.config.**get_environment**()
Returns `production`, `staging`, `testing` or `development` depending on the Server Name.

See *production version*, *staging version*, testing version, and *production version* for meaning.

gaetk2.config.**is_production**()
checks if we can assume to run on a production version instance.

... unless called by the resttest-client. See *production version* what this means.

There are suble differences to *get_environment()* - read the code for details.

gaetk2.config.**get_config**(*key*, *default=None*)
Get configuration value for key

gaetk2.config.**set_config**(*key*, *value*)
Set configuration value for key

gaetk2.config.runtime.**get_config**(*key*, *default=None*)
Get configuration value for key

gaetk2.config.runtime.**set_config**(*key*, *value*)
Set configuration value for key

gaetk2.views - pre made request handlers

class gaetk2.views.default.**RobotTxtHandler**(**args*, ***kwargs*)
Bases: *gaetk2.handlers.DefaultHandler*

Handler for *robots.txt*.

Assumes that only the default version should be crawled. For the default version the contents of the file *robots.txt* are sent. For all other versions *Disallow: /* is sent.

get()
Deliver robots.txt based on application version.

class gaetk2.views.default.**VersionHandler**(**args*, ***kwargs*)
Bases: *gaetk2.handlers.DefaultHandler*

Version Handler - allows clients to see the git revision currently running.

get()
Returns the current version.

class gaetk2.views.default.**RevisionHandler**(**args*, ***kwargs*)
Bases: *gaetk2.handlers.DefaultHandler*

Version Handler - allows clients to see the git revision currently running.

get()
Returns the current revision.

class gaetk2.views.default.**ReleaseHandler**(**args*, ***kwargs*)
Bases: *gaetk2.handlers.DefaultHandler*

Release Handler - allows clients to see the git release currently running.

get()
Returns the current release.

class gaetk2.views.default.**BluegreenHandler**(**args*, ***kwargs*)
Bases: *gaetk2.handlers.DefaultHandler*

Allows clients to see the if blue or green is currently running.

get()
Returns the current release.

class gaetk2.views.default.**WarmupHandler**(**args*, ***kwargs*)
Bases: *gaetk2.handlers.DefaultHandler*

Initialize AppEngine Instance.

warmup()
Common warmup functionality. Loads big/slow Modules.

get()
Handle warm up requests.

class gaetk2.views.default.**HeatUpHandler**(**args*, ***kwargs*)
Bases: *gaetk2.handlers.DefaultHandler*

Try to import everything ever referenced by an url.

get()
Import all Modules.

class gaetk2.views.backup.**BackupHandler**(**args*, ***kwargs*)
Bases: *gaetk2.handlers.DefaultHandler*

Handler to start scheduled backups.

get()
To be called by cron and only by cron.

WSGI Application

The code in here is basically a vanilla

webapp2.WSGIApplication class with additional error handling capabilites. See *Error Handling Guide* for a reference.

Also Route is included for your convenience.

For example in app.yaml add:

```
handlers:
    - url: /
      script: home.app
```

`home.py` should look like this:

```
from gaetk2.handlers import DefaultHandler
from gaetk2.application import WSGIApplication, Route

class HomeHandler(DefaultHandler):
    def get(self):
        self.return_text('it worked')

app = WSGIApplication([Route('/', handler=HomeHandler)])
```

gaetk2.application package

class gaetk2.application.**WSGIApplication**(*routes=None*, *debug=False*, *config=None*)

Overwrite exception handling.

For further information see the paret class at http://webapp2.readthedocs.io/en/latest/api/webapp2.html#webapp2.WSGIApplication

handle_exception(*request*, *response*, *e*)

Handles a uncaught exception occurred in `__call__()`.

Uncaught exceptions can be handled by error handlers registered in `error_handlers`. This is a dictionary that maps HTTP status codes to callables that will handle the corresponding error code. If the exception is not an `HTTPException`, the status code 500 is used.

The error handlers receive (request, response, exception) and can be a callable or a string in dotted notation to be lazily imported.

If no error handler is found, the exception is re-raised.

Parameters

- **request** – A `Request` instance.
- **response** – A `Response` instance.
- **e** – The uncaught exception.

Returns The returned value from the error handler.

default_exception_handler(*request*, *response*, *exception*)

Exception aufbereiten und loggen.

get_sentry_addon(*request*)

Try to extract additional data from the request for Sentry after an Exception tootk place.

Parameters **request** – The Request Object

Returns a dict to be sent to sentry as *addon*.

classify_exception(*request*, *exception*)

Based on the exception raised we classify it for logging.

We not only return an HTTP Status code and *level*, but also a *fingerprint* and *dict* of *tags* to help snetry group the errors.

setup_logging(*request*, *response*)

Provide sentry early on with information from the context.

Called at the beginning of each request. Some information is already set in *sentry.py* during initialisation.

fix_unicode_headers(*response*)

Ensure all Headers are Unicode.

class gaetk2.application.**Route**(*template*, *handler=None*, *name=None*, *defaults=None*, *build_only=False*, *handler_method=None*, *methods=None*, *schemes=None*)

A route definition that maps a URI path to a handler.

The initial concept was based on **'Another Do-It-Yourself Framework'_**, by Ian Bicking.

defaults = None

Default parameters values.

methods = None
Sequence of allowed HTTP methods. If not set, all methods are allowed.

schemes = None
Sequence of allowed URI schemes. If not set, all schemes are allowed.

regex
Lazy route template parser.

match(*request*)
Matches this route against the current request.

Raises `exc.HTTPMethodNotAllowed` if the route defines *methods* and the request method isn't allowed.

See also:

`BaseRoute.match()`.

build(*request*, *args*, *kwargs*)
Returns a URI for this route.

See also:

`Router.build()`.

WSGI Middlewares

Adds Session- and Error-Handling to your App Engine Infrastructure. See *Error Handling Guide*.

gaetk2.wsgi package

`gaetk2.wsgi.`**wrap_errorhandling**(*application*)
If Sentry is to be activated wrap the app in it.

`gaetk2.wsgi.`**wrap_session**(*application*)
Put gaesession around the app.

`gaetk2.wsgi.`**webapp_add_wsgi_middleware**(*application*)
Called with each WSGI application initialisation.

The most common usage pattern is to just import it in `appengine_config.py`:

```
from gaetk2.wsgi import webapp_add_wsgi_middleware
```

gaetk2.handlers - WSGI Request Handlers

The *gaetk2.handlers* package aims to be the working horse on which you build your application. Instead of a monolytic approach like GAETK1 we work with mixins here. All of this is based on webapp2 request handlers. Basically you overwrite `get() post()`. In there you do a `self.response.write('foo')`. gaetk2 provides you with the convinience functions *return_text()* for simple replies and *render()* for rendering jinja2 templates.

BasicHandler provides basic functionality and template variables. *JsonBasicHandler* is specialized to produce JSON.

Usually you would use *DefaultHandler* for your public pages. This includes *BasicHandler*, *MessagesMixin*, `AuthenticationReaderMixin`. For JSON output you would use *JsonBasicHandler* based on *JsonBasicHandler* and `AuthenticationReaderMixin`.

If you want to ensure Users are authenticated use *AuthenticatedHandler* which extends *DefaultHandler* with AuthenticationRequiredMixin

Mix-Ins

Mix-Ins provide specialized functionality to handlers. They are mostly implemented using gaetk2.handlers.basic.BasicHandler *Hook Mechanism*. Some Mix-Ins provide just methods manually called by your method handlers.

- *PaginateMixin* provides pagination of ndb-Queries.
- *gaetk2.handlers.mixins.messages.MessagesMixin* provide short term feedback to a user displayed on the "next page". The concept is similar to flask's "Message flashing".

General Flow

Based on you app.yaml Google App Engine executes a WSGI Application. Usually the Application is wraped by a WSGI middleware. This happens via webapp_add_wsgi_middleware() in appengine_config.py. Usually via *gaetk2.wsgi* session management and error reporting (see *Error Handling Guide*) are added.

Usually the *WSGI Application* is gaetk2.applicationWSGIApplication which finds out the handler to call via the *Route*, calls the handler and does a lot of error handling and information gathering. No user-serviceable inside.

All your handlers should inherit their main functionality from gaetk2.handler.base.BasicHandler which is a heavily modified webapp2.RequestHandler.

Usually you will overwrite get(), post(), head(), options(), put(), delete() or trace() to handle the respective HTTP-Methods. See the webapp2 Request Handler Documentation for an overview. These functions access the request via self.request - see webapp2 Request data Documentation. Output is generated via self.respomse or returning a response Object - see webapp2 Documentation on Returned values.

To allow easy subclassing and Multiple inheritance BasicHandler will ensure that a list of hook function in all parent classes is called. Before the request the following functions are called:

1. pre_authentication_hook() - done before any authentication is done. E.g. redirecting moved URLs. __init__()-like setup.
2. authentication_preflight_hook() - used by AuthenticationReaderMixin to load authentication Information from the request headers.
3. authentication_hook() - to handle and ensure Authentication. Used by AuthenticationRequiredMixin to ensure that the current user is authenticated.
4. authorisation_hook() - to check that the authenticated user is allowed to do the request.
5. method_preperation_hook() - this is probably the most often overwritten hook. It is meant to load data for a set of derived handlers. See below for examples.
6. **The HTTP request method** - usually get() or post()
7. response_overwrite() - this is not a hook so without super() magic only the top level implementation of the method resolution order (MRO) is called - like usual in Python classes. This is used to transform the response to the client. For example in JsonBasicHandler.
8. finished_hook() - called, even if a HTTP-Exception with code < 500 happens. Used to flush buffers etc.
9. handle_exception() - is called in case of an Exception. See webapp2 documentation.

When you call `render()` `build_context()` in all parent classes and Mix-Ins is called to construct the render context.

