adhocracy-3 Documentation

Release 0.0$_{prototype}$

...

June 14, 2017

Note: This isn't meant for general consumption at this stage. Many expected things do not work yet!

CHAPTER 1

What is adhocracy?

Adhocracy 3 aims to be a:

- python framework to build a REST-API *backend* for cms-like applications with a focus on *participation processes* and *collaborative text work*.
- javascript framework to build SinglePageApplication *frontends*.

It comes with these features out of the box:

- Generic REST-API based on the following concepts:
  - *Hypermedia REST API*: loose coupling frontend/backend, no fixed endpoints, (only half implemented, possible future:A3 Hypermedia REST-API)
  - *Supergraph*: Resources are versioned and build non hierachical data structures with other Resources, Versions never change, see Concept: The Supergraph and Concept: Simulating Patches with The Supergraph. (only half implemented, Problem: build a usable REST-API on top of this concept).
- Generic *API Specifiaction to build generic frontend* (see REST-API)
- Generic *Admin interface* (not implemented yet)
- *Resource and workflow modellings* for participation processes.
- *SinglePageApplication* frontends and backend customizations for *specific participation projects*

CHAPTER 2

Contents

Concepts

resource

Anything that exists in adhocracy is a resource: A proposal, a comment, but also users or individual rates.

resource type

Think of a resource type as a blueprint, and a resource as the actual building you build by following the blueprint. Example: The proposal "Better food in the cafeteria" would have the type `adhocracy_core.resources.proposal.IProposal`. Note that all resources of the same type have the same sheets, so there might be a lot of similar types with slightly different sheets (e.g. a simple proposal, a proposal with a budget, a proposal with a geographical location, ...).

sheet

Sheets are the features of resources. A proposal may for example have the sheet `adhocracy_core.sheets.title.ITitle` that allows it to have a title and the sheet `adhocracy_core.sheets.comment.ICommentable` that allows it to be commented on. A resource is really not much more than the sum of its sheets.

backend / frontend

The backend is the part of the software that stores the data. The frontend on the other hand is in charge of showing the data to users. Having a clear separation between these makes development simpler and theoretically allows to have more than one frontend, e.g. a website and a mobile app.

core / customization

Not all projects implemented with adhocracy are the same. That is why it is very easy to customize it for each individual project. The shared functionality is called "core" while the special code is called "customization".

process

A process resource represents a participation process. There are very different kinds of these. Idea Collections, where users can enter proposals and get feedback, and giving feedback on prepared documents, are probaly the most common ones.

permissions

What a user is permitted to do depends on their role. The roles a user has often depend on the context. Example: Amelia (user) may be the creator (role) of one proposal (context) and therefore permitted to edit it (permission). Proposals that she hasn't created, she may not edit, but she may comment on them.

workflow states

Participation processes typically have multiple phases: For example, you may want to first publish an announcement, then have the actual participation for some time, and display the results once that is over. This is possible by using workflows that can have different states.

Workflows can be used with processes, but also with any other kind of resource. An important feature of workflows is that you can change the permissions for each role based on the state.

Development

Installation

Installation

Requirements (Tested on DebianUbuntu, 64-Bit is mandatory):

1. git
2. python python-setuptools python-docutils
3. build-essential libssl-dev libbz2-dev libyaml-dev libncurses5-dev libxml2-dev libxslt-dev python3-dev
4. graphviz
5. ruby ruby-dev
6. gettext
7. libmagic1

If you don't use the custom compiled python (see below) you need some basic dependencies to build PIL (python image library):

8. libjpeg8-dev zlib1g-dev (http://pillow.readthedocs.org/en/latest/installation.html)

Create SSH key and upload to GitHub

```
ssh-keygen -t rsa -C "your_email@example.com"
```

Checkout source code

```
git clone git@github.com:liqd/adhocracy3.git
cd adhocracy3
git submodule update --init
```

Create virtualenv

```
pyvenv-3.5 .
```

If you don't have python 3.5 on your system, you may compile python 3.5 and Pillow instead of creating a virtualenv

```
cd python
python ./bootstrap.py
./bin/buildout
./bin/install-links
cd ..
```

Install adhocracy

```
./bin/pip install -r requirements.txt
./bin/python ./bootstrap.py
./bin/buildout -N
```

Update your shell environment:

```
source ./source_env
```

Run the application

Start supervisor (which manages the ZODB database, the Pyramid application and the Autobahn websocket server):

```
./bin/supervisord
./bin/supervisorctl start adhocracy:*
```

Check that everything is running smoothly:

```
./bin/supervisorctl status
```

Get information about the current workflow:

```
./bin/ad_set_workflow_state --info etc/development.ini <path-to-process>
# Example
./bin/ad_set_workflow_state --info etc/development.ini /mercator
```

Change the workflow state (most actions are not allowed for a normal user in the initial 'draft' state):

```
./bin/ad_set_workflow_state etc/development.ini <path-to-process> <states-to-transition>
# Example
./bin/ad_set_workflow_state etc/development.ini /mercator announce participate
```

Open the javascript front-end with your web browser:

```
xdg-open http://localhost:6551/
```

Shutdown everything nicely:

```
./bin/supervisorctl shutdown
```

Troubleshooting

If you encounter this error when starting adhocracy

> Problem connecting to WebSocket server: ConnectionRefusedError: [Errno 111] Connection refused

delete the *var/WS_SERVER.pid* file and retry again. This happens when the Websocket server is not shutdown properly.

Development Tasks

General Remarks

When changing the api, the frontend needs to re-generate the TypeScript modules providing the resource classes. This may trigger compiler errors that have to be resolved manually. For more details, see comment in the beginning of mkResources.ts.

Frontend wokflow

Run buildout once after switching project or the git branch. For changes to the TypeScript code it will now be sufficient to use *bin/tsc* or *bin/tsc –watch* (see tsconfig.json for settings used).

Running the Testsuite

frontend unit tests:

1. In node:

```
bin/polytester jsunit
```

2. In browser:

```
bin/supervisorctl start adhocracy:frontend
xdg-open http://localhost:6551/static/test.html
```

Note: For debugging, it helps to disable blanket.

Note: Running JS unit test in the browser with blanket enabled is currently broken.

protractor acceptance tests:

```
bin/polytester acceptance
```

Note: You need to have chrome/chromium installed in order to run the acceptance tests.

run backend functional tests:

```
bin/polytester pyfunc
```

run backend unit tests and show python test code coverage:

```
bin/polytester pyunit
xdg-open ./htmlcov/index.html
```

run all test:

```
bin/polytester
```

to display console output:

```
bin/polytester -v
```

modify test config:

> tests.ini (run all tests with polytester) pytest.ini (python/jasmin tests with pytest) etc/protractorConf (acceptantance tests with protractor)

delete database (works best on development systems without valuable data!):

```
rm -f ./var/Data.*
bin/supervisorctl restart adhocracy:*
```

If you are using the *supervisor group adhocracy_test:**, you don't have to delete anything. The database is in-memory and will die with the test_zodb service.

Generate html documentation

Recreate api documentation source files:

```
bin/ad_build_api_rst_files
```

Generate html documentation:

```
bin/ad_build_doc
```

Open html documentation:

```
xdg-open docs/build/html/index.html
```

Create scaffold for extension packages

1. Run the following commands:

   ```
   bin/pcreate -s adhocracy adhocracy_xx
   bin/pcreate -s adhocracy_frontend xx
   ```

   In the current repository layout, you then need to move the generated directories (`adhocracy_xx/` and `xx/`) to `src/`.

2. Add the new paths to `develop` and `eggs` in `base.cfg`.
3. Create `buildout-xx.cfg`
4. Add `src/adhocracy_xx` to `.coveragerc`
5. Add `src/xx/xx/build` to `.gitignore`

You may then want to run `bin/buildout -c buildout-xx.cfg` to check that everything works fine.

Update packages

python

Check whether new Python versions exist:

```
bin/requires.io update-site -t a54831113b039e9edbb2d26c2d2f9a9c99887437 -r adhocracy3
xdg-open https://requires.io/github/liqd/adhocracy3/requirements/?branch=master
```

You may then update the pinned Python versions in *versions.cfg* if appropriate.

ruby

```
bin/gem outdated  # binary may also be called bin/gem1.9.1 or bin/gem2.1
```

node.js

```
bin/npm --prefix node_modules --depth 0 outdated
```

bower

```
cd .../lib  # where bower installs the libraries
bower list
```

Release Adhocracy

Adhocracy uses semantic versions with one extra rule:

> Versions 0.0.* are considered alpha and do not have to follow the major-minor-patch rules of semantic versioning.

Git tag and *setup.py*-version must be the same string.

In order to create a new version, first make sure that:

1. you are on master. (this rule is motivated by the fact that rebasing tags is really nothing we want to have to deal with.)
2. the last commit contains everything you want to release and nothing else.
3. you have git-pushed everything to origin.

Then, to upgrade to version 0.0.3, carry out the following steps:

4. update *setup.py* to the new version (search for *name*=... and *version*=...). Commit this change.
5. *git tag -a 0.0.3 -m '...'*. The commit comment can be literally '...' if there is nothing special to say about this release, or something like e.g. *Presentation <customer> <date>*.
6. *git push –tags* (I think *git push* and *git fetch* treat tags and commits separately these days; for the convoluted details, consult the man pages).

Browse existing tags and check out a specific release:

```
git tag
git checkout 1.8.19
```

Apply a hotfix to an old release:

```
git checkout -b 1.8.19-hotfix-remote-root-exploit 1.8.19
...  # (edit)
git commit ...
git tag -a 1.8.20 -m 'Fix: remote-root exploit'
```

There is more to tags, such as deleting and signing. See *git tag –help*.

Update translations backend

create new language:

```
bin/ad_i18n en
```

extract message ids, update po and create mo files:

```
bin/ad_i18n
```

compile custom po file in extension package:

```
cd src/adhocracy_meinberlin/adhocracy_meinberlin/locale/en/LC_MESSAGES/
msgfmt --statistics -o adhocracy.mo adhocracy.po
```

#TODO helper script that updates/compiles all po files

Coding Style Guides

General coding style definitions.

Example Vim config according to coding guideline:

```
https://github.com/liqd/vim_config
```

Python

Test Driven Development

- 100% unit test coverage (must)
- use pytest fixtures to mock/create dependencies, functional tests have the *functional* marker, integration are using a fixture called *integration*.
- Test driven development with functional/integration and unit test (should)

  concept: http://en.wikipedia.org/wiki/Test-driven_development

  1. write function/integration test
  2. write unit test (simplest statement first)
  3. switch between writing code and change/extend tests until all test pass
  4. refactor

Refactor towards Clean Code

see (Refactoring & Clean Code)

Imports

- one import per line
- don't use * to import everything from a module
- don't use relative import paths

- dont catch `ImportError` to detect wheter a package is available or not, as it might hide circular import errors. Instead use `pkgresources.getdistribution` and catch `DistributionNotFound`. (http://do3.cc/blog/2010/08/20/do-not-catch-import-errors,-use-pkg_resources/)

Code formatting

- 4 spaces instead of tabs (must)
- no trailing white space (must)
- pep8 (must)
- pyflakes (must)
- pylint (should)
- mcabe (should)
- Advances String Formatting pep3101 (must)
- Single Quotes for strings except for docstrings (must)

Docstring formatting

- pep257 (must, bei tests und zope.Interface classes should)
- python 3 type annotation (must) according to https://pypi.python.org/pypi/sphinx_typesafe
- javadoc-style parameter descriptions, see http://sphinx-doc.org/domains.html#info-field-lists (should)
- example:

```
    def methodx(self, a: dict, flag=False) -> str:
        """Do something.

        :param a: description for a
        :param flag: description for flag
        :return: something special
        :raise ValueError: if a is invalid
        """
```

JavaScript

General considerations

- this document is split in multiple sections
  - general JavaScript
  - TypeScript
  - Angular
    - Angular templates
  - Adhocracy 3
  - Tests
- We prefer conventions set by 3rd party tools (e.g. tslint) over our own preferences.

- We try to be consistent with other guidelines from the adhocracy3 project

General JavaScript

We follow most rules proposed by tslint (see tslint config for details). However, there are some rules we want to adhere to that can not (yet) be checked with tslint.

- Use strict mode everywhere
  - There seem to be multiple issues with strict mode and TypeScript
    - http://typescript.codeplex.com/workitem/2003
    - http://typescript.codeplex.com/workitem/2176
- No implicit boolean conversions: `if (typeof x === "undefined")` instead of `if (!x)`
- Chaining is to be preferred.
  - If chain elements are many lines long, it is ok to avoid chaining. In this case, if chaining is used anyway, newlines and comments between chain elements are encouraged.
  - Layout: Each function (also the first one) starts a new line. The first line (without a `.`) is indented at n+0, all functions at n+1 (4 spaces deeper).

  Example:

```
adhHttp.get(url)
    .then(update)
    .then(exit);
```

- Each new identifier has its own `var`. (rationale: `git diff` / conflicts)

  Example:

```
// bad
var someVariable
    someOtherVariable;

// good
var someVariable;
var someOtherVariable;
```

- No whitespace immediately inside parentheses, brackets or braces (this includes empty blocks):

```
Yes: spam(ham[1], {eggs: 2})
No:  spam( ham[ 1 ], { eggs: 2 } )
```

- Do not align your code. Use the following indentation rules instead (single-line option is always allowed if reasonably short):
  - objects:

```
foo = {
    a: 1,
    boeifj: 2,
    cfhe: 3
}
```

  - lists:

```
foo = [
    138,
    281128
]
```

  - function definitions:

```
var foo(a : number) : number => a + 1;

var foo = (arg : number) : void => {
    return;
};

var foo = (
    arg : number,
    otherarg : Class
) : void => {
    return;
};
```

- The last item in a list or in function parameters may be split across multiple lines:

```
app.directive('myDirective', ["$q", "$http", ($q, $http) => {
    ...
}]);
```

- Do not use named functions. Assign anonymous functions to variables instead. This is less confusing. Further reading
- If you need an alias for `this`, always use `self` (as in knockout) or `_self` (in TypeScript classes). (`_this` is used by TypeScript in compiled code and is disallowed in typescript source in e.g. class instance methods.)

  If more than one nested self is needed, re-assign outer `self`s locally.

TypeScript

- imports at top
  - standard libs first (if such a thing ever exists), then external modules, then a3-internal modules.
  - only import from lower level. (FIXME: "lower level" does not mean file directory hierarchy, but something to be clarified. This rule is to be re-evaluated at some point.)
- imported adhocracy modules must be prefixed with "Adh".
- nested generic types are allowed up to 2 levels (`Foo<Bar<Baz>>`). Fewer is to be preferred where possible.
- Type functions, not the variables they are assigned to.
- Use `type[]` rather than `Array<type>`.
- A colon used for types must always be surrounded by single spaces:

```
// bad
var x: number;
var y:number;

// good
var x : number;
```

Lambdas TypeScript has its own lambda syntax. It has two differences from JavaScript's functions:

- The result of the final statement is returned automatically.
- `this` is the `this` from the enclosing scope.

Example:

```
var lambda = () => {
    var nested_fn = function() {
         return this;
    };
    var nested_lambda = () => this;
}

var fn = function() {
    var nested_fn = function() {
         return this;
    };
    var nested_lambda = () => this;
}
```

is compiled to:

```
var _this = this;
var lambda = function () {
    var nested_fn = function () {
        return this;
    };
    var nested_lambda = function () {
        return _this;
    };
};

var fn = function () {
    var _this = this;
    var nested_fn = function () {
        return this;
    };
    var nested_lambda = function () {
        return _this;
    };
};
```

These lambdas *should always be preferred* over functions because they avoid common mistakes like this:

```
class Greeter {
    greeting = "Hello";

    greet = function() {
        alert(this.greeting);
    };
}

var greeter = new Greeter();
setTimeout(greeter.greet, 1000);  // will alert 'undefined'
```

Still you should not use this behaviour extensively. Prefer to use the explicit aliases `_self` and `_class` in class methods:

```
class Greeter {
    public static greeting = "Hello";

    constructor(public name) {}

    greet = function() {
        var _self = this;
        var _class = (<any>_self).constructor;

        setTimeout(() => {
            console.log(_class.greeting + " " + _self.name + "!");
        }, 1000);
    }
}
```

Angular

- prefer isolated scope in directives and pass in variables explicitly.
- direct DOM manipulation/jQuery is only allowed inside directives.
- dependency injection
  - always use `["$q", function($q) {...}]` style
- do not use `$` in your variable names (leave it to angular).
- all directives, filters and services are prefixed with "adh".

> **Note:** In the future, this prefix may be split up in several ones, making refactoring necessary. Client-specific prefixes may be added without the need for refactoring.

- angular scopes should be typed with interfaces.

link vs. controller When writing directives, `link` and `controller` do mostly the same:

```
var linkDirective = (service) => {
    return {
        link: (scope, element) => {
            scope.foo = "bar";
            service.something();
        }
    };
};

var ctrlDirective = () => {
    return {
        controller: ["$scope", "$element", "service", ($scope, $element, service) => {
            $scope.foo = "bar";
            service.something();
        }]
    };
};
```

Note that dependency injection happens in different places in the two examples.

In general, `link` is to be preferred. There is one case where `controller` must be used. See Creating Directives that Communicate in the angular docs for an in depth discussion. In that case, the controller should not be defined inline but as a separate `class`:

```
class FooController {
    constructor(private depedency) {}

    public something() {
        ...
    }
}

var ctrlDirective = () => {
    return {
        controller: ["dependency", FooController]
    };
};

var subDirective = (service) => {
    return {
        require: "^ctrlDirective",
        link: (scope, element, attrs, ctrl) => {
            ctrl.something();
        }
    };
};
```

Template

- write polyglot HTML5.
  - prefix any angular-specific attributes with `data-`:

```
<span data-ng-bind="foo"></span>
```

  - Exception: The preferred way to use angular directives is the element syntax:

```
<adh-proposal data-path="/adhocracy/proposal/1"></adh-proposal>
```

    * This needs special care in IE8 and below. See https://docs.angularjs.org/guide/ie

- prefer `{{...}}` over `ngBind` (except for root template).

FIXME: when to apply which classes (should be in balance with CSS)

- apply classes w/o a specific need/by default?
- CSS and JavaScript are not allowed in templates. This includes ngStyle.
- Since templates (1) ideally are to be maintained by designers rather than software developers, and (2) are not type-checked by typescript, they must contain as little code as possible.

Documentation

- Use JSDoc-style comments in your code.
  - Currently, no tool seems to be available to include JSDoc comments in sphinx.
  - TypeScript has only limited JSDoc support

CSS

Preface

In recent years, methodologies like OOCSS, SMACSS, BEM, SUIT, Pattern driven Design, and Atomic Design have shifted the focus from designing pages to designing systems.

The best known application of this is probably bootstrap. But as one of bootstraps main developers said: You are encouraged to build your own bootstrap. This document describes the details of how we create our own design system for adhocracy3.

Common Terminology

To work together it is important to share a common language. Unfortunately, JavaScript programmers, CSS developers, and graphic designers sometimes have very different angles on the same things. The following terminology is therefore based on the tried and tested systems mentioned above.

Base Styling Base styling is the styling that applies to HTML elements when no additional classes are added. It sets the prevailing mood for a product. This involves general *text styling* as well as *links*, *headings*, and *input boxes*.

Layout The layout defines the position of elements on a page. It is typically based on a grid system.

Components There are many synonyms for this in the different methodologies: Object (OOCSS), module (Smacss), block (BEM), atom/molecule/organism (Atomic Design), or pattern (pattern driven design).

Components are independent of their context and can be reused throughout the UI. A typical component is a *button* or a *login dialog*. The rule of thumb is: If in doubt, it is a component.

Element An element is a part of a component that can not be used on its own. A typical example is a menu item (always part of a menu component).

States Components may have different *states* (e.g. *hover*, *active*, or *hidden*). States are always bound to specific components (e.g. there is no general *active* state).

Modifiers Components can have derived, modified versions. For example, there could be a button and a *call-to-action* button. In this case, call-to-action would be a modifier. (If you know about object oriented programming: this is similar to a subclass).

Modifiers are very similar to states because both modify a component. The rule of thumb to distinguish the two is that whereas the state of a component usually changes over time, modifiers don't.

Variables A variable can be used to define a value in a single place and then use it wherever we want. We could for example define the variable `primary-color` and use it throughout the UI. This allows us to change that color in a single place which makes theming easy.

Core and Themes The project may create multiple CSS-themes for the software. All themes share a common core. Themes can theoretically overwrite every aspect of the core. Since overwrites have maintenance cost, they should be kept at a minimum.

CSS and Design

This section describes the collaboration between designers and frontend developers. All the rules apply to core and any additional themes.

- UI designers ...
  - must mark any components, states, modifiers, and variables in wireframes.
  - may request new components, states, ... from the team.
    - They must decide whether the new component, state, ... should be part of core or theme.
    - They must provide semantically rich names for all new features. (e.g. "light-foreground" instead of "grey"; see Robert C Martin, Clean Code, Chapter 2)
    - Variables are mandatory for all colors and font sizes.
  - must provide the contents of a view in a linearized and thus prioritized sequence in addition to the layout structure. This is needed e.g. for screen readers (assistive technology for the blind) and web crawlers.
- Graphic designers ...
  - must provide values for all variables.
  - must provide designs for all components, states, ...
  - They must provide all necessary information and files as soon as possible (to avoid delays, preliminary dummy files may be provided). This includes:
    - fonts
    - icons
    - background images/logos
    - FIXME: define file formats, image resolution, ...
- CSS developers ...
  - must provide a living style guide (breakdown of all existing components, states, ...).
  - must report implementation issues as soon as possible.
  - must implement features as requested.

CSS, HTML, and JavaScript

This section describes the collaboration between CSS developers and JavaScript programmers.

- JavaScript does not set any CSS on elements. Instead it adds/removes states.
- Some CSS testing should be done in browser tests, i.e. CSS and JavaScript developers should work together on this.

Selectors This section describes which selectors must be used for different types. All classes are lowercase and hyphen-separated.

- component: class (no prefix)
- layout: class (prefix: `l-`)
- element: class (prefix: component name)

- state: pseudo-class, attribute, class (prefix: `is-` or `has-`)
- modifier: class (prefix: `m-`)

CSS Specifics

Preprocessor CSS preprocessors help a great deal in writing modular, maintainable CSS code by offering features like variables, imports, nesting, and mixins. Major contenders are Sass, Less and Stylus. We had good experiences with Sass so we will stick with it. CSS developers must read the Sass documentation.

Documentation and Style Guide A style guide in (web)design is an overview of all available colors, fonts, and components used in a product. In the context of CSS it can be generated from source code comments. In some way this is similar to doctests in python.

There is a long list of style guide generators. We chose to use hologram because it integrates well with our existing CSS tools.

Hologram is automatically installed when running buildout. You can use `bin/buildout install styleguide` to build the style guide to `docs/styleguide/`.

All variables, components, base styles, states, and modifiers must be documented (including HTML examples). Variables also need documentation and examples. As these do not expose selectors which could be used in examples it might be necessary to create `styleguide-*`-classes.

Common Terminology Considerations These are some CSS/SCSS specific thoughts on the common language terms defined above.

Modules A module is a SCSS file. Each component should have its own module including its states and modifiers. Several base styles may be included in a single module if they are closely related. The same goes for layout, variables, and mixins.

Variables

- Do not add variable definitions like `$color-default: blue !default;` to your modules because this may hide errors. Define all global variables in a central place instead.
- You should use local variables if you need to use the same value multiple times. Still in most cases it is possible to avoid these situations by grouping selectors or similar.

  Bad:

```
$padding: 2em;

.box1 {
    padding: $padding;
}
.box2 {
    padding: $padding;
}
```

  Worse:

```
.box1 {
    padding: 2em;
}
.box2 {
```

```
        padding: 2em;
    }
```

Good:

```
    .box1,
    .box2 {
        padding: 2em;
    }
```

States and Modifiers States and modifiers are always specific to a component. They have to be defined within the scope of the component.

Mixins There are two ways to implement mixins in Sass: `@mixin` and `@extend`. There are basically three differences:

- a `@mixin`, once defined, can be used everywhere. `@extend`s are are compiled into selector groups, which may not be possible depending on what you are trying to do.
- `@mixin` allows parameters and content blocks.
- `@extend` may produce more efficient (less redundant) CSS.

There is no rule about which one is preferred. As `@mixin` is simpler to use you might by tempted to use it exclusively. Always stop and also consider `@extend`.

Formatting We have a pre-commit hook with most of the sass-lint linters with their default settings, except for the following modifications:

- 4 space indentation.
- Include leading zero.
- Double quotes instead of single quotes.
- Comma-separated selectors need not be on their own lines. Still this is a must for composite selectors.
- A strict property sort order is not enforced. Still the properties should appear in roughly the following order:
  - `content` (only used on pseudo-selectors)
  - box – `display`, `float`, `position`, `left`, `top`,
  - `border` `height`, `width`, `margin`, `padding`
  - text – `font-family`, `font-size`, `line-height`, `text-transform`, `letter-spacing`, ...
  - color – `background`, `color`
  - other

The following additional rules apply:

- similar to pep8
  - only one property per line;
  - no trailing whitespace
  - two spaces between rule and comment, one after comment initialiser (good: `color:  white; // foo`; bad: `color:  white; //foo`)
  - prefer lines < 80 chars if possible

    - spaces around binary operators
- opening bracket at the end of the last selector line
- closing bracket in its own line
- avoid vendor specific prefixes/hacks in your code. You may however use mixins that create compatible code for exactly one thing (e.g. `border-radius` mixin by compass)

Units This gives an order of preference for the units that must be used with different types of values starting from preferred.

- length:
    - layout: `%`
    - distances relative to element font-size `em`
    - else: `rem`
    - for thin lines or in the context of images, `px` may be used to avoid low-quality rescaling
- font-size: variable, `rem`, `%`
- 0 length: no unit
- line-height: no unit, `em`, `rem`
    - see explanation by Eric Meyer.
- color: keyword, short hex, long hex, `rgba`, `hsla`
- generally prefer variables to keywords to numeric values
    - keywords are easier to apprehend when skipping through the code

---

Note: For all `rem` units the `rem()` mixin should be used, e.g.:

```
@include rem(margin, 10px 5px);
@include rem(margin-bottom, 2rem);
@include rem(border, 3px solid $color-function-positive);
```

This automatically calculates `rem` units with a `px` fallback for older browsers.

