Bericht Documentation

Release 0.1

Bericht team

December 06, 2014

Contents

"Bericht" is the codename for a custom django project, aiming to provide the following to small or medium-sized communities:

- an RSS-aggregator with full-content archiving features.
- cms features to mix aggregated content with custom articles.
- a open calendar system with ical import and export.
- a forum in the style of popular question and answer sites.

Contents:

CHAPTER 1

General Overview

Bericht is a platform that aims to provide a sort of "hub" for small to medium communites. Core for this "hub" is de-centralized content and debates around this content. Current plans include articles obtained from news feeds (using the RSS and Atom file format) and calendars that provide iCalendar export.

Besides imported content, articles and events can be created on the platform. Users can pose Questions and provide Answers (*Q&A*). Furthermore, dossiers provide a means to tie together topic-specific content from various sources augmented with an introductory article. A live ticker should provide means to cover live events on-line.

Users can register at the platform. In order to ensure that content on the platform stays on topic, users are divided into three groups:

- **Normal users** can author non-imported content, i.e. comments and Q&A. They can also propose new sources for imported content.
- **Trusted users** can vote on content from normal users and sources that are not fully trusted. Their content is also subject to the vote of other trusted users.
- **Editors** can put users into the trusted users group, can author articles and static pages (e.g. terms of service, about, etc.) and organize the content in various ways.

CHAPTER 2

Content Types

This section details the individual content types.

2.1 Global Content Types

These content types form the backbone of the platform and are connected to several other content types.

- **Entry** is an abstract concept that links to the more specific content type (i.e. they sub-class Entry). The trust system (voting, etc.) is linked to this content type and all inheriting classes. Every inheriting content type has a method to return teaser-like HTML.
- **Comment** can be appended to any of the following content types: Article, Q&A questions and answers, and events.
- **Static Page** are for some static stuff, like 'About', 'Contact', 'Terms of Service', etc.

Static Pages and *Comments* are provided by mezzanine, *Entry* is, together with the *voting system* and the *front page* contained in an app.

2.2 Articles

Articles are long(ish) texts with formatting, possibly multimedia, etc. They are shown to the public and users. We decided not to use abstract article as a common parent class to reduce complexity, instead use Entry (see at the top).

- **Article** is authored on Bericht
- **ImportedArticle** is created from a news feed item (*FeedItem*) and holds the feed item's link HTML

2.3 Q & A

This should provide questions & answers similar to the various http://stackexchange.com sites. But here it should be the case that people ask questions anonymously (to all but the administrators), i.e. the nickname should not be visible. Answers and comments (except from the person who asked) are with names. It is not yet clear if we include up- and down-voting of answers and approval of "the correct" answer.

The sub-content for Q&A is

- **Question**
- **Answer**

2.4 Event Calendar

An event calendar that supports singular and recurring events. Input/Editing is done on the EventDefinition while EventInstance is used for display (i.e. a recurring event has several EventInstances for one EventDefinition). It should support export for individual EventInstances, full calendars and all events from all calendars (which is the collection of events from one user) in the iCalendar format. Goal is also to support iCalendar import someday in the future.

- **EventDefintion** holds most information about an event, including how/if it is recurring
- **EventInstance** is used for display and can be N for 1 *EventDefinition*. Title, Body, etc. is taken from *EventDefinition*, what changes is date (and possibly time).

2.5 Dossier

A *Dossier* is a topic-specific collection of content with an introductory text, a small number of manually curated "must-reads" and the possibility to show a list of items based on keywords/tags. Dossiers should always have the same url but the content is meant to be updated regularly. Content types that can be connected are Articles, Q&A and Bookmarks. As it should be 'timeless', no events (? up for debate). Multiple users should be able to edit dossiers, revisions should be stored for a editing history (display of that history is not a priority).

