Ekklesia

Release 20.06.0

Jul 19, 2020

Contents

This is the development and operations documentation for the **Ekklesia e-democracy platform**.

View the full documentation at ReadTheDocs

CHAPTER 1

Goals

The aim of the Ekklesia project is to provide an open, extensible platform for direct electronic democracy. Organisations have different requirements for their policy drafting and decision making process. Instead of trying to build a monolithic one-fits-all solution, we want to integrate existing free software and provide open interfaces. Ekklesia is a framework for building e-democracy systems but should still be usable out-of-the-box for common workflows. Applications developed as part of the platform should be highly customizable themselves, either by configuration or easy extension on the source code level.

CHAPTER 2

Subprojects

The platform consists of multiple applications and supporting projects which use separate repositories.

- ekklesia: Shared documentation
- ekklesia-portal: Motion portal, public API and administrative interface. You can visit a running instance at antrag.piratenpartei.de
- ekklesia-voting: Pseudonymous voting component
- ekklesia-common: Common code for the ekklesia platform
- discourse-ekklesia: Discourse plugin for the Ekklesia platform

CHAPTER 3

External Software

The projects aims to integrate with externally developed applications that serve the purpose of creating a e-democracy platform. Currently, we use or are working on integrating the following projects:

- Discourse: platform for community discussion
  - collaborative development of motion drafts
  - export/import of motion drafts
  - general discussion tool
- Keycloak: identity and access management with OpenID Connect support
- OpenSlides: digital motion and assembly system
  - motion export to Openslides
  - voting result import from Openslides
- VVVote: cryptographic anonymized online voting system

3.1 Operations

Note: If not stated otherwise, the operations documentation currently applies to all Ekklesia Python web apps, *ekklesia-portal* and *ekklesia-voting*. *ekklesia-portal* is used as an example in the documentation.

- Backend: Python 3.8
- Web framework: Morepath
- Frontend
  - Pyjade (like Pug)
  - Jinja

  - Bootstrap 4
  - Sass
  - Javascript
- Database: PostgreSQL 12
- Dependency management and build tool: Nix
- (Optional) Docker / Kubernetes for running Docker images built by Nix

3.1.1 Running In Production

A production environment can be built by Nix. The generated output doesn't have additional requirements. The application can be run by a start script directly or using the Docker image built by Nix. Static assets are built separately and can be served by the included minimal Nginx. As for the application itself, we can build a standalone start script or a Docker image.

Without Docker

- Build and run app with the config file `config.yml`:

```
nix-build nix/serve_app.nix -o serve_app && serve_app/bin/run config.yml
```

- Build static assets and run Nginx:

```
nix-build nix/serve_static.nix -o serve_static && serve_static/bin/run
```

- Build static assets for use with another web server:

```
nix-build nix/static_files.nix -o static_files
```

With Docker

By default, Docker images are tagged with a version string derived from the last Git tag. You can set a different tag by adding `--argstr tag mytag` to the `docker*.nix` calls.

- Build and import app image:

```
./docker.nix -argstr tag mytag docker load -i docker-image-ekklesia-portal.tar
```

- Build and import static file image:

```
./docker_static.nix -argstr tag mytag docker load -i docker-image-ekklesia-portal-
↪static.tar
```

- Run app container:

```
docker run -p 127.0.0.1:8080:8080 -it -v $(pwd)/config-docker.yml:/config.yml␣
↪ekklesia-portal:mytag
```

- Run static file container:

```
docker run -p 127.0.0.1:8081:8080 -it ekklesia-portal-static:mytag
```

The app is now served at port 8080 and static files at port 8081 which are only reachable from localhost (127.0.0.1).

3.2 Development

Note: If not stated otherwise, the development documentation currently applies to all Ekklesia Python web apps, *ekklesia-portal* and *ekklesia-voting*. *ekklesia-portal* is used as an example in the documentation.