Accessibility

- Be careful about hiding things (`hidden` vs. `visually-hidden`) (see http://a11yproject.com/posts/how-to-hide-content/).
- Use fluid and responsive design (relative units like `%`, `em`, and `rem`).
- Prefer to define foreground and background colors in the same spot. Use color-contrast by sass-planifolia.
- While no support for IE < 9 is planned, do not introduce incapabilities where not needed (robust).

Icons You should avoid using pixel images as they are inflexible in size. If possible, prefer iconfonts. You can use Font Custom to easily generate an icon font from SVG files.

Context One of the most complicated issues in CSS in general is whether styles should change depending on context. On the one hand we talk about *responsive design*, on the other, components should be decoupled (Law of Demeter) to keep the code maintainable.

It is important to understand that there are two different kinds of context awareness involved here:

1. Elements inherit CSS rules from their context (e.g. `font-family` is shared across the whole document if set on the `html` element).
2. CSS code can apply additional styling to an element if it appears in a specific context (e.g. `#sidebar h2 {color: red;}`).

Inheritance is hard to avoid and does little damage. So we should embrace it.

I am not so sure about child selectors: OOCSS and SMACSS both recommend to avoid them. Still it is a powerful feature. This is still an open question.

Restructured Text (RST)

- 4 spaces instead of tabs (must)
- no trailing white space (must)
- Headline hierarchy: ===== — +++++ ~~~~~~~ ** (must)

Refactoring & Clean Code

Introduction

After changing code (or tests) it must be better than before to ensure *Extensibility* and *Maintainability* in the long run.

You should:

- Extract ClassMethodFunction
- MoveReuseRenameRemove
- Replace condition with polymorphism/Strategy

with the *clean code* principals in mind.

Clean Code summary

We follow mostly the rules from the chapter "Smells and hints" from the book "Clean Code" by Robert C. Martin, 2008. It should be read by every new developer. For a short summary you can read this document or have a look at the cheat sheet: http://www.planetgeek.ch/wp-content/uploads/2014/11/Clean-Code-V2.4.pdf.

Common Principles:

- Single Responsibility Principle (SRP): only one reason to change behavior/code
- Open Closed Principle (OCP): open for extensions (new classes,..), closed for modification
- Don't Repeat Yourself (DRY)
- Separation of Concern
- follow/create standards for naming, code structure and styles

- You ain't gonna need it (YAGNI)

Readability:

- easy to understand and extend by others
- readable code instead of comments
- less code
- good placed and clear responsibility (place code where the reader expects it)

Variable naming:

- explicit, show intention and maybe context information
- no misleading names
- distinction between concepts (get, append, add,..)

Additional guidelines:

- **do not** translate names and terms from the problem domain; **do** translate everything else
- **do** use singular
- **do** convert umlauts to ae, ue, ...

Variables:

- define close to where there are uses

Function name/arguments:

- verb with nouns (explain abstraction level and if possible arguments)
- name shouldn't be too long; using expressive named (keyword) arguments might help making an overly long name unnecessary
- prefer 0 or 1 arguments, at most 3
- Use kwargs for optional arguments, never use them like positional arguments (omitting the name)

Functions:

- short
- max 2 indention level
- do only "one thing"
  - one level of abstraction (can you divide it into sections? or extract a helper function with different name?)
  - one down story of to paragraphs (TO X do a, TO X do b,.. X == function name)
  - just one return statement (or several ones, but close together)
  - no switch statement (or at most one for each functionality/class, if unavoidable)

  - Command Query Separation (no side effects, use descriptive naming):
    - change state object
    - query information
    - change/return argument
    - create object
  - separate error handling (easier to read and to extend)
- prefer not to change the argument objects, never mutate default kwargs

Class names:

- show responsibility
- explicit distinction of generic "concepts", if needed domain specific concepts
- no context information (for example domain specific suffix ("Adhocracy") or type ("String")

Classes:

- SRP, OCP
- high cohesion: all methods should share the class variables, if not split class
- small
- private functions below first public function that depends on it

Objects:

- data structures: direct access to variables
- objects: hide data structure, present public "behavior" methods for this objects
- procedural:
  - easy to add new functions
  - difficult to add new data structures (every functions need to check datatype, maybe Open/Close Principle violated
- OO with polymorphic methods:
  - easy to add new data structures
  - difficult to add new functions (need to extend all subclasses/implementer)
- Law of Demeter:
  - own variables / methods: ok
  - foreign data structures: ok
  - foreign object: use only public methods
  - no train wrecks: call().call()

Exceptions:

- do not return/accept None without need or accept wrong arguments (exeption: ease unit tests) (makes it hard to find/debug errors)
- do not use Exception to handle special cases (use Wrapper Classes or throw exception)
- exception class should make it easy for the caller to handle exception, give contect information, hide third party errors

Third party code:

- make Facade to access, catch errors
- Learning Test to play around and test new versions

Unit Tests:

- first draft +> test success +> refactor code and tests
- first test with simplest statement +> code +> more tests +> code,.. (only what is needed to pass test)
- clean code, Domain Specific Test+API
- structure: Given When Then
- assert one thing

System:

- Separation of concern
- Split Creation (factories, start application) , Running (assume every thing is alread created)

Contribute Code with Git

make usefull commits

Git commits serve different purposes:

- Allow reviewers to quickly go through your changes.
- Help developers in the future to understand the intention of your change using `git blame`.
- Use the break of writing a commit message as an opportunity to reflect on what you have just coded.
- Backup what you did and allow reverting to an earlier state, if necessary.

These goals may be in conflict with other goals (such as “have more time for writing tests and code”), and sometimes even with each other (“small commits” vs. “test suite always works”). Therefore, this section does not contain any strict rules, but suggestions. The reader is encouraged to decide in which contexts they make sense. (In particular, “should” is not the RFC-all-caps “SHOULD”, but something to consider.)

Suggestions:

- Aim at making small commits containing only one semantic change.

  In order to do that, you may want to use helper tools such as tig, git-cola or plain `git add --interactive` or `git add --patch`, allowing for easy line-by-line staging. Interactive rebasing (`git rebase -i`) may help with cleaning up history in retrospect, i.e. splitting / combining / reordering commits. Be aware of not pushing published non-volatile branches (as described in Code Review Process).

- The test suite should run through successfully on every commit. Test coverage doesn't necessarily need to be 100% on each commit, as some developers may want to split commits in functional code and testing code and write the latter later. Of course writing the tests first is preferred.

- For the actual commit message, we follow the rules, which are codified as an example by tpope:

  > Capitalized, short (50 chars or less) summary
  >
  > More detailed explanatory text, if necessary. Wrap it to about 72 characters or so. In some contexts, the first line is treated as the subject of an email and the rest of the text as the body. The blank line separating the summary from the body is critical (unless you omit the body entirely); tools like rebase can get confused if you run the two together.
  >
  > Write your commit message in the imperative: "Fix bug" and not "Fixed bug" or "Fixes bug." This convention matches up with commit messages generated by commands like git merge and git revert.
  >
  > Further paragraphs come after blank lines.
  >
  > 1. Bullet points are okay, too
  > 2. Typically a hyphen or asterisk is used for the bullet, preceded by a single space, with blank lines in between, but conventions vary here
  > 3. Use a hanging indent

- Referring to other commits can be done by using their hash ID. Be aware that the hash ID changes on rebase.

Descriptive summary prefix keywords are encouraged, but there is no strict rule as to which keywords exist and where to use them. Here is a list of options:

- Refactor (optionaly followed by a commit hash)
- Fixup (optionally followed by a commit hash to squash this one into; defaults to previous commit):

```
Fixup a93bhd34: typo
```

- Gardening (for changes that do not significantly change the meaning or structure of the code, such as style guide fixes)

Note that there's already standard messages for commits created by git (Revert "...") and conventions for review commits (`[R] prefix`) as described in the Code Review Process.

Add feature branch

Terminology

If branch A is branched from branch B, then B is called A's *base branch*.

A branch is called *published* if it has been pushed to a repository that is accessed by more than one user. Usually, this means the project-specific central upstream repository, but a branch is also considered published if one developer has pushed changes to another developer's laptop.)

Branch types

Branch naming follows a pattern that makes it easier to process branch lists automatically. The pattern consists of year (`YYYY`), month (`MM`), a developer name shortcut (`DEV`), keywords (small letter words), and descriptive free text (`[-a-z]+`).

The following branch types exist:

Story branches (`YYYY-MM-story-[-a-z]+`) For each user story, there is a story branch that must be based on `master`. Story branches may sprout personalized or volatile branches (see below).

Fix branches (`YYYY-MM-fix-[-a-z]+`) For each bug on the story board, a fix branch is created. It must be based on `master`.

Personalized branches (`YYYY-MM-DEV-[-a-z]+`) Developers create personalized branches in order to work on tasks. Personalized branches may be based anywhere. It is **not** allowed to `push --force` a personalized branch.

Volatile branches (`YYYY-MM-_DEV-[-a-z]+`) Personalized branches with `push --force` option. The developer must announce that this branch may change arbitrarily by adding an underscore mark before the developer name shortcut in the branch name. Volatile branches may be based anywhere.

Finding branch points

For the processes defined in this document, it is interesting to find the points in the repository where a branch branched off other branches in the past. We call these points *branch points*.

Note that the information at which point a branch branched off its direct base branch is not maintained by git. This does not make the question of the direct base branch any less meaningful, but it makes it tricky to answer.

If the base branch is `master`, then you can get a reference to the branch point of the current branch like this:

```
export BRANCHPOINT=`git rev-list HEAD ^master --topo-order | tail -n 1`~1
git show $BRANCHPOINT
```

(`git show-branch` yields more relevant data, but in a less machine-readable form.)

Rebase and +n-branch logic

To keep the code history clean, a personalized branch may be rebased before it is merged into its base. (Volatile branches may always be rebased, because there is no guarantee that they behave in any way as branches should.)

Rebasing has two advantages:

- You can move your branch to the HEAD of the base branch as an alternative to merging. This way you keep a near-linear commit history;
- with the `-i` option, rebasing allows to re-order and clean up individual commits, and thus make the life of the reviewer (and anyone else looking at the history) easier.

In order to avoid that `rebase` changes repository state destructively (instead of just adding additional commits), the rebase must happen according to *+n-branch logic*:

```
# (complete work on branch, say, 2014-05-mf-bleep based on, say, master)
# (make sure that upstream is set to origin/2014-05-mf-bleep)
git push -v
git checkout -b 2014-05-mf-bleep+1
git rebase master
git push -v origin 2014-05-mf-bleep+1
```

Remarks:

- the un-rebased branch has no +n suffix, the first rebase has '+1', the second '+2' and so on.
- if you call rebase with argument `-i`, you can do a lot of rebase magic (squashing and dropping and reordering and all that). This feature is quite self-explanatory – just try it! [FIXME: there was an oddity when you are in the editor and want to cancel. @nidi, can you fill that in here? i think you've explained this to me once.]
- if you call `git rebase -i $BRANCHPOINT`, you can do rebase magic without actually changing the branch point.

Dos and Don'ts

1. `push --force` is forbidden. The only exception are volatile branches.
2. `rebase` is generally forbidden on published branches. Exceptions: `rebase` is allowed in volatile branches; `rebase` with +n-branch logic is allowed in personalized branches and allowed-but-discouraged in story branches.
3. Always use `git merge` with `--no-ff` when merging a branch into its base branch.

   (When merging the base branch into a story or personalized branch to benefit from code recently added elsewhere, fast-forward is usually not possible since the histories of two merged branches have diverged. `--no-ff` usually does not apply in this case.)

   If you want to make `--no-ff` the default (you can still explicitly enable it with `--ff`):

```
git config --global merge.ff true
```

4. Merging ancestor branches into a current branch is ok. This makes it feasible to keep up to date with changes in a base branch in long-living story or personalized branches. The merge commit will be eliminated if the current branch is rebased on the ancestor branch HEAD at any point in time after the merge.
5. Fixes to trivial issues may be committed by a developer directly to master without branching. The commit must be at least mentioned to one more developer, who must check whether the issue qualifies as trivial and the commit is sound.

Merge feature branch to master branch

Before merging your feature branch please check:

- code follows general *coding style* guidelines and specific development hints in *backend*, *frontend*.
- all test pass and code coverage backend stays at 100%
- travis test builds should pass (github hook)
- new features are documented
- style checks pass (or let the git after commit hook do its work):

  bin/ad_check_code -a src/adhocracy src/adhocracy_sample

- code reviewed by other developer

Administration

Several console scripts are provided to facilitate the administration of Adhocracy 3.

Importing Resources

User, Groups and normal Resources can be imported with the *ad_fixtures* script. You choose between fixtures registered in adhocracy python packages:

```
./bin/ad_fixtures etc/development.ini -c adhocracy_core:test_fixtures
```

Or from an absolute file system path:

```
./bin/ad_fixtures etc/development.ini -c /home/user/adhocracy_core/test_fixtures
```

The *-h* flag can be used to see a full description of the options:

```
Traceback (most recent call last):
  File "/home/docs/checkouts/readthedocs.org/user_builds/adhocracy3/envs/latest/lib/python3.5/site-pa
    ws.require(__requires__)
  File "/home/docs/checkouts/readthedocs.org/user_builds/adhocracy3/envs/latest/lib/python3.5/site-pa
    needed = self.resolve(parse_requirements(requirements))
  File "/home/docs/checkouts/readthedocs.org/user_builds/adhocracy3/envs/latest/lib/python3.5/site-pa
    raise VersionConflict(dist, req).with_context(dependent_req)
pkg_resources.ContextualVersionConflict: (pyramid 1.5.2 (/home/docs/checkouts/readthedocs.org/user_bu

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "/home/docs/checkouts/readthedocs.org/user_builds/adhocracy3/envs/latest/bin/ad_fixtures", lin
    from pkg_resources import load_entry_point
  File "/home/docs/checkouts/readthedocs.org/user_builds/adhocracy3/envs/latest/lib/python3.5/site-pa
    @_call_aside
  File "/home/docs/checkouts/readthedocs.org/user_builds/adhocracy3/envs/latest/lib/python3.5/site-pa
    f(*args, **kwargs)
  File "/home/docs/checkouts/readthedocs.org/user_builds/adhocracy3/envs/latest/lib/python3.5/site-pa
    working_set = WorkingSet._build_master()
  File "/home/docs/checkouts/readthedocs.org/user_builds/adhocracy3/envs/latest/lib/python3.5/site-pa
    return cls._build_from_requirements(__requires__)
  File "/home/docs/checkouts/readthedocs.org/user_builds/adhocracy3/envs/latest/lib/python3.5/site-pa
    dists = ws.resolve(reqs, Environment())
  File "/home/docs/checkouts/readthedocs.org/user_builds/adhocracy3/envs/latest/lib/python3.5/site-pa
    raise VersionConflict(dist, req).with_context(dependent_req)
pkg_resources.ContextualVersionConflict: (pyramid 1.5.2 (/home/docs/checkouts/readthedocs.org/user_bu
```

Import Badges

Badges can be imported with the *ad_import_resources* script:

```
./bin/ad_import_resources etc/development.ini src/adhocracy_core/adhocracy_core/resources/user_badges
```

Badges can be assigned to resources with the *ad_assign_badges* script:

```
./bin/ad_assign_badges etc/development.ini ./src/adhocracy_core/adhocracy_core/scripts/user_badge_ass
```

Set Workflow state

The state of the workflow can be changed with the *set_workflow_state* command. The *-h* flag can be used to see a full description of the options:

Command 'set_workflow_state -h' failed: [Errno 2] No such file or directory: 'set_workflow_state'

Show Auditlog

The entries of the auditlog can be shown with the *ad_auditlog* command. The *-h* flag can be used to see a full description of the options:

Command 'set_workflow_state -h' failed: [Errno 2] No such file or directory: 'set_workflow_state'

Backend

Overview

The adhocracy backend is a python framework to build a REST-API *backend* for cms-like applications.

It was developed with the use cases and limitations of the policy drafing and decision making tool adhocracy2 and the Concept: The Supergraph concept in mind. The main focus lies on being *extensible*, allow modelling complex *participation workflows* and *graph data structures* , and ensure *privacy* and *data integrity*.

Note:

```
The implementation is largely based on `substanced D`_, but has a lot
of customization. One possible refactoring would be to make it a
nice behaving REST-API extension.
```

You can work with the following concepts.

Resource Handling

resource tree to ease working with hierarchical data

URL Routing supports both, url dispatch (fixed endpoints) and resource hierarchy traversal .

Fine grained security system:

- permission protect operations (like 'view' a resource, sheet or field)
- granted to *role* s, which in turn are granted to *principal* s.
- local permission and roles: grants can be modified for every resource and its ancestors in the resource tree .

Workflows:

- Finite State machine for resources
- change local permissions or run scripts on phase transition

Resources:

- composed by a set of sheets (see Concept: Modelling a Simple Use-Case with The Supergraph)
- runtime adding/removing of sheets possible
- open/close principal for resource modelling

Sheets:

- interface for a specific resource behaviour:
  - api methods
  - **data structure (`colander.Schema`) with following field types**
    - data

        * metadata
        * computed data
        * references
        * backreferences
    - encapsulate data/reference storage

TODO:: Here it would be great to have a small overview of what sheets do and how they work. Maybe give a concrete example of how they are used in combination with Colander for the JSON serialization and how they are used by the object factory. Also explain the link between resources and sheets and how they reference each other could be explain (with a diagram?).

Versioned Resources:

- lineare history or allow forking and merging (not implemented)
- data fields do never change

References:

- allow complex non hierachical data structures
- references (unidirectional) and backreferences (computed) between resource sheets

Reference Update policies if referenced Resource has new version:

- No Update
- Auto Update (new Version is created / reference is updated)
- Optional Update, User has to comfirm (examle "like reference") (not implemented)

Data Storage

Auditing:

- every data/reference change is logged
- no lost data for versioned Resources

Optimistic Concurency Control, atomic requests

- no manual data lock or transaction handling needed

Object database for persistence storage and search

- no sync problems, easy to debug

alternative storages for sheet data/references/search indexes (not implemented)

- support databases with more sophisticaed reference graph/search features

import/export scripts

Code

Type hinting

- play nice with code autocompletion (and static type checks).

Extensible:

- Pyramid extensibility

- Resource/Sheet concept, type definitions are easy to customize in extension packages

Softwarestack

- Python 3 (programming language)
- Pyramid (web framework)
- substance D (application server)
- hypatia (search)
- ZODB (object database)
- colander schema (data structures and validation)
- AutobahnlPython (websocket server)
- Varnish (http proxy cache server)
- buildout (build system)

History

We started 2012 with the plan to port adhocracy2 to pyramid. This become a long discussion how to build a framework for particpation processes based on fancy graph data structures Legacy concepts. Mid 2013 we started serious efforts to start developing. We compared multiple framework - database combinations to find the right technical base that allows to start quickly but does not stand in they way if the project grows. Doing this we had the following in mind:

- python 3 support
- active community and good documentation
- good extensibility -> *zope component* style like architecture
- fast references to resources and complex reference queries
- one system to search & store python objects and references
- ACID transactions
- resource tree, url traversal
- workflows, local permissions

We did two prototypes to play with the neo4j graph database and dropped it mainly due to cutting edge python support and transaction features. So we came to the ZODB database. It is stable and can do python object references in a very simple way because it is an object database. That lead to using pyramid and substanced as small framework that matched many of our requirements.

Evaluated framework - database combinations

(restored version from mid 2013, original evaluation report is lost)

General

feature:	python 3	active developed	documentation	full-stack	simple code	admin interface	high level api	REST-API	extensibility (every behaviour can be extendend/replaces without changing the core code)
cubicweb - sql	-	+	+	++	-	+	++	++	+
django - sql	-	+	+	++	-	+	++	+	+-
pyramid/kotti - sql	+	+	+	++	+	+	+	-	+
pyramid/bulbflow - neo4j	+	-	+	-	+	-	+	-	-
pyramid - rexster/neo4j	+	+-	+	-	-	-	+	+	+-
pyramid - ZODB	+	+	-	+	-	–	-	-	+
pyramid/substanced - ZODB	+	+-	+	++	+	++	+	-	+
Zope2/Plone - ZODB	–	+-	-	++	–	+	++	–	++
ZTK/Grok - ZODB	+-	-	+	++	–	+-	+	–	++

References

feature:	complex queries/graph traversal	fast	scalibility
cubicweb/sql	++	+-	++
django - sql	-	+-	+
pyramid/kotti - sql	-	+-	+
pyramid/bulbflow - neo4j	++	-	++
pyramid - rexter/neo4j	++	–	++
pyramid - ZODB	-	++	++
pyramid/substanced - ZODB	-	+	++
Zope2/Plone - ZODB	+	+-	+-
ZTK/Grok - ZODB	-	++	++

Resources

feature:	“sheets”	resource tree	local permissions	traversal	workflow	ACID transactions
cubicweb/sql	+	-	-	-	-	+
django - sql	-	-	+-	-	?	+
pyramid/kotti - sql	-	+	+	+	+	+
pyramid/bulbflow - neo4j	-	-	-	-	-	+
pyramid - rexter/neo4j	-	-	-	-	-	-
pyramid - ZODB	-	+	+	+	-	+
pyramid/substanced - ZODB	+	+	+	+	+	+
Zope2/Plone - ZODB	+	+	+	+	++	+
ZTK/Grok - ZODB	+	+	+	+	+	+

Search

feature:	checks permissions	full text index	attribute index	same transaction like resources
cubicweb/sql	-	++	+	-
django - sql	-	++	+	-
pyramid/kotti - sql	+	++	+	-
pyramid/bulbflow - neo4j	-	-	+	-
pyramid - rexter/neo4j	-	-	+	-
pyramid - ZODB	-	-	-	-
pyramid/substanced - ZODB	+	+	+	+
Zope2/Plone - ZODB	+	+	++	+
ZTK/Grok - ZODB	+	+	+	+

Notes

cubicweb	SemanticWeb web framework
django	full stack web framework
pyramid	micro web framework, internally based on zope components
pyramid / kotti	small cms project
pyramid / bulbflow - neo4j	Resource Modelling for graph database neo4j
pyramid - rexter / neo4j	REST-API for graph database neo4j
pyramid / substanced	small application server project
Zope2 / Plone	Big cms project/full stack framework based on zope components, permission checks enforced in application code
ZTK (ZopeToolkit) / Grok	full stack framework based on zope components, not active anymore, permission checks enforced in application code

Other evaluated frameworks without ZODB: pyramid - cubicweb database, pyramid - rdflib, pyramid/repoze.workflow/plone.behavior - neo4j

Others with ZODB: w20e.pycms, Karl Project, pyramid/repoze.workflow/plone.behavior, Zope2/repoze.workflow/plone.dexterity

More recent frameworks not considered

If we start a rewrite we would focus on full-stack frameworks for REST-APIs, standards, and simplified requirements. The following more recent projects are look promising.

- http://ramses.tech/
  - full stack solution for REST-APIs
  - easy prototyping/api specification
  - good ElasticSearch "frontend" to handle all kind of requests
- http://morepath.readthedocs.org/
  - flexible micro framework for REST-API/HTML rendering
  - combine/extend small application (like processXY, document management, user management, ...)
- django rest framework v3 / (or json-api extension) http://www.django-rest-framework.org/

  - full stack solution for REST-APIs
- json-api https://py-jsonapi.readthedocs.org/en/latest/
  - full stack solution for REST-APIs
- http://pythonhosted.org/jsondata/
  - data structure and patches based on JSON-Schema

Architecture

Software packages

The backend and frontend is released with the following python packages:

adhocracy_core framework and generic rest api, admin frontend

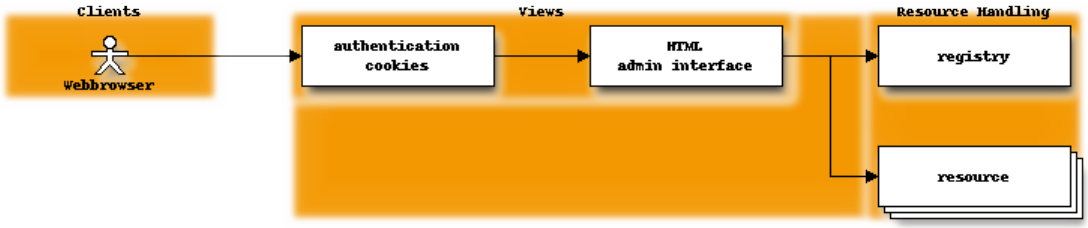
adhocracy_sample examples how to customize resource/sheet types

adhocracy_frontend framework for the javascript frontend

adhocracy_xyz Backend extensions for project specific application

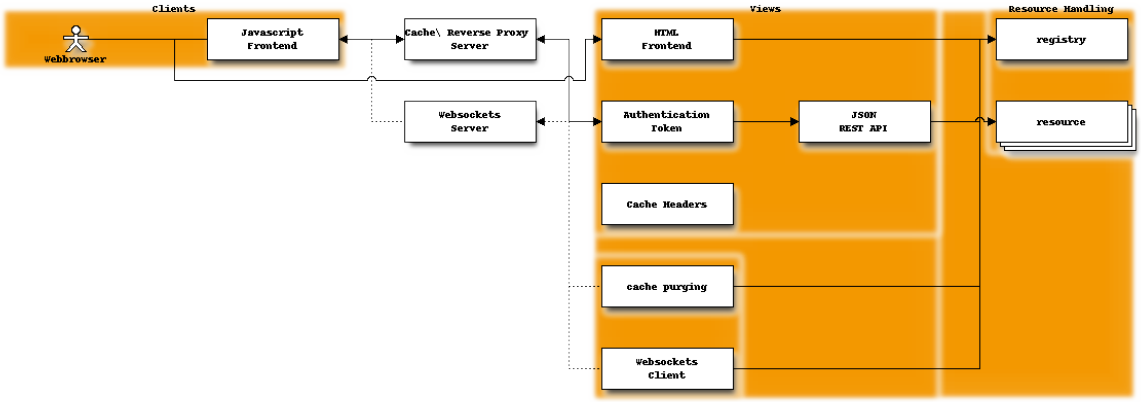
xyz projects specific application with javascript frontend

Frontend Technical Admin Interface (substanced)



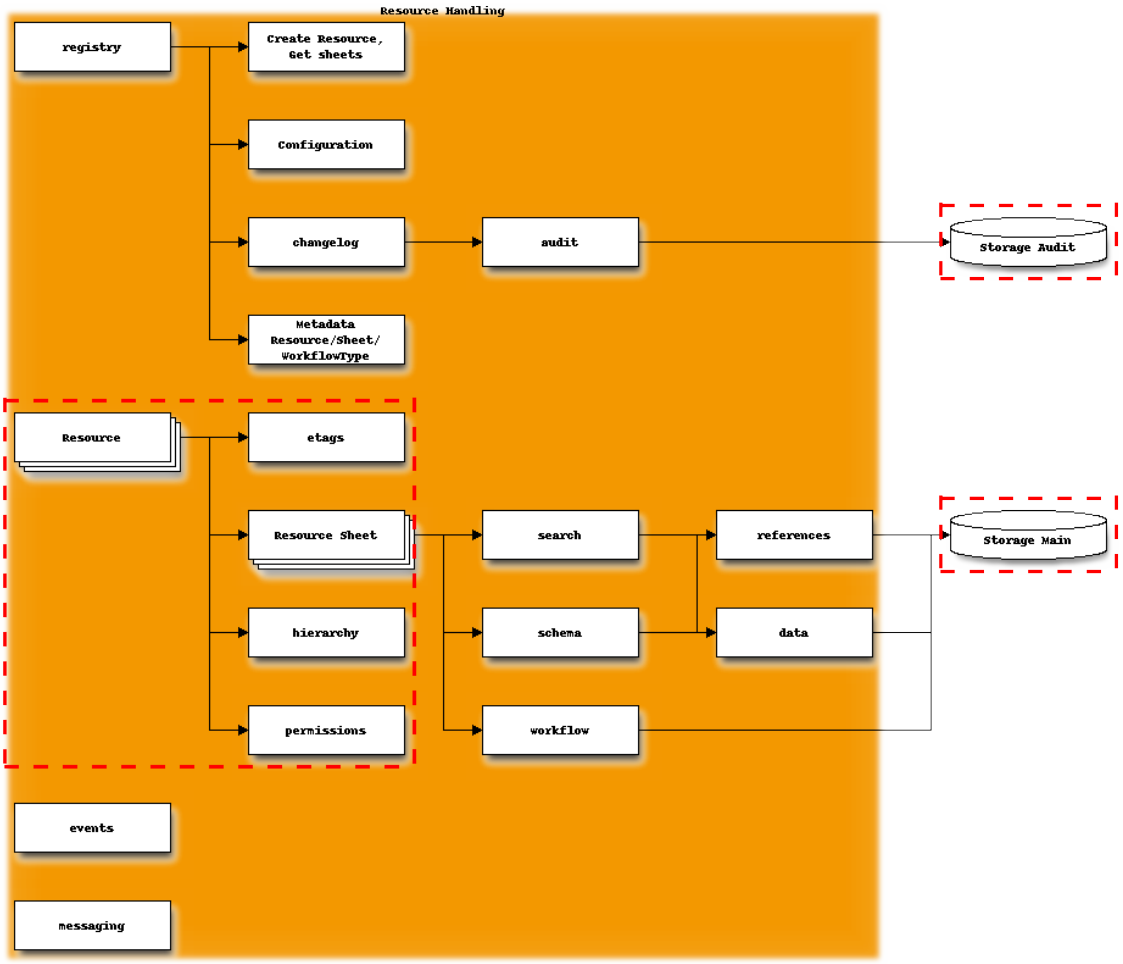
HTML Admin interface	Technical admin interface (serverside rendering)
Authentication Cooky	Authentication based on cookies and session id.