The following Sample Implementation implements (parts) of a shopping cart to illustrate usage:

```
from gaetk2 import exc
from gaetk2.handlers import AuthenticatedHandler
from gaetk2.application import WSGIApplication, Route

class AbstractSiteHandler(AuthenticatedHandler):
    """General stuff used all over the site."""

    def pre_authentication_hook(self, method_name, *args, **kwargs):
        self.analytics = analytics.Client()

    def build_context(self, values):
        if 'title' not in values:
            values['title'] = 'My Cool Site'
        return values

    def finished_hook(self, *args, **kwargs):
        self.analytics.flush()

class BaseCartHandler(AbstractSiteHandler):
    """Functionality used for the cart."""

    def method_preperation_hook(self, method_name, *args, **kwargs):
        self.cart = self.session.get('cart3', {'items': []})

    def finished_hook(self, *args, **kwargs):
        if hasattr(self, 'cart'):
            # might be missing on authentication failures
            self.session['cart3'] = self.cart  # this marks session as dirty
            # store cart len so client side code can read it
            cart3_len = str(len(self.cart.get('items', [])))
            self.response.set_cookie('cart3_len', cart3_len, max_age=7*24*60*60)

    def handle_exception(self, e, debug):
        """On Exceptions flush the cart to provide a clean 'reboot'."""
        if not getattr(e, 'code', 500) < 500:
            # no fluch on redirect etc.
            self.flush_cart()
        raise


class AddToCartHandler(BaseCartHandler):

    def get(self):
        sku = self.request.get('sku', '')
        menge = int(self.request.get('menge_%s' % sku, 1))
        self.cart['items'].append((sku, menge))
        raise exc.HTTP302_Found(location='/cart3')


class UpdateCartHandler(BaseCartHandler):

    def get(self):
        self.update(self.request.GET)
```

(continues on next page)

(continued from previous page)

```
        raise exc.HTTP302_Found(location='/cart3')

    def post(self):
        self.update(self.request.POST)
        if self.request.POST.get('action') == 'checkout':
            raise exc.HTTP302_Found(location='/cart3/checkout/')
        raise exc.HTTP302_Found(location='/cart3')


class CheckoutHandler(BaseCartHandler):

    def get(self):
        self.render(
            dict(title='Your cart', cart=self.cart),
            'cart3/checkout.html')
        self.analytics.track('Checkout Started')

    def post(self):
        orderid = generate_order(self.cart)
        self.cart = {'items': []}
        defer(inform_about_order, orderid)
        # from :class:`~gaetk2.handlers.mixins.messages.MessagesMixin`
        self.add_message(
            'success', jinja2.Markup('Order {} created.'.format(orderid)))
        self.track('Order Completed')
        raise exc.HTTP302_Found(location='/')


application = WSGIApplication([
    Route('/cart3/addtocart/', AddToCartHandler),
    Route('/cart3/updatecart/', UpdateCartHandler),
    Route('/cart3/checkout/', CheckoutHandler),
    Route('/cart3/', ShowCartHandler),
    Route('/cart3.<fmt>', ShowCartHandler),
])
```

An other common usage is that you have a group of pages with a common URL prefix and all of them extracting information from the UlRL, e.g. customer number. So you might have:

```
* /k/<userid>/dettings/
* /k/<userid>/orders/
* /k/<userid>/orders/<orderid>
* /k/<userid>/orders/<orderid>/invoice.pdf
```

TBD

gaetk2.handlers package

class gaetk2.handlers.**DefaultHandler**(**args*, ***kwargs*)
Handle Requests and load self.credential if Authentication is provided.

class gaetk2.handlers.**JsonHandler**(**args*, ***kwargs*)
Send JSON data to client and load self.credential if Authentication is provided.

class gaetk2.handlers.**AuthenticatedHandler**(**args*, ***kwargs*)

class gaetk2.handlers.**AuthenticatedJsonHandler**(**args*, ***kwargs*)

class gaetk2.handlers.base.**BasicHandler**(**args*, ***kwargs*)

Generic Handler functionality.

You usually overwrite `get()` or `post()` and call *render()* in there. See gaetk2.handlers for examples.

For Returning Data to the user you can access the *self.response* object or use *return_text()* and *render()*. See *get_jinja2env()* to understand the jinja2 context being used.

Helper functions are *abs_url()* and `is_production()`.

See also:

is_sysadmin(), *is_staff()* and *has_permission()* are meant to work with `AuthenticationReaderMixin` for `self.request` see webapp2 Documentation

request
See webapp2 documentation

credential
authenticated user, see `AuthenticationReaderMixin`

session
current session which is based on https://github.com/dound/gae-sessions.

default_cachingtime
None or int – Class Variable. Which cache headers to generate, see *_set_cache_headers()*.

Note: gaetk2 adds various variables to the template context. Other mixins provide additional template variables. See the Index genindex under "Template Context" to get an overview.

These Template Variables are provided:

- request
- credential
- is_staff (self.is_staff())
- is_sysadmin (self.is_sysadmin())
- gaetk_production (is_production())
- gaetk_development (is_development())
- gaetk_release (get_release())
- gaetk_app_name (gaetkconfig.APP_NAME)
- gaetk_gae_version (CURRENT_VERSION_ID)
- gaetk_sentry_dsn (gaetkconfig.APP_NAME)
- gaetk_logout_url
- gaetk_path (self.request.path)

Warning: *BasicHandler* implements a rather unusual way to implement Multi-Inherance/Mix-Ins. Instead of insisting that every parent class and everty Mix-In implements all possible methods and calls super on them *BasicHandler* uses a custom dispatch mechanism which calls all methods in all parent and sibling classes.

The following functions are called on all parent and sibling classes:

- *pre_authentication_hook()*.
- *authentication_preflight_hook()*.
- *authentication_hook()*.
- *authorisation_hook()*.
- *method_preperation_hook()*.
- *finished_hook()* - called even if a `exc.HTTPException` < 500 occurs.

build_context() is special because the output is "chained". So the rendering is done with something like the output of `Child.build_context(Parent.build_context(MixIn.build_context({})))`

response_overwrite() and *finished_overwrite()* can be overwritten to provide special functionality like in *JsonBasicHandler*.

You are encouraged to study the source code of *BasicHandler*!

abs_url(*url*)

Converts an relative into an qualified absolute URL.

Parameters **url** (*str*) – an path to a web resource.

Returns A fully qualified url.

Return type str

Example

```
>>> BasicHandler().abs_url('/foo')
'http://server.example.com/foo'
```

is_sysadmin()

Checks if the current user is logged in as a SysOp/SystemAdministrator.

We use various souces to deterine the Status of the user. Returns *True* if:

- google.appengine.api.users.is_current_user_admin()
- the request came from *127.0.0.1* local address
- *self.credential.sysadmin == True*

Returns the status of the currently logged in user.

Return type boolean

is_staff()

Returns if the current user is considered internal.

This means he has access to not only his own but to all settings pages, etc.

- *is_sysadmin()*
- *self.credential.staff == True*

Returns the status of the currently logged in user is considered internal.

Return type boolean

has_permission(*permission*)
Checks if user has a given permission.

render(*values*, *template_name*)
Render a Jinja2 Template and write it to the client.

If rendering takes an unusual long time this is logged.

Parameters

- **values** (*dict*) – variables for template context.
- **template_name** (*str*) – name of the template to render.

See also:

`build_context()`also provides data to the template context and is often extended by plugins. See :class:`BasicHandler()` docsting for standard template variables.

return_text(*text*, *status=200*, *content_type=u'text/plain'*, *encoding=u'utf-8'*)
Quick and dirty sending of some plaintext to the client.

Parameters

- **text** (*str or unicode*) – Data to be sent to the cliend. A NEWLINE is appended.
- **status** (*int*) – status code to be sent to the client. Defaults to 200.
- **content_type** – to be sent to the client in respective header.
- **encoding** – to be used when sending to the client.

build_context(*values*)
Helper to provide additional request-specific values to HTML Templates.

Will be called on all Parents and MixIns, no *super()* needed.

def build_context(self, values): myvalues = dict(navsection='kunden', ...) myvalues.update(values) return myvalues

_add_jinja2env_globals(*env*)
Helper to provide additional Globals to Jinja2 Environment.

This should be considered one time initialisation.

Example:: env.globals['bottommenuurl'] = '/admin/' env.globals['bottommenuaddon'] = ' Admin' env.globals['profiler_includes'] = gae_mini_profiler.templatetags.profiler_includes

debug(*message*, **args*)
Detailed logging for development.

This logging will only happen, if `WSGIApplication` was initialized with `debug=True`. Is meant for local inspection of the stack during development. Messages are prefixed with the method name from where they are called.

pre_authentication_hook(*method_name*, **args*, ***kwargs*)
Might do redirects before even authentication data is loaded.

Called on all parent and sibling classes.

authentication_preflight_hook(*method_name*, **args*, ***kwargs*)
Might load Authentication data from Headers.

Called on all parent and sibling classes.

authentication_hook(*method_name*, **args*, ***kwargs*)
Might verify Authentication data.

Called on all parent and sibling classes.

authorisation_hook(*method_name*, **args*, ***kwargs*)
Might check if authenticated user is authorized.

Called on all parent and sibling classes.

method_preperation_hook(*method_name*, **args*, ***kwargs*)
Is Called just before GEP, POST, PUT, DELETE etc. is called.

Used to provide common data in child classes. E.g. to set up variables, load Date etc.

response_overwrite(*response*, *method*, **args*, ***kwargs*)
Function to transform response. To be overwritten.

finished_hook(*method*, **args*, ***kwargs*)
To be called at the end of an request.

finished_overwrite(*response*, *method*, **args*, ***kwargs*)
Function to allow logging etc. To be overwritten.

clear_session()
Terminate the session reliably.

get_jinja2env()
Initialise and return a jinja2 Environment instance.

_jinja2_exception_handler(*traceback*)
Is called during Jinja2 Exception processing to provide logging.