- Backend: Python 3.8
- Web framework: Morepath
- Frontend
  - Pyjade (like Pug)
  - Jinja
  - Bootstrap 4
  - Sass
  - Javascript
- Database: PostgreSQL 12
- Dependency management and build tool: Nix
- (Optional) Docker / Kubernetes for running Docker images built by Nix

3.2.1 Development Quick Start

The following instructions assume that *Nix* and *lorri* are already installed, and an empty + writable PostgreSQL database can be accessed somehow.

If you don't have *Nix* and *lorri* or can't use an existing PostgreSQL server, have a look at *Development Environment*.

1. Clone the repository and enter nix shell in the project root folder to open a shell which is your dev environment:

```
git clone https://github.com/Piratenpartei/ekklesia-portal
cd ekklesia-portal
lorri shell
```

2. Compile translations and CSS:

```
makebabel.ipy compile
sassc -I $SASS_PATH src/ekklesia_portal/sass/portal.sass \
    src/ekklesia_portal/static/css/portal.css
```

3. Create a config file named `config.yml` using the config template from `src/ekklesia_portal/config.example.yml` or skip this to use the default settings from `src/ekklesia_portal/default_settings.py`. Make sure that the database connection string points to an empty + writable database.
4. Initialize the dev database with a custom config file:

```
python tests/create_test_db.py -c config.yml
```

5. The development server can be run with a custom config file by executing:

```
python src/ekklesia_portal/runserver.py -debug -c config.yml
```

3.2.2 Development Environment

To get a consistent development environment, we use Nix to install Python and the project dependencies. The development environment also includes PostgreSQL 12, linters, a SASS compiler and pytest for running the tests.

The following code snippets are written for *ekklesia-portal* but also work for *ekklesia-voting* when you change the project name.

Install Nix

Installation instructions taken from Getting Nix. *See the link for other installation methods.*

Nix is currently supported on Linux and Mac. The quickest way to install Nix is to open a terminal and run the following command (as a user other than root with sudo permission):

```
curl -L https://nixos.org/nix/install | sh
```

Make sure to follow the instructions output by the script. The installation script requires that you have sudo access to root.

Install Lorri

The best way to get a development shell is to use Lorri which improves the builtin `nix-shell` command.

Install `lorri` (also works for updates):

```
nix-env -if https://github.com/target/lorri/archive/master.tar.gz
```

This is enough to use `lorri shell` needed for the quick start section below.

We also recommend using the `direnv` integration. This will automatically enter the development shell when changing to the project directory. Please follow the Lorri Installation Instructions.

Running PostgreSQL as User

You can run a PostgreSQL database server with your user permissions if you don't want to use an existing database server. Run the pg_ctl commands from the nix shell.

Run as user:

```
pg_ctl -D ~/postgresql init
postgres -D ~/postgresql -k /tmp -h ''
```

Create database (in another terminal):

```
createdb -h /tmp ekklesia_portal
```

You can connect to the database with psql now:

```
psql -h /tmp ekklesia_portal
```

Use the following connection string in the app config file:

```
database:
  uri: "postgresql+psycopg2:///ekklesia_portal?host=/tmp"
```

Updating The Development Environment

`lorri shell` always installs changed dependencies and tools before entering the development shell which takes some seconds.

When using the `direnv` integration, running `lorri daemon` in the background automatically updates the development shell when something changes. Press Enter in the development shell to trigger the first daemon build or to see the changes in the shell made by `direnv`.

You can also trigger an update by running `lorri watch --once` if you don't want to run `lorri daemon`.

Editor / IDE Integration

Tested with VSCode, Pycharm

Run this to build the environment:

```
./python_dev_env.nix
```

This creates a directory `pyenv` that is similar to a Python virtualenv. The Environment should be picked up by the IDE using the Python interpreter in the directory. A restart may be required.

3.2.3 Concept Tutorial

A concept is a way to organize code. Technically it's a package that bundles the code needed to view, create, modify and delete (or CRUD: create, read, update, delete) a *thing* or *entity* in your application's domain. Most applications will consist of multiple concepts. Typical concepts would be user, profile, comment, item, document or department.