Frontend Javascript (Single Page Application)



Javscript Frontend	Single Page Application (client side rendering)
Cache Proxy	Proxy to cache http requests (varnish)
Cache Headers	Set http caching headers, compute etag)
Cache Purging	Send purge request to Cache server when resources are updated
Authentication Token	Authentication based on request header token.
REST API	JSON representation of resources to Create/Read/Update/Delete.
HTML Frontend	HTML representation of resources (only root, serves javascript/settings/routings)
Websockets client	Send notification mesages to the websockets server when resources are updated

Backend Resource Handling



The red line groups responsibility for persistence data storage (Note: all perstence data access should be done with the sheets). For further explenations see Modules.

Modules

API and separation of responsibility

responsibility (means reason to change code if functionality changes) should lay at one single point of code (packages/modules in this case), see also Refactoring & Clean Code.

layer loosly group of modules that follow these rules: * must not import from upper layer * should not import from same layer * may import interfaces from all layers * may import from lower layer

Application Layer

`adhocracy_core`	Configure, add dependency packages/modules, start application.

Frontend Views, Client Communication Layer

adhocracy_core.rest	Configure rest api packages.
adhocracy_core.rest.views	GET/POST/PUT requests processing.
adhocracy_core.rest.batchview	POST batch requests processing.
adhocracy_core.rest.schemas	Data structures / validation specific to rest api requests.
adhocracy_core.rest.subscriber	Subscriber to modify the http response object.
adhocracy_core.rest.exceptions	HTTP Exception (500, 310, 404,..) processing.
adhocracy_core.caching	Adapter and helper functions to set the http response caching headers.
adhocracy_core.authentication	Authentication with support for token http headers.
adhocracy_core.websockets	Asynchronous client-server communication via Websockets.

Registry, Factories, Access to Metadata Layer

adhocracy_core.content	Create resources, get sheets/metadata, permission checks.
adhocracy_core.changelog	Transaction changelog for resources.

Resource Handling Layer

adhocracy_core.resources	Resource types mapped to sheets (OpenClosePrinciple), object hierarchy.
adhocracy_core.resources.base	Resource base implementation with zodb persistence.
adhocracy_core.resources.simple	Basic simple type without children and non versionable.
adhocracy_core.resources.pool	Basic type with children typically to create process structures.
adhocracy_core.resources.item	Basic Pool for specific Itemversions typically to create process content.
adhocracy_core.resources.itemversion	Basic versionable type typically for process content.
adhocracy_core.resources.root	Root type to create initial object hierarchy and set global Permissions.
adhocracy_core.resources.principal	Principal types (user/group) and helpers to search/get user information.
adhocracy_core.resources.subscriber	Autoupdate resources.
adhocracy_core.sheets	Data structures/validation, set/get for an isolated set of resource data.
adhocracy_core.catalog	Configure search catalogs.
adhocracy_core.catalog.adhocracy	Adhocracy catalog and index views.
adhocracy_core.catalog.subscriber	Reindex subscribers.
adhocracy_core.authorization	Authorization with roles/local roles mapped to adhocracy principals.
adhocracy_core.messaging	Send messages to Principals.
adhocracy_core.graph	Set/Get Resource References / versions graph (DAG) helpers.
adhocracy_core.workflows	Finite state machines for resources.

Interfaces, Utils Layer

adhocracy_core.interfaces	Interfaces for plugable dependencies, basic metadata structures.
adhocracy_core.utils	Helper functions shared between modules.
adhocracy_core.events	Hooks to modify runtime behavior (use 'subscriber.py' in you package).
adhocracy_core.schema	Basic data structures and validation.
adhocracy_core.exceptions	Internal Exceptions.

Other stuff

adhocracy_core.scaffolds	Create scaffolds to customize or extend adhocracy.
adhocracy_core.scripts	scripts.
adhocracy_core.stats	Send runtime statistics to *statsd <http://statsd.readthedocs.org>*.
adhocracy_core.auditing	Log which user modifies resources in additional 'audit' database.
adhocracy_core.evolution	Scripts to migrate legacy objects in existing databases.
adhocracy_core.registry	
adhocracy_core.renderers	Additional pyramid renderers.
adhocracy_core.templates	

TODO: move all scripts to adhocracy_core.scripts

Substanced dependencies

- substanced.evolution (migration, see adhocracy_core.evolution)
- substanced.catalog (search, extended by adhocracy_core.catalog)
- substanced.workflow (state machines mapped to resource types, extended by adhocracy_core.workflows)
- substanced.content (provide content types factories, extendend by adhocracy_core.content)
- substanced.objectmap (reference resources, extented by adhocracy_core.graph)
- substanced.folder (Persistent implemention for adhocracy_core.interfaces.IPool resources)

Extend/Customize

- must follow Rules for extensible pyramid apps: configuration, configuration extentensions, view/asset overriding, event subscribers. Use imperative-configuration, except for views configuration-declaration.
- may use the underlaying zope component architecture provided by the application registry directly. may not use the global *zope component* registry, see also ZCA in pyramid.
- must follow rules for module *layer* (see above)
- make code dependencies pluggable to allow different implementations (other authentication, references storage, data storage, search, ..) Dependencies should have an interface to describe public methods.
- override resource/sheet metadata, see adhocracy_sample

Note: You can use the script *bin/ad_check_forbidden_imports* to list suspicious imports

Naming conventions

- Non-versionable resources types are named resource.x.IX with a sheet named sheet.x.IX.
- Versionable resources types are named resource.x.IXVersion (inherits from IITemVersion) with a sheet named sheet.x.IX. They belong to the container (parent) resource type called resource.x.IX (inherits from IItem).
- Resource/sheet types to express RDF like statements are named after the *verb*, for example: IRate.

API

Embedding

Adhocracy 3 is designed to be "embed first". This means that it will usually not be used on its own, but embedded in some website, e.g. a content management system (CMS). But it should also be possible to embed individual widgets.

Terms

The following terms are used in the context of embedding:

Host The website where adhocracy is embedded.

Widget A piece of adhocracy that can be embedded on its own. Some functionality (e.g. registration) will require the user to switch to the *platform*. See also CSS.

Frontend URL This is where the actual adhocracy frontend is available. Users are not supposed to interact with this directly. Instead, the frontend should be embedded somewhere.

Platform URL This is where the full adhocracy is available (as opposed to a widget). The term *platform* also refers to the complete set of functionality and navigation that sets it apart from a mere widget.

Canonical URL Content can show up in different places (i.e. with different URLs), namely at the frontend URL, the platform URL and embedded in many more places. The canonical URL is the *default* URL. In most cases (but not in all!), it will point to the platform. See also RFC6596.

Embed-API

The general idea consists of two parts: the SDK javascript code, which has to be loaded once, and widget markers in the DOM. On initialization, the widget markers are replaced by iframes, which show the actual content.

SDK snippet

This is our JavaScript code that runs in the host page. It was carefully written to not interfere with the hosts own JavaScript code.

- Bootstraps everything, initializes widgets
- Selects Adhocracy version to be used
- Creates `window.adhocracy` namespace
- Resizes widgets on the fly

Example:

```
<script type="text/javascript" src="https://adhocracy.lan/static/js/AdhocracySDK.js"></script>
<script type="text/javascript">
    adhocracy.init('http://adhocracy.lan', function(adhocracy) {
        adhocracy.embed('.adhocracy_marker');
    });
</script>
```

Widget markers

In order to embed actual widgets, you need to add one or more markers anywhere in the document. Each marker must define a widget and optionally one or more parameters:

```
<div class="adhocracy_marker"
     data-widget="document-workbench">
</div>
<div class="adhocracy_marker"
     data-widget="paragraph-version-detail"
     data-locale="en"
     data-ref="..." data-viewmode="display">
</div>
```

Note: Syntax should exist for both HTML5 (*data-* parameters) and HTML4

Parameters The available parameters depend on the respective widget. However, the following parameters are always available:

- the special widget `"plain"` will embed the full platform instead of a single widget:

```
    <div class="adhocracy_marker" data-widget="plain"></div>
```

- `autoresize` will control whether the iframe will automatically be resized to fit its contents. Defaults to `true`. It is recommended to set this to `false` if the embedded widget contains moving columns. In that case, an explicit height may be provided instead:

```
    <div class="adhocracy_marker" data-widget="plain" data-autoresize="false" style="height: 400px">
```

- `locale` can be used to set a locale.
- `autourl`: If set to `true`, the URL of the embedded adhocracy will be appended (and constantly updated) to the host URL via `#!`. This is only possible once per host page for obvious reasons.
- `nocenter`: By default, the widget will be centered in the iframe. If this option is set to `true`, it will fill the iframe instead.
- `noheader`: By default, a header will be shown above the widget. If this option is set to `true`, it will be omitted.
- `initial-url` will set the initial URL (i.e. path, query and anchor) for the embedded platform if widget is `"plain"`.

What happens internally

Say we use the following marker:

```
<div class="adhocracy_marker" data-widget="proposal-workbench" data-content="/proposal"></div>
```

This will be converted to the following URL for the iframe:

```
//example.com/embed/proposal-workbench?content=/proposal
```

The template inside of that iframe will look roughly like this:

```
<adh-proposal-workbench data-content="/proposal"></adh-proposal-workbench>
```

Allowing a directive to be embedded

Not every directive is allowed to be embedded. You need to register it with the embed provider:

```
import * as AdhEmbed from "../Embed/Embed";

export var myDirective = () => {
    // your directive's code
};


export var moduleName = "adhMyModule";

export var register = (angular) => {
    angular
        .module(moduleName, [
            AdhEmbed.moduleName
        ])
        .config(["adhEmbedProvider", (adhEmbedProvider : AdhEmbed.Provider) => {
            adhEmbedProvider.registerEmbeddableDirectives(["my-directive"]);
        }])
        .directive("adhMyDirective", [myDirective]);
};
```

Embed Widget for testing

As a side effect, the embed API can be used to develop and test functionalities of frontend widgets in an isolated way.

Say you have registered a directive as described in the previous section. Now you can see your widget under:

```
/embed/my-directive
```

Maybe you would also like to add data to your directive using attributes. As there is no surrounding scope to your directive, this needs to be mocked. You can do that by appending some GET parameters to your URL:

```
/embed/my-directive?variable1=1&variable2=2
```

The HTML element that is added to the embed page will look like this:

```
<adh-my-directive data-variable1="1" data-variable2="2" ></adh-my-directive>
```

In your directive you can now for example use this like this:

```
export var myDirective = () => {
    return {
        scope: {
            variable1: "@",
            variable2: "@"
        },
        // more code
    };
};
```

General notes

- Accout activation (after registration) and password reset require that the backend sends a URL to the user via email. So the backend needs to know canonical URLs for that.
- If a feature is not available in an embedded widget, all aspects of that widget that rely on that feature need to be modified. For example, whenever a user is referenced, we include a link to their profile page. If profile pages are not available in an embedded widget, these links either need to be removed or point to the platform instead.

doctest: +ELLIPSIS # doctest: +NORMALIZE_WHITESPACE

User Registration and Login

Prerequisites

Some imports to work with rest api calls:

```
>>> from copy import copy
>>> from pprint import pprint
>>> from adhocracy_core.testing import broken_header
```

Start adhocracy app and log in some users:

```
>>> anonymous = getfixture('app_anonymous')
>>> participant = getfixture('app_participant')
>>> moderator = getfixture('app_moderator')
>>> admin = getfixture('app_admin')
```

Test that the relevant resources and sheets exist:

```
>>> resp = anonymous.get('/meta_api').json
>>> 'adhocracy_core.sheets.versions.IVersions' in resp['sheets']
True
>>> 'adhocracy_core.sheets.principal.IUserBasic' in resp['sheets']
True
>>> 'adhocracy_core.sheets.principal.IUserExtended' in resp['sheets']
True
>>> 'adhocracy_core.sheets.principal.IPasswordAuthentication' in resp['sheets']
True
```

User Creation (Registration)

A new user is registered by creating a user object under the `/principals/users` pool. On success, the response contains the path of the new user:

```
>>> data = {'content_type': 'adhocracy_core.resources.principal.IUser',
...         'data': {
...              'adhocracy_core.sheets.principal.IUserBasic': {
...                  'name': 'Anna Müller'},
...              'adhocracy_core.sheets.principal.IUserExtended': {
...                  'email': 'anna@example.org'},
...              'adhocracy_core.sheets.principal.IPasswordAuthentication': {
...                  'password': 'EckVocUbs3'}}}
>>> resp = anonymous.post('/principals/users', data).json
>>> resp['content_type']
'adhocracy_core.resources.principal.IUser'
>>> user_path = resp['path']
```

```
>>> user_path
'.../principals/users/00...
```

The "name" field in the "IUserBasic" schema is a non-empty string that can contain any characters except '@' (to make user names distinguishable from email addresses). The username must not contain any whitespace except single spaces, preceded and followed by non-whitespace (no whitespace at begin or end, multiple subsequent spaces are forbidden, tabs and newlines are forbidden).

The "email" field in the "IUserExtended" sheet must be a valid email address.

Creating a new user will not automatically log them in. First, the backend will send a registration message to the specified email address. Once the user has clicked on the activation link in the message, the user account is ready to be used (see "Account Activation" below).

On failure, the backend responds with status code 400 and an error message. E.g. when we try to register a user with an empty password:

```
>>> data = {'content_type': 'adhocracy_core.resources.principal.IUser',
...         'data': {
...              'adhocracy_core.sheets.principal.IUserBasic': {
...                  'name': 'Other User'},
...              'adhocracy_core.sheets.principal.IUserExtended': {
...                  'email': 'annina@example.org'},
...              'adhocracy_core.sheets.principal.IPasswordAuthentication': {
...                  'password': ''}}}
>>> resp = anonymous.post('/principals/users', data)
>>> resp.status_code
400
>>> pprint(resp.json)
{'errors': [{'description': 'Required',
             'location': 'body',
             'name': 'data.adhocracy_core.sheets.principal.IPasswordAuthentication.password'}],
 'status': 'error'}
```

<errors> is a list of errors. The above error indicates that a required field (the password field) is missing or empty. The following other error conditions can occur:

- username does already exist
- email does already exist
- email is invalid (doesn't look like an email address)
- couldn't send a registration mail to the email address (description starts with 'Cannot send registration mail')
- password is too short (less than 6 chars)
- password is too long (more than 100 chars)
- internal error: something went wrong in the backend

For example, if we try to register a user whose email address is already registered:

```
>>> data = {'content_type': 'adhocracy_core.resources.principal.IUser',
...         'data': {
...              'adhocracy_core.sheets.principal.IUserBasic': {
...                  'name': 'New user with old email'},
...              'adhocracy_core.sheets.principal.IUserExtended': {
...                  'email': 'anna@example.org'},
...              'adhocracy_core.sheets.principal.IPasswordAuthentication': {
...                  'password': 'EckVocUbs3'}}}
>>> resp = anonymous.post('/principals/users', data)
```

```
>>> resp.status_code
400
>>> pprint(resp.json)
{'errors': [{'description': 'The user login email is not unique',
             'location': 'body',
             'name': 'data.adhocracy_core.sheets.principal.IUserExtended.email'}],
 'status': 'error'}
```

Note: in the future, the registration request may contain additional personal data for the user. This data will probably be added to the "IUserBasic" sheets, if it's generally public, to the "IUserExtended" sheet otherwise (or maybe it'll be store in additional new sheets); e.g.:

```
'data': {
    'adhocracy_core.sheets.principal.IUserBasic': {
        'name': 'Anna Müller',
        'forename': '...',
        'surname': '...'},
    'adhocracy_core.sheets.principal.IPasswordAuthentication': {
        'password': '...'},
    'adhocracy_core.sheets.principal.IUserExtended': {
        'email': 'anna@example.org',
        'day_of_birth': '...',
        'street': '...',
        'town': '...',
        'postcode': '...',
        'gender': '...'
    }
 }
```

Account Activation

Before they have confirmed their email address, new users are invisible (hidden). They won't show up in user listings, and retrieving information about them manually leads to a *410 Gone* response (see Deleting Resources):

```
>>> resp = anonymous.get(user_path)
>>> resp.status_code
410
>>> resp.json['reason']
'hidden'
```

On user registration, the backend sends a mail with an activation link to the specified email address and sends a 2xx HTTP response to the frontend, so the frontend can tell the user to expect an email. The user has to click on the activation link to activate their account. The *path* component of all such links starts with `/activate/`. Once the frontend receives a click on such a link, it must post a JSON request containing the path to the `activate_account` endpoint of the backend:

```
>>> newest_activation_path = getfixture('newest_activation_path')
>>> data = {'path': newest_activation_path}
>>> resp = anonymous.post('/activate_account', data).json
>>> pprint(resp)
{'status': 'success',
 'user_path': '.../principals/users/...',
 'user_token': '...'}
```

The backend responds with either response code 200 and 'status': 'success' and 'user_path' and 'user_token', just like after a successful login request (see next section). This means that the user account has been activated and the user is now logged in.

```
>>> data = {'path': '/activate/blahblah'}
>>> resp = anonymous.post('/activate_account', data)
>>> resp.status_code
400
>>> pprint(resp.json)
{'errors': [{'description': 'Unknown or expired activation path',
             'location': 'body',
             'name': 'path'}],
 'status': 'error'}
```

Or it responds with response code 400 and 'status': 'error'. Usually the error description will be one of:

- 'String does not match expected pattern' if the path doesn't start with '/activate/'
- 'Unknown or expired activation path' if the activation path is unknown to the backend or if it has expired because it was generated more than 7 days ago. Note that activation links are deleted from the backend once the account has been successfully activated, and expired links may also be deleted. Therefore we don't know whether the activation link was never valid (the user mistyped it or just tried to guess one), or it used to be valid but has expired. The message displayed to the user should explain that.

If the link is expired, user activation is no longer possible for security reasons and the user has to call support or register again, using a different email. (More user-friendly options are planned but haven't been implemented yet!)

Since the user account has been activated, the public part of the user information is now visible to everybody:

```
>>> resp = anonymous.get(user_path).json
>>> resp['data']['adhocracy_core.sheets.principal.IUserBasic']['name']
'Anna Müller'
```

Like every resource, the user has a metadata sheet with creation information. In the case of users, the creator is the user themselves:

```
>>> resp_metadata = resp['data']['adhocracy_core.sheets.metadata.IMetadata']
>>> resp_metadata['creator']
'.../principals/users/00...
>>> resp_metadata['creator'] == user_path
True
```

User Login

To log-in an existing and activated user via password, the frontend posts a JSON request to the URL `login_username` with a user name and password:

```
>>> data = {'name': 'Anna Müller',
...         'password': 'EckVocUbs3'}
>>> resp = anonymous.post('/login_username', data).json
>>> pprint(resp)
{'status': 'success',
 'user_path': '.../principals/users/...',
 'user_token': '...'}
>>> user_path = resp['user_path']
>>> user_token_via_username = resp['user_token']
>>> headers = {'X-User-Token': user_token_via_username}
>>> user = copy(anonymous)
>>> user.header = headers
```

Or to `login_email`, specifying the user's email address instead of name:

```
>>> data = {'email': 'anna@example.org',
...         'password': 'EckVocUbs3'}
>>> resp = anonymous.post('/login_email', data).json
>>> pprint(resp)
{'status': 'success',
 'user_path': '.../principals/users/...',
 'user_token': '...'}
>>> user_token_via_email = resp['user_token']
```

On success, the backend sends back the path to the object representing the logged-in user and a token that must be used to authorize additional requests by the user.

An error is returned if the specified user name or email doesn't exist or if the wrong password is specified. For security reasons, the same error message (referring to the password) is given in all these cases:

```
>>> data = {'name': 'No such user',
...         'password': 'EckVocUbs3'}
>>> resp = anonymous.post('/login_username', data)
>>> resp.status_code
400
>>> pprint(resp.json)
{'errors': [{'description': "User doesn't exist or password is wrong",
             'location': 'body',
             'name': 'password'}],
 'status': 'error'}
```

A different error message is given if username and password are valid but the user account hasn't been activated yet:

```
{'description': 'User account not yet activated',
 'location': 'body',
 'name': 'name'}
```

User Authentication

Once the user is logged in, the backend must add add header field to all HTTP requests made for the user: "X-User-Token". Its value is the received "user_token", respectively. The backend validates the token. If it's valid and not expired, the requested action is performed in the name and with the rights of the logged-in user.

Without authentication we may not post anything:

```
>>> resp = anonymous.options('/').json
>>> 'POST' not in resp
True
```

With authentication instead we may::

```
    >>> resp = admin.options('/').json
    >>> pprint(resp['POST']['request_body'])
    [...'adhocracy_core.resources.organisation.IOrganisation',...]
```

If the token is not valid or expired the backend responds with an error status that identifies the "X-User-Token" header as source of the problem:

```
>>> broken = copy(anonymous)
>>> broken.header = broken_header
>>> resp = broken.get('/meta_api')
>>> resp.status_code
400
>>> sorted(resp.json.keys())
```

```
['errors', 'status']
>>> resp.json['status']
'error'
>>> resp.json['errors'][0]['location']
'header'
>>> resp.json['errors'][0]['name']
'X-User-Token'
>>> resp.json['errors'][0]['description']
'Invalid user token'
>>> anonymous.header = {}
```

Tokens will usually expire after some time. (In the current implementation, they expire by default after 30 days, but configurations may change this.) Once they are expired, they will be considered as invalid so any further requests made by the user will lead to errors. To resolve this, the user must log in again.

Viewing Users

Without authorization, only very limited information on each user is visible:

```
>>> resp = anonymous.get(user_path).json
>>> resp['data']['adhocracy_core.sheets.principal.IUserBasic']
{'name': 'Anna Müller'}
>>> 'adhocracy_core.sheets.principal.IUserExtended' in resp['data']
False
>>> 'adhocracy_core.sheets.principal.IPermissions' in resp['data']
False
```

Only admins and the user herself can view extended information such as her email address:

```
>>> resp = admin.get(user_path).json
>>> pprint(resp['data']['adhocracy_core.sheets.principal.IUserExtended'])
{'email': 'anna@example.org', 'tzname': 'UTC'}
>>> 'adhocracy_core.sheets.principal.IPermissions' in resp['data']
True
>>> resp = user.get(user_path).json
>>> 'adhocracy_core.sheets.principal.IUserExtended' in resp['data']
True
>>> 'adhocracy_core.sheets.principal.IPermissions' in resp['data']
True
```

Other users, even if logged in, cannot:

```
>>> resp = participant.get(user_path).json
>>> 'adhocracy_core.sheets.principal.IUserExtended' in resp['data']
False
>>> 'adhocracy_core.sheets.principal.IPermissions' in resp['data']
False
```

Editing Users

User can edit their own data:

```
>>> headers = {'X-User-Token': user_token_via_username}
>>> user = copy(anonymous)
>>> user.header = headers
>>> data = {'data': {'adhocracy_core.sheets.principal.IUserBasic': {'name': 'edited_name'}}}
>>> resp = user.put(user_path, data).json
```

```
>>> len(resp['updated_resources']['modified'])
1
```

If they want to edit security-related information they need to pass their passwords in a custom header:

```
>>> headers = {'X-User-Token': user_token_via_username,
...            'X-User-Password': 'EckVocUbs3'}
>>> user = copy(anonymous)
>>> user.header = headers
>>> data = {'data': {'adhocracy_core.sheets.principal.IPasswordAuthentication': {'password': 'edited_
>>> resp = user.put(user_path, data).json
>>> len(resp['updated_resources']['modified'])
1
```

If the header is missing the change is silently dropped:

```
>>> headers = {'X-User-Token': user_token_via_username}
>>> user = copy(anonymous)
>>> user.header = headers
>>> data = {'data': {'adhocracy_core.sheets.principal.IPasswordAuthentication': {'password': 'edited_
>>> resp = user.put(user_path, data).json
>>> len(resp['updated_resources']['modified'])
0
```

Password Reset

If users forget their passwords, they can request a reset email:

```
>>> data = {'email': 'anna@example.org'}
>>> resp = anonymous.post('/create_password_reset', data).json
>>> resp['status']
'success'
```

The email contains a link that will allow them to enter a new password. Password reset also returns the credentials so that a user can login directly:

```
>>> newest_reset_path = getfixture('newest_reset_path')
>>> data = {'path': newest_reset_path(),
...         'password': 'new_password'}
>>> resp = anonymous.post('/password_reset', data).json
>>> pprint(resp)
{'status': 'success',
 'user_path': '.../principals/users/...',
 'user_token': '...'}
```

Security Considerations

- The password-reset mechanism allows attackers that have access to a user's email address to take over an account.
- The password-edit mechanism allows attackers that have access to a user's session and password to change the password. However, the user receives an email informing them about the change and about ways to recover their password (i.e. password-reset).
- In the future we may want to allow users to change their email address. In this case attackers with access to a user's session and password would be able to take over an account.