_render_to_fd(*values*, *template_name*, *fd*)
Sends the rendered content of a Jinja2 Template to Output.

Per default the template is provided with output of `build_context(values)`.

_set_cache_headers(*caching_time=None*)
Set Cache Headers.

Parameters **caching_time** (*None or int*) – the number of seconds, the result should be cachetd at frontend caches. `None` means no Caching-Headers. See also *default_cachingtime*. *0* or negative Values generate an comand to disable all caching.

_call_all_inherited(*funcname*, **args*, ***kwargs*)
In all SuperClasses call *funcname* - if it exists.

_reduce_all_inherited(*funcname*, *initial*)
In all SuperClasses call *funcname* with the output of the previus call.

dispatch()
Dispatches the requested method fom the WSGI App.

Meant for internal use by the stack.

handle_exception(*exception*, *debug*)
Called if this handler throws an exception during execution.

The default behavior is to re-raise the exception to be handled by `WSGIApplication.handle_exception()`.

Parameters

- **exception** – The exception that was thrown.

- **debug_mode** – True if the web application is running in debug mode.

Returns response to be sent to the client.

class gaetk2.handlers.base.**JsonBasicHandler**(**args*, ***kwargs*)

Handler which is specialized for returning JSON.

Excepts the method to return

- dict(), e.g. *{'foo': bar}*

Dict is converted to JSON. *status* is used as HTTP status code. *cachingtime* is used to generate a *Cache-Control* header. If *cachingtime is None*, no header is generated. *cachingtime* defaults to 60 seconds.

serialize(*content*)

convert content to JSON.

response_overwrite(*response*, *method*, **args*, ***kwargs*)

Function to transform response. To be overwritten.

class gaetk2.handlers.mixins.paginate.**PaginateMixin**

Show data in a paginated fashion.

Call *paginate()* in your Request-Method handler.

Example

Build a view function like this:

```
from ..handlers import AuthenticatedHandler
from ..handlers.mixins import PaginateMixin

class MyView(AuthenticatedHandler, PaginateMixin):
    def get(self):
        query = MyModel.query().order('-created_at)
        template_values = self.paginate(query)
        self.render(template_values, 'template.html')
```

Your `template.html` could look like this:

```
<ul>
  {% for obj in object_list %}
    <li>{{ obj }}</li>
  {% endfor %}
</ul>

{{ paginator4 }}
```

The `{{ paginator4 }}` expression renders a Bootstrap 4 Paginator object. If you dont want that you can add your own links:

```
{% if prev_objects %}
  <a href="?{{ prev_qs }}">&larr; Prev</a>
{% endif %}
{% if next_objects %}
  <a href="?{{ next_qs }}">Next &rarr;</a>
{% endif %}
```

paginate(*query*, *defaultcount=25*, *datanodename=u'objects'*, *calctotal=True*, *formatter=None*)

Add NDB-based pagination to Views.

Provides a template environment by calling `self.paginate()`.

Parameters

- **query** – a ndb query object
- **defaultcount** (*int*) – how many items to display per page
- **datanodename** (*string*) – name of template variable to hold the entties
- **calctotal** (*boolean*) – do you want to provide the total number of entities
- **formatter** – function to transform entities for output.

formatter is called for each object and can transform it into something suitable. If no *formatter* is given and objects have a *as_dict()* method, this is used for formating.

if *calctotal* == *True* then the total number of matching rows is given as an integer value. This is a ecpensive operation on the AppEngine and results might be capped at 1000.

datanodename is the key in the returned dict, where the Objects resulting form the query resides.

defaultcount is the default number of results returned. It can be overwritten with the HTTP-parameter *limit*.

We handle the additional query parameters `start`, `cursor`, `cursor_start` from the HTTP-Request to note what is currently displayed. `limit` can be used to overwrite `defaultcount`.

The *start* HTTP-parameter can skip records at the beginning of the result set.

If the *cursor* HTTP-parameter is given we assume this is a cursor returned from an earlier query.

Returns paginator4 paginator.current_start (int): paginator.prev_objects (boolean): paginator.prev_qs (string): paginator.prev_start (int): paginator.next_objects (boolean): paginator.next_start (int): paginator.next_qs (string): paginator.limit (int): paginator.total (int): paginator.cursor (string):

`paginator4` is generated by rendering *gaetk_fragments/PaginateMixin.paginator.html* with the other return values. You can overwrite the output by providing your own *gaetk_fragments/PaginateMixin.paginator.html* in your search path.

..image:: http://filez.foxel.org/3w330i0Z0T36/Image%202018-04-10%20at%207.48.22%20AM.jpg

See also:

http://mdornseif.github.com/2010/10/02/appengine-paginierung.html

http://blog.notdot.net/2010/02/New-features-in-1-3-1-prerelease-Cursors

http://code.google.com/appengine/docs/python/datastore/queryclass.html#Query_cursor

---

Note: Somewhat obsoleted by *listviewer*.

---

class gaetk2.handlers.mixins.messages.**MessagesMixin**

MessagesMixin provides the possibility to send messages to the user.

Like Push-Notifications without the pushing.

add_message(*typ*, *text*, *ttl=15*)

Sets a user specified message to be displayed to the currently logged in user.

typ can be *error*, *success*, *info* or *warning text* is the text do be displayed *ttl* is the number of seconds after we should stop serving the message.

If you want to pass in HTML, you need to use *jinja2.Markup([string])*.

build_context(*uservalues*)
Add Messages to context.

class gaetk2.handlers.mixins.multirender.**MultirenderMixin**
Provide rendering for a variety of formats with minimal code.

For the three major formats HTML, JSON, CSV and XML und you can get away with virtually no code.

Still nowadays we discourage the habit of massaging a single view into providing different formats of the same data.

multirender(*fmt*, *data*, *mappers=None*, *contenttypes=None*, *filename='download'*, *defaultfmt='html'*, *html_template='data'*, *html_addon=None*, *xml_root='data'*, *xml_lists=None*, *tabular_datanodename='objects'*)
Send Data formated in different ways to the client.

Some real-world view method might look like this:

URL matches '/empfaenger/([A-Za-z0-9_-]+)/rechnungen.?(json|xml|html)?', def get(self, kundennr, fmt):

query = models.Rechnung.all().filter('kundennr = ', kundennr) values = self.paginate(query, 25, datanodename='rechnungen') self.multirender(fmt, values,

filename='rechnungen-%s' % kundennr, html_template='rechnungen.html', tabular_datanodename='rechnungen')

/empfaenger/12345/rechnungen and */empfaenger/12345/rechnungen.html* will result in *rechnungen.html* beeing rendered. */empfaenger/12345/rechnungen.json* results in JSON being returned with a *Content-Disposition* header sending it to the file *rechnungen-12345.json*. Likewise for */empfaenger/12345/rechnungen.xml*. If you add the Parameter *disposition=inline* no Content-Desposition header is generated.

If you use fmt=json with a *callback* parameter, JSONP is generated. See http://en.wikipedia.org/wiki/JSONP#JSONP for details.

If you give a dict in *html_addon* this dict is additionaly passed the the HTML rendering function (but not to the rendering functions of other formats).

You can give the *xml_root* and *xml_lists* parameters to provide *dict2xml()* with defenitions on how to name elements. See the documentation of *roottag* and *listnames* in dict2xml documentation.

For tabular formats like XLS and CSV we assume that *data[tabular_datanodename]* contains a list of dicts to be rendered.

For more sophisticated layout you can give customized mappers. Using functools.partial is very helpfiull for thiss. E.g.

from functools import partial multirender(fmt, values,

mappers=dict(xml=partial(dict2xml, roottag='response',

listnames={'rechnungen': 'rechnung', 'odlines': 'odline'}, pretty=True),

html=lambda x: '<body><head><title>%s</title></head></body>' % x))

gaetk2.jinja_filters module - template filters

These filters do a lot of formatting and conversion. They are Build with German localisation and HTML in mind to extend Jinja's own filters

Use them like this in your templates:

```
{{ body|markdown }}
<div class="{{ obj.designator|cssencode }}">
```

If you use gaetk2.handlers these filters are made available automatically. If not youcan include them via *register_custom_filters()*.

Spacing Issues

We currently use u'u202F' NARROW NO-BREAK SPACE U+202F to separate numbers. Unfortunately this is missing in most fonts and not well supported in browsers.

Contents

Services provided

Access Control

These Access Control filters are somewhat more involved because they need the cooperation of the rest of gaetk2. They are meant to show certain parts of a template only to certain users.

See `gaetk2.handler.base.is_staff()` and `gaetk.models.gaetk_Credential` for further Reference.

- *onlystaff()* - display content only if the currently logged in user `is_staff()`.

Todo: This functionality is not finalized so far.

- *authorize()*
- Write Authorisation tutorial.

Encoding

Ensure a variable is a valid value for CSS, URL, XML attribute.

- *cssencode()*
- urlencode - legacy, now part of Jinja >= 2.7.

See also urlencode, escape, xmlattr and tojson in jinja2.

Date-Formatting

- *dateformat()* - formats a `date` object.
- *datetimeformat()* - formats a `datetime` object.
- *tertial()* - outputs a tertial (opposed to quater).

Number-Formating

User-Readable Number formatting. All of these assume you are outputting HTML.

- *nicenum()* - seperates thousands by spaces
- *intword()* - 1200000 becomes '1.2 Mio' and
- *euroword()* - divides cents by 100 and returns an *intword()*
- *eurocent()* - divides cants by 100 and returns an *nicenum()*.
- *g2kg()* - convert to a human readable weigth measurment in g/kg/t. See also filesizeformat in jinja2.
- *percent()* - a `None` tollerant `"%.0f"`
- *iban()* - Format An International Banking Code.

Text-Formatting

Many of these functions are most relevant for settings where you have <pre>> or want to reach a similar effect in HTML.