2.6 Live Ticker

A *Live Ticker* is a feature that should provide infrastructure to do live coverage of events. As such it provides means to add *updates* and show these in (reversed) chronological order with timestamps.

On the public front end, a live ticker is displayed on an mobile-optimized page that automatically loads new updates at top using JavaScript/AJAX. If a *Live Ticker* is finished, the representation is different in that the updates are shown in chonological order.

It is possible to put the *Live Ticker* on the front page (showing the latest k updates) and also show it as breaking, i.e. always on the top of the page on *Bericht*, unless the user decides to hide it (clicking on an 'x').

On the back end, an interface is provided that is tailored to quick data entry, setting the time to the current time as default and providing autocomplete for the optional metadata of the updates (based on previous updates). It is also well usable from mobile devices.

A *Live Ticker* can be created by everyone with a certain trust level (either editors or trusted users) and the creator can add any users as contributors. This information is not shown publicly.

A *Live Ticker* constists of the following content types:

- **LiveTicker** provides a description of the event covered and optional information that is always visible at top as long as the ticker is active. It also holds information about who created the ticker and who is allowed to post updates.
- **Update** is an individual update and provides a timestamp, the update content and a field for optional additional data that can be used for location information.

2.7 Bookmarks

Bookmarks provides the possibility for users to post links that they think are of interest to the community.

For every *Bookmark* it is required that the user states, with a comment, why this link is interesting. Users are encouraged to add tags to a Bookmark. Bookmarks are subject to voting and the link's HTML is fetched for archival purposes and article extraction is run on the HTML. This extracted article is not shown in full (mainly due to copyright issues), but used for search and the start of the article is used as a teaser.

The content type is thus as follows:

- **Bookmark** holds the link, user comment, HTML, extracted article and metadata.

CHAPTER 3

Content Handling

Content on *Bericht* is often imported from external sources and much of the content needs approval from trusted users. A few words on how imported content is handled and how the workflow for user-interaction is currently designed.

3.1 Aggregator

Aggregator takes care of importing content from news feeds. This should be as much separated from *ImportedArticle* as possible: Aggregator models store the "original" data from the news feed, while *ImportedArticle* (which is created from *FeedItem*) fetches the link's HTML and runs the article extraction. *ImportedArticle* is what is being displayed and maybe edited/augmented (with additional tags, better teaser, fixed content, etc.)

- **FeedFile** stores the feed file and archives it, time-stamped
- **Feed** stores the parsed feed
- **FeedItem** holds individual feed items

3.2 Artex: Article Extraction

Artex extracts articles from HTML pages. It is based on readability-lxml which itself is based on the readability library from arc90. Because many news feeds provide only teasers, we decided to use article extraction for all news feed items. Article extraction is done when creating an *ImportedArticle* from a *FeedItem*: The linked website's HTML is fetched, stored and then *artex* is run on the HTML.

Artex wraps around the readability-lxml library and adds parameters that proved useful during our tests. First are additional 'negative' keywords that can are stored in `settings.ARTEX_NEGATIVE_KEYWORDS` and are used to identify non-article HTML elements. Another are the `settings.ARTEX_METADATA_TERMS` that are used to identify HTML elements at the start and end of an article that contain article metadata (or ads).

3.3 Source Re-Check

Often it happens that blog posts, articles, etc. get modified (shortly) after their first publishing. Often these changes are meaningful, e.g. correction of facts, additional relevant information, etc.

The goal for bericht is, on the one hand, to quickly show new content. On the other hand, we want to reflect these changes. To accomplish this, we re-run the article extraction *n* hours after the first import.

As editors can manually edit an *ImportedArticle*, these manual changes could get lost if the content was simply overridden. To prevent this, the editors get a notification if there is an update at the source for a manually edited article. The changes are then shown in a diff-like view (highlighting additions and deletions between the current local article and the updated source article).

3.4 Voting

We decided to use a customized fork of django_voting, after we looked into a bunch of available solutions, because we need to record different types of votes - yes, no, veto and abstain - not just up- and downvotes which could be represented by integer values.