- Once an account has been compromised it is not possible to recover. Legitimate users have no way to prove their legitimacy.

doctest: +ELLIPSIS # doctest: +NORMALIZE_WHITESPACE

REST-API

Prerequisites

Some imports to work with rest api calls:

```
>>> import copy
>>> from functools import reduce
>>> from operator import itemgetter
>>> import os
>>> import requests
>>> from pprint import pprint
```

Start Adhocracy testapp and login admin:

```
>>> log = getfixture('log')
>>> app_admin = getfixture('app_admin')
>>> rest_url = getfixture('rest_url')
>>> rest_url
'http://localhost/api'
```

Resource structure

Resources have one content interface to set its type, like "adhocracy_core.resources.organisation.IOrganisation".

Terminology: we refer to content interfaces and the objects specified by content interfaces as "resources"; resources consist of "sheets" which are based on the substance-d concept of property sheet interfaces.

Every Resource has multiple sheets that define schemata to set/get data.

There are 5 base types of resources:

- *Pool*: folder in the resource hierarchy, can contain other Pools of any kind.
- *Item*: container Pool for ItemVersions of a specific type that belong to the same *DAG Sub-Items* that are closely related (e.g. Sections within Documents)
- *ItemVersion*: a specific version of an item (SectionVersion, DocumentVersion)
- *Simple*: Anything that is neither versionable/item nor a pool.

To model the application domain we have some frequently use derived types with semantics:

- *Organisation*: a subtype of Pool to do basic structuring for the Resource tree. Typical subtypes are other Organisations or *Process*.
- *Process*: a subtype of Pool to add configuration and resources for a specific participation process. Typical subtypes are *Proposal*.
- *Proposal*: a subtype of Item, this is normally content created by participants during a paticipation process.

Example resource tree:

```
Pool:          locations
Simple:        locations/berlin

Pool:          proposals
Item:          proposals/proposal1
ItemVersion:   proposals/proposal1/v1

Item:          proposals/proposal1/document1
ItemVersion:   proposals/proposal1/document1/v1
```

Meta-API

The backend needs to answer to kinds of questions:

1. Globally: What resources (content types) exist? What sheets may or must they contain? (What parts of) what sheets are read-only? mandatory? optional?
2. In the context of a given session and URL: What HTTP methods are allowed? With what resource objects in the body? What are the authorizations (display / edit / vote-on / ...)?

The second kind is implemented with the OPTIONS method on the existing URLs. The first is implemented with the GET method on a dedicated URL.

Global Info

The dedicated prefix defaults to "/meta_api/", but can be customized. The result is a JSON object with two main keys, "resources" and "sheets":

```
>>> resp_data = app_admin.get('/meta_api/').json
>>> sorted(resp_data.keys())
['resources', 'sheets', 'workflows']
```

The "resources" key points to an object whose keys are all the resources (content types) defined by the system:

```
>>> sorted(resp_data['resources'].keys())
[...'adhocracy_core.resources.organisation.IOrganisation'...]
```

Each of these keys points to an object describing the resource. If the resource implements sheets (and a resource that doesn't would be rather useless!), the object will have a "sheets" key whose value is a list of the sheets implemented by the resource:

```
>>> organisation_desc = resp_data['resources']['adhocracy_core.resources.organisation.IOrganisation']
>>> sorted(organisation_desc['sheets'])
['adhocracy_core.sheets.asset.IHasAssetPool', 'adhocracy_core.sheets.description.IDescription'...]
```

In addition we get the listing of resource super types (excluding IResource):

```
>>> document_desc = resp_data['resources']['adhocracy_core.resources.document.IDocument']
>>> sorted(document_desc['super_types'])
['adhocracy_core.interfaces.IItem', 'adhocracy_core.interfaces.IPool']
```

If the resource is an item, it will also have a "item_type" key whose value is the type of versions managed by this item (e.g. a Section will manage SectionVersions as main element type):

```
>>> document_desc['item_type']
'adhocracy_core.resources.document.IDocumentVersion'
```

If the resource is a pool or item that can contain resources, it will also have an "element_types" key whose value is the list of all resources the pool/item can contain (including the "item_type" if it's an item). For example, a pool can contain other pools; a document can contain tags.

```
>>> organisation_desc['element_types']
[...adhocracy_core.resources.process.IProcess...
>>> sorted(document_desc['element_types'])
[...'adhocracy_core.resources.paragraph.IParagraph']
```

The "sheets" key points to an object whose keys are all the sheets implemented by any of the resources:

```
>>> sorted(resp_data['sheets'].keys())
[...'adhocracy_core.sheets.name.IName', ...'adhocracy_core.sheets.pool.IPool'...]
```

Each of these keys points to an object describing the resource. Each of these objects has a "fields" key whose value is a list of objects describing the fields defined by the sheet:

```
>>> pprint(resp_data['sheets']['adhocracy_core.sheets.name.IName']['fields'][0])
{'creatable': True,
 'create_mandatory': True,
 'editable': False,
 'name': 'name',
 'readable': True,
 'valuetype': 'adhocracy_core.schema.Name'}
```

Each field definition has the following keys:

name The field name

create_mandatory Flag specifying whether the field must be set if the sheet is created (post requests).

readable Flag specifying whether the field can be read (get requests).

editable Flag specifying whether the field can be set to edit an existing sheet (put requests).

creatable Flag specifying whether the field can be set if the sheet is created (post requests).

valuetype The type of values stored in the field, either a basic type (as defined by Colander) such as "String" or "Integer", or a custom-defined type such as "adhocracy_core.schema.AbsolutePath"

There also are some optional keys:

containertype Only present if the field can store multiple values (each of the type specified by the "valuetype" attribute). If present, the value of this attribute is either "list" (a list of values: order matters, duplicates are allowed) or "set" (a set of values: unordered, no duplicates).

targetsheet Only present if "valuetype" is a path ("adhocracy_core.schema.AbsolutePath"). If present, it gives the name of the sheet that all pointed-to resources will implement (they might possibly be of different types, but they will always implement the given sheet or they wouldn't be valid link targets).

For example, the 'subdocuments' field of IDocument is an ordered list pointing to other IDocument's:

```
>>> secfields = resp_data['sheets']['adhocracy_core.sheets.document.IDocument']['fields']
>>> for field in secfields:
...     if field['name'] == 'elements':
...         pprint(field)
...         break
{'containertype': 'list',
 'creatable': True,
 'create_mandatory': False,
 'editable': True,
 'name': 'elements',
 'readable': True,
```

```
 'targetsheet': 'adhocracy_core.sheets.document.ISection',
 'valuetype': 'adhocracy_core.schema.AbsolutePath'}
```

The 'follows' field of IVersionable is an unordered set pointing to other IVersionable's:

... >>> verfields = resp_data['sheets']['adhocracy_core.sheets.versions.IVersionable']['fields'] ... >>> for field in verfields: ... ... if field['name'] == 'follows': ... ... pprint(field) ... ... break ... {'containertype': 'set', ... 'creatable': True, ... 'create_mandatory': False, ... 'name': 'follows', ... 'editable': True, ... 'readable': True, ... 'targetsheet': 'adhocracy_core.sheets.versions.IVersionable', ... 'valuetype': 'adhocracy_core.schema.AbsolutePath'}

In addition we get the listing of sheet super types (excluding ISheet):

```
>>> pprint(resp_data['sheets']['adhocracy_core.sheets.comment.IComment']['super_types'])
['adhocracy_core.interfaces.ISheetReferenceAutoUpdateMarker']
```

OPTIONS

Returns possible methods for this resource, example request/response data structures and available interfaces with resource data. The result is a JSON object that has the allowed request methods as keys:

```
>>> resp_data = app_admin.options('/').json
>>> sorted(resp_data.keys())
['DELETE', 'GET', 'HEAD', 'OPTIONS', 'POST', 'PUT']
```

If a GET, POST, or PUT request is allowed, the corresponding key will point to an object that contains at least "request_body" and "response_body" as keys:

```
>>> sorted(resp_data['GET'].keys())
[...'request_body', ...'response_body'...]
>>> sorted(resp_data['POST'].keys())
[...'request_body', ...'response_body'...]
```

The "response_body" sub-key returned for a GET request gives a stub view of the actual response body that will be returned:

```
>>> pprint(resp_data['GET']['response_body'])
{'content_type': '',
 'data': {...'adhocracy_core.sheets.name.IName': {}...},
 'path': ''}
```

"content_type" and "path" will be filled in responses returned by an actual GET request. "data" points to an object whose keys are the property sheets that are part of the returned resource. The corresponding values will be filled during actual GET requests; the stub contains just empty objects ('{}') instead.

If the current user has the right to post new versions of the resource or add new details to it, the "request_body" sub-key returned for POST points to a array of stub views of allowed requests:

```
>>> data_post_pool = {'content_type': 'adhocracy_core.resources.organisation.IOrganisation',
...                   'data': {'adhocracy_core.sheets.metadata.IMetadata': {},
...                            'adhocracy_core.sheets.title.ITitle': {},
...                            'adhocracy_core.sheets.name.IName': {},
...                            'adhocracy_core.sheets.description.IDescription': {},
...                            'adhocracy_core.sheets.image.IImageReference': {},
...                            'adhocracy_core.sheets.workflow.IWorkflowAssignment': {}}}
>>> data_post_pool in resp_data['POST']['request_body']
True
```

The "response_body" sub-key again gives a stub view of the response body:

```
>>> pprint(resp_data['POST']['response_body'])
{'content_type': '', 'path': ''}
```

If the current user has the right to modify the resource in-place, the "request_body" sub-key returned for PUT gives a stub view of how the actual request should look like:

```
..      >>> pprint(resp_data['PUT']['request_body'])
..      {'data': {...'adhocracy_core.sheets.name.IName': {}...}}
```

FIXME: PUT is missing, because the current test pool resource type has not editable sheet.

The "response_body" sub-key gives, as usual, a stub view of the resulting response body:

```
..      >>> pprint(resp_data['PUT']['response_body'])
..      {'content_type': '', 'path': ''}
```

Basic calls

We can use the following http verbs to work with resources.

HEAD

Returns only http headers:

```
>>> resp = app_admin.head('/adhocracy')
>>> resp.headerlist
[...('Content-Type', 'application/json; charset=UTF-8'), ...
>>> resp.text
''
```

The caching headers are set to no-cache to ease testing:

```
>>> resp.headers['X-Caching-Mode']
'no_cache'
```

GET

Returns resource and child elements meta data and all sheet with data:

```
>>> resp_data = app_admin.get('/').json
>>> pprint(resp_data['data'])
{...'adhocracy_core.sheets.metadata.IMetadata': ...
```

POST

Create a new resource

```
>>> prop = {'content_type': 'adhocracy_core.resources.process.IProcess',
...         'data': {'adhocracy_core.sheets.name.IName': {'name': 'Documents'}}}
>>> resp_data = app_admin.post('/', prop).json
>>> resp_data['content_type']
'adhocracy_core.resources.process.IProcess'
```

The response object has 3 top-level entries:

- The content type and the path of the new resource:

```
>>> resp_data['content_type']
'adhocracy_core.resources.process.IProcess'
>>> resp_data['path']
'.../Documents/'
```

- A listing of resources affected by the transaction:

```
>>> sorted(resp_data['updated_resources'])
['changed_descendants', 'created', 'modified', 'removed']
```

The subkey 'created' lists any resources that have been created by the transaction:

```
>>> sorted(resp_data['updated_resources']['created'])
['.../', '.../Documents/assets/', '.../Documents/badges/']
```

The subkey 'modified' lists any resources that have been modified:

```
>>> sorted(resp_data['updated_resources']['modified'])
['...', '.../principals/users/00...']
```

Modifications also include that case that a reference from another resource has been added or removed, since references are often exposed in both directions (the reserve direction is called "backreference"). In this case, the user is shown as modified since the new resource contains a reference to its creator.

The subkey 'removed' lists any resources that have been removed by marking them as hidden (see Deleting Resources):

```
>>> resp_data['updated_resources']['removed']
[]
```

A resource will be shown it at most *one* of the 'created', 'modified', or 'removed' lists, never in two or more of them.

The subkey 'changed_descendants' lists the parent (and grandparent etc.) pools of all the resources that have been created, modified, or removed. Any *query* to such pools may have become outdated as a result of the transaction (see "Filtering Pools" document below):

```
>>> sorted(resp_data['updated_resources']['changed_descendants'])
['...', '.../principals/', '.../principals/users/']
```

PUT

Modify data of an existing resource

```
FIXME: disable because IName.name is not editable. use another example!
FIXME: what we do here is a `patch` actually, so we should rename this.
```

... >>> data = {'content_type': 'adhocracy_core.resources.pool.IBasicPool', ... 'data': {'adhocracy_core.sheets.name.IName': {'name': 'youdidntexpectthis'}}} ... >>> resp_data = app_admin.put_json('/Documents', data).json ... >>> pprint(resp_data) ... {'content_type': 'adhocracy_core.resources.pool.IBasicPool', ... 'path': '/Documents'}

Check the changed resource

```
...    >>> resp_data = app_admin.get('/Documents').json
...    >>> resp_data['data']['adhocracy_core.sheets.name.IName']['name']
...    'youdidntexpectthis'
```

FIXME: write test cases for attributes with "create_mandatory", "editable", etc. (those work the same in PUT and POST, and on any attribute in the json tree.)

PUT responses have the same fields as POST responses.

Note: When putting multiple sheets in a request some changes might be currently dropped when the request does not have sufficient permissions, e.g. cannot be edit by the user or requires an additional header.

ERROR Handling

FIXME: ... is not working anymore in this doctest

The normal return code is 200

```
>>> data = {'content_type': 'adhocracy_core.resources.process.IProcess',
...         'data': {'adhocracy_core.sheets.name.IName': {'name': 'Documents'}}}
```

If you submit invalid data the return error code is 400

```
>>> data = {'content_type': 'adhocracy_core.resources.pool.IBasicPool',
...         'data': {'adhocracy_core.sheets.example.WRONGINTERFACE': {'name': 'Documents'}}}
```

and you get data with a detailed error description:

```
{
  'status': 'error',
  'errors': errors.
}
```

With errors being a JSON dictionary with the keys "location", "name" and "description".

location is the location of the error. It can be "querystring", "header" or "body" name is the eventual name of the value that caused problems description is a description of the problem encountered.

If all goes wrong the return code is 500.

Create and Update Versionable Resources

Introduction and Motivation

This section explains updates to resources with version control. Two sheets are central to version control in adhocracy: IDAG and IVersion. IVersion is in all resources that support version control, and IDAG is a container that manages all versions of a particular content element in a directed acyclic graph.

IDAGs as well as IVersions need to be created explicitly by the frontend.

The server supports updating a resource that implements IVersion by letting you post a content element with missing IVersion sheet to the DAG (IVersion is read-only and managed by the server), and passing a list of parent versions in the post parameters of the request. If there is only one parent version, the new version either forks off an existing branch or just continues a linear history. If there are several parent versions, we have a merge commit.

Example: If a new versionable content element has been created by the user, the front-end first posts an IDAG. The IDAG works a little like an IPool in that it allows posting versions to it. The front-end will then simply post the initial version into the IDAG with an empty predecessor version list.

IDAGs may also implement the IPool sheet for containing further IDAGs for sub-structures of structured versionable content types. Example: A document may consist of a title, description, and a list of references to sections. There is

a DAG for each document and each such dag contains one DAG for each document that occurs in any version of the document. Section refs in the document object point to specific versions in those DAGs.

When posting updates to nested sub-structures, the front-end must decide for which parent objects it wants to trigger an update. To stay in the example above: If we have a document with two sections, and update a section, the post request must contain both the parent version(s) of the section, but also the parent version(s) of the document that it is supposed to update.

To see why, consider the following situation:

```
Doc     v0        v1       v2
                 /        /
Par1    v0    v1        /
                       /
Par2    v0          v1

      >-----> time >-------->
```

We want Doc to be available in 3 versions that are linearly dependent on each other. But when the update to Par2 is posted, the server has no way of knowing that it should update v1 of Doc, BUT NOT v0!

Create

Create a Document (a subclass of Item which pools DocumentVersions)

```
>>> pdag = {'content_type': 'adhocracy_core.resources.document.IDocument',
...          'data': {},
...          }
>>> resp = app_admin.post('/Documents', pdag)
>>> pdag_path = resp.json['path']
>>> pdag_path
'.../Documents/document_0000000/'
```

The return data has the new attribute 'first_version_path' to get the path first Version:

```
>>> pvrs0_path = resp.json['first_version_path']
>>> pvrs0_path
'.../Documents/document_0000000/VERSION_0000000/'
```

Version IDs are numeric and assigned by the server. The front-end has no control over them, and they are not supposed to be human-memorable. For human-memorable version pointers that also allow for complex update behavior (fixed-commit, always-newest, ...), consider sheet ITags.

The Document has the IVersions and ITags interfaces to work with Versions:

```
>>> resp = app_admin.get(pdag_path)
>>> resp.json['data']['adhocracy_core.sheets.versions.IVersions']['elements']
['.../Documents/document_0000000/VERSION_0000000/']

>>> resp.json['data']['adhocracy_core.sheets.tags.ITags']['LAST']
'.../Documents/document_0000000/VERSION_0000000/'

>>> resp.json['data']['adhocracy_core.sheets.tags.ITags']['FIRST']
'.../Documents/document_0000000/VERSION_0000000/'
```

Update

Fetch the first Document version, it is empty

```
>>> resp = app_admin.get(pvrs0_path)
>>> pprint(resp.json['data']['adhocracy_core.sheets.document.IDocument'])
{'elements': []}

>>> pprint(resp.json['data']['adhocracy_core.sheets.versions.IVersionable'])
{'follows': []}
```

but owned by the Document item creator:

Create a new version of the proposal that follows the first version

```
>>> pvrs = {'content_type': 'adhocracy_core.resources.document.IDocumentVersion',
...         'data': {'adhocracy_core.sheets.document.IDocument': {
...                     'elements': []},
...                  'adhocracy_core.sheets.versions.IVersionable': {
...                     'follows': [pvrs0_path]}},
...          'root_versions': [pvrs0_path]}
>>> resp = app_admin.post(pdag_path, pvrs)
>>> pvrs1_path = resp.json['path']
>>> pvrs1_path != pvrs0_path
True
```

Add and update child resource

We expect certain Versionable fields for the rest of this test suite to work

```
>>> resp = app_admin.get('/meta_api')
>>> vers_fields = resp.json['sheets']['adhocracy_core.sheets.versions.IVersionable']['fields']
>>> pprint(sorted(vers_fields, key=itemgetter('name')))
[{'containertype': 'list',
  'creatable': True,
  'create_mandatory': False,
  'editable': True,
  'name': 'follows',
  'readable': True,
  'targetsheet': 'adhocracy_core.sheets.versions.IVersionable',
  'valuetype': 'adhocracy_core.schema.AbsolutePath'}]
```

The 'follows' element must be set by the client when it creates a new version that is the successor of one or several earlier versions.

Create a Section item inside the Document item

```
>>> sdag = {'content_type': 'adhocracy_core.resources.paragraph.IParagraph',
...         'data': {}
...         }
>>> resp = app_admin.post(pdag_path, sdag)
>>> sdag_path = resp.json['path']
>>> svrs0_path = resp.json['first_version_path']
```

and a second Section

```
>>> sdag = {'content_type': 'adhocracy_core.resources.paragraph.IParagraph',
...         'data': {}
...         }
>>> resp = app_admin.post(pdag_path, sdag)
>>> s2dag_path = resp.json['path']
>>> s2vrs0_path = resp.json['first_version_path']
```

Create a third Document version and add the two Sections in their initial versions

```
>>> pvrs = {'content_type': 'adhocracy_core.resources.document.IDocumentVersion',
...         'data': {'adhocracy_core.sheets.document.IDocument': {
...                     'elements': [svrs0_path, s2vrs0_path]},
...                  'adhocracy_core.sheets.versions.IVersionable': {
...                     'follows': [pvrs1_path],}
...                 },
...          'root_versions': [pvrs1_path]}
>>> resp = app_admin.post(pdag_path, pvrs)
>>> pvrs2_path = resp.json['path']
```

If we create a second version of kapitel1

```
>>> svrs = {'content_type': 'adhocracy_core.resources.paragraph.IParagraphVersion',
...         'data': {
...              'adhocracy_core.sheets.document.IParagraph': {
...                  'title': 'Kapitel Überschrift Bla',
...                  'elements': []},
...               'adhocracy_core.sheets.versions.IVersionable': {
...                  'follows': [svrs0_path]
...                  }
...          },
...          'root_versions': [pvrs2_path]
...         }
>>> resp = app_admin.post(sdag_path, svrs)
>>> svrs1_path = resp.json['path']
>>> svrs1_path != svrs0_path
True
```

Whenever a IVersionable contains 'follows' link(s) to preceding versions, there should be a top-level 'root_versions' element listing the version of their root elements. 'root_versions' is a set, which means that order doesn't matter and duplicates are ignored. In this case, it points to the proposal version containing the document to update.

The 'root_versions' set allows automatical updates of items that embedding or otherwise linking to the updated item. In this case, a fourth Document version is automatically created along with the updated Section version:

```
>>> resp = app_admin.get(pdag_path)
>>> pprint(resp.json['data']['adhocracy_core.sheets.versions.IVersions'])
{'count': '4',
 'elements': ['.../Documents/document_0000000/VERSION_0000000/',
              '.../Documents/document_0000000/VERSION_0000001/',
              '.../Documents/document_0000000/VERSION_0000002/',
              '.../Documents/document_0000000/VERSION_0000003/']}

>>> resp = app_admin.get('/Documents/document_0000000/VERSION_0000003')
>>> pvrs3_path = resp.json['path']

>>> s2vrs1_path = resp.json['path']
>>> s2vrs1_path != s2vrs0_path
True
```

More interestingly, if we try to create a second version of kapitel2 we get an error because this would automatically create two new version for pvrs3 and pvrs2 (both contain s2vrs0_path):

```
>>> svrs = {'content_type': 'adhocracy_core.resources.paragraph.IParagraphVersion',
...         'data': {
...              'adhocracy_core.sheets.document.IParagraph': {
...                  'title': 'on the hardness of version control',
...                  'elements': []},
```

```
...                 'adhocracy_core.sheets.versions.IVersionable': {
...                     'follows': [s2vrs0_path]
...                     }
...             },
...             'root_versions': []
...           }
>>> resp = app_admin.post(s2dag_path, svrs)
>>> pprint(resp.json['errors'][0])
{'description': 'No fork allowed - The auto update ...
```

But if we set the *root_version* to the last Document version (pvrs3)::

```
    >>> svrs = {'content_type': 'adhocracy_core.resources.paragraph.IParagraphVersion',
    ...          'data': {
    ...               'adhocracy_core.sheets.document.IParagraph': {
    ...                   'title': 'on the hardness of version control',
    ...                   'elements': []},
    ...                'adhocracy_core.sheets.versions.IVersionable': {
    ...                   'follows': [s2vrs0_path]
    ...                   }
    ...             },
    ...             'root_versions': [pvrs3_path]
    ...           }
    >>> resp = app_admin.post(s2dag_path, svrs)
```

a new version pvrs4 is automatically created following only pvrs3, not pvrs2:

```
>>> resp = app_admin.get(pdag_path)
>>> pprint(resp.json['data']['adhocracy_core.sheets.versions.IVersions'])
{'count': '5',
 'elements': ['.../Documents/document_0000000/VERSION_0000000/',
              '.../Documents/document_0000000/VERSION_0000001/',
              '.../Documents/document_0000000/VERSION_0000002/',
              '.../Documents/document_0000000/VERSION_0000003/',
              '.../Documents/document_0000000/VERSION_0000004/']}

>>> resp = app_admin.get('/Documents/document_0000000/VERSION_0000004')
>>> pvrs4_path = resp.json['path']
>>> resp.json['data']['adhocracy_core.sheets.versions.IVersionable']['follows']
[.../Documents/document_0000000/VERSION_0000003/']

>>> resp = app_admin.get('/Documents/document_0000000/VERSION_0000003')
>>> resp.json['data']['adhocracy_core.sheets.versions.IVersionable']['follows']
[.../Documents/document_0000000/VERSION_0000002/']
```

FIXME: If two frontends post competing documents simultaneously, neither knows which proposal version belongs to whom. Proposed solution: the post response must tell the frontend the changed `root_version`.

Tags

Each Versionable has a FIRST tag that points to the initial version:

```
>>> resp = app_admin.get('/Documents/document_0000000')
>>> pprint(resp.json['data']['adhocracy_core.sheets.tags.ITags']['FIRST'])
'.../Documents/document_0000000/VERSION_0000000/'
```

It also has a LAST tag that points to the newest versions – any versions that aren't 'followed_by' any later version:

```
>>> pprint(resp.json['data']['adhocracy_core.sheets.tags.ITags']['LAST'])
'.../Documents/document_0000000/VERSION_0000004/'
```

Forks and forkability

This api has been designed to allow implementation of complex merge conflict resolution, both automatic and with user-involvement. Many resource types, however, only supports a simplified version control strategy with a *linear history*: If any version that is not head is used as a predecessor, the backend responds with an error. The frontend has to handle these errors, as they can always occur in race conditions with other users.