The *thing* we want to implement here is a blog post. We use *ekklesia-portal* as target project here but this also works for other Python Ekklesia applications with a Web UI.

Building Blocks

Concepts use the following objects passed in from the outside:

- **Request**: A HTTP request object, provided by WebOb and extended by Morepath's Request. Gives access to a database session, POST data, a browser session and application settings, for example.
- **Model**: Typically, this is an object from a database or a collection of them. But can be of any class that is used to carry around information belonging to a concept.
- **Cell**: Renders a HTML view of a concept by using data from a model object.
- **Template**: Cells usually produce HTML output by using a *Pyjade Template*. Templates use code from their associated cell and model fields to display stuff.
- **Path**: Maps URL pattern to a model. It's easy to link to model instances from cells by using the `request.link` method. (-> Morepath: Paths and Linking)
- **View**: Contains code to prepare and render a HTTP response. In most cases, HTML views should delegate the actual rendering to a cell. (-> Morepath: Views)
- **Form**: Renders a HTML form based on a schema, captures the HTTP response and passes the input to the schema for validation. (-> Deform: Basic Usage)
- **Schema**: Defines properties that are rendered by a form, validates incoming data that should be written to a model object.

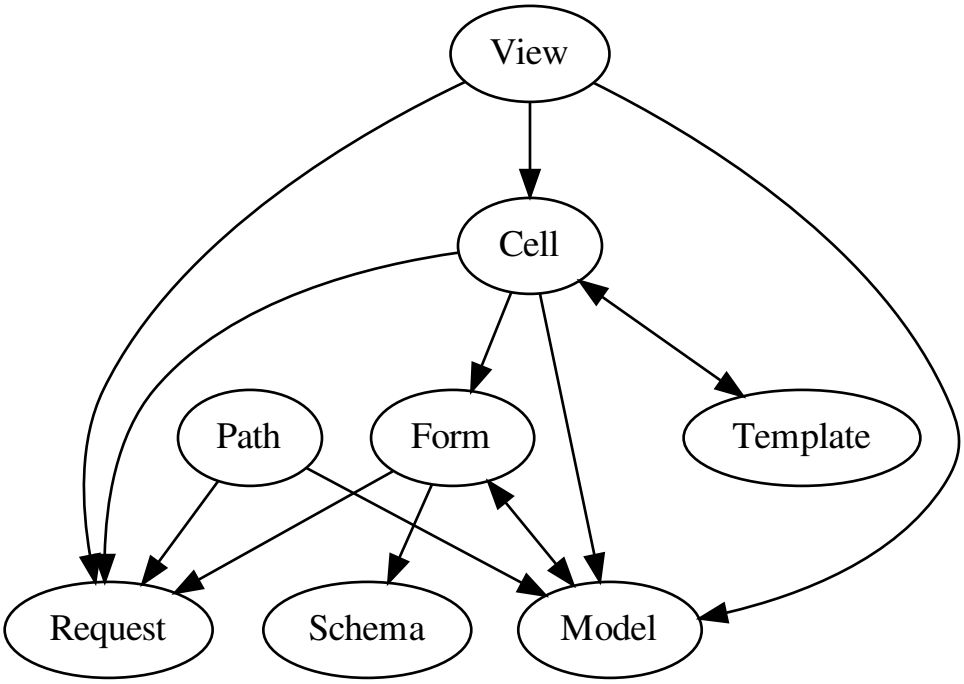

Fig. 1: Building blocks of a concept

Generate a Concept

The Command **`ekklesia-generate-concept`** generates a working concept implementation together with a test. It contains views for creating, editing and displaying instances of the concept.

