Ekklesia

Release 23.07.0

Ekklesia contributors

Jul 05, 2023

CONTENTS

1 Goals **3**

2 Features **5**
- 2.1 For All Users . . . 5
- 2.2 For Administrators . . . 5
- 2.3 Planned Features . . . 6

3 Ekklesia Projects **7**

4 External Software **9**
- 4.1 Operations . . . 9
- 4.2 Development . . . 11
- 4.3 Identity Management / Authentication . . . 28

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

CHAPTER

ONE

GOALS

The aim of the Ekklesia project is to provide an open, extensible platform for direct electronic democracy. Organizations 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 provides usable out-of-the-box configuration 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.

Ekklesia is designed with privacy in mind: applications have to be usable with pseudonyms, anonymous voting should be supported and personal data must only be shared with system components if it's really required and with user consent. Encryption must be used wherever possible, preferably end-to-end encryption, especially for sensitive content like personal voting confirmations.

CHAPTER

TWO

FEATURES

The available feature set depends on the used components. See the list in the next section. Most of the listed features are implemented in ekklesia-portal.

2.1 For All Users

- Log in using Single-Sign-On for all applications.
- Discuss ideas and collaborate on the development of propositions (motions).
- Search for draft, in-progress and finished propositions using full text and meta data.
- Gather submitters/supporters for a proposition.
- Submit propositions for commission review.
- Write user-generated content using plain text/Markdown.
- Discuss submitted propositions with user-rated/sorted pro and contra arguments.
- Amend propositions and submit counter-propositions.
- Vote anonymously using score voting on ballots with multiple options.
- View documents (like statutes, programs and more) and propose changes.
- View propositions for a specific voting phase (event).
- Change user interface language (currently English, German and partly French support).

2.2 For Administrators

- Set up multiple sub-departments of an organization with different workflow settings.
- Configure proposition types with different properties.
- Customize text content in the user interface, in multiple languages.
- Modify proposition content and meta data for administrative reasons.
- Build custom navigation/overview pages using Markdown in multiple languages.
- Export propositions to OpenSlides (CSV) and import voting results.

2.3 Planned Features

- Switch between multiple languages for user-submitted content with manual and automated translations.
- Dynamic visualization of amendments and document changes.
- Track changes to documents and propositions in version control repositories (Git).
- (Semi-)automated merging of changes into documents.
- Gather arguments and vote recommendations from your social contacts.

CHAPTER

THREE

EKKLESIA PROJECTS

The platform consists of multiple applications and supporting projects which use separate repositories. Projects developed by the Ekklesia project can be found in the edemocracy organization on GitHub.

- ekklesia: Shared documentation
- ekklesia-portal: Motion portal, public API and administrative interface.
- ekklesia-voting: Pseudonymous voting component
- ekklesia-notify: API for sending notifications to users
- ekklesia-common: Common code for the ekklesia platform
- nix-ekklesia-vvvote: Automated Nix/NixOS deployment for the VVVote voting system.
- discourse-ekklesia: Discourse plugin for the Ekklesia platform

CHAPTER

FOUR

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
- Matrix: encrypted notifications

4.1 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:
  - Main language: Python 3.11
  - Web framework: Morepath
  - Web API framework for ekklesia-notify: FastAPI
  - Testing: pytest, WebTest
- Frontend
  - Templates PyPugJS (similar to Pug) with Jinja as template engine.
  - Sass Framework Bootstrap 4
  - htmx for "AJAX" requests directly from HTML.
- Database: PostgreSQL 15

- Dependency management and build tool: Nix
- Documentation: Sphinx with MyST Markdown parser.
- (Optional) Docker / Kubernetes for running Docker images built by Nix

4.1.1 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.

NixOS Module

There's a NixOS module in `nix/modules/default.nix` to run the backend.

See ekklesia.nix in ekklesia-deploy for an example on how to use the module. In ekklesia-deploy, you can also see how you can pin the version of ekklesia-portal using niv and how to service static files with Nginx.