Current and potential future conflict resolution strategies are:

1. If a race condition is reported by the backend, the frontend updates the predecessor version to head and tries again. (In the unlikely case where lots of post activity is going on, it may be necessary to repeat this several times.)

   Example: IRatingVersion can only legally be modified by one user and should not experience any race conditions. If it does, the second post wins and silently reverts the previous one.

2. (Future work) Like 1., but the frontend posts two new versions on top of HEAD. If this is the situation of the conflict:

```
Doc      v0----v1
             \
              -----v1'

       >-----> time >-------->
```

   Then it is resolved as follows (by the frontend of the author of v1'):

```
Doc      v0----v1
                 \
                  -----v0'----v1'

       >-----> time >-------->
```

   v0' is a copy of v0 that differs only in its predecessor. It is called a 'revert' version. (FIXME: is there a way to enrich the data with a 'is_revert' flag?)

   This must be done in a batch request (a transaction) in order to avoid that only the revert is successfully posted, but the actual change fails. Again, it is possible that this batch request fails, and has to be attempted several times.

   Example: IDocumentVersion can be modified by many users concurrently.

3. (Future work) Both authors of the conflict are notified (email, dashboard, ...), and explained how they can inspect the situation and add new versions. (The email should probably contain a warning that it's best to get on the phone and talk it through before generating more merge conflicts.)

4. (Future work) Ideally, the user would to be notified that there is a conflict, display the differences between the three versions, and allow the user to merge his changes into the current HEAD.

5. (Future work) It is allowed to have multiple heads in the DAG, e.g. different preferred versions by different principals. This however still requires a lot of UX work to be done.

To give an example, *Comments* only allow a linear version history (just a single heads). Lets create a comment with an initial version (see below for more on comments and *post pools*):

```
>>> resp = app_admin.get('/Documents/document_0000000/VERSION_0000004')
>>> commentable = resp.json['data']['adhocracy_core.sheets.comment.ICommentable']
>>> post_pool_path = commentable['post_pool']
>>> comment = {'content_type': 'adhocracy_core.resources.comment.IComment',
...            'data': {}}
>>> resp = app_admin.post(post_pool_path, comment)
>>> comment_path = resp.json['path']
>>> first_commvers_path = resp.json['first_version_path']
>>> first_commvers_path
'.../Documents/document_0000000/comments/comment_000.../VERSION_0000000/'
```

We can create a second version that refers to the first (auto-created) version as predecessor:

```
>>> commvers = {'content_type': 'adhocracy_core.resources.comment.ICommentVersion',
...             'data': {
...                 'adhocracy_core.sheets.comment.IComment': {
...                     'refers_to': pvrs4_path,
...                     'content': 'Bla bla bla!'},
...                 'adhocracy_core.sheets.versions.IVersionable': {
...                     'follows': [first_commvers_path]}},
...             'root_versions': [first_commvers_path]}
>>> resp = app_admin.post(comment_path, commvers)
>>> snd_commvers_path = resp.json['path']
>>> snd_commvers_path
'.../Documents/document_0000000/comments/comment_000.../VERSION_0000001/'
```

However, if we try to add another version that *also* gives the first version (no longer head) as predecessor, we get an error:

```
>>> resp_data = app_admin.post(comment_path, commvers).json
>>> pprint(resp_data)
{'errors': [{'description': 'No fork allowed ...
             'location': 'body',
             'name': 'data.adhocracy_core.sheets.versions.IVersionable.follows'}],
 'status': 'error'}
```

The *description* of the error will always be 'No fork allowed'. This allows distinguishing this error from other kinds of errors.

Only resources that implement the *adhocracy_core.sheets.versions.IForkableVersionable* sheet (instead of *adhocracy_core.sheets.versions.IVersionable*) allow forking (multiple heads). For now, none of our standard resource types does this.

Resources with PostPool, example Comments

To give another example of a versionable content type, we can write comments about proposals. The proposal has a commentable sheet:

```
>>> resp = app_admin.get(pvrs4_path)
>>> commentable = resp.json['data']['adhocracy_core.sheets.comment.ICommentable']
```

This sheet has a special field *post_pool* referencing a pool:

```
>>> post_pool_path = commentable['post_pool']
```

We can post comments to this pool only:

```
>>> comment = {'content_type': 'adhocracy_core.resources.comment.IComment',
...            'data': {}}
>>> resp = app_admin.post(post_pool_path, comment)
>>> comment_path = resp.json['path']
>>> comment_path
'.../Documents/document_0000000/comments/comment_000...'
>>> first_commvers_path = resp.json['first_version_path']
>>> first_commvers_path
'.../Documents/document_0000000/comments/comment_000.../VERSION_0000000/'
```

The first comment version is empty (as with all versionables), so lets add another version to say something meaningful. A comment contains *content* (arbitrary text) and *refers_to* a specific version of a proposal.

```
>>> commvers = {'content_type': 'adhocracy_core.resources.comment.ICommentVersion',
...             'data': {
...                 'adhocracy_core.sheets.comment.IComment': {
...                     'refers_to': pvrs4_path,
...                     'content': 'Gefällt mir, toller Vorschlag!'},
...                 'adhocracy_core.sheets.versions.IVersionable': {
...                     'follows': [first_commvers_path]}},
...             'root_versions': [first_commvers_path]}
>>> resp = app_admin.post(comment_path, commvers)
>>> snd_commvers_path = resp.json['path']
>>> snd_commvers_path
'.../Documents/document_0000000/comments/comment_000.../VERSION_0000001/'
```

Comments can be about any versionable that allows posting comments. Hence it's also possible to write a comment about another comment:

```
>>> metacomment = {'content_type': 'adhocracy_core.resources.comment.IComment',
...                'data': {}}
>>> resp = app_admin.post(post_pool_path, metacomment)
>>> metacomment_path = resp.json['path']
>>> metacomment_path
'.../Documents/document_0000000/comments/comment_000...'
>>> comment_path != metacomment_path
True
>>> first_metacommvers_path = resp.json['first_version_path']
>>> first_metacommvers_path
'.../Documents/document_0000000/comments/comment_000.../VERSION_0000000/'
```

As usual, we have to add another version to actually say something:

```
>>> metacommvers = {'content_type': 'adhocracy_core.resources.comment.ICommentVersion',
...                 'data': {
...                     'adhocracy_core.sheets.comment.IComment': {
...                         'refers_to': snd_commvers_path,
...                         'content': 'Find ich nicht!'},
...                     'adhocracy_core.sheets.versions.IVersionable': {
...                         'follows': [first_metacommvers_path]}},
...                 'root_versions': [first_metacommvers_path]}
>>> resp = app_admin.post(metacomment_path, metacommvers)
>>> snd_metacommvers_path = resp.json['path']
>>> snd_metacommvers_path
'.../Documents/document_0000000/comments/comment_000.../VERSION_0000001/'
```

Let view all the comments referring to the proposal with a query on the comments pool:

```
>>> resp_data = app_admin.get(post_pool_path,
...     params={'content_type': 'adhocracy_core.resources.comment.ICommentVersion',
...             'depth': 2}).json
>>> commvers = resp_data['data']['adhocracy_core.sheets.pool.IPool']['elements']
>>> snd_commvers_path in commvers
True
```

Since comments can refer to other comments, we can also find out which other comments refer to this comment version:

```
>>> resp_data = app_admin.get(post_pool_path,
...     params={'content_type': 'adhocracy_core.resources.comment.ICommentVersion',
...             'adhocracy_core.sheets.comment.IComment:refers_to': snd_commvers_path,
...             'depth': 2}).json
>>> comlist = resp_data['data']['adhocracy_core.sheets.pool.IPool']['elements']
>>> comlist == [snd_metacommvers_path]
True
```

Rates

We can rate objects that provide the *adhocracy_core.sheets.rate.IRateable* sheet (or a subclass of it), e.g. comment versions. Rateables have their own post pool, so we ask the comment where to send rates about it:

```
>>> resp = app_admin.get(snd_commvers_path)
>>> rateable_post_pool = resp.json['data']['adhocracy_core.sheets.rate.IRateable']['post_pool']
```

IRate objects are versionable too, so we first have to create a *IRate* resource and then post a *IRateVersion* resource below it:

```
>>> rate = {'content_type': 'adhocracy_core.resources.rate.IRate',
...         'data': {}}
>>> resp = app_admin.post(rateable_post_pool, rate)
>>> rate_path = resp.json['path']
>>> first_ratevers_path = resp.json['first_version_path']
>>> ratevers = {'content_type': 'adhocracy_core.resources.rate.IRateVersion',
...             'data': {
...                 'adhocracy_core.sheets.rate.IRate': {
...                     'subject': app_admin.user_path,
...                     'object': snd_commvers_path,
...                     'rate': '1'},
...                 'adhocracy_core.sheets.versions.IVersionable': {
...                     'follows': [first_ratevers_path]}},
...             'root_versions': [first_ratevers_path]}
>>> resp = app_admin.post(rate_path, ratevers)
>>> snd_ratevers_path = resp.json['path']
>>> snd_ratevers_path
'...Documents/document_0000000/rates/rate_0000000/VERSION_0000001/'
```

If we want to change our rate, we can post a new version:

```
>>> ratevers['data']['adhocracy_core.sheets.rate.IRate']['rate'] = '0'
>>> ratevers['data']['adhocracy_core.sheets.versions.IVersionable']['follows'] = [snd_ratevers_path]
>>> ratevers['root_versions'] = [snd_ratevers_path]
>>> resp = app_admin.post(rate_path, ratevers)
>>> third_ratevers_path = resp.json['path']
>>> third_ratevers_path != snd_ratevers_path
True
```

But creating a second rate is not allowed to prevent people from voting multiple times:

```
   >>> resp = app_admin.post(rateable_post_pool, rate)
   >>> rate2_path = resp.json['path']
   >>> first_rate2vers_path = resp.json['first_version_path']
   >>> ratevers['data']['adhocracy_core.sheets.versions.IVersionable']['follows'] = [first_rate2vers_
   >>> ratevers['root_versions'] = [first_rate2vers_path]
   >>> resp_data = app_admin.post(rate2_path, ratevers).json
   >>> resp_data['errors'][0]['name']
   'data.adhocracy_core.sheets.rate.IRate.object'
   >>> resp_data['errors'][0]['description']
   '; Another rate by the same user already exists'

...TODO: remove ';' suffix of error description, :mod:`colander` bug
```

The *subject* of a rate must always be the user that is currently logged in – it's not possible to vote for other users:

```
>>> ratevers['data']['adhocracy_core.sheets.rate.IRate']['subject'] = '/principals/users/0000005/'
>>> ratevers['data']['adhocracy_core.sheets.versions.IVersionable']['follows'] = [third_ratevers_path
>>> ratevers['root_versions'] = [third_ratevers_path]
>>> resp_data = app_admin.post(rate_path, ratevers).json
>>> resp_data['errors'][0]['name']
'data.adhocracy_core.sheets.rate.IRate.subject'
>>> resp_data['errors'][0]['description']
'; Must be the currently logged-in user'
```

Batch requests

The following URL accepts batch requests

```
>>> batch_url = '/batch'
```

A batch request a POST request with a json array in the body that contains certain HTTP requests encoded in a certain way.

A success response contains in its body an array of encoded HTTP responses. This way, the client can see what happened to the individual POSTS, and collect all the paths of the individual resources that were posted.

Batch requests are processed as a transaction. By this, we mean that either all encoded HTTP requests succeed and the response to the batch request is a success response, or any one of them fails, the database state is rolled back to the beginning of the request, and the response is an error, explaining which request failed for which reason.

Things that are different in individual requests

Forks and multiple versions

During one Batch request you can create only one new version. The first version created (with an explicit post request or auto updated) is used to store all modifications.

Preliminary resource paths: motivation and general idea.

All requests with methods POST, GET, PUT as allowed in the rest of this document are allowed in batch requests. POST differs in that it yields *preliminary resource paths*. To understand what that is, consider this example: In step 4 of a batch request, the front-end wants to post to the path that resulted from posting the parent resource in step 3 of the same request, so batch requests need to allow for an abstraction over the resource paths resulting from POST requests. POST yields preliminary paths instead of actual ones, and POST, GET, and PUT are all allowed to use preliminary paths in addition to the "normal" ones. Apart from this, nothing changes in the individual requests.

Preliminary resource paths: implementation.

The encoding of a request consist of an object with attributes for method (aka HTTP verb), path, and body. A further attribute, 'result_path', defines a name for the preliminary path of the object created by the request. The preliminary path is like an *AbsolutePath*, but it starts with '@' instead of '/'. If the preliminary name will not be used, this attribute can be omitted or left empty.

```
>>> encoded_request_with_name = {
...     'method': 'POST',
...     'path': '/Proposal/document_0000000',
...     'body': { 'content_type': 'adhocracy_core.resources.sample_paragraph.IParagraph' },
...     'result_path': '@par1_item',
...     'result_first_version_path': '@par1_item/v1'
... }
```

Preliminary paths can be used anywhere in subsequent requests, either in the 'path' item of the request itself, or anywhere in the json data in the body where the schemas expect to find resource paths. It must be prefixed with "@" in order to mark it as preliminary. Right before executing the request, the backend will traverse the request object and replace all preliminary paths with the actual ones that will be available by then.

In order to post the first *real* item version, we must use 'first_version_path' as the predecessor version, but we can't know its value before the post of the item version. This would not be a problem if the item would be created empty.

FIXME: change the api accordingly so that this problem goes away!

In order to work around you can set the optional field 'result_first_version_path' with a *preliminary resource path*.

Examples

Let's add some more paragraphs to the second document above

```
>>> document_item = s2dag_path
>>> batch = [ {
...              'method': 'POST',
...              'path': pdag_path,
...              'body': {
...                  'content_type': 'adhocracy_core.resources.paragraph.IParagraph',
...                  'data': {}
...              },
...              'result_path': '@par1_item',
...              'result_first_version_path': '@par1_item/v1'
...            },
...            {
...              'method': 'POST',
...              'path': '@par1_item',
...              'body': {
...                  'content_type': 'adhocracy_core.resources.paragraph.IParagraphVersion',
...                  'data': {
...                      'adhocracy_core.sheets.versions.IVersionable': {
...                          'follows': ['@par1_item/v1']
...                      },
...                      'adhocracy_core.sheets.document.IParagraph': {
...                          'text': 'sein blick ist vom vorüberziehn der stäbchen'
...                      }
...                  },
...              },
...              'result_path': '@par1_item/v2'
...            },
...            {
```

```
...             'method': 'GET',
...             'path': '@par1_item/v2'
...           },
...         ]
```

The batch response is a dictionary with two fields:

```
>>> batch_resp = app_admin.post(batch_url, batch).json
>>> sorted(batch_resp)
['responses', 'updated_resources']
```

'responses' is an array of the individual responses.

'updated_resources' lists all the resources affected by the POST and PUT requests in the batch request. If the batch requests doesn't contain any such requests (only GET etc.), all of its sub-entries will be empty.

```
>>> updated_resources = batch_resp['updated_resources']
>>> rest_url + '/Documents/' in updated_resources['changed_descendants']
True
>>> rest_url + '/Documents/document_0000000/PARAGRAPH_0000002/' in updated_resources['created']
True
```

Lets inspect some of the responses. The 'code' field contains the HTTP status code. The 'body' field contains the JSON dict that would normally be sent as body of the request, except that its 'updated_resources' field (if any) is omitted:

```
>>> len(batch_resp['responses'])
3
>>> pprint(batch_resp['responses'][0])
{'body': {'content_type': 'adhocracy_core.resources.paragraph.IParagraph',
          'first_version_path': '.../Documents/document_0000000/PARAGRAPH_0000002/VERSION_0000000/',
          'path': '.../Documents/document_0000000/PARAGRAPH_0000002/'},
 'code': 200}
>>> pprint(batch_resp['responses'][1])
{'body': {'content_type': 'adhocracy_core.resources.paragraph.IParagraphVersion',
          'path': '.../Documents/document_0000000/PARAGRAPH_0000002/VERSION_0000000/'},
 'code': 200}
>>> pprint(batch_resp['responses'][2])
{'body': {'content_type': 'adhocracy_core.resources.paragraph.IParagraphVersion',
          'data': {...},
          'path': '.../Documents/document_0000000/PARAGRAPH_0000002/VERSION_0000000/'},
 'code': 200}
 >>> batch_resp['responses'][2]['body']['data']['adhocracy_core.sheets.document.IParagraph']['text']
 'sein blick ist vom vorüberziehn der stäbchen'
```

New Versions are only created once within one batch request. That means the second subrequest does not create a second version, but updates the existing first version:

```
>>> v0 = batch_resp['responses'][0]['body']['first_version_path']
>>> v0_again = batch_resp['responses'][1]['body']['path']
>>> v0 == v0_again
True
```

The follow reference points to None:

```
>>> batch_resp['responses'][2]['body']['data']['adhocracy_core.sheets.versions.IVersionable']['follow
[]
```

The LAST tag should point to the last version we created within the batch request:

```
>>> resp_data = app_admin.get('/Documents/document_0000000/PARAGRAPH_0000002').json
>>> resp_data['data']['adhocracy_core.sheets.tags.ITags']['LAST']
'.../Documents/document_0000000/PARAGRAPH_0000002/VERSION_0000000/'
```

All creation and modification dates are equal for one batch request:

```
>>> pdag_metadata = app_admin.get(pdag_path).json['data']['adhocracy_core.sheets.metadata.IMetadata']
>>> pv0_path =  batch_resp['responses'][0]['body']['first_version_path']
>>> pv0_metadata = app_admin.get(pv0_path).json['data']['adhocracy_core.sheets.metadata.IMetadata']
>>> pv1_path =  batch_resp['responses'][0]['body']['path']
>>> pv1_metadata = app_admin.get(pv1_path).json['data']['adhocracy_core.sheets.metadata.IMetadata']
>>> pv0_metadata['creation_date'] \
... == pv0_metadata['modification_date']\
... == pv1_metadata['creation_date']\
... == pv1_metadata['modification_date']
True
```

Post another paragraph item and a version. If the version post fails, the paragraph will not be present in the database

```
>>> invalid_batch = [ {
...             'method': 'POST',
...             'path': pdag_path,
...             'body': {
...                 'content_type': 'adhocracy_core.resources.paragraph.IParagraph',
...                 'data': {}
...             },
...             'result_path': '@par2_item'
...           },
...           {
...             'method': 'POST',
...             'path': '@par2_item',
...             'body': {
...                 'content_type': 'NOT_A_CONTENT_TYPE_AT_ALL',
...                 'data': {
...                     'adhocracy_core.sheets.versions.IVersionable': {
...                         'follows': ['@par2_item/v1']
...                     },
...                     'adhocracy_core.sheets.document.IParagraph': {
...                         'content': 'das wird eh nich gepostet'
...                     }
...                 }
...             },
...             'result_path': '@par2_item/v2'
...           }
...         ]
>>> invalid_batch_resp = app_admin.post(batch_url, invalid_batch).json
>>> pprint(sorted(invalid_batch_resp['updated_resources']))
['changed_descendants', 'created', 'modified', 'removed']
>>> pprint(invalid_batch_resp['responses'])
[{'body': {'content_type': 'adhocracy_core.resources.paragraph.IParagraph',
           'first_version_path': '...',
           'path': '...'},
  'code': 200},
 {'body': {'errors': [...],
           'status': 'error'},
  'code': 400}]
>>> get_nonexistent_obj = app_admin.get(invalid_batch_resp['responses'][0]['body']['path'])
>>> get_nonexistent_obj.status
'404 Not Found'
```

Note that the response will contain embedded responses for all successful encoded requests (if any) and also for the first failed encoded request (if any), but not for any further failed requests. The backend stops processing encoded requests once the first of them has failed, since further processing would probably only lead to further errors.

Filtering Pools / Search

By default resources with IPool sheets do not list the child elements but only the *count*:

```
>>> resp_data = app_admin.get('/Documents/document_0000000/comments/').json
>>> pprint(resp_data['data']['adhocracy_core.sheets.pool.IPool'])
{'count': '3', 'elements': []}
```

Note: due to limitations of our (de)serialization library (Colander), -the count is returned as a string, though it is actually a number.

To list child elements you have to do a search query with *elements=paths* (see below for more detailed examples):

```
    >>> resp_data = app_admin.get( '/Documents/document_0000000/comments',
    ...     params={'elements': 'paths'}).json
    >>> pprint(resp_data['data']['adhocracy_core.sheets.pool.IPool'])
    {'count': '3',
     'elements': ['http://...]}
```

It is possible to filter and aggregate the elements listed in the IPool sheet by additional GET parameters. For example, we can only retrieve children that have specific resource type (*content_type*):

```
>>> resp_data = app_admin.get('/Documents/document_0000000',
...     params={'content_type': 'adhocracy_core.resources.paragraph.IParagraph'}).json
>>> pprint(resp_data['data']['adhocracy_core.sheets.pool.IPool']['elements'])
['.../Documents/document_0000000/PARAGRAPH_0000000/',
 '.../Documents/document_0000000/PARAGRAPH_0000001/',
 '.../Documents/document_0000000/PARAGRAPH_0000002/']
```

Note that multiple filters are combined by AND. If we specify a content_type filter and a sheet filter, only the elements matched by *both* filters will be returned. The same applies to all other filters as well.

For more sophisticated queries you can add various comparator suffix to your parameter value. The available comparators depend on the choosedn filter.

eq 'equal to' is the default comparator we already used implicit:

```
>>> resp_data = app_admin.get('/Documents/document_0000000',
...     params={'content_type': '["eq", "adhocracy_core.resources.paragraph.IParagraph"]'}).json
>>> pprint(resp_data['data']['adhocracy_core.sheets.pool.IPool']['elements'])
['.../Documents/document_0000000/PARAGRAPH_0000000/'...
```

noteq not equal to:

```
>>> resp_data = app_admin.get('/Documents/document_0000000',
...     params={'content_type': '["noteq", "adhocracy_core.resources.paragraph.IParagraph"]'}).json
>>> pprint(resp_data['data']['adhocracy_core.sheets.pool.IPool']['elements'])
['.../Documents/document_0000000/VERSION_0000000/',...
```

gt greater then:

```
>>> resp_data = app_admin.get('/Documents/document_0000000/rates/',
...     params={'name': '["gt", "rate_0000000"]'}).json
>>> pprint(resp_data['data']['adhocracy_core.sheets.pool.IPool']['elements'])
['.../Documents/document_0000000/rates/rate_0000001/']
```

ge greater or equal to:

```
>>> resp_data = app_admin.get('/Documents/document_0000000/rates/',
...     params={'name': '["ge", "rate_0000000"]'}).json
>>> pprint(resp_data['data']['adhocracy_core.sheets.pool.IPool']['elements'])
['.../Documents/document_0000000/rates/rate_0000000/',
 '.../Documents/document_0000000/rates/rate_0000001/']
```

lt lower then:

```
>>> resp_data = app_admin.get('/Documents/document_0000000/rates/',
...     params={'name': '["lt", "rate_0000001"]'}).json
>>> pprint(resp_data['data']['adhocracy_core.sheets.pool.IPool']['elements'])
['.../Documents/document_0000000/rates/rate_0000000/']
```

le lower or equal to:

```
>>> resp_data = app_admin.get('/Documents/document_0000000/rates/',
...     params={'name': '["le", "rate_0000001"]'}).json
>>> pprint(resp_data['data']['adhocracy_core.sheets.pool.IPool']['elements'])
['.../Documents/document_0000000/rates/rate_0000000/',
 '.../Documents/document_0000000/rates/rate_0000001/']
```

Some comparators can handle a list of query values.

any:

```
>>> resp_data = app_admin.get('/Documents/document_0000000/rates/',
...     params={'name': '["any", ["rate_0000000", "rate_0000001"]]'}).json
>>> pprint(resp_data['data']['adhocracy_core.sheets.pool.IPool']['elements'])
['.../Documents/document_0000000/rates/rate_0000000/',
 '.../Documents/document_0000000/rates/rate_0000001/']
```

notany:

```
>>> resp_data = app_admin.get('/Documents/document_0000000/rates/',
...     params={'name': '["notany", ["rate_0000000", "rate_0000001"]]'}).json
>>> pprint(resp_data['data']['adhocracy_core.sheets.pool.IPool']['elements'])
[]
```

By default, only direct children of a pool are listed as elements, i.e. the standard depth is 1. Setting the *depth* filter to a higher value allows also including grandchildren (depth=2) or even great-grandchildren (depth=3) etc. Allowed values are arbitrary positive numbers and *all*. *all* can be used to get nested elements of arbitrary nesting depth:

```
>>> resp_data = app_admin.get('/Documents',
...     params={'content_type': 'adhocracy_core.resources.document.IDocumentVersion',
...             'depth': 'all'}).json
>>> pprint(resp_data['data']['adhocracy_core.sheets.pool.IPool']['elements'])
[...'.../Documents/document_0000000/VERSION_0000001/'...]

>>> resp_data = app_admin.get('/Documents',
...     params={'content_type': 'adhocracy_core.resources.document.IDocumentVersion',
...             'depth': '2'}).json
>>> pprint(resp_data['data']['adhocracy_core.sheets.pool.IPool']['elements'])
[...'.../Documents/document_0000000/VERSION_0000001/'...]
```

Without specifying a deeper depth, the above query for IDocumentVersions wouldn't have found anything, since they are children of children of the pool:

```
>>> resp_data = app_admin.get('/Documents',
...     params={'content_type': 'adhocracy_core.resources.document.IDocumentVersion'
```

```
...             }).json
>>> pprint(resp_data['data']['adhocracy_core.sheets.pool.IPool']['elements'])
[]
```

If you specify *sort* you can set a <*custom*> filter (see below) that supports sorting to sort the result:

```
>>> resp_data = app_admin.get('/Documents/document_0000000',
...     params={'sort': 'name'}).json
>>> resp_data['data']['adhocracy_core.sheets.pool.IPool']['elements']
['.../Documents/document_0000000/PARAGRAPH_0000000/',...
```

Note All resource in the result set must have a value in the chosen sort filter. For example if you use *rates* you have to limit the result to resources with `adhocracy_core.sheets.rate.IRateable` sheet.