- *markdown()* - convert Markdown to HTML.
- *nl2br()* - basically get the output of <pre> without using <pre>.
- *left_justify()*
- *right_justify()*

See also urlize, indent and center in jinja2.

Boolean-Formatting (and None)

Displaying Booleans with the ability to distinguish between `(True, False, None)`.

- *yesno()* - output yes, no, maybe
- *onoff()* - unse Font Awesome icons to display boolean
- *none()* - Supress `None` output. See also default in jinja2.

Datastore Protocol

gaetk2 has certain conventions how to structure your datastore models when followed certain things work automagically.

- `otag()` - Link to an Object by *designator*.
- `datastore()` - Link to Admin-Console.

Misc

- `plural()` - Pluralize (works for German).

GAETK1 Compability

- `datetime` has been renamed to datetimeformat.
- `to_json` is gone, use tojson in jinja2 2.9.
- `urlencode` is gone, use urlencode in jinja2 2.7. the urlencode provided by jinja has much more features than we had.
- `attrencode` is gone, use xmlattr in jinja2 2.9.
- generally we now return only Unicode Plain Text, no HTML. `nicenum`, `eurocent``and ``g2kg` are changed by that.
- the urlencode provided by jinja2

Module contents

`gaetk2.jinja_filters.`**`authorize`**(*ctx*, *value*, *permission_types*, *tag=None*)
Display content only if the current logged in user has a specific permission.

This means if all strings in *permission_types* occur in *credential.permissions*.

`gaetk2.jinja_filters.`**`onlystaff`**(*ctx*, *value*, *tag=None*)
Display content only if the current logged in user *is_staff()*.

This tag generatyes HTML. If you don't wan't HTML use this construct:

```
{% if is_staff() %}Internal Info{% endif %}
```

The Tag encloses content in a `<span>` or `<div>` depending on it's contents:

```
{{ "bla"|onlystaff }}
<!-- is rendered to: -->
<span class="gaetk_onlystaff">bla</span>

{% filter onlystaff %}
    <form  ...></form>
{% endfilter %}
<!-- is rendered to: -->
<div class="gaetk_onlystaff">
    <form ...></form>
</div>
```

(continues on next page)

(continued from previous page)

```
{% filter onlystaff %}
    <i>test text</i>
{% endfilter %}
<!-- is rendered to: -->
<span class="gaetk_onlystaff"><i>test text</i></span>
```

If you not happy with how the filter chooses between `<span>` and `<div>` you can provide a tag to be used. Or you can provide empty data to avoid all markup:

```
{% filter onlystaff('p') %}
    <i>test text</i>
{% endfilter %}
<!-- is rendered to: -->
<p class="gaetk_onlystaff">bla</p>

{% filter onlystaff('') %}
    foo
{% endfilter %}
<!-- is rendered to: -->
foo
```

Automatic detection does not work perfectly within tables. Your milage may vary.

If the user is not staff an empty tag is generated:

```
{% filter onlystaff %}
    supersecret
{% endfilter %}
<!-- is rendered to: -->
<span class="gaetk_onlystaff-denied"><!-- !is_staff() --></span>

{% filter onlystaff('') %}
    supersecret
{% endfilter %}
<!-- is rendered to: (nothing) -->
```

gaetk2.jinja_filters.**cssencode**(*value*)

Makes a string valid as an CSS class name.

This ensured only valid characters are used and the class name starts with an character. This is enforced by prefixing *CSS* if the string does not start with an character:

```
<div class="{{ 5711|cssencode }} {{ 'root beer'|cssencode }}">
>>> '<div class="CSS5711 root-beer">'
```

gaetk2.jinja_filters.**dateformat**(*value*, *formatstring=u'%Y-%m-%d'*, *nonchar=u''*)

Formates a date.

Tries to convert the given `value` to a `date` object and then formats it according to `formatstring`:

```
{{ date.today()|dateformat }}
{{ "20171224"|dateformat('%Y-%W') }}
```

gaetk2.jinja_filters.**datetimeformat**(*value*, *formatstring=u'%Y-%m-%d %H:%M'*, *nonchar=u''*)

Formates a datetime.

Tries to convert the given `value` to a `datetime` object and then formats it according to `formatstring`:

```
{{ datetime.now()|datetimeformat }}
{{ "20171224T235959"|datetimeformat('%H:%M') }}
```

gaetk2.jinja_filters.**tertial**(*value*, *nonchar=u'\u2400'*)
Change a Date oder Datetime-Objekt into a Tertial-String.

Tertials are third-years as opposed to quater years:

```
{{ "20170101"|tertial }} {{ "20170606"|tertial }} {{ "20171224"|tertial }}
>>> "2017-A" "2017-B" "2017-C"
```

gaetk2.jinja_filters.**nicenum**(*value*, *spacer=u'\u202f'*, *nonchar=u'\u2400'*)
Format the given number with spacer as delimiter, e.g. *1 234 456*.

Default spacer is NARROW NO-BREAK SPACE U+202F. Probably *style="white-space:nowrap; word-spacing:0.5em;"* would be an CSS based alternative.

gaetk2.jinja_filters.**intword**(*value*, *nonchar=u'\u2400'*)
Converts a large integer to a friendly text representation.

Works best for numbers over 1 million. For example, 1000000 becomes '1.0 Mio', 1200000 becomes '1.2 Mio' and '1200000000' becomes '1200 Mio'.

gaetk2.jinja_filters.**eurocent**(*value*, *spacer=u'\u202f'*, *decimalplaces=2*, *nonchar=u'\u2400'*)
Format the given cents as Euro with spacer as delimiter, e.g. '1 234 456.23'.

Obviously works also with US$ and other 100-based. currencies.

This is like :func:nicenum. Use `decimalplaces=0` to cut of cents, but even better use :func:euroword.

Default spacer is NARROW NO-BREAK SPACE U+202F. Probably *style="white-space:nowrap; word-spacing:0.5em;"* would be an CSS based alternative.

gaetk2.jinja_filters.**euroword**(*value*, *plain=False*, *nonchar=u'\u2400'*)
Fomat Cents as pretty Euros.

gaetk2.jinja_filters.**g2kg**(*value*, *spacer=u'\u202f'*, *nonchar=u'\u2400'*)
Wandelt meist g in kg um, aber auch in andere Einheiten.

gaetk2.jinja_filters.**percent**(*value*, *nonchar=u'\u2400'*)
Fomat Percent and handle None.

gaetk2.jinja_filters.**iban**(*value*, *spacer=u'\u202f'*, *nonchar=u'\u2400'*)
Format the given string like an IBAN Account Number.

Default spacer is NARROW NO-BREAK SPACE U+202F.

Eg:

```
{{ "DE77123413500000567844"|iban }}
DE77 1234 1350 0000 5678 44
```

gaetk2.jinja_filters.**markdown**(*value*)
Renders a string as Markdown.

Syntax: {{ valuelmarkdown }}

We are using markdown2 to do the rendering.

gaetk2.jinja_filters.**nl2br**(*eval_ctx*, *value*)
Newlines in
-Tags konvertieren.

gaetk2.jinja_filters.**left_justify**(*value*, *width*)
Prefix the given string with spaces until it is width characters long.

gaetk2.jinja_filters.**right_justify**(*value*, *width*)
Postfix the given string with spaces until it is width characters long.

gaetk2.jinja_filters.**yesno**(*value*, *answers=u'yes*, *no*, *maybe'*)
Output a text based on Falsyness, Trueyness and `is None`.

Example:: {{ value|yesno:"yeah,nope,maybe" }}.

gaetk2.jinja_filters.**onoff**(*value*)
Display Boolean as Font Awesome Icon Icon darstellen.

We use Font Awesome toogle-on and toogle-of to indicate state.

gaetk2.jinja_filters.**none**(*value*, *nonchar=u''*)
Converts `None` to `''`.

Similar to `|default('', true)` in jinja2 but more explicit.

gaetk2.jinja_filters.**otag**(*obj*)
Link like this: *obj.designator*.

gaetk2.jinja_filters.**datastore**(*entity*, *attr=None*, *value=None*, *text=None*)
Generate HTML a-Tag to Google Datastore Query.

Example:: {{ credential|datastore }} -> queries for key {{ credential|datastore('email') }} -> queries for email {{ credential|datastore('name', '') }} -> queries for credential.name == '' {{ credential|datastore(text='Search in Datastore') }} -> changes Link-Text

gaetk2.jinja_filters.**plural**(*value*, *singular_str*, *plural_str*)
Return value with singular or plural form.

```
{{ l|length|plural('Items', 'Items') }}
```

gaetk2.jinja_filters.**register_custom_filters**(*jinjaenv*)
Register the filters to the given Jinja environment.

gaetk2.forms package - form handling via WTForms

gaetk2.forms aims at making Bootstrap 4 Forms <'Bootstrap 3 Forms and WTForms play nice together. This means for the normal form you don't have to write any HTML.

There is also some unmaintained legacy code for Bootstrap 3 Forms.

Together with wtforms-appengine you can get a very smooth form handling experience.

wtfbootstrap4(*form*)
Takes a form instance and changes the widgets within to conform to bootstrap / HTML5 layout including labels, error-messages, etc.