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 app image as `docker-image-ekklesia-portal.tar` and import it:

```
./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).

Directly From Source

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

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

To change the config location, set the environment variable `EKKLESIA_PORTAL_CONFIG`.

- 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
```

4.2 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:
  - Main language: Python 3.11
  - Web framework: Morepath
  - Web API framework for ekklesia-notify: FastAPI
  - Testing: pytest, WebTest
- Frontend
  - Templates PyPugJS (similar to Pug) with Jinja as template engine.
  - Sass Framework Bootstrap 4
  - htmx for "AJAX" requests directly from HTML.
- Database: PostgreSQL 15
- Dependency management and build tool: Nix
- Documentation: Sphinx with MyST Markdown parser.
- (Optional) Docker / Kubernetes for running Docker images built by Nix

4.2.1 Development Quick Start

This chapter describes briefly how to set up a development environment to run a local instance of the application.

Setting up the environment for testing and running tests is described in *Testing*.

The following instructions assume that *Nix* is already installed, has Nix flakes enabled, and an empty + writable PostgreSQL database can be accessed somehow.

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

It's strongly recommended to also follow the instructions at *Setting up the Cachix Binary Cache* to speed up the installation.

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

```
git clone https://github.com/edemocracy/ekklesia-portal
cd ekklesia-portal
nix develop
```

2. Compile translations and CSS (look at `dodo.py` to see what this does):

```
doit
```

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. Set up the database for testing (look at `flake.nix` to see what this does):

```
create_test_db
```

5. Run tests:

```
pytest
```

6. Run the development server (look at `flake.nix` to see what this does):

```
run_dev
```

Run `help` to see all commonly used dev shell commands.

4.2.2 Development Environment

Note: To get an overview on how to set up a development environment, please read *Development Quick Start* or see *NixOS Virtualbox Dev VM* to learn how to set up a NixOS development VM from start to finish.

To get a consistent development environment, we use Nix to install Python and the project dependencies. The development environment also includes PostgreSQL, 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

Note: There's a new guide, Zero to Nix: Get Nix running on your system which uses an alternative installer that wants to make it easier to set up Nix in the correct way. This looks promising but we haven't tested it, yet. Please try it out!

Official Nix installer

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.

Enable Nix Flakes support

We use Nix Flakes, which are a widely used feature of Nix. They are still marked as experimental, though, so we need to enable the feature flag first if you used the official installation method.

You can add the required config to your local Nix config file like this:

```
mkdir -p ~/.config/nix
echo "experimental-features = nix-command flakes" >> ~/.config/nix/nix.conf
```

See Flakes - NixOS Wiki for more details and other methods.

if you want to know more about Nix Flakes, see Zero to Nix: Nix flakes

Direnv

Use direnv together with nix-direnv to get a automatically updating development shell. See the linked pages for instruction on how to set them up.

There's a :file:`envrc.example` in the project repository root which shows how to configure `direnv` for the project.

Setting up the Cachix Binary Cache

To speed up installation, you should add the edemocracy binary cache hosted on Cachix.

If your user can edit Nix config files (when using a single-user installation on a non-NixOS system, for example), just install the Cachix client and add the edemocracy cache:

```
nix-env -if https://github.com/cachix/devenv/tarball/latest
cachix use edemocracy
```

If cachix cannot change the config, it will instructions on how to do it. On NixOS, you have to run cachix as root (`sudo cachix use edemocracy`) or another trusted Nix user.

Install devenv

We use devenv to provide a development shell environment. You don't have to install this separately to work on the project and a limited version of devenv is available in the development shell itself but you might find some devenv commands useful, especially when working on project dependencies. We will probably use this tool more in the future.

```
nix-env -if https://github.com/cachix/devenv/tarball/latest
```

See Getting Started - devenv for more details. The documentation there also repeats some of the steps also included in here, like setting up Cachix and Nix.

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 dev 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

When using the `direnv` integration, the shell is automatically updated when something changes in the Python dependencies or the Nix code of the project. Press *Enter* in the development shell to trigger the build or run `direnv allow` if direnv didn't recognize a change.

If you don't use `direnv`, exit your current shell and run `nix develop` again.

Editor / IDE Integration

Tested with VSCode, Pycharm

Run this in the dev shell to build the environment:

```
build_python_venv
```

This creates a directory `venv` that looks like a Python virtualenv. The Environment should be picked up by the IDE using the Python interpreter in the directory. A restart of the IDE may be required.