Let's generate our Comment concept:

```
$ ekklesia-generate-concept blog_post

generated concept /home/ts/git/ekklesia-portal/src/ekklesia_portal/concepts/blog_post,
tests are located at tests/concepts/blog_post
```

The generate source tree looks like this:

```
$ tree /home/ts/git/ekklesia-portal/src/ekklesia_portal/concepts/blog_post
/home/ts/git/ekklesia-portal/src/ekklesia_portal/concepts/blog_post
├── blog_post_cells.py
├── blog_post_contracts.py
├── blog_post_helper.py
├── blog_posts.py
├── blog_post_views.py
├── __init__.py
└── templates
    ├── blog_post.j2.jade
    ├── blog_posts.j2.jade
    ├── edit_blog_post.j2.jade
    └── new_blog_post.j2.jade
```

- `blog_post_views.py`: path and view functions
  - path blog_posts: handles listing blog posts and creating new ones
  - path blog_post: handles viewing a blog post and editing it
  - view index: list blog posts
  - view new: show form for new blog post
  - view create: handle POST request from the new blog post form
  - view edit: show form for editing an existing blog post
  - view update: handle POST request from the edit blog post form
- `blog_post_contracts.py`: bundles blog post schema and form
- `blog_posts.py`: collection model used by the blog posts path
- `blog_post_helper.py`: utilities that can be used in cells and from other concepts

Tests live outside of the application package:

```
$ tree /home/ts/git/ekklesia-portal/tests/concepts/blog_post
/home/ts/git/ekklesia-portal/tests/concepts/blog_post
├── __init__.py
└── test_blog_posts.py
```

Test it

- unit test a cell: given a model instance, when using an option or not, then produces the expected HTML output
- functional testing of views: fill out forms, submit and check output with WebTest

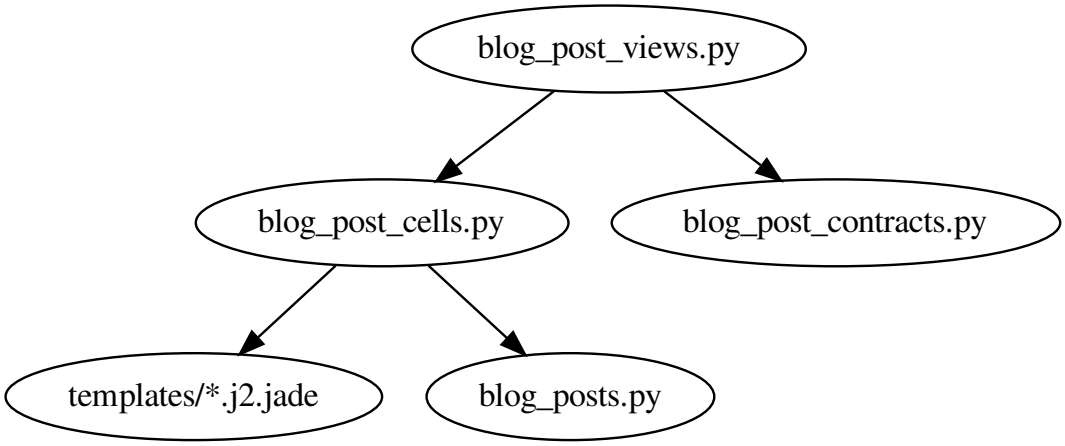

Fig. 2: Generated files for the Comment concept

Extend it

- implement cell properties that can be used in a template
- extend the schema and set custom widgets in blog_post_contracts
- put more path and view functions in blog_post_views
- (a bit more advanced) refactor templates with fragments

3.2.4 Testing

Tests are written and executed using Pytest.

Running Tests

1. Enter nix shell in the project root folder to open a shell which is your test environment:

```
cd ekklesia-portal
nix-shell
```

2. Compile translations:

```
ipython makebabel.ipy compile
```

3. Create a config file named `testconfig.yml` using the config template from `tests/testconfig.example.yml` Make sure that the database connection string points to an empty + writable database.
4. Initialize the test database:

```
python tests/create_test_db.py -c testconfig.yml
```

5. The tests can be run with `pytest` from the repository root directory.

3.2.5 Templates

We use PyJade as a Jinja extension. Templates with the extension `.j2.jade` are passed to PyJade and converted to a Jinja template.