So usage would look like this:

```
# Define an Datastore / NDB - Model
from google.appengine.ext import ndb
class pay_Lastschriftmandat(ndb.Model):
    kundennr = ndb.StringProperty(required=True)
    kontoinhaber = ndb.StringProperty(required=True)
    iban = ndb.StringProperty()
    bic = ndb.StringProperty()
    datum = ndb.DateTimeProperty()
```

(continues on next page)

(continued from previous page)

```
    mandatsreferenz = ndb.StringProperty(required=True)
    glaeubigernr = ndb.StringProperty(required=True)
    updated_at = ndb.DateTimeProperty(auto_now=True)
    created_at = ndb.DateTimeProperty(auto_now_add=True)

# build the WTForm from Model
from wtforms_appengine.ndb import model_form
LastschriftmandatForm = model_form(
    pay_Lastschriftmandat,
    only=['bic', 'iban', 'datum']
)

# Render form
from gaetk2.forms import wtfbootstrap3
class View(gaetk2.handlers.DefaultHandler):
    def get(self):
        # instantiate form
        form = LastschriftmandatForm()
        # style form
        form = wtfbootstrap3(form)
        self.render({'form': form}, 'view.html')
```

Now you could render it in your template like this:

```
<form method="POST">
  <div class="form-body form-group">
    {% for field in form %}
      {% if field.flags.required %}
        {{ field(required=True) }}
      {% else %}
          {{ field() }}
      {% endif %}
    {% endfor %}
  </div><!-- /form-body -->
  <div class="text-right">
    <button type="submit" id="{{ domid }}-submit-button" form="{{ domid }}-form"␣
↪data-loading-indicator="true" class="btn btn-primary" autocomplete="off">{{␣
↪buttonname }}</button>
  </div>
</form>
```

See also `wtfbootstrap3()` for legacy support.

Todo:

- Add validation. Use https://validators.readthedocs.io/en/latest/
- Add some standard way to render a complete Form not just fields.

Module contents

`gaetk2.forms.`**`wtfbootstrap3`**(*form*)

changes a WTForms.Form Instance to use html5/bootstrap Widgets.

gaetk2.forms.**wtfbootstrap4**(*form*)
changes a WTForms.Form Instance to use html5/bootstrap Widgets.

gaetk2.helpers module

gaetk2.helpers provides support for writing more concise, elegant code for gaetk2.

Module contents

gaetk2.helpers.**check404**(*obj*, *message='Object not found.'*)
Raises 404 if `bool(obj)` is `False`.

The major usecase is to replace:

```
def post(self, kundennr):
    kunde = m_api.get_kunde(kundennr)
    if not kunde:
        raise HTTP404_NotFound
    do_some_work()
```

with:

```
def post(self, kundennr):
    kunde = check404(m_api.get_kunde(kundennr))
    do_some_work()
```

This has the potential to make view-Functions much more readable.

gaetk2.helpers.**abs_url**(*url*)
Convert a relative URL to an absolute URL.

You really should prefer `gaetk2.handler.base.BasicHandler.abs_url()` because it has better information about the request and host.

gaetk2.tools.taskqueue - access App Engine taskqueues

This are convinience functions to work with App Engine taskqueues. Also *`defer()`* provides much better error reporting See *Error Handling Guide*.

Contents

Module contents

gaetk2.taskqueue.**taskqueue_add_multi**(*qname*, *url*, *paramlist*, ***kwargs*)
Adds more than one Task to the same Taskqueue/URL.

This helps to save API-Calls. Usage pattern:

```
tasks = []
for kdnnr in kunden.get_changed():
    tasks.append(dict(kundennr=kdnnr))
taskqueue_add_multi('softmq', '/some/path', tasks)
```

gaetk2.taskqueue.**taskqueue_add_multi_payload**(*name*, *url*, *payloadlist*, ***kwargs*)
like taskqueue_add_multi() but transmit a json encoded payload instead a query parameter.

In the Task handler you can get the data via `zdata = json.loads(self.request.body)`. See http://code.google.com/appengine/docs/python/taskqueue/tasks.html

gaetk2.taskqueue.**defer**(*obj*, **args*, ***kwargs*)
Defers a callable for execution later.

like https://cloud.google.com/appengine/articles/deferred but adds the function name to the url for easier debugging.

Add this to *app.yaml*:

handlers: # needed to allow abritary postfixes and better error handling - url: /_ah/queue/deferred(.*)

script: gaetk2.views.default.application login: admin

Parameters starting with _ are handed down to taskqueue.add()

gaetk2.taskqueue.**defer_once_per_hour**(*obj*, **args*, ***kwargs*)
Like *defer()* but only once per hour.

Executes the same function with the same parameters not more often than once per hour. The heuristic for doing so are not exact so do not rely on this mechanism for anything importatant.

This is more for updating cloud services with statistics etc.

gaetk2.taskqueue.**defer_once_per_day**(*obj*, **args*, ***kwargs*)
Like *defer_once_per_hour()* but only once per day.

gaetk2.datastore module

gaetk2.datastore tries to codify a common set of expectations and usages for gaetk2.

Inherit from `gaetk2.datastore.gaetkModel` instead of ndb.Model to get some added functionality. The rationale there is that e common interface and thus admin- and programmer-time is more important than savings on space and and processing time. To we add possible unneded database fields. You can remove them on a case by case basis in derivered classes.

- `query_iterator()` - helps to iterate over big query results
- `get_or_insert_if_new()` helps you to see if a new Entity was created.
- `copy_entity()` - can write an entity with a different key to the datastore
- `update_obj()` - basically implements conditional `put()`
- `reload_obj()` - forces an object to be re-read from disk
- `apply_to_all_entities()` - iterates over a table executing a function (“mapper”)

Data Model Conventions

- url

- created_at, updated_at
- name, nicename
- designator

Module contents

gaetk2.datastore.**query_iterator**(*query*, *limit=50*)
Iterates over a datastore query while avoiding timeouts via a cursor.

Especially helpful for usage in backend-jobs.

gaetk2.datastore.**copy_entity**(*e*, ***extra_args*)
Copy ndb entity but change values in kwargs.

Usage:: b = copy_entity(a, id='new_id_here') b.put()

gaetk2.datastore.**get_or_insert_if_new**(*cls*, *id*, ***kwds*)
Like ndb.get_or_insert()' but returns *(entity, new)*.

This allows you to see if something has been created or if there was an already existing entity:

```
>>> get_or_insert_if_new(Model, 'newid')
(<instance>, True)
>>> get_or_insert_if_new(Model, 'newid')
(<instance>, False)
```

gaetk2.datastore.**write_on_change2**(*obj*, *data*)
Apply new data to an entity and write to datastore if anything changed.

This should save you money since reads are 3 times cheaper than writes. It also helps you do leave not given attributes unchanged.

Usage:

```
instance = ndb.Model...get()
dirty = write_on_change2(instance, ...,
  dict(id=123, amout_open=500, score=5, ...)
```

gaetk2.datastore.**update_obj**(*obj*, ***kwargs*)
More modern Interface to *write_on_change2()*.

gaetk2.datastore.**reload_obj**(*obj*)
Returns a reloaded Entity from disk.

gaetk2.datastore.**apply_to_all_entities**(*func*, *model*, *batch_size=0*, *num_updated=0*, *num_processed=0*, *cursor=None*)
Appliy a certain task all entities of *model*.

It scans every entity in the datastore for the model, exectues *func(entity)* on it and re-saves it if func trturns true. Tries to keep *updated_at* and *updated_by* unchanged.

Example

def _fixup_MyModel_updatefunc(obj):

if obj.wert_eur is not None: obj.wert_eur = int(obj.wert_eur) return True

return False

def fixup_MyModel(): apply_to_all_entities(_fixup_app_angebotspos_updatefunc, MyModel)

or

def execute(_now):

datastore.apply_to_all_entities(_fixup_bestandsbuch_updatefunc, ic_bestandsbuch.ic_BestandsbuchEintrag)

def _fixup_bestandsbuch_updatefunc(obj): changed = False # Attribute, die es als string und text in der datebnbank gibt normalisieren for attrname in ‘’‘ausloeser vorhergehender_bestandsbucheintrag info’‘’.split():

if getattr(obj, attrname, None) is not None: setattr(obj, attrname, unicode(getattr(obj, attrname))) changed = True

return changed

gaetk2.resttestlib - Simple Acceptance Tests

This module allows you to run simple non-interactive tasks against an installed Version of your application. We found that it helps to catch most simple programming errors and regressions prior to production deployment.

Simple tests look like this:

```
from gaetk2.resttestlib import create_testclient_from_cli
client = create_testclient_from_cli('myserver.appspot.com')

client.GET('/_ah/warmup').responds_http_status(200)

client.run_checks(max_workers=4)
print len(client.responses), "URLs tested"
sys.exit(client.errors)
```

This uses the low-level *Response* interface. But usually you will work with the *TestClient.check()* family of functions. Check can handle more than one URL at once:

```
client.check(
    '/mk/pay/start/a6LP3L',
    '/mk/pay/paypal/init/a6LP3L'
)
```

Based on file extension we check not only the content type, but also that the response is well formed - at least to a certain degree:

```
client.check(
    '/k/SC10001/artikel',
    '/api/marketsuche.json'
    '/k/SC10001/artikel.csv',
    '/k/SC10001/artikel.html',
    '/k/SC10001/artikel.xml'
)
```

TestClient.check_redirect() takes a list of sources and destinations and ensures that the server redirects to the desired destination:

```
client.check_redirect(
    dict(url='/artnr/73005/', to='/artnr/73000/'),
    dict(url='/artnr/73000/', to='/artnr/73000/01/'),
)
```

The framework is meant to check for fine grained access controls via HTTP-Basic-Auth. You can provide a list of `handle=username:password` pairs during instantiation or via the command line. You can then refer to them in your checks the the `auth` parameter:

```
users = [
    'usera=CK101:FNYBMAMPVC6EU',
    'userb=u1001:TEABPVPGPVGBFE',
    'admin=u2001:LQASNAJC6GUUP4VY',
    'inactiveuser=u22730o:MATLEU4BJA756']
client = create_testclient_from_cli('myserver.appspot.com', users)
client.check(
    '/pay/start/testingClassic',
    '/mk/pay/paypal/init/testingMarket',
    auth='usera')
client.check_redirect(dict(url='/', to='/inactive.html'), auth='inactiveuser')
```

One of the main uses of *resttestlib* is to check that certain resources are allowed for some users and denied for others:

```
client.check_allowdeny(
    '/k/SC10001/auftraege',
    allow=['usera', 'admin'],
    deny=['userb', None]
)
```

The special user `None` means unauthenticated.