4.2.3 NixOS Virtualbox Dev VM

You can use the official NixOS VirtualBox appliance if you want to run the project on our preferred production system NixOS.

This will guide you from downloading the appliance to a running application.

Note: You need VirtualBox installed on your system.

Note: If you want to run the application elsewhere, please look at *Development Quick Start*

VM Preparation

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

1. Get the NixOS 22.11 VirtualBox appliance and follow the instructions there to import and start the VM. Enable clipboard integration in the VirtualBox menu bar with *Devices -> Shared Clipboard -> Host to Guest* so you can copy-paste longer commands from here.
2. In the VM, open **konsole** (press **Alt-F1** and type *konsole*) and run the following commands as the demo user.
3. Add the *edemocracy* binary cache:

```
# password for sudo is `demo`
nix-shell -p cachix --run "sudo cachix use edemocracy"
# ignore the instructions shown by cachix, we do that later.
```

4. Edit `/etc/nixos/configuration.nix` (using `sudo nano /etc/nixos/configuration.nix`, for example) and change the `imports` line to:

```
# no commas between list items in Nix!
imports = [
  <nixpkgs/nixos/modules/installer/virtualbox-demo.nix>
  ./ekklesia_dev.nix
];
```

5. Fetch our recommended NixOS configuration with curl and put it in `/etc/nixos` (you can also `download it here`):

```
curl -O https://raw.githubusercontent.com/edemocracy/ekklesia/master/docs/
↪development/ekklesia_dev.nix
sudo mv ekklesia_dev.nix /etc/nixos
```

6. Rebuild the NixOS system to activate the new configuration and run zsh as new shell:

```
sudo nixos-rebuild switch
zsh
```

7. Tell direnv where nix-direnv is located:

```
echo "source /run/current-system/sw/share/nix-direnv/direnvrc" > ~/.direnvrc
```

Setting up the Project

1. Clone the repository and change to the checked out directory:

```
git clone https://github.com/edemocracy/ekklesia-portal
cd ekklesia-portal
```

2. Tell `direnv` to use `nix-direnv` whenever you enter the directory. `nix-direnv` starts building immediately which may take a while:

```
cp envrc.example .envrc
direnv allow
# direnv runs nix build now
```

   You can run `direnv allow` again to activate changes to the .envrc file or get an up-to-date shell when direnv couldn't run automatically.

3. Compile translations and CSS (look at dodo.py to see what this does):

```
doit
```

4. Create a config file named `config.yml` using the config template from `src/ekklesia_portal/config.example.yml`. Under `database`, you have to change `uri`. Under `app`, change `force_ssl` to *false* and `insecure_development_mode` to *true*. The config file should look like this:

```
database:
    uri: "postgresql+psycopg2:///ekklesia_portal?host=/run/postgresql"
app:
    instance_name: my_ekklesia_portal
    insecure_development_mode: true
    login_visible: true
    force_ssl: false
browser_session:
    secret_key: dev
    cookie_secure: false
    permanent_lifetime: 999999
```

5. Initialize the dev database:

```
create_dev_db
```

   This command populates the `ekklesia_portal` database.

6. Run the development server (look at `flake.nix` to see what this does):

```
run_dev
```

   You can use a browser, for example Firefox which is pre-installed on the VM to go to the application running at `http://localhost:8080`. Log in as `testuser`, or `testadmin` for a privileged admin user.

7. To compile changes to stylesheets and translations automatically, run in a second shell tab/window:

```
doit_auto
```

Setting up Tests

1. To set the test database connection URL {file}, open `.envrc` in an editor and uncomment the line with `EKKLESIA_PORTAL_TEST_DB_URL` (remove the `#`) . The line in the file should look like this now:

```
EKKLESIA_PORTAL_TEST_DB_URL="postgresql+psycopg2:///test_ekklesia_portal?host=/run/
↪postgresql"
```

2. Run `direnv allow` to activate changes to `.envrc`.
3. Set up test data:

```
create_test_db
```

   This command populates the `test_ekklesia_portal` database.

4. Run all tests:

```
pytest
```

4.2.4 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 *PyPugJS 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.