Our templates mostly support the syntax of Pug which was called *Jade* before. Instead of Javascript, pieces of code in templates are interpreted as Jinja code. Keep that in mind when reading the language reference for Pug. As a basic rule, keep complicated logic away from templates and use Python code from the cell associated with a template. Reading existing templates and following their style is a good idea.

HTML Tags and Attributes

By default, text at the start of a line (or after only white space) represents an HTML tag. Indented tags are nested, creating the tree structure of HTML.

```
ul
  li Item A
  li Item B
  li Item C
```

Tag attributes look similar to HTML, with optional commas between them. Simple values are strings with quotes:

```
a(href='//example.com', target='_blank') Example
```

Class and ID Literal

In addition to using the class attribute like in HTML, classes can be added to tags with the `.classname` syntax:

```
a.button.button-primary
```

```
<a class="button button-primary"></a>
```

For divs, you can skip writing `div` and just use a class literal:

```
.content
```

```
<div class="content"></div>
```

ID literals follow the same rules start with a `#`.

Values from Expressions

Attribute values support Jinja expressions:

```
a.link(href=example_url, class=("extra" if options.extra else "")) Example
```

Adds “extra” to class attribute only if options.extra is true:

```
<a class="link extra">Example</a>
```

Boolean Attributes

```
input(type='checkbox' checked) [x]
|
input(type='checkbox' checked=true) [x]
|
input(type='checkbox' checked=false) []
```

Differences to Pug

- Jinja expressions instead of Javascript
- no multiline attributes
- no special handling of class and style attributes
- no support for *&attributes* syntax

Code

Buffered (With Output)

= is used to produce output from a Jinja expression. Pug calls this *buffered code*. This works like `{{ }}` in Jinja. For security, output is automatically escaped by Jinja.

```
p
  = 'This code is <escaped>!'
```

For translations, `gettext` with _ as alias and `ngettext` from Babel are available in all templates.

Templates that are used by a Cell have access to all methods of the Cell that don't start with _. Methods that only have the self argument are automatically turned into cached properties and can be used without `()`.

```
a(href=index_url)= _('back_to_index')
```

This uses the return value of a cell method `def index_url(self): ...` as link target and the translation for 'back_to_index' as link text.

Warning: Don't put a space before the =!

Often it's nicer to put the code in the next line which has the same effect as the last example:

```
a(href=index_url)
  = _('back_to_index')
```

Unescaped HTML

To output HTML without escaping, wrap the string in Python code with `Markup()` from `markupsafe`.

Unbuffered

- can be used to run Jinja code directly, without output. Pug calls this *unbuffered code*. This works like `{% %}` in Jinja.

We use it for layout extension and blocks on the Jinja level but this may go away in the future.

```
- extends "ekklesia/layout.j2.jade"
```

```
- block content
  .content
```

In Jinja syntax, that block looks like this:

```
{% block content %}
<div class="content"></div>
{% end block %}
```

Try to avoid putting code in templates, most of the time there are better ways to express things. Logic should be implemented in a Cell method or code called by a cell method.

Conditionals

Conditionals can check values for "truthyness" like in Jinja/Python:

```
if output
  = output
```

Writes `output` only if output is *truthy* (not empty string or not None, for example).

If ... else is also supported:

```
if options.show_full_output
  = full_output
else
  = short_output
```

Plain Text

Text after a tag is just rendered as plain text:

```
p Some Text
```

You won't need that often because we want to translate most of the strings.