Not supported filters cannot be used for sorting:

```
>>> resp_data = app_admin.get('/Documents/document_0000000',
...                           params={'sort': 'path'}).json
>>> resp_data['errors'][0]['description']
'"path" is not one of content_type, name, text,...
```

If *reverse* is set to `True` the sorting will be reversed:

```
>>> resp_data = app_admin.get('/Documents/document_0000000',
...     params={'sort': 'name', 'reverse': True}).json
>>> resp_data['data']['adhocracy_core.sheets.pool.IPool']['elements']
['.../Documents/document_0000000/rates/',...
```

You can also specifiy a *limit* and an *offset* for pagination:

```
>>> resp_data = app_admin.get('/Documents/document_0000000',
...     params={'sort': 'name', 'limit': 1, 'offset': 0}).json
>>> resp_data['data']['adhocracy_core.sheets.pool.IPool']['elements']
['.../Documents/document_0000000/PARAGRAPH_0000000/']
```

The *count* is not affected by *limit*:

```
>>> resp_data = app_admin.get('/Documents/document_0000000',
...     params={'count': 'true', 'limit': 1}).json
>>> child_count = resp_data['data']['adhocracy_core.sheets.pool.IPool']['count']
>>> assert int(child_count) >= 10
```

The *elements* parameter allows controlling how matching element are returned. By default, 'elements' in the IPool sheet contains nothing. This corresponds to setting *elements=omit*

```
>>> resp_data = app_admin.get('/Documents/document_0000000',
...     params={'content_type': 'adhocracy_core.resources.document.IDocumentVersion',
...             'elements': 'omit'}).json
>>> pprint(resp_data['data']['adhocracy_core.sheets.pool.IPool']['elements'])
[]
```

Setting *elements=paths* will yield a response with a listing of resource paths.

```
>>> resp_data = app_admin.get('/Documents/document_0000000',
...     params={'content_type': 'adhocracy_core.resources.document.IDocumentVersion',
...             'elements': 'paths'}).json
>>> pprint(resp_data['data']['adhocracy_core.sheets.pool.IPool']['elements'])
['.../Documents/document_0000000/VERSION_0000000/',...
```

Setting *elements=content* will instead return the complete contents of all matching elements – what you would get by making a GET request on each of their paths:

```
>>> resp_data = app_admin.get('/Documents/document_0000000',
...     params={'content_type': 'adhocracy_core.resources.document.IDocumentVersion',
...             'elements': 'content'}).json
>>> pprint(resp_data['data']['adhocracy_core.sheets.pool.IPool'])
{'count': '5',
 'elements': [{'content_type': 'adhocracy_core.resources.document.IDocumentVersion',
               'data': ...
```

sheet filter resources with a specific sheet type:

```
>>> resp_data = app_admin.get('/Documents/document_0000000',
...     params={'content_type': 'adhocracy_core.sheets.document.IDocument'}).json
>>> pprint(resp_data['data']['adhocracy_core.sheets.pool.IPool']['elements'])
['.../Documents/document_0000000/VERSION_0000000/',...
```

Valid query comparables: 'eq', 'noteq', 'lt', 'le', 'gt', 'ge', 'any', 'notany'

tag is a filter that allows filtering only resources with a specific tag. Often we are only interested in the newest versions of Versionables. We can get them by setting *tag=LAST*. Let's find the latest versions of all documents:

```
>>> resp_data = app_admin.get('/Documents/document_0000000',
...     params={'content_type': 'adhocracy_core.resources.paragraph.IParagraphVersion',
...             'depth': 'all', 'tag': 'LAST'}).json
>>> pprint(resp_data['data']['adhocracy_core.sheets.pool.IPool']['elements'])
['.../Documents/document_0000000/PARAGRAPH_0000000/VERSION_0000001/',
 '.../Documents/document_0000000/PARAGRAPH_0000001/VERSION_0000001/',
 '.../Documents/document_0000000/PARAGRAPH_0000002/VERSION_0000000/']
```

Valid query comparables: 'eq', 'noteq', 'any', 'notany'

<*custom*> filter: depending on the backend configuration there are additional custom filters:

- *rate* the rate value of resources with `adhocracy_core.sheets.rate.IRate` sheet. This is mostly useful for the requests with the *aggregated* filter. Supports sorting. Valid query comparable: 'eq', 'noteq', 'lt', 'le', 'gt', 'ge', 'any', 'notany'
- *rates* the aggregated value of all `adhocracy_core.sheets.rate.IRate` resources referencing a resource with `adhocracy_core.sheets.rate.IRateable`. Only the LAST version of each rate is counted. Supports sorting. Valid query comparable: 'eq', 'noteq', 'lt', 'le', 'gt', 'ge', 'any', 'notany'
- *controversiality* controversy metrics based on rates and number of comments for all commentable and rateable resources. Supports sorting. Valid query comparable: 'eq', 'noteq', 'lt', 'le', 'gt', 'ge', 'any', 'notany'
- *name* the identifier value of all resources (last part in the resource url). This is the same value like the name in the `adhocracy_core.sheets.name.IName` sheet. Valid query comparable: 'eq', 'noteq', 'lt', 'le', 'gt', 'ge', 'any', 'notany' Supports sorting.
- *creator* the *userid* of the resource creator. This is the path of the user resource url. Valid query comparable: 'eq' Supports sorting.

```
>>> resp_data = app_admin.get('/Documents', params={'creator': '/principals/users/0000003'}).jso
>>> pprint(resp_data['data']['adhocracy_core.sheets.pool.IPool']['elements'])
['.../Documents/badges/',
 '.../Documents/document_0000000/']
```

- *item_creation_date* the the item_creation_date value of resources with `adhocracy_core.sheets.metadata.IMetadata`. Valid query comparable: 'eq', 'noteq', 'lt', 'le', 'gt', 'ge', 'any', 'notany'
- *workflow_state* workflow state, see Workflows, the state of versions is the same as for its item. Valid query comparable: 'eq', 'noteq', 'lt', 'le', 'gt', 'ge', 'any', 'notany'

- *badge* the badge names of resources with `adhocracy_core.sheets.badge.IBadgeable` sheet. Valid query comparable: 'eq', 'noteq', 'any', 'notany'
- *title* the title of resources with `adhocracy_core.sheets.title.ITitle` sheet. Valid query comparable: 'eq', 'noteq', 'lt', 'le', 'gt', 'ge', 'any', 'notany'
- *user_name* the login name of users. Valid query comparable: 'eq', 'noteq', 'lt', 'le', 'gt', 'ge', 'any', 'notany'

<package.sheets.sheet.ISheet:FieldName> filters: you can add arbitrary custom filters that refer to sheet fields with references. The key is the name of the isheet plus the field name separated by ':' The value is the wanted reference target.

First we create more paragraphs versions:

```
>>> pvrs0_path = '/Documents/document_0000000/PARAGRAPH_0000002/VERSION_0000000/'
>>> pvrs = {'content_type': 'adhocracy_core.resources.paragraph.IParagraphVersion',
...         'data': {'adhocracy_core.sheets.versions.IVersionable': {
...                  'follows': [pvrs0_path]}},
...          'root_versions': [pvrs0_path]}
>>> resp = app_admin.post('/Documents/document_0000000/PARAGRAPH_0000002',
...                       pvrs)
>>> pvrs1_path = resp.json['path']
```

Now we can search references:: def get(self, path: str, params={}, extra_headers={}) -> TestResponse:

> """Send get request to the backend rest server."""
> url = self._build_url(path)
> headers = copy(self.header)
> headers.update(extra_headers)
> resp = self.app.get(url,
>
> headers=headers, params=params, expect_errors=True)
>
> return resp

```
    >>> resp_data = app_admin.get('/Documents/document_0000000',
    ...     params={'content_type': 'adhocracy_core.resources.paragraph.IParagraphVersion',
    ...             'adhocracy_core.sheets.versions.IVersionable:follows':
    ...             '/Documents/document_0000000/PARAGRAPH_0000002/VERSION_0000000/',
    ...             'depth': 'all', 'tag': 'LAST'}).json
    >>> pprint(resp_data['data']['adhocracy_core.sheets.pool.IPool']['elements'])
    ['.../Documents/document_0000000/PARAGRAPH_0000002/VERSION_0000001/']
```

Valid query comparable: 'eq'

If the specified sheet or field doesn't exist or if the field exists but is not a reference field, the backend responds with an error:

```
>>> resp_data = app_admin.get('/Documents/document_0000000',
...     params={'adhocracy_core.sheets.NoSuchSheet:nowhere':
...             '.../Documents/document_0000000/PARAGRAPH_0000002/VERSION_0000000/'}).json
>>> resp_data['errors'][0]['description']
'No such sheet or field'
>>> resp_data['errors'][0]['location']
'querystring'

>>> resp_data = app_admin.get('/Documents/document_0000000',
...     params={'adhocracy_core.sheets.name.IName:name':
...             '.../Documents/document_0000000/kapitel2/VERSION_0000000/'}).json
>>> resp_data['errors'][0]['description']
'Not a reference node'
>>> resp_data['errors'][0]['name']
'adhocracy_core.sheets.name.IName:name'
```

You'll also get an error if you try to filter by a catalog that doesn't exist:

```
>>> resp_data = app_admin.get('/Documents/document_0000000',
...     params={'content_type': 'adhocracy_core.resources.paragraph.IParagraphVersion',
...             'foocat': 'whatever'}).json
>>> resp_data['errors'][0]['description']
'Unrecognized keys in mapping: "{\'foocat\': \'whatever\'}"'
```

aggregateby allows you to add the additional field *aggregateby* with aggregated index values of all result resources. You have to set the value to an existing filter like *aggregateby=tag*. Only index values that exist in the query result will be reported, i.e. the count reported for each value will be 1 or higher.

```
>>> resp_data = app_admin.get('/Documents/document_0000000',
...     params={'content_type': 'adhocracy_core.resources.paragraph.IParagraphVersion',
...             'depth': 'all', 'aggregateby': 'tag'}).json
>>> pprint(resp_data['data']['adhocracy_core.sheets.pool.IPool']['aggregateby'])
{'tag': {'FIRST': 3, 'LAST': 3}}
```

Asynchronous Backend-Frontend Communication Via Web Sockets

The basic idea is very simple: clients need to be able to subscribe and unsubscribe to (changes of) a given object. If an object changes and a client is subscribed to it, that client will receive a notification.

We implement this by opening one web-socket per client (= frontend) at the beginning of a session. Subscribe and unsubscribe requests are sent from client to server (= backend), and change notifications are sent from server to client. The client is responsible to handle and dispatch each particular change to the parts of the GUI that care about it.

Client Messages

Both client and server send messages in JSON format.

Client messages have the following structure:

```
{ "action": "ACTION", "resource": "RESOURCE_PATH" }
```

ACTION is one of:

- "subscribe" to start receiving updates about a resource. If the client has already sent an earlier subscribe request for that resource, the new request is silently ignored.
- "unsubscribe" to stop receiving updates about a resource. If the client is not currently subscribed to that resource, the request is silently ignored.

For example:

```
{ "action": "subscribe", "resource": "/adhocracy/prop1/" }
```

And later:

```
{ "action": "unsubscribe", "resource": "/adhocracy/prop1/" }
```

Server Messages

Responses to Client Messages

Status Confirmations If a client request was processed successfully by the server, it sends a status confirmation:

```
{ "status": "STATUS", "action": "ACTION", "resource": "RESOURCE_PATH" }
```

STATUS is either:

- "ok" if the request was processed successfully and changed the internal state of the server.
- "redundant" if the request was unnecessary since it already corresponded to internal state of the server (the client tried to subscribe to a resource it has already subscribed or to unsubscribe from a resource it hasn't subscribed).

The "action" and "resource" fields repeat the corresponding values from the client request.

Error Messages Otherwise, if the server didn't understand a request sent by the client or could not handle it, it responds with an error message:

```
{ "error": "ERROR_CODE", "details": "DETAILS" }
```

ERROR_CODE will be one of the following:

- "unknown_action" if the client asked for an action that the server doesn't understand (neither "subscribe" nor "unsubscribe"). DETAILS contains the unknown action.
- "unknown_resource" if a client specified a resource path that is unknown to the server. DETAILS contains the unknown resource path.
- "malformed_message" if the client sent a message that cannot be parsed as JSON. DETAILS contains a parsing error message.
- "invalid_json" if the client sent a message that is JSON but doesn't contain the expected information (for example, if it's a JSON array instead of a JSON object or if "action" or "resource" keys are missing or their values aren't strings). DETAILS contains a short description of the problem.
- "internal_error" if an internal error occurred at the server. DETAILS contains a short description of the problem. In an ideal world, this will never happen.

Note that it is not always possible to provide action and resource of the respective request (e.g. with "invalid_jason"). The client needs to keep track of the order in which it sends the requests, and has to associate the responses with that list. Responses (errors or not) are guaranteed to be sent to the frontend in the same order as requests are sent to the backend.

Notifications

Whenever one of the subscribed resources is changed, the server sends a message to the client. Which messages are sent depends on the type of the resource that has been subscribed.

- If resource is a Simple (e.g. a Tag):
  - If the value of the Simple has changed:

```
{ "event": "modified", "resource": "RESOURCE_PATH" }
```

  - If the Simple has been removed:

```
{ "event": "removed", "resource": "RESOURCE_PATH" }
```

    In practice this usually means that the resource has been deleted or marked hidden (see Deleting Resources).

- If resource is a Pool:
  - If some of the Pool's metadata has changed (e.g. its title):

```
{ "event": "modified", "resource": "RESOURCE_PATH" }
```

(Same as with Simples.)

- If the Pool has been removed:

```
{ "event": "removed", "resource": "RESOURCE_PATH" }
```

(Same as with Simples.)

- If a new child (sub-Pool or Item) is added to the Pool:

```
{ "event": "new_child",
  "resource": "RESOURCE_PATH",
  "child": "CHILD_RESOURCE_PATH" }
```

- If a child (sub-Pool or Item) is removed from the Pool:

```
{ "event": "removed_child",
  "resource": "RESOURCE_PATH",
  "child": "CHILD_RESOURCE_PATH" }
```

In practice this usually means that the resource has been deleted or marked as hidden (see Deleting Resources).

- If a child (sub-Pool or Item) in the Pool is modified:

```
{ "event": "modified_child",
  "resource": "RESOURCE_PATH",
  "child": "CHILD_RESOURCE_PATH" }
```

(Rationale for modify: a pool is probably rendered as a table of contents, and if the title of an object changes, the table of contents must be re-rendered.)

- If anything that lies below the pool (children, grandchildren etc.) has been added, removed, or modified:

```
{ "event": "changed_descendants", "resource": "RESOURCE_PATH" }
```

This event is sent only once per transaction and pool, even if multiple of its descendants have been modified. It tells the frontend that any *queries* previously sent to the pool should now be considered outdated, as query results can refer to grandchildren and other resources that lie below the pool, but aren't its direct children.

- If resource is an Item (e.g. a Proposal):
  - If a new sub-Item is added to the Item (e.g. a Section):

```
{ "event": "new_child",
  "resource": "RESOURCE_PATH",
  "child": "CHILD_RESOURCE_PATH" }
```

(Same as with Pool.)

  - If a new ItemVersion is added to the Item:

```
{ "event": "new_version",
  "resource": "RESOURCE_PATH",
  "version": "VERSION_RESOURCE_PATH" }
```

  - The other events sent as the same as for Pools, since all Items are also pools: "modified", "removed", "removed_child", "modified_child", "changed_descendant". The "modified_child" and "removed_child" events don't distinguish between sub-Items and ItemVersions – both are considered children.

- If resource is an ItemVersion:
  - If a backreference in the version has changed:

```
{ "event": "modified", "resource": "RESOURCE_PATH" }
```

    This happens e.g. if a successor version has been created that refers to the subscribed version as its predecessor.

    Otherwise, versions are immutable, so updated backreferences (the reverse direction for a reference from another resource to this one) are the only thing that can trigger a "modified" event.

A note about resource removal: if a resource is removed (deleted or hidden), any subscribers to it will automatically be unsubscribed, so they won't receive further updates about this resource, even if it later "revealed" (unhidden) again. Subscribers to the parent pool will receive a "new_child" or "new_version" message notifying them about the revealed resource just as if it had been newly created.

Re-Connects

There is no way to recover the state of a broken connection. The backend handles every disconnect by discarding all subscriptions.

Therefore, if the WS connection ends for any reason, the frontend must re-connect, flush its cache, and reload and re-subscribe to every resource that is still relevant.

(POSSIBLE FUTURE WORK: If WS connections prove to be unstable enough to make the above approach cause too much overhead, the backend may maintain the session for a configurable amount of time. If the frontend re-connects in that time window and presents a session key, it will receive a list of change notifications that it missed during the broken connection, and it won't have to flush its cache. The session key could either be negotiated over the WS, or there may be some token provided by substance_d / angular / somebody that can be used for this.)

Permission system

Principals

There are two types of principals users and groups (*principal*). On the technical level, roles are also called principles.

groups (set of users):

- authenticated (all authenticated users)
- system.Everyone (all authenticated and anonymous users, standard group)
- gods (initial custom group, no permission checks)
- admins (custom group)
- managers (custom group)
- ...

users:

- god (initial user)
- ...

Principals are mapped to a set of global permissions(*role*) and local permissions for a specific context (*local role*)

Roles (mapping to permissions)

Roles with example permission mapping:

- **reader: can view:** view the proposal
- **annotator: can add content metadata/annotations** add comment to the proposal add voting to the proposal add rating to the proposal add tag to the proposal
- **contributor: can add content:** add proposal
- **editor: can edit content:** edit proposal
- **creator: edit meta stuff: permissions, transition to workflow states, ...:** edit proposal change workflow state to draft change permissions
- **reviewer: do transition to specific workflow states:** change workflow state to accepted/denied
- **manager: delete, edit meta stuff: permissions, transition to workflow states, ...:** 'delete' illegal content change workflow state .. change permissions
- **admin: create an configure the participation process, manage principals:** add participation process set workflow manage principals

TODO the role definition is outdated

Mappings of principals to local roles are associate with resources and are inherited within the object hierarchy in the database. The creator is the principal who created the local context. The creator role is automatically set for a specific local context and is not inherited.

Permission mappings for roles with high priority override those with lower priorty. The order is determined by `adhocracy_core.schema.ROLE_PRINCIPALS` from left (low) right (high).

ACL (Access Control List)

List with ACEs (Access Control Entry): [<Action>, <Principal>, <Permission>]

Action: Allow | Deny Principal: UserId | group:GroupID | role:RoleID Permission: view, edit, add, ...

Every resource in the object hierarchy has a local ACL.

To check permission all ACEs are searched starting with the ACL of the requested resource, and then searching the parent's ACLs recursively. The Action of the first ACE with matching permission is returned.

Customizing

1. map users to group
2. map roles to principals
3. use workflow system to locally add roles to principals.
4. locally add *local role* s (change permission to allow others to edit)
5. map permissions to roles:
    - use only configuration for this
    - default mapping should just work for most use cases

Questions

What is the difference (conceptually) between a role and a group?

- For the basic pyramid authorization system there are only principals, no matter if you call them user/group or role. On our conceptual level we have a different semantic for user, group and role. You can see roles as groups with a default set of permissions.

is there multiple inheritance?

- no

does "inheritance" always mean "content type inheritance"?

- in this context *inheritance* means inheritance from parent to child in the object hierarchy

can groups be members of groups?

- no. but it would be easy to implement that.

Do we need workflows at all? or can we assume ACLs and roles don't change at run time?

- For the year 2014: ACL won't change during runtime and workflows are not needed

API

The user object must contain a list of roles and a list of groups she is a member of. This is necessary because the UI looks different for different roles (at the very least, we want to see a different icon for every role in the login widget).

If the FE sends a request to the BE that it has no authorization for, it will receive an error (depending on the situation either 4xx to conceal the existence of secret resources, or 3xx to explicitly deny access).

There are (at least) four approaches to implement an API that the FE can use to query BE about permissions without actually performing an access operation an observing the response:

1. OPTIONS protocol. This is expressive enough to decide if user is allowed to edit a resource or not, but not enough to inspect or edit permissions of self (by ordinary users) or other users (by admin).
2. (future work) Add permission object to meta API (CAVEAT: this makes version resources change unexpectedly).
3. (future work) Change HTTP response to contain not only the resource but also permission information in a larger JSON object.
4. (future work) New HTTP end-point for permission requests.

doctest: +ELLIPSIS # doctest: +NORMALIZE_WHITESPACE

Default permissions

Basic functional tests for roles and default permission settings.

Prerequisites

Some imports to work with rest api calls:

```
>>> from pprint import pprint
>>> from adhocracy_core.resources.document import IDocument
>>> from adhocracy_core.resources.document import IDocumentVersion
>>> from adhocracy_core.resources.organisation import IOrganisation
```

Start adhocracy app and log in some users:

```
>>> anonymous = getfixture('app_anonymous')
>>> participant = getfixture('app_participant')
>>> participant2 = getfixture('app_participant2')
>>> moderator = getfixture('app_moderator')
>>> initiator = getfixture('app_initiator')
>>> admin = getfixture('app_admin')
>>> god = getfixture('app_god')
```

Create participation process structure by god (sysadmin) (like `adhocracy_core.interfaces.IPool` subtypes)

```
>>> prop = {'content_type': 'adhocracy_core.resources.organisation.IOrganisation',
...         'data': {'adhocracy_core.sheets.name.IName': {'name': 'organisation'}}}
>>> resp = god.post('/', prop).json
>>> prop = {'content_type': 'adhocracy_core.resources.process.IProcess',
...         'data': {'adhocracy_core.sheets.name.IName': {'name': 'process'}}}
>>> resp = god.post('/organisation', prop)
```

Create participation process content by participant:

```
>>> prop = {'content_type': 'adhocracy_core.resources.document.IDocument',
...         'data': {}}
>>> resp = participant.post('/organisation/process', prop).json
>>> participant_proposal = resp['path']
>>> participant_proposal_comments = resp['path'] + 'comments'
>>> participant_proposal_rates = resp['path'] + 'rates'

>>> prop = {'content_type': 'adhocracy_core.resources.document.IDocumentVersion',
...         'data': {}}
>>> resp = participant.post(participant_proposal, prop).json
```

Create content annotations by participant:

```
>>> prop = {'content_type': 'adhocracy_core.resources.comment.IComment',
...         'data': {}}
>>> resp = participant.post(participant_proposal_comments, prop).json
>>> participant_comment = resp['path']
>>> prop = {'content_type': 'adhocracy_core.resources.comment.ICommentVersion',
...         'data': {'adhocracy_core.sheets.comment.IComment': {'comment': 'com'}}}
>>> resp = participant.post(participant_comment, prop)

>>> prop = {'content_type': 'adhocracy_core.resources.rate.IRate', 'data': {}}
>>> resp = participant.post(participant_proposal_rates, prop).json
>>> participant_rate = resp['path']
```

Anonymous

Can read resources and normal sheets:

```
>>> resp = anonymous.options('/organisation').json
>>> pprint(resp['GET']['response_body']['data'])
{...'adhocracy_core.sheets.metadata.IMetadata': {}...}
```

Cannot create comments annotations for participation process content:

```
>>> 'POST' in anonymous.options(participant_proposal_comments).json
False
```

Cannot create rate annotations for participation process content:

```
>>> 'POST' in anonymous.options(participant_proposal_rates).json
False
```

Cannot edit annotations for participation process content:

```
>>> 'POST' in anonymous.options(participant_comment).json
False
```

Cannot create process content:

```
>>> 'POST' in anonymous.options('/organisation/process').json
False
```

Cannot edit process content:

```
>>> 'POST' in anonymous.options(participant_proposal).json
False
```

Cannot create process structure:

```
>>> 'POST' in anonymous.options('/organisation/process').json
False
```

Cannot edit process structure:

```
>>> 'PUT' in anonymous.options('/organisation/process').json
False
```

Participant

Can read resources and normal sheets:

```
>>> resp = participant.options('/organisation').json
>>> pprint(resp['GET']['response_body']['data'])
{...'adhocracy_core.sheets.metadata.IMetadata': {}...}
```

Can create comments annotations for participation process content:

```
>>> resp = participant.options(participant_proposal_comments).json
>>> pprint(sorted([r['content_type'] for r in resp['POST']['request_body']]))
['adhocracy_core.resources.comment.IComment']
```

Can create rate annotations for participation process content:

```
>>> resp = participant.options(participant_proposal_rates).json
>>> pprint(sorted([r['content_type'] for r in resp['POST']['request_body']]))
['adhocracy_core.resources.rate.IRate']
```

Can edit his own annotations:

```
>>> resp = participant.options(participant_comment).json
>>> pprint(sorted([r['content_type'] for r in resp['POST']['request_body']]))
['adhocracy_core.resources.comment.ICommentVersion']
```

Cannot edit annotations:

```
>>> 'POST' in participant2.options(participant_comment).json
False
```

Can create process content:

```
>>> resp = participant.options('/organisation/process').json
>>> pprint(sorted([r['content_type'] for r in resp['POST']['request_body']]))
['adhocracy_core.resources.document.IDocument',
 'adhocracy_core.resources.document.IGeoDocument',
 'adhocracy_core.resources.external_resource.IExternalResource',
 'adhocracy_core.resources.proposal.IGeoProposal',
 'adhocracy_core.resources.proposal.IProposal',
 'adhocracy_core.resources.relation.IPolarization']
```

Can edit his own process content:

```
>>> resp = participant.options(participant_proposal).json
>>> pprint(sorted([r['content_type'] for r in resp['POST']['request_body']]))
['adhocracy_core.resources.document.IDocumentVersion',
 'adhocracy_core.resources.paragraph.IParagraph']
```

Cannot edit process content:

```
>>> 'POST' in participant2.options(participant_proposal).json
False
```

Cannot create process structure:

```
>>> 'POST' in participant.options('/organisation').json
False
```

Cannot edit process structure:

```
>>> 'PUT' in participant.options('/organisation').json
False
```

Moderator

Can create comments annotations for participation process content:

```
>>> resp = moderator.options(participant_proposal_comments).json
>>> pprint(sorted([r['content_type'] for r in resp['POST']['request_body']]))
['adhocracy_core.resources.comment.IComment']
```

#Cannot create rate annotations for participation process content:: # # >>> 'POST' in moderator.options(participant_proposal_rates).json # False

Cannot edit annotations for participation process content:

```
>>> 'POST' in moderator.options(participant_comment).json
False
```

#Cannot create process content:: # # >>> 'POST' in moderator.options('/organisation/process').json # False

Cannot edit process content:

```
>>> 'POST' in moderator.options(participant_proposal).json
False
```

Can hide process content

```
    >>> resp = moderator.options(participant_proposal).json
    >>> 'adhocracy_core.sheets.metadata.IMetadata' \
```

```
    ...     in resp['PUT']['request_body']['data']
    True
```

Initiator

Cannot create process structure organisation:

```
>>> resp = initiator.options('/organisation').json
>>> postables = sorted([r['content_type'] for r in resp['POST']['request_body']])
>>> IOrganisation.__identifier__ not in postables
True
```

Cannot edit process structure organisation (except the workflow state):

```
>>> pprint(sorted([r for r in resp['PUT']['request_body']['data']]))
['adhocracy_core.sheets.workflow.IWorkflowAssignment']
```

Can create process structure process:

```
>>> resp = initiator.options('/organisation').json
>>> pprint(sorted([r['content_type'] for r in resp['POST']['request_body']]))
['adhocracy_core.resources.document_process.IDocumentProcess',
 'adhocracy_core.resources.process.IProcess']
```

Admin

Cannot create rate annotations for participation process content:

```
#    >>> 'POST' in admin.options(participant_proposal_rates).json
#    False
```

Can edit annotations for participation process content:

```
>>> 'POST' in admin.options(participant_comment).json
True
```

Can create process structure:

```
>>> resp = admin.options('/organisation').json
>>> pprint(sorted([r['content_type'] for r in resp['POST']['request_body']]))
['adhocracy_core.resources.document_process.IDocumentProcess',
 'adhocracy_core.resources.organisation.IOrganisation',
 'adhocracy_core.resources.process.IProcess']
```

Cannot edit process structure:

```
>>> 'PUT' in admin.options('/organisation').json
True

>>> 'PUT' in admin.options('/organisation/process').json
True
```

Can create groups:

```
>>> resp = admin.options('/principals/groups').json
>>> pprint(sorted([r['content_type'] for r in resp['POST']['request_body']]))
['adhocracy_core.resources.principal.IGroup']
```

Can create users:

```
>>> resp = admin.options('/principals/users').json
>>> pprint(sorted([r['content_type'] for r in resp['POST']['request_body']]))
['adhocracy_core.resources.principal.IUser']
```

Can assign users to groups, and roles to users:

```
>>> god_user = '/principals/users/0000000'
>>> resp = admin.options(god_user).json
>>> pprint(sorted([s for s in resp['PUT']['request_body']['data']]))
[...'adhocracy_core.sheets.principal.IPasswordAuthentication',
 'adhocracy_core.sheets.principal.IPermissions',
 'adhocracy_core.sheets.principal.IUserBasic',
 'adhocracy_core.sheets.principal.IUserExtended',
 'adhocracy_core.sheets.rate.ICanRate'...]
```

Workflows

Preliminaries

Some imports to work with rest api calls:

```
>>> from pprint import pprint
>>> from adhocracy_core.resources.process import IProcess
>>> from adhocracy_core.resources.document import IDocument
```

Start adhocracy app and log in some users:

```
>>> app_god = getfixture('app_god')
>>> app_god.base_path = '/'
```

Lets create some content:

```
>>> data = {'adhocracy_core.sheets.name.IName': {'name': 'process'}}
>>> resp = app_god.post_resource('/', IProcess, data)
>>> data = {}
>>> resp = app_god.post_resource('/process', IDocument, data)
```

Workflows

Workflows are finite state machines assigned to a resource. States can set the local permissions. States can have metadata (title, date,...). State transitions can have a callable to execute arbitrary tasks.