Describe how this is part of the general test and deployment strategy.

Module contents

gaetk2.resttestlib.**create_testclient_from_cli**(*default_hostname*, *users*)

Creates a Testclient with it's arguments from the Commandline.

the CLI understands the options, –hostname, –credentials-user, their default values are taken from this functions args

default_hostname: hostname, on wich to run tests, if none is provided via CLI

returns a *TestClient*

class gaetk2.resttestlib.**TestClient**(*host*, *users*, *debug=False*)

Hilfsklasse zum Ausfuehren von HTTP-Requests im Rahmen von Tests.

add_credentials(*auth*, *creds*)

Stellt dem Client credentials zur Verfügung, die in GET genutzt werden können.

auth: key der Credentials creds: HTTP-Credentials in der Form 'username:password'

GET(*path*, *auth=None*, *accept=None*, *headers={}*, ***kwargs*)

Führt einen HTTP-GET auf den gegebenen [path] aus. Nutzt dabei ggf. die credentials zu [auth] und [accept].

check(**args*, ***kwargs*)

check_allowdeny(**args*, ***kwargs*)

check_redirect(**args*, ***kwargs*)

check_statuscode(**args*, ***kwargs*)

run_checks(*max_workers=6*)
run queued checks.

errors
Anzahl der fehlgeschlagenen Zusicherungen, die für Anfragen dieses Clients gefroffen wurden.

class gaetk2.resttestlib.**Response**(*client*, *method*, *url*, *status*, *headers*, *content*, *duration*, *response*)
Repräsentiert das Ergebnis einer REST-Anfrage. Mittels responds_* koennen zusicherungen geprueft werden:

r.responds_http_status(200) r._responds_html()

responds_normal()
Sichert zu, dass ein Dokument gefunden wurde.

responds_not_found()
Sichert zu, dass kein Dokument gefunden wurde.

responds_access_denied()
Sichert zu, dass der Zugriff verweigert wurde.

responds_forbidden()
Sichert zu, dass der Zugriff verweigert wurde.

responds_with_content_location(*expected_location*)
Sichert zu, dass die Antwort einen location-header hat.

responds_http_status(*expected_status*)
sichert zu, dass mit dem gegebenen HTTP-status geantwortet wurde.

responds_content_type(*expected_type*)
sichert zu, dass mit dem gegebenen Content-Type geantwortet wurde.

converter_succeeds(*converter*, *message*)
Sichert zu, dass content mittels converter(self.content) ohne exception konvertiert werden kann.

gaetk2.modelexporter module

This module provides functionality to write datastore Models or Queries to the client as XLS or CSV Files. Usage like this in your handler:

```
exporter = ModelExporter(ic_AuftragsPosition)
filename = '%s-%s.xls' % (compat.xdb_kind(ic_AuftragsPosition), datetime.datetime.
↪now())
handler.response.headers['Content-Type'] = 'application/msexcel'
handler.response.headers['content-disposition'] = \
    'attachment; filename=%s' % filename
exporter.to_xls(handler.response)
# or:
# exporter.to_csv(handler.response)
```

Module contents

class gaetk2.modelexporter.**ModelExporter**(*model*, *query=None*, *uid=u''*, *only=None*, *ignore=None*, *additional_fields=None*, *maxseconds=40*)
Bases: object

Export all entities of a Model as XLS, CSV.

Parameters

- **model** (*ndb.Model*) – Model to be exported, required.
- **query** (*ndb.Query or None*) – Query to limit the records to be exported.
- **uid** (*str*) – Encodes the person doing the Export in the Output.
- **only** (*list(str) or None*) – List of Field-/Propertynames to export
- **ignore** (*list(str) or None*) – List of Field-/Propertynames not to export
- **additional_fields** (*list(str) or None*) – The priority of the message, can be a number 1-5
- **maxseconds** (*int*) – Truncate exporting after this many seconds.

Intatiate a *ModelExporter* and call *to_xls()* or *to_csv()* to get an export of the Entities on Disk.

ModelExporter can also be used to create a automated tabular HTML view like in the admin interfave or in the ListViewer.

fields
Property with list of files to export.

Can be overwritten. Current implementation is cached.

create_header(*output*, *fixer=<function <lambda>>*)
Generates one or more header rows in *output*.

Can be overwritten.

create_row(*output*, *data*, *fixer=<function <lambda>>*)
Generates a single output row.

Can be overwritten.

create_csvwriter(*fileobj*)
Generates an outputstream from fileobj.

Can be overwritten to change the `csv.writer` csv.writer options.

to_csv(*fileobj*)
Generate CSV in fileobj.

Overwrite *create_csvwriter()* to change CSV Style.

to_xls(*fileobj*)
generate XLS in fileobj

gaetk2.admin package

This package implemets automatic administration facilities. It aims to be a mix of the concepts of Django Admin and the Google App Engine Admin Console. It is aimed to be used not by Developers, Dev Ops or System Administrators but by the regular staff using your application - so it apptempts to give you less opportunity to shoot your self in the foot.

It also aims at giving you building blocks for your own user facing pages.

These Services provided by the Admin-Package are automatically available at */admin2/* in your URL-Tree.

With minimal configuration you can get an admin site as above. Just add a file `admin_gaetk2.py` e.g. in `modules/pay/` or any other directory within your the `modules` directory:

```
from gaetk2.admin import site
from gaetk2.admin.layout import AdminLayout
from . import pay_models

class MyLayout(AdminLayout):
    links = [
        ('SEPA-Dateien',
         'https://console.cloud.google.com/storage/browser/foobar'),
    ]
site.registerlayoutclass(MyLayout)
site.registermodel(pay_models.pay_Lastschriftmandat)
site.registermodel(pay_models.pay_IPNRecord)
site.registermodel(pay_models.pay_Kontovorgang)
```

Files named `modules/**/admin_gaetk2.py` are automatically found an included in the Admin Site.

Adding Datastore Models to the Admin Site

You have to manually add all ndb models you want to have in the Admin Site in like this to `admin_gaetk2.py`:

```
from gaetk2.admin import site
from . import pay_models
site.registermodel(pay_models.pay_Lastschriftmandat)
```

gaetk2.admin.modeladmin.ModelAdmin is the main mechanism for changing the automatically generated admin interface. You intantiate it for each model you want to have administered:

```
class LastschriftmandatAdmin(ModelAdmin):
    list_fields = ['ist_aktiv', 'last_used',
        'kundennr', 'kontoinhaber', 'iban', 'updated_at', 'created_at']
    queries = {
        'aktiv': pay_models.pay_Lsm.query(pay_models.pay_Lsm.ist_aktiv==True),
        'nicht aktiv': pay_models.pay_Lsm.query(pay_models.pay_Lsm.ist_aktiv==False),
        'alle': pay_models.pay_Lsm.query(),
    }
site.registermodel(pay_models.pay_Lsm, LastschriftmandatAdmin)
```

Todo:

KundeForm = model_form(m_Kunde, exclude=[‘designator’, ‘empfaengernr’, ‘updated_at’, ‘created_at’, ‘name1’, ‘name2’], field_args={ ‘owner’: {‘default’: ‘cyberlogi’}, ‘email’: {‘validators’: [express_email_validator]}, })

Todo:

- rename `application_id` to topic everywhere
- reimplement search

Package contents

class gaetk2.admin.modeladmin.**ModelAdmin**(*model*, *admin_site*, *topic=None*)

Admin Model - Implements CRUD for NDB

read_only = True
User is not allowed to do any changes to the database for this Models Entities.

deletable = False
User is allowed to delete Entities via the admin interface.

list_per_page = 50
Number of items per page.

order_field = u'-created_at'
Sorting. Beware of datastore indices!

ordering = u''
TBD Mit 'order_field' laesst sich die Sortierung bei der Anzeige der Model-Instanzen im Admin-Bereich anpassen. Als Default werden die Datensaetze in absteigender Reihenfolge ihrer Erzeugung sortiert, jedoch kann jede Admin-Klasse die Sortierung mit 'order_field' beeinflussen, indem sie ein bel. anderes Feld dort angibt.

post_create_hooks = []
List of functions to be called with the newly created object as the sole parameter.

db_key_field = None
Standardmaessig lassen wir die App Engine fuer das Model automatisch einen Key generieren. Es besteht jedoch in der Admin-Klasse die Moeglichkeit, via 'db_key_field=[propertyname]' ein Feld festzulegen, dessen Inhalt im Formular als Key beim Erzeugen der Instanz genutzt wird.

topic = None
The Topic (Application Name in Django) under which the Model is listed in the admin GUI.

queries = {}
TBD

list_fields = ()
Names of fields to show in Entity listing.

If you do not want to show all the files you can give a tuple of fields to show:

```
list_fields = ('designator', 'name', 'plz', 'ort', 'email')
```

TBD: relation to fields / only.

detail_fields = ()
TBD

get_ordering(*request*)
Return the sort order attribute

get_queryset(*request*)
Gib das QuerySet für die Admin-Seite zurück

Es wird die gewünschte Sortierung durchgeführt.

get_form(***kwargs*)
Erzeuge Formularklasse für das Model

get_object(*encoded_key*)
Ermittle die Instanz über den gegeben ID

handle_blobstore_fields(*handler*, *obj*, *key_name*)
Upload für Blobs

change_view(*handler*, *object_id*, *extra_context=None*)
View zum Bearbeiten eines vorhandenen Objekts

add_view(*handler*, *extra_context=None*)
View zum Hinzufügen eines neuen Objekts

delete_view(*handler*, *extra_context=None*)
Request zum Löschen von (mehreren) Objekten behandeln.