Use a pipe to put plain text on a separate line. Whitespace after the pipe is included in the output.

```
p
  |Some Text
  | More Text
```

Whitespace at the end of the line is also rendered! This should be avoided because many editors will strip trailing whitespace and change semantics of the template.

```
<p>Some Text More Text</p>
```

Mixing Code with Text

Let's say we want to output two values separated by a space:

```
p
  = first_part
  | 
  = second_part
```

Iteration

Use for to iterate over a sequence:

```
ul
  for item in items
    = li.list-item= item.text
```

Comments

```
// comment which is rendered to a HTML comment

//- comment which is just for the template, not rendered to HTML
```

Code Style

- Always use 2 spaces for indentation. Do not mix!
- Line length should not exceed 120, but sometimes it's neccessary to write longer lines.
- Avoid putting `= output` directly after a HTML tag, especially when it has multiple attributes. Put it on a indented line instead.

3.2.6 Stylesheets

Generate CSS

CSS is compiled from Sass files that include files from Bootstrap and Font-Awesome. sassc is used as Sass compiler.

Generate CSS with:

```
sassc -I $SASS_PATH src/ekklesia_portal/sass/portal.sass \
  src/ekklesia_portal/static/css/portal.css
```

3.2.7 Dependency Management

Note: You only have to work with Poetry commands if you want to change Python dependencies. The Nix development shell provides a working Python interpreter with all needed packages.

Python dependencies are managed by Poetry. We use poetry2nix to integrate Poetry with our Nix automation code. Changes in the Poetry dependencies specification are picked up by Nix on the next run. If you use the Lorri daemon, adding, updating or removing packages automatically rebuilds the development shell which provides a Python interpreter with the wanted Python packages.

Python package names and version constraints are listed in the `[tool.poetry.dependencies]` and `[tool.poetry.dev-dependencies]` sections of `pyproject.toml`.

Common Python packages for all applications are provided by *ekklesia-common*. To see what's available, use `poetry show ekklesia-common` or have a look at `pyproject.toml`.

`poetry show` lists all packages that are pulled in directly or are dependencies of dependencies:

ekklesia-common is a Git dependency, so Poetry shows the version and the short Git commit id.

`poetry update` updates dependencies to their latest versions allowed by the version constraints.

`poetry add <package>` and `poetry remove <package>` should work as expected.

After changing dependencies, it's a good idea to run *nix-build* to see if everything builds with Nix. Using `lorri daemon` also helps finding problems. Some "exotic" packages need further help to build with Nix. Problems can be caused by build dependencies that are not recognized automatically by poetry2nix. In this case, the overrides in `deps.nix` must be changed. We already have some examples of doing that. Poetry2nix provides a lot of common overrides so most popular packages should just work.

3.3 Identity Management / Authentication

Ekklesia applications that need to identify users and have access to protected information like the eligibilty for voting act as OpenID Connect Client (based on OAuth 2). Currently, we are working with Keycloak but other OpenID Connect compliant solutions with good configurability may work as well.

Users that want to log in to an Ekklesia application are redirected to the OpenID Connect provider where they enter their credentials, username and password. Turning on two-factor-authentication (2FA) should be recommended to users or even be required to log in.

3.3.1 Expected Claims in the ID Token

Ekklesia applications use the standard `sub` claim in the OpenID Connect ID token. This should be configured as an "pairwise subject identifier" which means that it's unique in the context of a single application and is only known to that application.

Applications can save the relationship between `sub` and an internal user object that is used to save additional user data needed by the application.

There could be additional user identifiers that are known to multiple applications if they need to exchange information about users. This is not used yet but may be needed for implementing a notification system that can be used by multiple applications.

Portal and voting components need to know if a user can use certain functions that are restricted. They may use the following claims in the ID token:

- `verified`: is this an verified user? (for example: has been personally identified by some process)
- `eligible`: is this user eligible to vote? (for example: has paid their membership fee for this year?)
- `external_voter`: is using an external means of voting (for example: uses mail-in ballot which prevents voting online)

Applications may need to know the `roles` of an user. Role names may map to the name of a department. Other roles are ignored. There may be additional supported role name to specify users with special permission, for example department administrations and global admins.