The MetaAPI gives us the states and transitions metadata for each workflow:

```
>>> resp = app_god.get('/meta_api').json
>>> workflow = resp['workflows']['sample']
```

State metadata contains a human readable title:

```
>>> state = workflow['states']['participate']
>>> state['title']
'Participate'
```

a description:

```
>>> state['description']
'This phase is...
```

a local ACM (see doc:*glossary*) that is set when entering this state:

```
>>> state['acm']['principals']
['participant', ...
>>> state['acm']['permissions']
[['create_proposal',...
```

a hint for the frontend if displaying this state in listing should be restricted:

```
>>> state['display_only_to_roles']
[]
```

The initial workflow state:

```
>>> workflow['initial_state']
'participate'
```

Transition metadata determines the possible state flow and can provide a callable to execute arbitrary tasks:

```
>>> transition = workflow['transitions']['to_frozen']
>>> pprint(transition)
{'callback': None,
 'from_state': 'participate',
 'permission': 'do_transition',
 'to_state': 'frozen'}
```

Workflow Assignment

Pool have a WorkflowAssignment sheet to get the registered workflow:

```
>>> resp = app_god.get('/process/').json
>>> workflow_data = resp['data']['adhocracy_core.sheets.workflow.IWorkflowAssignment']
>>> workflow_data['workflow']
'sample'
```

and get the current state:

```
>>> workflow_data['workflow_state']
'participate'
```

in addition it can have custom metadata for specific workflow states:

```
>>> workflow_data['state_data']
[]
```

this metadata can be set:

```
>>> data = {'data': {'adhocracy_core.sheets.workflow.IWorkflowAssignment':  {'state_data':
...                   [{'name': 'participate', 'description': 'new',
...                     'start_date': '2015-05-26T12:40:49.638293+00:00'}]
...          }}}
>>> resp = app_god.put('/process', data)
>>> resp.status_code
200

>>> resp = app_god.get('/process').json
```

```
>>> workflow_data = resp['data']['adhocracy_core.sheets.workflow.IWorkflowAssignment']
>>> pprint(workflow_data['state_data'][0])
{'description': 'new',
 'name': 'participate',
 'start_date': '2015-05-26T12:40:49.638293+00:00'}
```

Workflow transition to states

We can also modify the state if the workflow has a suitable transition. First we check the available next states:

```
>>> resp = app_god.options('/process').json
>>> resp['PUT']['request_body']['data']['adhocracy_core.sheets.workflow.IWorkflowAssignment']
{'workflow_state': ['frozen']}
```

Then we can put the wanted next state:

```
    >>> data = {'data': {'adhocracy_core.sheets.workflow.IWorkflowAssignment': {'workflow_state': 'f
    >>> resp = app_god.put('/process', data)
    >>> resp.status_code
    200
```

```
    >>> resp = app_god.get('/process').json
    >>> resp['data']['adhocracy_core.sheets.workflow.IWorkflowAssignment']['workflow_state']
    'frozen'
```

NOTE: The available next states depend on the workflow transitions and user permissions. NOTE: To make this work every state may have only one transition to another state.

Workflow State filtering

Filtering Pools allow to search for resource with specific workflow state:

```
>>> resp_data = app_god.get('/', {'workflow_state': 'WRONG'}).json
>>> pprint(resp_data['data']['adhocracy_core.sheets.pool.IPool']['elements'])
[]
```

```
>>> resp_data = app_god.get('/', {'workflow_state': 'frozen'}).json
>>> pprint(resp_data['data']['adhocracy_core.sheets.pool.IPool']['elements'])
['.../process/']
```

Assets and Images

Introduction

Assets are files of arbitrary type that can be uploaded to and downloaded from the backend. From the viewpoint of the backend, they are just "blobs" – binary objects without any specific semantic.

Images are a subtype of assets; they can be resized and cropped to different target formats.

To manage assets, the backend has the *adhocracy_core.resources.asset.IAsset* resource type, which is a special kind of *Pool*.

Assets can be uploaded to an *asset pool*. Resources that provide an asset pool implement the *adhocracy_core.sheets.asset.IHasAssetPool* sheet, which has a single field:

asset_pool path to the asset pool where assets can be posted

The *adhocracy_core.resources.asset.IAsset* resource type provides three sheets:

- *adhocracy_core.sheets.metadata.IMetadata*: provided by all resources, automatically created and updated by the backend
- *adhocracy_core.sheets.asset.IAssetMetadata* with only readonly fields:

  mime_type the MIME type of the asset; The mime_type provided by the uploaded asset file will be sanity-check. The backend rejects the asset in case of a detectable mismatch (e.g. if the frontend posts a Word file the image mimetype "image/jpeg" is given). Not all mismatches will be detectable, e.g. different "text/" subtypes can be hard to distinguish.

  size the size of the asset (in bytes)

  filename the name of the file uploaded by the frontend (in the backend, the asset will have a different, auto-generated path)

  attached_to a list of backreferences pointing to resources that refer to the asset

- *adhocracy_core.sheets.asset.IAssetData* with a single field:

  data the binary data of the asset ("blob")

  This sheet is POST/PUT-only, see below on how to download/view the binary data.

For testing, we import the needed stuff and start the Adhocracy app:

```
>>> from pprint import pprint
>>> log = getfixture('log')
>>> admin = getfixture('app_admin_filestorage')
>>> rest_url = getfixture('rest_url')
```

And an http server to test image download:

```
>>> import os
>>> import adhocracy_core
>>> httpserver = getfixture('httpserver')
>>> base_path = adhocracy_core.__path__[0]
>>> test_image_path = os.path.join(base_path, '../', 'docs', 'test_image.png')
>>> httpserver.serve_content(open(test_image_path, 'rb').read())
>>> httpserver.headers['Content-Type'] = 'image/png'
>>> test_image_url = httpserver.url
```

We need a pool with an asset pool:

```
>>> data = {'content_type': 'adhocracy_core.resources.process.IProcess',
...         'data': {'adhocracy_core.sheets.name.IName': {
...                       'name':  'process'}}}
>>> resp_data = admin.post('/', data).json
>>> proposal_pool_path = resp_data['path']
>>> proposal_pool_path
'.../process/'
```

We can ask the pool for the location of the asset pool:

```
>>> resp_data = admin.get(proposal_pool_path).json
>>> asset_pool_path = resp_data['data'][
...          'adhocracy_core.sheets.asset.IHasAssetPool']['asset_pool']
>>> asset_pool_path
'.../process/assets/'
```

Asset Subtypes, MIME Type Validators, resizing

Note: this section is mostly backend-specific.

The generic *adhocracy_core.sheets.asset.IAssetMetadata* sheet doesn't limit the MIME type of assets. Since this is rarely desirable, it is considered abstract and cannot be instantiated – only subclasses that provide a *MIME Type Validator* can. Check out the *adhocracy_core.sheets.image* module for an example of how to do that.

To prevent confusing the frontend, you should also define a subclass of the *adhocracy_core.resources.asset.IAsset* resource type that uses the subclassed sheet instead of the generic one. See *adhocracy_core.resources.image* for an example.

In the examples that follow, we will use the subclassed example resource type and sheet.

The image will be automatically resized to all of the specified sizes. If the target aspect ratio is different from the original aspect ratio, the size that is wider/higher is cropped so that only the middle part of it remains. For example, if the original image has 1500x500 pixel and the target size is 500x250 ('detail' size in the above example), it will be scaled to 50% (750x250 pixel) and then 125 pixel to the left and 125 to the right will be cropped to reach the target size.

Uploading Assets

Assets are uploaded (POST) and updated (PUT) in a special way. Instead of sending a JSON document, the field names and values are flattened into key/value pairs that are sent as a "multipart/form-data" request. Hence, the request will have keys similar to the following:

content_type the type of the resource that shall be created, e.g. "adhocracy_core.resources.image.IImage"

data:adhocracy_core.sheets.asset.IAssetData:data the binary data of the uploaded file, as per the HTML *<input type="file" name="asset">* tag.

But note that a concrete subsheet must be used instead of the generic IAssetMetadata sheet, matching the given resource type.

For example, lets upload a little picture and create a proposal version that references it. But first we have to create a proposal:

```
>>> prop_data = {'content_type': 'adhocracy_core.resources.document.IDocument',
...              'data': {}}
>>> resp = admin.post(proposal_pool_path, prop_data)
>>> prop_path = resp.json['path']
>>> prop_v0_path = resp.json['first_version_path']
```

Now we can upload a sample picture:

```
>>> upload_files = [('data:adhocracy_core.sheets.asset.IAssetData:data',
...     'python.jpg', open('docs/_static/python.jpg', 'rb').read())]
>>> request_body = {'content_type': 'adhocracy_core.resources.image.IImage'}
>>> resp_data = admin.post(asset_pool_path, request_body,
...                        upload_files=upload_files).json
```

In response, the backend sends a JSON document with the resource type and path of the new resource (just as with other resource types). The resource name is generated randomly:

```
>>> resp_data['content_type']
'adhocracy_core.resources.image.IImage'
>>> pic_path = resp_data['path']
>>> pic_path
'.../process/assets/.../'
```

If the frontend tries to upload an asset that is overly large (more than 16 MB), the backend responds with an error. Stricter size limits may be appropriate for some asset types, but they are left to the frontend.

Downloading Assets

Assets can be downloaded in different ways:

- As a JSON document containing just the metadata
- In case of images, in one of the cropped sizes defined by the ImageSizeMapper

The frontend can retrieve the JSON metadata by GETting the resource path of the asset:

```
>>> resp_data = admin.get(pic_path).json
>>> resp_data['content_type']
'adhocracy_core.resources.image.IImage'
>>> resp_data['data']['adhocracy_core.sheets.metadata.IMetadata']['modification_date']
'20...'
>>> resp_image_meta = resp_data['data']['adhocracy_core.sheets.image.IImageMetadata']
>>> pprint(resp_image_meta)
{'attached_to': [],
 'detail': '.../process/assets/.../0000000/',
 'filename': 'python.jpg',
 'mime_type': 'image/jpeg',
 'size': '159041',
 'thumbnail': '.../process/assets/.../0000001/'}
```

The actual binary data is *not* part of that JSON document:

```
>>> 'adhocracy_core.sheets.asset.IAssetData' in resp_data['data']
False
```

In case of images, it can retrieve the image binary data in one of the predefined cropped sizes by asking for one of the keys defined by the ImageSizeMapper as child element:

```
>>> resp_data = admin.get(resp_image_meta['detail'])
>>> resp_data.content_type
'image/jpeg'
>>> detail_size = len(resp_data.body)

>>> resp_data = admin.get(resp_image_meta['thumbnail'])
>>> thumbnail_size = len(resp_data.body)
>>> thumbnail_size > 2000
True
>>> thumbnail_size < detail_size
True
```

Referring to Assets

Sheets can have fields that refer to assets of a specific type. This is done in the usual way be setting the type of the field to *Reference* (to refer to a single asset) or *UniqueReferences* (to refer to a list of assets) and defining a suitable *reftype* (e.g. with *target_isheet* = *IImageMetadata*).

Lets post a new proposal version that refers to the image:

```
>>> vers_data = {'content_type': 'adhocracy_core.resources.document.IDocumentVersion',
...              'data': {'adhocracy_core.sheets.document.IDocument': {
...                       'title': 'We need more pics!',
```

```
...                       'description': 'Or maybe just nicer ones?',
...                       'elements': []},
...                   'adhocracy_core.sheets.image.IImageReference': {
...                       'picture': pic_path},
...                   'adhocracy_core.sheets.versions.IVersionable': {
...                       'follows': [prop_v0_path]}},
...           'root_versions': [prop_v0_path]}
>>> resp = admin.post(prop_path, vers_data)
>>> prop_v1_path = resp.json['path']
>>> prop_v1_path
'...0/VERSION_0000001/'
```

If we re-download the image metadata, we see that it is now attached to the proposal version:

```
>>> resp_data = admin.get(pic_path).json
>>> resp_data['data']['adhocracy_core.sheets.image.IImageMetadata']['attached_to']
[...0/VERSION_0000001/']
```

Replacing Assets

To upload a new version of an asset, the frontend sends a PUT request with enctype=”multipart/form-data” to the asset URL. The PUT request may contain the same keys as a POST request used to create a new asset.

The *data:adhocracy_core.sheets.asset.IAssetData:data* key is required, since the only use case for a PUT request is uploading a new version of the binary data (everything else is just metadata).

If the *content_type* key is given, it *must* be identical to the current content type of the asset (changing the type of resources is generally not allowed).

Only those who have *editor* rights for an asset can PUT a replacement asset. If an image is replaced, all its cropped sizes will be automatically updated as well.

Since assets aren't versioned, the old binary “blob” will be physically and irreversibly discarded once a replacement blob is uploaded.

Lets replace the uploaded python with another one:

```
>>> upload_files = [('data:adhocracy_core.sheets.asset.IAssetData:data',
...     'python2.jpg', open('docs/_static/python2.jpg', 'rb').read())]
>>> request_body = {'content_type': 'adhocracy_core.resources.image.IImage'}
>>> resp_data = admin.put(pic_path, request_body,
...                       upload_files=upload_files).json
```

As usual, the response lists the resources affected by the transaction:

```
>>> updated_resources = resp_data['updated_resources']
>>> sorted(updated_resources)
['changed_descendants', 'created', 'modified', 'removed']
>>> resp_data['updated_resources']['modified']
['.../process/assets/.../']
>>> rest_url + '/process/' in updated_resources['changed_descendants']
True
```

If we download the image metadata again, we see that filename and size have changed accordingly:

```
>>> resp_data = admin.get(pic_path).json
>>> resp_data['data']['adhocracy_core.sheets.image.IImageMetadata']['size']
'112107'
```

Predefined scaled+cropped views are automatically updated as well:

```
>>> thumbnail = resp_data['data']['adhocracy_core.sheets.image.IImageMetadata']['thumbnail']
>>> resp_data = admin.get(thumbnail)
>>> len(resp_data.body) > 2000
True
>>> len(resp_data.body) == thumbnail_size
False
```

Deleting and Hiding Assets

Assets can be deleted or censored ("hidden") in the usual way, see Deleting Resources.

Referring to external images

The image reference sheet also allows to refer to an external image url.

```
>>> resp = admin.get(prop_v1_path).json
>>> resp['data']['adhocracy_core.sheets.image.IImageReference']['picture']
'.../process/assets/.../'
>>> resp['data']['adhocracy_core.sheets.image.IImageReference']['external_picture_url']
''
```

If we set this field

```
>>> vers_data = {'content_type': 'adhocracy_core.resources.document.IDocumentVersion',
...              'data': {'adhocracy_core.sheets.image.IImageReference': {
...                          'external_picture_url': test_image_url},
...                       'adhocracy_core.sheets.versions.IVersionable': {
...                          'follows': [prop_v1_path]}}}
>>> resp = admin.post(prop_path, vers_data)
>>> prop_v2_path = resp.json["path"]
>>> resp = admin.get(prop_v2_path).json
>>> resp['data']['adhocracy_core.sheets.image.IImageReference']['external_picture_url']
'http:/...
```

the backend downloads and references the given image url. The old picture reference is replaced with the newly created image.

```
>>> resp['data']['adhocracy_core.sheets.image.IImageReference']['picture']
'.../process/assets/.../'
```

Caching strategy

Caching is realized on different levels:

- Caching of static resources (javascript, html, css, images)
- Caching of content resources

Caching static resources

The rough idea is to cache static resources (javascript, HTML, CSS, images) forever by adding a timestamp or checksum to a query string to each static resource file, which changes each time the file has changed. This allows both browser and proxy (e.g. varnish) caching.

Two mechanisms are used for static file caching at the moment:

- HTML template files are joined together as a Javascript module, which can be used to prefill the angular template cache.
- JS and CSS files are cached through *pyramid_cachebust* and custom code in *adhocracy_frontend/__init__.py*. This adds a query string with a timestamp (frontend webserver start time) to each resource to be loaded.

  Pyramid 1.6 will contain native cachebusting functionality, so some things might be implemented differently then.

In the future, we may want to add individual checksums for each file instead of one timestamp for all to allow more fine-grained caching and decrease load after server restarts. However this would require a more sophisticated RequireJS setup, or concatenation of all Javascript files.

Caching content resources

Both backend and frontend cache content resources. Both caches are manually invalidated, triggered by the backend.

Backend resource caching (Varnish)

To be described.

Frontend resource caching

To be described.

Deleting Resources

Anyone with the *delete_resource* permission can *delete* it by using the HTTP DELETE verb. Deleted resources can not be recovered.

Deleting an existing resource is only possible for updatable resources like Simple, Pools, Items, i.e. *not* for Versions as this would mess up the version *DAG* and *not* for AssetDownloads.

The effect of deleting is as follows:

- All child/descendant resources (whose resource path includes the path of the deleted ancestor as prefix) are also deleted.
- Deleted resources are no longer listed in parent pools or search queries.
- If the frontend attempts to retrieve a *deleted* resource via GET, the backend responds with HTTP status code *404 Not Found*, just as if the resource had never existed.
- *Deleted* resources may not be referenced from other resources. If the frontend follows an outdated references it must therefore be prepared to encounter *404 Not Found* responses and deal with them appropriately (e.g. by silently skipping them or by showing an explanation such as “Comment deleted”).
- *DELETE* http method is idempotent

Hiding Resources

Apart from physically deleting a resource, it can also be marked as "hidden" using a boolean field (flag) defined in the *adhocracy_core.sheets.metadata.IMetadata* sheet. It default to false. If this sheet is omitted when POSTing new resources, the default value is used.

The usecase for deleting is that users want to withdraw some content. The usecase for hiding is that moderators want to hide unapproriate content.

Hiding an existing resource is only possible for updatable resources, i.e. *not* for Versions (which are immutable and hence don't allow PUT).

Anyone with the *hide* permission (typically granted to the manager role) can *hide* a resource by PUTting an update with *IMetadata { hidden: true }*. Likewise they can un-hide a hidden resource by PUTting an update with *IMetadata { hidden: false }*. Nobody else can change the value of the *hidden* field.

The effect of these flags is as follows:

- A positive value of the *hidden* flags is inherited by child/descendant resources (whose resource path includes the path of the hidden ancestor as prefix). Hence a *hidden* resource is one that has its own *hidden* flag set to true or that has an ancestor whose *hidden* flag is true.
- Normally, only resources that are not *hidden* are listed in parent pools and search queries.
- FIXME Not implemented yet, since the frontend doesn't yet need it: The parameter *include=hidden* can be used to include hidden resources in pool listings and other search queries. If its value is *hidden*, resources will be found regardless of the value of their *hidden* flag. However, only those with *hide* permission are ever able to view the contents of hidden resources. It's also possible to set *include=visible* to get only non-hidden resources, but it's not necessary since that is the default.
- If the frontend attempts to retrieve a *hidden* resource via GET, the backend normally responds with HTTP status code *410 Gone*. FIXME Not implemented yet, since the frontend doesn't yet need it: The frontend can override this by adding the parameter *include=hidden* to the GET request, just as in search queries. Managers (those with *hide* permission) can view hidden resources in this way. Those without this permission will still get a *410 Gone* if the resource is hidden.
- The body of the *410 Gone* is a small JSON document that explains why the resource is gone (for future use if there may be other reasons than *hidden*). It also shows who made the last change to the resource and when:

```
{ 'reason': 'hidden',
  'modified_by': '<path-to-user>',
  'modification_date': '<timestamp>'}
```

  Often the last modification will have been the hiding of the resource, but there is no guarantee that this is always the case. Especially, the resource may be marked as hidden because one of its ancestors was hidden (as that status is inherited). In that case, the person who last modified the child resource likely has nothing to do with the person who hid the ancestor resource.

- *Hidden* resources may still be referenced from other resources. If the frontend follows such references it must therefore be prepared to encounter *410 Gone* responses and deal with them appropriately (e.g. by silently skipping them or by showing an explanation such as "Comment deleted").
- FIXME Not implemented yet, since the frontend doesn't yet need it. *Hidden* resources will normally not be shown in backreferences, which are calculated on demand. The *include=hidden* parameter can be used to change that and include backreferences to hidden resources. The same restrictions apply, i.e. normal users can use this parameter to find out whether hidden backreferences exist, but they won't be able to see their contents. In any case the frontend should be prepared to deal with *410 Gone* when following backreferences in the same way as when following forward reference – even if it didn't explicitly ask to include them, they might show up due to caching.

FIXME We should extend the Meta API to expose the distinction between references and backreferences to the frontend, currently only the backend knows this.

Notes:

- This document is about deletion of resources (JSON documents). Deletion of uploaded assets (images, PDFs etc.) is outside its current scope.
- Currently, the hidden status of resources isn't treated as special by the Websocket server. So, if an resource is flagged as hidden, a "modified" event is sent to subscribers of that resource and a "modified_child" event is sent to subscribers of the parent pool. FIXME At same point in the future, we might want to change that and send "removed"/"removed_item" messages instead.

A Censorship Example

Lets put the above theory into practice by hiding (censoring) some content!

Some imports to work with rest api calls:

```
>>> from pprint import pprint
```

Start adhocracy app and log in some users:

```
>>> log = getfixture('log')
>>> anonymous = getfixture('app_anonymous')
>>> participant = getfixture('app_participant')
>>> moderator = getfixture('app_moderator')
>>> admin = getfixture('app_admin')
>>> rest_url = getfixture('rest_url')
```

Lets create some content:

```
>>> data = {'content_type': 'adhocracy_core.resources.organisation.IOrganisation',
...         'data': {'adhocracy_core.sheets.name.IName': {'name':  'pool2'}}}
>>> resp = admin.post('/', data)
>>> data = {'content_type': 'adhocracy_core.resources.process.IProcess',
...         'data': {'adhocracy_core.sheets.name.IName': {'name': 'child'}}}
>>> resp = admin.post('/pool2', data)
>>> data = {'content_type': 'adhocracy_core.resources.organisation.IOrganisation',
...         'data': {'adhocracy_core.sheets.name.IName': {'name': 'pool1'}}}
>>> resp = admin.post('/', data)
>>> data = {'content_type': 'adhocracy_core.resources.process.IProcess',
...         'data': {'adhocracy_core.sheets.name.IName': {'name': 'child'}}}
>>> resp = admin.post('/pool1', data)
>>> data = {'content_type': 'adhocracy_core.resources.document.IDocument',
...         'data': {}}
>>> resp = participant.post('/pool1/child', data)
>>> document_creator = participant.user_path
>>> document_item = resp.json['path']
>>> document_first_version = resp.json['first_version_path']
```

As expected, we can retrieve the pool and its child:

```
>>> resp = anonymous.get('/pool2').json
>>> 'data' in resp
True
>>> resp = anonymous.get('/pool2/child').json
>>> 'data' in resp
True
```

Both pools show up in the pool sheet:

```
>>> resp = anonymous.get('/',  params={'elements': 'paths'}).json
>>> pprint(sorted(resp['data']['adhocracy_core.sheets.pool.IPool']
...                           ['elements']))
['.../pool1/',.../pool2/'...
```

Lets check whether we have the permission to delete resources. The person who has created a resource (creator role) has the right to delete it:

```
>>> resp = participant.options(document_item).json
>>> 'DELETE' in resp
True
```

But they cannot hide it:

```
>>> 'PUT' not in resp
True
```

– that special right is reserved to managers:

```
>>> resp = moderator.options(document_item).json
>>> 'adhocracy_core.sheets.metadata.IMetadata' \
...     in resp['PUT']['request_body']['data']
True
```

Note: normally the sheets listed in the OPTIONS response are just mapped to empty dictionaries, the contained fields are not listed. But IMetadata is a special case since not everybody who can delete a resource can hide it. Therefore, the presence of the 'deleted' and/or 'hidden' fields indicates that PUTting a new value for this field is allowed. Once more, the corresponding value is just a stub (the empty string) and doesn't have any meaning.