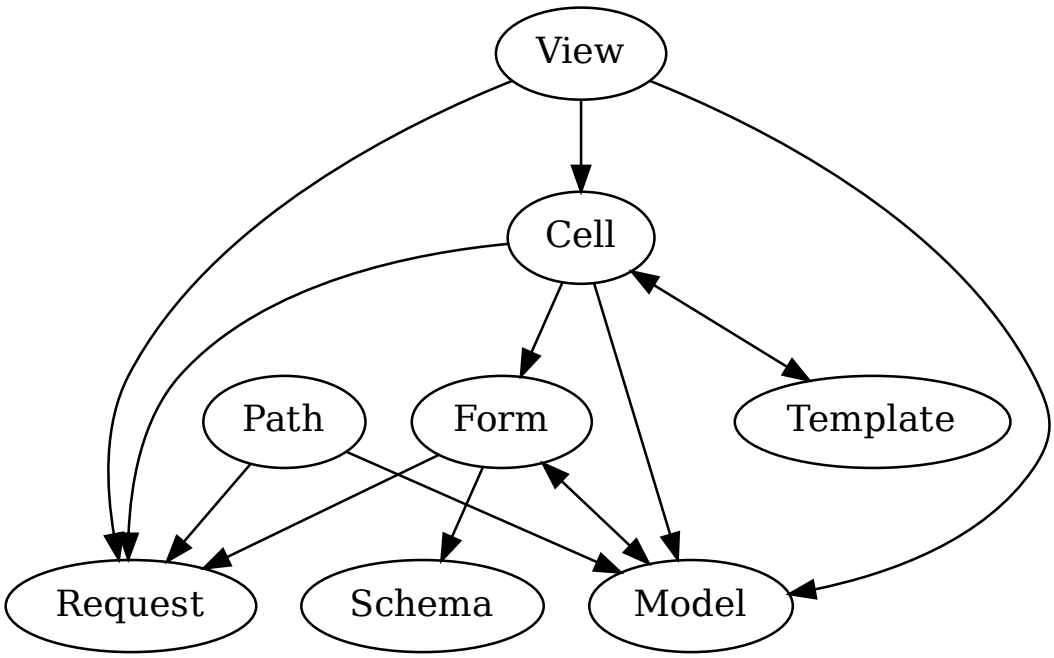

Fig. 1: Building blocks of a concept

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
```

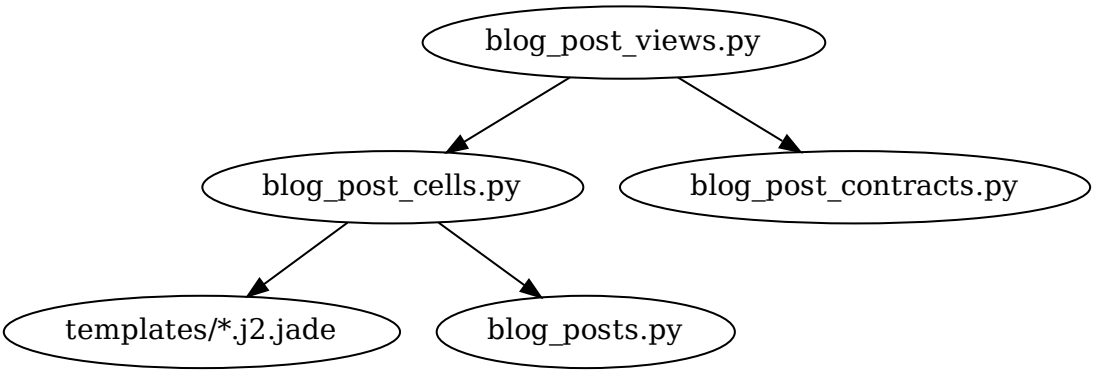

Fig. 2: Generated files for the Comment concept

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

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

4.2.5 Testing

Tests are written and executed using pytest.

Setup

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

Steps 1 and 2 can be skipped when you already followed *Development Quick Start*. Steps 3 and 4 set up a separate database for running the tests.

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

```
git clone https://github.com/edemocracy/ekklesia-portal
cd ekklesia-portal
nix develop
```

2. Compile translations and CSS (look at `dodo.py` to see what this does):