Redirectet bei Erfolg zur Objektliste. *extra_context* ist für die Signatur erforderlich, wird aber nicht genutzt.

export_view_csv(*handler*, *extra_context=None*)
Request zum Exportieren von allen Objekten behandeln.

extra_context ist für die Signatur erforderlich, wird aber nicht genutzt.

export_view_xls(*handler*, *extra_context=None*)
Request zum Exportieren von allen Objekten behandeln.

extra_context ist für die Signatur erforderlich, wird aber nicht genutzt.

get_template(*action*)
Auswahl des zur *action* passenden templates.

class gaetk2.admin.sitemodel.**AdminSite**
Registry for Models and other Stuff to be administered via Web GUI.

Cenceptually Our Grandparent - Django Admin - Lives in a world of "Applications" out of which your Django Installation is composed.

GAETK2 does not follow this approach very much. We assum each Model/Kind Name is unique in the whole deployed Web-Application and don't use djangos term "application" to avoid confusion. We speak of "Topics" whose sole purpose is to organize contant in the admin interface.

registerlayoutclass(*layout_class*, *topic=None*)

registermodel(*model_class*, *admin_class=None*, *topic=None*)
Registers the given model with the given admin class.

topics()

get_layout_by_topic(*topic*)

get_admin_by_topic(*topic*)

kinds()

get_admin_by_kind(*kind*)

get_model_by_kind(*kind*)

get_model_class(*application*, *model*)
Klasse zu 'model' zurückgeben.

class gaetk2.admin.layout.**AdminLayout**

links = []

gaetk2.tools Package

Thees package contains functionality mostly used intenally.

Contents

Todo:

- ids.py
- hujson2.py
- http.py
- config.py
- auth0tools.py
- sentry.py
- structured.py

gaetk2.tools.caching - smart caching

Caching on Google App Engine makes your application faster and cheaper. While for *key.get()* operations ndp provides caching for you, queries are nwver cached by the datastore infrastructure.

After years of experimentation we come to the conclusion that you should always use some time-based cache invalidation. This will result in "eventual consitency" even if you do not get your cache invalidation strategy perfectly right.

We provide `lru_cache()` with a default TTL of 12 hours. It does local instance memory caching and is an extension of `functools` from Python 3.3.

`lru_cache_memcache()` is an extension using a two-level strategy: content which is not found in the local instance cache is pulled from the shared memcache. Cache entries are not shared between different versions of your application.

It is suggested, that you use a relatively small *maxsize* with `lru_cache_memcache()` to save on instance memory.

`gaetk2.tools.caching.`**`lru_cache`**(*maxsize=64*, *typed=False*, *ttl=43200*)

Least-recently-used cache decorator.

Parameters

- **`maxsize`** (*`int or None`*) – if None, the LRU features are disabled and the cache can grow without bound.

- **typed** (*boolean*) – if *True*, arguments of different types will be cached separately. For example, f(3.0) and f(3) will be treated as distinct calls with distinct results.
- **ttl** (*int or None*) – if set, cache entries are only served for *ttl* seconds.

Arguments to the cached function must be hashable.

View the cache statistics named tuple (hits, misses, maxsize, currsize) with f.cache_info(). Clear the cache and statistics with f.cache_clear(). Access the underlying function with f.__wrapped__.

See: http://en.wikipedia.org/wiki/Cache_algorithms#Least_Recently_Used

Example

```
@lru_cache(maxsize=6)
def _fetchbrands():
    query = mk_models.mk_Brand.query()
    return [brand.name for brand in query.iter() if not brand.deleted]
```

class gaetk2.tools.caching.**lru_cache_memcache**(*maxsize=8*, *typed=False*, *ttl=43200*)

Use *lru_cache()* with memcache as an fallback.

Arguments are the same as *lru_cache()*.

Example

```
@lru_cache_memcache(ttl=3600)
def _fetchbrands():
    query = mk_models.mk_Brand.query()
    return [brand.name for brand in query.iter() if not brand.deleted]
```

gaetk2.tools.datetools

Todo:

- Explain what gaetk2.tools.datetools is for

gaetk2.tools.datetools.**tertial**(*date*)

Wandelt ein Date oder Datetime-Objekt in einen Tertial-String

gaetk2.tools.datetools.**rfc3339_date**(*date=None*)

Formates a datetime object according to RfC 3339.

gaetk2.tools.datetools.**rfc3339_date_parse**(*date*)

Parses an RfC 3339 timestamp into a datetime object.

gaetk2.tools.datetools.**convert_to_date**(*date*)

Converts argument into a date object.

Assumes argument to be a RfC 3339 coded date or a date(time) object.

gaetk2.tools.datetools.**convert_to_datetime**(*date*)

Converts argument into a datetime object.

Assumes argument to be a RfC 3339 coded date or a date(time) object.

gaetk2.tools.datetools.**rfc2616_date**(*date=None*)
Formates a datetime object according to RfC 2616.

RfC 2616 is a subset of RFC 1123 date. Weekday and month names for HTTP date/time formatting; always English!

gaetk2.tools.datetools.**rfc2616_date_parse**(*data*)
Parses an RfC 2616/2822 timestapm into a datetime object.

gaetk2.tools.datetools.**date_trunc**(*trtype*, *timestamp*)
Truncate date or datetime object. Truncated object of the given type.

This function is inspired by date_trunc from PostgreSQL, see http://www.postgresql.org/docs/8.1/static/functions-datetime.html#FUNCTIONS-DATETIME-TRUNC

Supported types are year, quarter, month, week, day, hour, minute, second.

```
>>> date_trunc('week', datetime.datetime(1974, 8, 21))
datetime.datetime(1974, 8, 19, 0, 0)
>>> date_trunc('week', datetime.date(1973, 8, 8))
datetime.date(1973, 8, 6)
```

gaetk2.tools.datetools.**get_tertial**(*date*)
Calculates the tertial

```
>>> get_tertial(datetime.date(2015, 1, 9))
1
>>> get_tertial(datetime.datetime(2015, 2, 19))
1
>>> get_tertial(datetime.date(2015, 3, 9))
1
>>> get_tertial(datetime.datetime(2015, 4, 20))
1
>>> get_tertial(datetime.datetime(2015, 5, 4))
2
>>> get_tertial(datetime.datetime(2015, 6, 11))
2
>>> get_tertial(datetime.datetime(2015, 7, 22))
2
>>> get_tertial(datetime.date(2015, 8, 3))
2
>>> get_tertial(datetime.date(2015, 9, 23))
3
>>> get_tertial(datetime.datetime(2015, 10, 24))
3
>>> get_tertial(datetime.date(2015, 11, 11))
3
>>> get_tertial(datetime.datetime(2015, 12, 6))
3
```

gaetk2.tools.datetools.**get_quarter**(*date*)
Calculates the quarter

```
>>> get_quarter(datetime.date(2015, 1, 9))
1
>>> get_quarter(datetime.datetime(2015, 2, 19))
1
>>> get_quarter(datetime.date(2015, 3, 9))
1
```

(continues on next page)

(continued from previous page)

```
>>> get_quarter(datetime.datetime(2015, 4, 20))
2
>>> get_quarter(datetime.datetime(2015, 5, 4))
2
>>> get_quarter(datetime.datetime(2015, 6, 11))
2
>>> get_quarter(datetime.datetime(2015, 7, 22))
3
>>> get_quarter(datetime.date(2015, 8, 3))
3
>>> get_quarter(datetime.date(2015, 9, 23))
3
>>> get_quarter(datetime.datetime(2015, 10, 24))
4
>>> get_quarter(datetime.date(2015, 11, 11))
4
>>> get_quarter(datetime.datetime(2015, 12, 6))
4
```

gaetk2.tools.datetools.**get_yearspan**(*date*)
Gibt den ersten und letzten Tag des Jahres zurück in dem *date* liegt

```
>>> get_yearspan(datetime.date(1980, 5, 4))
(datetime.date(1980, 1, 1), datetime.date(1980, 12, 31))
>>> get_yearspan(datetime.date(1986, 3, 11))
(datetime.date(1986, 1, 1), datetime.date(1986, 12, 31))
```

gaetk2.tools.datetools.**get_tertialspan**(*date*)
Gibt den ersten und den letzten Tag des Tertials zurück in dem *date* liegt

```
>>> get_tertialspan(datetime.date(1978, 9, 23))
(datetime.date(1978, 9, 1), datetime.date(1978, 12, 31))
```

gaetk2.tools.datetools.**get_quarterspan**(*date*)
Gibt den ersten und den letzten Tag des Quartals zurück in dem *date* liegt

```
>>> get_quarterspan(datetime.date(1978, 6, 12))
(datetime.date(1978, 4, 1), datetime.date(1978, 6, 30))
```

gaetk2.tools.datetools.**get_monthspan**(*date*)
Gibt den ersten und letzten Tag des Monats zurück in dem *date* liegt

```
>>> get_monthspan(datetime.date(1980, 5, 4))
(datetime.date(1980, 5, 1), datetime.date(1980, 5, 31))
```

gaetk2.tools.datetools.**get_weekspan**(*date*)
Gibt den ersten und den letzten Tag der Woche, in der *date* liegt, zurück.