A `Vote` is its own database object and has a one-to-one relationship to the `Entry`, or theoretically any other object, voted upon. A custom model manager provides methods to record a vote, get a users vote and to get all votes on a given object.

The decision what to do with those votes is up to the model on which the vote was casted, you can find an example in `Entry.is_public`

CHAPTER 4

Installation Instructions

Bericht is currently written in python 2, because mezzanine has not been ported to python 3 so far. So you need at least python 2 (including development headers) and pip, pythons package manager as well as two libraries for xml-parsing, libxml and libxslt. On Debian(-based) systems the following should be sufficient::

```
apt-get install python python-dev python-pip libxml2-dev libxslt-dev
```

Tip: Usage of virtualenv is strongly recommended!

Install the requirements:

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

Create a `local_settings.py` file:

```
DEBUG = True
DATABASES = {
    "default": {
        "ENGINE": "django.db.backends.sqlite3",
        "NAME": "dev.db",
    }
}
```

Initialize the database:

```
python manage.py createdb --nodata
python manage.py migrate
```

Run the development server:

```
python manage.py runserver
```

Have fun!

4.1 Ansible

We deploy our systems using ansible, the scripts are tested to run on debian stable and included in `deploy/`. Change `bericht.dev` in `deploy/hosts` to the IP or hostname of your debian server and run:

```
ansible-playbook -i deploy/hosts -K deploy/site.yml
```

CHAPTER 5

Coding Guidelines

You probably read this because you decided to contribute to Bericht. That's nice and highly appreciated, but please make sure that your code respects the following conventions before you push code into our repositories.

5.1 PEP8

Python comes with its own opinionated style guide, which is called pep8 and available online. We use flake8 to check if our code respects pep8 and does not contain known symptoms of common problems ("code smell").

Note: It is strongly recommended to use a git hook to ensure that you are only committing code which does satisfy flake8. We might reject code that does not!

To use such a git hook, create a file in `.git/hooks/pre-commit` with the following contents:

```
#!/usr/bin/env python

import os
import subprocess
import sys


def system(*args, **kwargs):
    kwargs.setdefault('stdout', subprocess.PIPE)
    proc = subprocess.Popen(args, **kwargs)
    out, err = proc.communicate()
    return out

def main():
    project_dir = os.path.dirname(os.path.dirname(
        os.path.dirname(os.path.realpath(__file__))))
    print(project_dir)
    output = system('flake8', '.', cwd=project_dir)
    if output:
        print output,
        sys.exit(1)

if __name__ == '__main__':
    main()
```

and make it executable:

```
chmod +x .git/hooks/pre-commit
```

Hint: If you are absolutely certain that a file should not be checked, add `# flake8: noqa` to the beginning of that file. This should only be used for configuration of used tools or auto-generated code like migrations, never for production code!

5.2 Tests

Please write tests whenever you add or change functionality and run existing tests before you push. We use Djangos tools which are based on Pythons unittest module for unit tests and behave and splinter for acceptance testing.

Run unit tests for the `aggregator` app with:

```
python manage.py test aggregator
```

...and acceptance tests with:

```
python manage.py test aggregator --testrunner=bericht.utils.test_runner.AcceptanceTestSuiteRunner
```

A test suite for client-side javascript tests must yet be chosen.

CHAPTER 6

Third Party Packages

Bericht uses a lot of software written by other people. For reference, links to their documentation are collected here. Please make sure that the right version is linked where available if you upgrade a package.

- Python 2.7
- Django 1.5.5
- Mezzanine
- requests
- feedparser
- django-taggit
- South
- Sphinx
- BeautifulSoup
- django-debug-toolbar
- django-extensions
- django-filter
- djang-rest-framework

CHAPTER 7

Indices and tables

- *genindex*
- *modindex*
- *search*