```
doit
```

3. Make sure that the database connection URL points to an empty + writable database. By default, the tests use the Postgres socket in `/tmp` and the database name is `test_ekklesia_portal`. You can set the environment variable `EKKLESIA_PORTAL_TEST_DB_URL` to customize the database connection.

4. Set up the database for testing (look at `flake.nix` to see what this does):

```
create_test_db
```

Running Tests

The tests can be run inside a Nix dev shell with the **`pytest`** command from the repository root directory.

You can supply a path to run only a part of the test suite, here only for the proposition concept:

```
pytest tests/concepts/proposition
```

This also works for single modules:

```
pytest tests/concepts/proposition/test_propositions.py
```

Or you can select tests by a part of the name:

```
pytest -k test_arg
```

See the Usage and Invocations in the pytest documentation for more info.

4.2.6 Templates

We use PyPugJS (formerly *Pyjade*) as a Jinja extension. Templates with the extension `.j2.jade` or `.j2.pug` are passed to PyPugJS 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 "truthiness" 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 necessary 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.

4.2.7 Stylesheets

CSS is compiled from Sass files in `src/ekklesia_portal/sass` which include files from Bootstrap and Font Awesome.

`sassc` is used as Sass compiler.

Generate CSS

Compile CSS (look at `dodo.py` to see what this does):

```
doit css_compile
```

To compile changes to stylesheets (and translations) automatically in the background, run:

```
doit_auto
```

4.2.8 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 *Direnv* integration, 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:

```
alembic               1.4.2           A database migration tool for SQLAlchemy.
dectate               0.14            A configuration engine for Python frameworks
deform                2.0.10          Form library with advanced features like nested␣
↪forms
ekklesia-common       20.07.0 b7c71cf
```

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 {ref}:`direnv` also helps finding problems. Some "exotic" packages need further help to build with Nix. Problems can be caused by build dependencies that aren't 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.

4.2.9 Database

Our primary database system is PostgreSQL. We use SQLAlchemy as ORM and SQL toolkit and Alembic for automated database schema migrations.

Setting up a Development Database

Create database schema and load example data for experimentation in development:

> **Warning:** Check the database section in the config file to avoid overwriting the wrong database!

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

Running Migrations

Check the current revision:

```
alembic current
```

Upgrade the schema in the configured database to the latest version (may include multiple migration steps):

```
alembic upgrade head
```

Use the `EKKLESIA_PORTAL_CONFIG` environment variable if you need to set the path to the app configuration file. The alembic command doesn't have an option for that.

Downgrading is also possible:

```
alembic downgrade -1
```

This rolls back the last migration step.

Schema Changes and Migrations

Migrations can be partially auto-generated by Alembic by comparing the SQLAlchemy model/schema code with the existing database.

First, change the model/schema code for the app in `src/ekklesia_*/datamodel.py`. Then, generate the migration file with a proper migration message (used for the filename and as a comment in the migration module):

```
alembic revision --autogenerate -m "Allow no target date for aborted voting phase"
```

Migration modules can be found in `alembic/versions/`.

Note: Check the generated code after generating it and remove the comments generated by Alembic!

Run the generated migration and check if it does the right thing. You should also test the downgrade.

4.2.10 Code Style

We use YAPF to check Python code style and apply automatic formatting. We plan to move to black.

Code should conform to PEP8, the style guide for Python code. YAPF goes beyond that and applies stylistic changes as configured in the style file `.style.yapf`. Our maximal line length is 120 but we want to reduce it to 88, the black default.

Auto-Formatting Python Code

Running **`yapf -dr src`** shows a diff for the entire source directory. **`yapf -ir src`** formats everything in-place (git commit first!).

An alias to auto-format only changed files as seen by Git may be helpful:

```
alias py-format-changed="git ls-files -m --others --exclude-standard | grep '\.py$' |␣
↪xargs yapf -i"
```

Style Tips

If you want each list value to stay on a separate line, use a trailing comma after the last element (here after `'created_at'`):

```
model_properties = [
    'abstract',
    'author',
    'ballot',
    'content',
    'created_at',
]
```

More About Code Style

- Recommended talk: Raymond Hettinger - Beyond PEP 8 – Best practices for beautiful intelligible code - PyCon 2015

4.3 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.

4.3.1 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.