Dabei ist Montag der erste Tag der woche und Sonntag der letzte.

```
>>> get_weekspan(datetime.date(2011, 3, 23))
(datetime.date(2011, 3, 21), datetime.date(2011, 3, 27))
```

gaetk2.tools.datetools.**get_timespan**(*period*, *date*)
Get given timespan for date

Convenience function as a wrapper for the other get_*span functions

gaetk2.tools.datetools.**tertial_add**(*date*, *tertials*)

Add number of tertials to date.

```
>>> date = datetime.date(1982, 11, 7)
>>> tertial_add(date, -1)
datetime.date(1982, 5, 1)
>>> tertial_add(date, 0)
datetime.date(1982, 9, 1)
>>> tertial_add(date, 1)
datetime.date(1983, 1, 1)
>>> tertial_add(date, 2)
datetime.date(1983, 5, 1)
>>> tertial_add(date, 3)
datetime.date(1983, 9, 1)
>>> tertial_add(date, 4)
datetime.date(1984, 1, 1)
```

```
>>> date = datetime.datetime(1982, 11, 7)
>>> tertial_add(date, 4)
datetime.datetime(1984, 1, 1)
```

gaetk2.tools.datetools.**month_add**(*date*, *months*)

Add number of months to date.

```
>>> import datetime
>>> date = datetime.date(1986, 3, 9)
>>> month_add(date, -12)
datetime.date(1985, 3, 9)
>>> month_add(date, -1)
datetime.date(1986, 2, 9)
>>> month_add(date, 0)
datetime.date(1986, 3, 9)
>>> month_add(date, 3)
```

```
>>> date = datetime.datetime(1986, 3, 9)
>>> month_add(date, 12)
datetime.datetime(1987, 3, 9)
```

gaetk2.tools.datetools.**year_add**(*date*, *years*)

Add number of years to date.

```
>>> import datetime
>>> year_add(datetime.datetime(2016, 2, 29), 1)
datetime.date(2017, 2, 28)
>>> year_add(datetime.date(2016, 2, 29), 1)
datetime.date(2017, 2, 28)
>>> year_add(datetime.date(2015, 2, 28), 1)
datetime.date(2016, 2, 28)
>>> year_add(datetime.date(2017, 2, 28, -1)
datetime.date(2016, 2, 28)
>>> year_add(datetime.datetime(2016, 2, 29), -1)
datetime.datetime(2015, 2, 28)
```

gaetk2.tools.datetools.**add_to_day**(*day*, *offset*)

Returns the date n days before or after day.

gaetk2.tools.datetools.**easter**(*year*)

Returns the day of Easter sunday for 'year'.

This function only works betweeen 1900 and 2099

gaetk2.tools.datetools.**easter_related_holidays**(*year*)
Returns a list of holidays which are related to easter for 'year'.

gaetk2.tools.datetools.**holidays_german**(*start*, *end*)
Returns a list of dates between start and end that are holidays.

gaetk2.tools.datetools.**workdays**(*start*, *end*)
Calculates the number of working days (Mo-Fr) between two given dates.

Whereas the workdays are calculated siilar to Python slice notation: [start : end[Example: >>> workdays(datetime.date(2007, 1, 26), datetime.date(2007, 1, 27)) # Fr - Sa 1 >>> workdays(datetime.date(2007, 1, 28), datetime.date(2007, 1, 29)) # Su - Mo 0

gaetk2.tools.datetools.**workdays_german**(*start*, *end*)
Calculates the number of working days between two given dates while considering german holidays.

gaetk2.tools.datetools.**is_workday_german**(*day*)
Checks if a day is a workday in germany (NRW).

```
>>> is_workday_german(datetime.date(2007, 1, 1))
False
>>> is_workday_german(datetime.date(2007, 1, 2))
True
```

gaetk2.tools.datetools.**next_workday_german**(*startday*)
Returns the next workday after startday.

```
>>> next_workday_german(datetime.date(2006, 12, 29))
datetime.date(2007, 1, 2)
```

gaetk2.tools.datetools.**previous_workday_german**(*startday*)
Returns the workday before startday.

```
>>> previous_workday_german(datetime.date(2007, 1, 2))
datetime.date(2006, 12, 29)
```

gaetk2.tools.datetools.**add_workdays_german**(*startday*, *count*)
Adds <count> workdays to <startday>.

gaetk2.tools.unicode - string handling

This are functions which help to handle data from a pre-Unicode world. Much of this code is acient and has no use in a worl where JSON and XML ensure somewhat clean encoding. But still there are so many places where you are allowd to send only ASCII subsets.

Data Cleanup

- `de_noise()` - removed Unicode Characters which normally have no place in buiseness documents (eg street names). This includes Emojii but also protected spaces unusual quotation marks etc. This data is usually included dut to cut and paste errors. Read source to see what is replaced.
- `de_umlaut()` - converts data to plain ASCII while converting german Umlauts to something reasonable.
- `de_utf8()` - "repair" wrongly decoded UTF-8.

Number Conversion

`num_encode()` and `num_decode()` convert arbitrary long numbers to strings and back again. Works nice for datastore IDs. Uses base 62 (lowwer and upper letters and numbers) to get a compact representation.

`num_encode_uppercase()` uses base36 which is less compact but case insensitive.

You can use these functions to getsomewhat easy to tipe compact datastore ids:

```
class SomeEntity(ndb.Model):
    nr = ndb.ComputedProperty(lambda num_encode(self: self.key.id()) if self.key.id()␣
↪else '?')
```

Module contents

`gaetk2.tools.unicode.`**`de_utf8`**(*data*)
This is meant to help with utf-8 data appearing where unicode should apperar.

`gaetk2.tools.unicode.`**`de_umlaut`**(*data*)
Converts a text to ASCII acting smart about Umlauts.

```
>>> de_umlaut('1 Über Hügel saß René äöüÄÖÜß')
'1 Ueber Huegel sass Rene aeoeueAeOeUess'
```

`gaetk2.tools.unicode.`**`de_noise`**(*data*)
Removes all stuff which should not appear in normal Western Text.

```
>>> de_noise(u'»Susie`s Giga\Super-Markt®¿«')
u">Susie's Giga/Super-Markt(R)?<"
>>> de_noise(u"ümlaut eins:{")
u'\xfcmlaut eins:'
>>> de_noise(u'«A» {C} ¿D? „E‟ ›F‹')
u'<A> (C) ?D? "E" >F<'
>>> de_noise(u'`A´')
u"'A'"
>>> de_noise(u'«» Umlaute kann doctest !gut {®} ¿? „‟ ›‹')
u'<> Umlaute kann doctest !gut ((R)) ?? "" ><'
>>> de_noise(u'DE37330513500010441422')
u'DE37330513500010441422'
```

`gaetk2.tools.unicode.`**`slugify`**(*value*)
Converts a string to be usable in URLs without excaping.

Normalizes string, converts to lowercase, removes non-alpha characters, and converts spaces to hyphens.

Inspired by Django's "django/template/defaultfilters.py".

`gaetk2.tools.unicode.`**`num_encode`**(*n*)
Convert an integer to an base62 encoded string.

`gaetk2.tools.unicode.`**`num_decode`**(*s*)
Convert the result of num_encode() back to an integer.

`gaetk2.tools.unicode.`**`num_encode_uppercase`**(*n*)
Convert an integer to an base36 (unly uppercase and numbers) encoded string.

gaetk2.tools.structured_xls package

Todo:

- Explain what gaetk2.tools.structured_xls is for

class gaetk2.tools.structured_xls.**XLSwriter**(*output=None*, *sheetname='This Sheet'*)

csv - Module compatible Interface to generate excel files.

... but you have to call *save()* oder *getvalue()* to generate the final XLS file.

Parameters

- **output** (*file or None*) – *optional* File-Like Object for *save()* to export to.
- **sheetname** (*str*) – *optional* Name of the single Worksheet we export to.

Uses the deprecated xlwt.

Usage:

```
xlswriter = XLSwriter()
xlswriter.writerow(['foo', 1, 2])
xlswriter.writerow(['bar', 3, datetime.date.today()])
xlswriter.save(open('test.xls')
```

writerow(*row*)
Eine Zeile schreiben. Row ist eine Liste von Werten.

save(*fd=None*)
Write rendered XLS file to *fd* or *self.output*.

getvalue()
Returns rendered XLS file as a StringIO().

1.10.2 gaetk2.exc module

exception gaetk2.exc.**HTTPException**(*message*, *wsgi_response*)
Bases: exceptions.Exception

exception

gaetk2.exc.**HTTP301_Moved**
alias of webob.exc.HTTPMovedPermanently

gaetk2.exc.**HTTP302_Found**
alias of webob.exc.HTTPFound

gaetk2.exc.**HTTP303_SeeOther**
alias of webob.exc.HTTPSeeOther

gaetk2.exc.**HTTP307_TemporaryRedirect**
alias of webob.exc.HTTPTemporaryRedirect

gaetk2.exc.**HTTP400_BadRequest**
alias of webob.exc.HTTPBadRequest

gaetk2.exc.**HTTP401_Unauthorized**
alias of webob.exc.HTTPUnauthorized

gaetk2.exc.**HTTP403_Forbidden**
alias of webob.exc.HTTPForbidden

gaetk2.exc.**HTTP404_NotFound**
alias of webob.exc.HTTPNotFound

gaetk2.exc.**HTTP405_HTTPMethodNotAllowed**
alias of webob.exc.HTTPMethodNotAllowed

gaetk2.exc.**HTTP406_NotAcceptable**
alias of webob.exc.HTTPNotAcceptable

gaetk2.exc.**HTTP307_TemporaryRedirect**
alias of webob.exc.HTTPTemporaryRedirect

gaetk2.exc.**HTTP409_Conflict**
alias of webob.exc.HTTPConflict

gaetk2.exc.**HTTP410_Gone**
alias of webob.exc.HTTPGone

gaetk2.exc.**HTTP413_TooLarge**
alias of webob.exc.HTTPRequestEntityTooLarge

gaetk2.exc.**HTTP415_UnsupportedMediaType**
alias of webob.exc.HTTPUnsupportedMediaType

gaetk2.exc.**HTTP500_ServerError**
alias of webob.exc.HTTPServerError

gaetk2.exc.**HTTP501_NotImplemented**
alias of webob.exc.HTTPNotImplemented

gaetk2.exc.**HTTP503_ServiceUnavailable**
alias of webob.exc.HTTPServiceUnavailable

gaetk2.exc.**HTTP504_GatewayTimeout**
alias of webob.exc.HTTPGatewayTimeout

1.10.3 gaetk2.models module

1.10.4 Module contents

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

g

Index

Symbols

A

B

C

D

E

F

G

H

I

S

T

U

V

W

X

Y