Lets hide pool2:

```
>>> data = {'content_type': 'adhocracy_core.resources.pool.IBasicPool',
...         'data': {'adhocracy_core.sheets.metadata.IMetadata':
...                          {'hidden': True}}}
>>> resp = admin.put('/pool2', data).json
```

Inspecting the 'updated_resources' listing in the response, we see that pool2 was removed:

```
>>> resp['updated_resources']['removed']
['.../pool2/']
```

Now we get an error message when trying to retrieve the pool2:

```
>>> resp = anonymous.get('/pool2')
>>> resp.status_code
410
>>> resp.json['reason']
'hidden'
>>> resp.json['modified_by']
'.../principals/users/000...'
>>> 'modification_date' in resp.json
True
```

Nested resources inherit the hidden flag from their ancestors. Hence the child of the pool2 is now hidden too:

```
>>> resp = anonymous.get('/pool2/child')
>>> resp.status_code
410
>>> resp.json['reason']
'hidden'
```

Only the pool1 is still visible in the pool:

```
>>> resp = anonymous.get('/', params={'elements': 'paths'}).json
>>> rest_url + '/pool1/' in resp['data']['adhocracy_core.sheets.pool.IPool']['elements']
True
>>> rest_url + '/pool2/' in resp['data']['adhocracy_core.sheets.pool.IPool']['elements']
False
```

Sanity check: internally, the backend uses a *private_visibility* index to keep track of the visibility/deletion status of resources. But this filter is private and cannot be directly queried from the frontend:

```
>>> resp = anonymous.get('/', {'private_visibility': 'hidden'})
>>> resp.status_code
400
>>> resp.json['errors'][0]['description']
'Unrecognized keys in mapping: "{\'private_visibility\': \'hidden\'}"'
```

Lets hide an item with referenced resources. Prior to doing so, lets check that there actually is a listed version:

```
>>> resp = anonymous.get(document_item)
>>> document_creator == resp.json['data']['adhocracy_core.sheets.metadata.IMetadata']['creator']
True
```

Now we hide the item:

```
>>> data = {'content_type': 'adhocracy_core.resources.document.IDocumentItem',
...         'data': {'adhocracy_core.sheets.metadata.IMetadata':
...                      {'hidden': True}}}
>>> resp = moderator.put(document_item, data)
>>> resp.status
'200 OK'
```

The referenced user resource is affected by this change since its back references have changed. Therefore, it shows up in the list of modified resources:

```
>>> document_creator in resp.json['updated_resources']['modified']
True
```

In the end we can cleanup with some real deletion:

```
>>> resp = admin.delete("/pool1")
>>> resp.status_code
200

>>> resp.json['updated_resources']['removed']
['.../pool1...

>>> resp = admin.get("/pool1")
>>> resp.status_code
404
```

doctest: +ELLIPSIS # doctest: +NORMALIZE_WHITESPACE

Messaging

Prerequisites

Some imports to work with rest api calls:

```
>>> from adhocracy_core import testing
```

Start Adhocracy testapp

```
>>> log = getfixture('log')
>>> anonymous = getfixture('app_anonymous')
>>> participant = getfixture('app_participant')
>>> moderator = getfixture('app_moderator')
>>> admin = getfixture('app_admin')
>>> rest_url = getfixture('rest_url')
```

Message to a User

The end point '/message_user' can be used to send messages from a user to another user or a group of users:

```
>>> data = {'recipient': rest_url + '/principals/users/0000000',
...         'title': 'Important notice regarding your Adhocracy account',
...         'text': '''Everything is fine.
... Thank you for your attention and have a nice day.'''}
>>> resp = participant.post('/message_user', data)
>>> resp.status_code
200
>>> resp.text
'""'
```

The fields are all required and have the following semantics:

recipient the name of a user (*.../principals/users/...*)

title the title (subject) of the message. An installation dependent prefix or suffix may be added to the subject (e.g. "Adhocracy Notification: ...").

text

the plain-text body of the message. An installation dependent prefix and/or suffix may be added to the text.

The backend checks that the user has sufficient permissions to send the message – only users with the *message_to_user* permission (typically granted

to the contributor role) may do so. If this is the case, it sends the message per e-mail to the specified user, or to every user in the specified group.

On success, the backend just sends an empty string back to the frontend. Otherwise (e.g. if the user is not allowed to send messages), an error message is sent back.

If a user doesn't have the necessary permissions (e.g. because they are not logged in), the backend responds with 403 Forbidden:

```
>>> data= {'recipient': rest_url + '/principals/users/0000000',
...        'title': 'Important notice regarding your Adhocracy account',
...        'text': '''Everything is fine.
... Thanks you for your attention and have a nice day.'''}
>>> resp = anonymous.post('/message_user', data)
>>> resp.status_code
403
```

Message to a Group of User

FIXME In the future, it'll be possible to send messages to groups of users, using the end point '/message_group'. The end point works just like '/message_user', except that the *recipient* is a group (*.../principals/groups/...*) instead of a single user. This requires the *message_to_group* permission, typically granted to the manager role. The message is sent per e-mail to every user in the specified group. This is not yet implemented because we haven't needed it yet.

Messages to All

FIXME The following is not implemented yet. Also, it will probably be implemented via an internal messaging system rather then by sending mails to anonymous.

The end point '/message_all' can be used to send messages from a user to *everybody*:

```
>> data = {'title': 'Call for participation',
...         'text': 'With great power comes great responsibility!'}
>> resp = moderator.post('/message_all', data)
>> resp.status_code
200
>> resp.text
'""'
```

The fields are both required and have the same semantics as above.

The backend checks that the user has sufficient permissions to send the message – only users with the *message_to_all* permission (typically granted to the admin role) may do so. If this is the case, it sends the message per e-mail to *all* users registered at the Adhocracy installation, so this function should really be used with care!

The backend responds with an empty string or an error message, as above.

> ...>>> data = {'title': 'Call for participation', ... 'text': 'With great power comes great responsibility!'}
> ...>>> resp = moderator.post('./../message_all', data) ...>>> resp.text ...403

doctest: +ELLIPSIS # doctest: +NORMALIZE_WHITESPACE

Badges

Badges are resources that can be badged to mark special process content.

Prerequisites

Some imports to work with rest api calls:

```
>>> from pprint import pprint
>>> from adhocracy_core.resources.document import IDocument
>>> from adhocracy_core.resources.document import IDocumentVersion
```

Start adhocracy app and log in some users:

```
>>> participant = getfixture('app_participant')
>>> moderator = getfixture('app_moderator')
>>> admin = getfixture('app_admin')
>>> log = getfixture('log')
```

Create participation process structure/content to get started:

```
>>> prop = {'content_type': 'adhocracy_core.resources.organisation.IOrganisation',
...          'data': {'adhocracy_core.sheets.name.IName': {'name': 'organisation'}}}
>>> resp = admin.post('/', prop).json
>>> prop = {'content_type': 'adhocracy_core.resources.process.IProcess',
...          'data': {'adhocracy_core.sheets.name.IName': {'name': 'process'}}}
>>> resp = admin.post('/organisation', prop)

>>> prop = {'content_type': 'adhocracy_core.resources.document.IDocument',
...          'data': {}}
>>> resp = participant.post('/organisation/process', prop).json
>>> proposal_item = resp['path']
>>> proposal_version = resp['first_version_path']

>>> prop = {'content_type': 'adhocracy_core.resources.document.IDocument',
...          'data': {}}
>>> resp = participant.post('/organisation/process', prop).json
>>> proposal2_version = resp['first_version_path']
```

Create Badge

Badges can be created in *badges* pools. The IHasBadgesPool sheet of the process gives us the right pool:

```
>>> resp = moderator.get('/organisation/process').json
>>> badges_pool = resp['data']['adhocracy_core.sheets.badge.IHasBadgesPool']['badges_pool']
```

Now we can create a Badge:

```
>>> prop = {'content_type': 'adhocracy_core.resources.badge.IBadge',
...          'data': {'adhocracy_core.sheets.name.IName': {'name': 'badge1'},
...                   'adhocracy_core.sheets.title.ITitle': {'title': 'Badge 1'},
...                   'adhocracy_core.sheets.description.IDescription': {'description': 'This is 1'},
...                   },
...          }
>>> resp = moderator.post(badges_pool, prop).json
>>> badge = resp['path']
```

To add a badge to a badge group we first create the group:

```
>>> prop = {'content_type': 'adhocracy_core.resources.badge.IBadgeGroup',
...          'data': {'adhocracy_core.sheets.name.IName': {'name': 'group1'},
...                   },
...          }
>>> resp = moderator.post(badges_pool, prop).json
>>> group = resp['path']
```

then create the badge inside this group:

```
>>> prop = {'content_type': 'adhocracy_core.resources.badge.IBadge',
...          'data': {'adhocracy_core.sheets.name.IName': {'name': 'badge1'},
...                   },
...          }
>>> resp = moderator.post(group, prop).json
>>> badge_with_group = resp['path']
```

The badge groups hierarchy is also shown with the badge sheet:

```
>>> resp = moderator.get(badge_with_group).json
>>> resp['data']['adhocracy_core.sheets.badge.IBadge']
{'groups': [.../group1/']}
```

Assign badges to process content

To assign badges we have to post a badge assignment between user, content and badge.

First we need the pool to post badge assignments to:

```
>>> resp = moderator.get(proposal_version).json
>>> post_pool = resp['data']['adhocracy_core.sheets.badge.IBadgeable']['post_pool']
```

To get assignable badges we send an options request to this post pool:

```
>>> resp = moderator.options(post_pool).json
>>> resp['POST']['request_body'][0]['data']['adhocracy_core.sheets.badge.IBadgeAssignment']['badge']
[.../process/badges/badge1/',.../process/badges/group1/badge1/']
```

The user is typically the current logged in user:

```
>>> user = moderator.user_path
```

Now we can post the assignment to a proposal version:

```
>>> prop = {'content_type': 'adhocracy_core.resources.badge.IBadgeAssignment',
...         'data': {'adhocracy_core.sheets.badge.IBadgeAssignment':
...                       {'subject': user,
...                        'badge': badge,
...                        'object': proposal_version}
...          }}
>>> resp = moderator.post(post_pool, prop)
>>> resp.status_code
200
```

or proposal item:

```
>>> prop = {'content_type': 'adhocracy_core.resources.badge.IBadgeAssignment',
...         'data': {'adhocracy_core.sheets.badge.IBadgeAssignment':
...                       {'subject': user,
...                        'badge': badge,
...                        'object': proposal_item}
...          }}
>>> resp = moderator.post(post_pool, prop)
>>> resp.status_code
200
```

Now the badged content shows the back reference targeting the badge assignment:

```
>>> resp = participant.get(proposal_version).json
>>> resp['data']['adhocracy_core.sheets.badge.IBadgeable']['assignments']
[...0/badge_assignments/0000000/']
```

It is not possible to assign twice the same badge:

```
>>> prop = {'content_type': 'adhocracy_core.resources.badge.IBadgeAssignment',
...         'data': {'adhocracy_core.sheets.badge.IBadgeAssignment':
...                       {'subject': user,
...                        'badge': badge,
```

```
...                         'object': proposal_version}
...           }}
>>> resp = moderator.post(post_pool, prop).json
>>> resp['errors'][0]['description']
'Badge already assigned'
```

We can also use the filtering pool api to search for content with specific badge names:

```
>>> prop = {'badge': 'badge1',
...         'depth': 'all'}
>>> resp = moderator.get('/organisation/process', params=prop).json
>>> resp['data']['adhocracy_core.sheets.pool.IPool']['elements']
['...document_0000000/',...document_0000000/VERSION_0000000/']
```

In addition we can search for versions that have an item with a specific badge:

```
>>> prop = {'item_badge': 'badge1',
...         'depth': 'all'}
>>> resp = moderator.get('/organisation/process', params=prop).json
>>> resp['data']['adhocracy_core.sheets.pool.IPool']['elements']
['...0/']
```

PostPool and Assignable validation

If we use the wrong post_pool we get an error:

```
>>> resp = moderator.get(proposal2_version).json
>>> wrong_post_pool = resp['data']['adhocracy_core.sheets.badge.IBadgeable']['post_pool']

>>> prop = {'content_type': 'adhocracy_core.resources.badge.IBadgeAssignment',
...         'data': {'adhocracy_core.sheets.badge.IBadgeAssignment':
...                       {'subject': user,
...                        'badge': badge,
...                        'object': proposal_version}
...          }}
>>> resp = moderator.post(wrong_post_pool, prop).json
>>> resp
{...'You can only add references inside ...0/badge_assignments...
```

TODO add validators for subject (assignable?) TODO add options to make badges from one group exclusive

User Badges

Badges can be assigned to users the same way as process content. The principals pool gives us the badges pool:

```
>>> resp = moderator.get('/principals').json
>>> badges_pool = resp['data']['adhocracy_core.sheets.badge.IHasBadgesPool']['badges_pool']
```

There the admin can create badges:

```
>>> prop = {'content_type': 'adhocracy_core.resources.badge.IBadge',
...         'data': {'adhocracy_core.sheets.name.IName': {'name': 'userbadge'},
...                  },
...         }
>>> resp = admin.post(badges_pool, prop).json
>>> badge = resp['path']
```

The user gives us the badge assignment post pool:

```
>>> user_with_badge = moderator.user_path
>>> resp = moderator.get(user_with_badge).json
>>> post_pool = resp['data']['adhocracy_core.sheets.badge.IBadgeable']['post_pool']
```

to create badge assignments:

```
>>> user = admin.user_path
>>> prop = {'content_type': 'adhocracy_core.resources.badge.IBadgeAssignment',
...         'data': {'adhocracy_core.sheets.badge.IBadgeAssignment':
...                         {'subject': user,
...                          'badge': badge,
...                          'object': user_with_badge}
...           }}
>>> resp = admin.post(post_pool, prop).json
```

Now the badged content shows the back reference targeting the badge assignment:

```
>>> resp = participant.get(user_with_badge).json
>>> resp['data']['adhocracy_core.sheets.badge.IBadgeable']['assignments']
[.../users/badge_assignments/0000000/']
```

Frontend

Overview

The adhocracy 3 frontend is an opinionated web framework based on angular.js and written in TypeScript. Its main goal is to provide all the building blocks that developers need to implement participation processes of all kinds. The user interface should be consistent and recognizable while providing enough flexibility for a wide range of target audiences as well as branding.

Angular (web framework)

AngularJS is an open source JavaScript framework mainly developed by Google.

In contrast to traditional frameworks like jQuery, you do not directly interact with the DOM. Instead, you only interact with data structures in JavaScript. The DOM is *bound* to these data structures and updates automatically.

Other notable features are services (singleton objects), dependency injection (a way to decouple your code), and *promise based APIs* (as opposed to callbacks).

Angular is somewhat similar to other client-side rendering frameworks like ember.js or react.

TypeScript (programming language)

TypeScript is an open-source language mainly developed by Microsoft.

It contains many features of upcoming JavaScript versions (ES6/7) as well as static typing while staying compatible with ES5. Notable features include:

- static type checking
- module system
- classes

- arrow functions
- default values for function arguments

In order to use static type checking with non-TypeScript code, the project DefinitelyTyped provides type definitions for many popular JavaScript libraries.

TypeScript is similar to CoffeeScript in that it compiles to JavaScript. It is similar to Babel in that it backports many future JavaScript features.

Sass / Compass (CSS preprocessor)

Ease writing CSS with support for nesting, mixins and variables and various helper tools.

RequireJS (module loader)

Loads javascript modules and bundles javascript/css files. You can find a comparsion with other (younger) projects here.

Backend API

Sheets

Resources in adhocracy are composed of *sheets*. Each sheet describes one aspect of the resource, e.g. that it has a title (`ITitle`) or that it can be rated (`IRateable`) or commented on (`ICommentable`).

Which sheets are available or required on a resource is defined by their *content type*. You can get a list of all content types and their sheets from the *Meta-API*.

Pool queries

All resources that can contain other resources have an `IPool` sheet and are generally refered to as pools. Pools can be *queried* for their contents. The results can be sorted and filtered by several conditions, some of which depend on the available sheets. Example: In order to get all comments that refer to resource `/my/resource`, you may use the following (simplified) query string:

```
?depth=all&content_type=IComment&IComment:refers_to=/my/resource&sort=creation_date
```

Deletion

In order to always allow users to recover from accidental actions, the backend does not physically delete content. This is why the `DELETE` HTTP method is generally not available. Instead, there are *ways to mark the content as deleted* so it is no longer accessible.

Batch requests

Sometimes you may want to change data in the backend, but it is not possible to do it in a single request. Doing it in two or more requests however has the risk that you end up with an inconsistent state because one of the later requests fails.

For this case, the adhocracy backend allows to encode several requests in a single *batch request* that is then processed in a single database transaction. This way the whole batch is rolled back if a single request fails.

Permissions

The backend has a sophisticated *permission system* with roles, groups and local permissions. The frontend ignores all this and is only interested in the result: Is the current user allowed to do this action? All information required for that can be obtained by sending an *OPTIONS request* to the relevant backend endpoint.

Websockets

The backend uses *websockets* to notify the frontend whenever a resource changes. This can be used to update the UI automatically.

Note: Updating the UI automatically is possible, but not always the right thing to do. If everything is changing all the time, users will only get confused.

Note: Websocket notifications are also used to do cache invalidation in the frontend. So if the websocket connection fails, the frontend stops caching completely and may get slow.

The build directory

Adhocracy is split into several python packages. For a specific project there are typically four packages:

Core		Customization	
Backend	Frontend	Backend	Frontend
adhocracy_core	adhocracy_frontend	adhocracy_foo	foo

When bin/buildout is run, the `static` directories from both frontend packages are merged into a single one called `build` that is located next to `static` in the customization package. Merging in this case means that files from both directories are symlinked into the build directory. If a file exists in both packages, the one from the customization overwrites the one from core.

Note: This mechanism allows the customization to replace any file from core. However, this is strongly discouraged in most cases as it is hard to maintain the overwrites.

Independent widgets

In order to provide reusable widgets, we try to make our directives as independent as possible. In practice that means that we always isolate the directive scope (with few exceptions) and pass a minimal number of parameters.

For example, a proposal directive would only get the `path` of a resource instead of relying on some parent directive to fetch it first. This of course means that many directives may trigger the same HTTP requests. This is mitigated by a caching system that is built into the `adhHttp` service.

Modules

Module Systems

The adhocracy frontend is based mainly on two technologies: TypeScript and angular.js. Both have their own module system. These two systems are very different.

TypeScript Modules

In TypeScript, each file is a module (TypeScript does in fact offer two module systems. We use external modules). A module `example.ts` can be imported like this:

```
import * as Example from "./example";
```

Static imports have the benefit of allowing to check for the existence of modules and for circular imports at compile time. But be aware that this is only true if you actually use the module for more than type-checking. If not, the import will be stripped after that step and no further checks will be done.

An important bit is that these imports are responsible for actually loading the required files in the browser. Without a non-stripped import, the module will just not be available.

Angular Modules

Angular modules are the place where services, directives and filters are registered. When a module depends on another one that means that it imports all of its services, directives and filters.

A module `example` that depends on module `dependency` is created like this:

```
var exampleModule = angular.module("example", ["dependency"]);
```

This mechanism happens at runtime and therefore missing dependencies and circular imports can only be detected at runtime.

How we use it

Packages

In adhocracy we create what we call a *package* for every reusable feature. A package may contain services, directives and filters. Each package has its own folder in `Packages/`.

A package may contain arbitrary TypeScript modules. These must not import any other TypeScript modules except for type-checking. There are few exceptions to that rules, e.g. `Util`.

In addition, there must be a TypeScript module named `Module.ts` that defines an angular module by exporting a variable `moduleName` and a function `register`. `moduleName` contains the name that should be used for this module. By convention the module name is the package name in camel case prefixed with 'adh'. `register` takes angular as a first argument and registers the module with all of its services and directives.

`Module.ts` must also take care of importing additional code, both from other packages and from third party libraries. For other packages that can be done by importing `Module.ts` from that package and adding it as a angular module dependency using its `moduleName`. This way it is also made sure that requirejs will actually load the code.

Resources

The backend defines a set of resource and sheet types and exposes them in a meta API. Matching TypeScript classes will be generated by the `mkResources.ts` script that is automatically run by buildout. The resulting code can be found in a top level folder called `Resources/`.

Further Reading

- Angular Best Practice for App Structure
- An AngularJS Style Guide for Closure Users at Google

Routing

As Adhocracy 3 is a single page application, the routing is done in the frontend. Unfortunately, the obvious choice angular-route does not meet our requirements so we wrote our own router. This document gives a brief overview over the routing system.

Basics

Adhocracy 3 consists of a backend with a rest API and a frontend with a server part and a client part (JavaScript). The frontend server serves static files for the most part. The frontend client is a complex application that manages most of the routing.

Routing in Adhocracy 3 relies on angular's `$location` service which in turn uses the browser's history API (or a fallback if that is not available). This allows us to change the URL from JavaScript code without triggering a browser redirect.

Server

The routing is done on the client. However, the server still needs to serve the frontend code on all valid URLs. The existing rules already cover quite a lot. If you want to add another rule (basically if you want to add another *area*, see below) you need to edit `src/adhocracy_frontend/adhocracy_frontend/__init__.py` or the corresponding file in a customization package. Simply add a line like this to `includeme()`:

```
add_frontend_route(config, name, rule)
```

where `name` is the name of this route and `rule` is a rule that the URL will have to match. See pyramid add_route documentation for further details.

Top Level State

The `adhTopLevelState` service provides the infrastructure for routing in the Adhocracy 3 frontend. It manages an internal flat object called the *state* and syncs it with the URL. So whenever the URL changes, the state is updated and vice versa. The service provides an interface to get, set and watch the state.

The service also defines a template that is rendered by the `adhView` directive.

In some respects, `adhTopLevelState` is similar to angular-route but much more flexible.

Areas

The actual work of syncing URL and state as well as defining the template is not done by `adhTopLevelState` itself but by so called *areas*. This allows us to have very different routing methods in different areas.

The current area is selected by the first part of the URL path. So if the URL is `http://example.com/foo/bar/1?key=value`, the area is `foo`.

Each area defines a function `route()` to convert the URL to a state object, `reverse()` to convert a state object to an URL, and a template.

Currently, we use some simple areas for login related functionality, one area for embedding and (the most important) one for resources.

Resource Area

The most important area is the resource area. URLs in this area are directly derived from backend paths. If a resource has the path `/some/path` in the backend, the corresponding route in the frontend would be `/r/some/path`.

Much like `adhTopLevelState`, the resource area only provides an infrastructure. You need to configure what the state should be based on *resource type*, *view*, *process type* and *embed context*.

While the resource type can be deduced from the path, view, process type and embed context are new concepts. *Process type* and *embed context* will be discussed later in this document.

Views allow to have multiple routes to a single resource. So while `/r/some/path` might point to a detail view of the resource, `/r/some/path/@edit` might point to a form where the resource can be edited. Note that the view is prefixed with an `@`.

Process

While the previous concepts were frontend specific, *processes* also exist in the backend. A process contains a resource subtree and defines roles and permissions for that subtree.

In the frontend it also defines which template should be used. The directive `adhProcessView` is used to render that template. It is currently used in the resource area's content space.

Note: The UI for a process is sometimes refered to as a *workbench*.

Embed Context

For a general discussion of embedding, see Embedding Python in Another Application.

When entering adhocracy through the embed area, an embed context is defined. This changes the frontend's behaviour: for example there might be a different header or different routes.

Conclusion

So here is a rough overview of what happens when I enter `/r/some/path` into my browser address bar:

1. The server serves an HTML bootstrap page.
2. `adhTopLevelState` notices a change to the URL and starts processing it. From the first part of the URL (`/r/`) it knows that it has to use the resource area.

3. The resource area converts the URL to a flat state object. This object contains information about the process type.
4. `adhView` renders the area template.
5. `adhProcessView` renders the process template.

Note that everything except for the first step also happens when I click on a link within Adhocracy.

Translation

Whenever you want to do translation of Adhocracy content, we strongly encorage you to do the following steps:

1. if necessary: mark the content you need to translate as translatable,
2. extract translatable strings for a translation session,
3. translate the strings in transifex,
4. if necessary: update the change_german_salutation script.

1. Markup Translatable Strings

For translations in the frontend we use angular-translate. It offers several ways of marking a string as translatable, but we mainly use the `translate` filter in templates:

```
<a href="#">{{ "TR__LOGIN" | translate }}</a>
<a href="#">{{ "TR__USERS_PROPOSALS" | translate:{name: adhUser.name} }}</a>
```

In few cases it is not possible to do translation in the template. In these cases you can also use the `$translate` service. Note that this service returns a promise:

```
$translate("TR__LOGIN").then((translated) => {
    ...
});
```

In our code we do not use actual human language. Instead, we use technical strings (uppercase, with underscores, prefixed with `TR__`).

2. String Extraction

angular-translate does not provide a script to extract translatable strings. So we hacked our own:

- `bin/ad_extract_messages` will output a list of all translatable strings in the current git subtree.
- You can pipe the output into `bin/ad_merge_messages` to update the JSON files. It takes two parameters: The package name and a filename prefix, e.g.:

```
    bin/ad_merge_messages adhocracy_frontend core
```

So in order to update the translation files in the "foo" package, you can use the following command:

```
cd src/foo/
../../bin/ad_extract_messages | ../../bin/ad_merge_messages foo foo
```

Note: Both scripts are not of good quality and may not cover all edge-cases.

3. Translation with Transifex

We generally use transifex for translation. Note that new strings in the code are not automatically displayed in transifex, and updates in transifex are not instantly reflected in either the code or any running platform.

Before you push local changes to transifex, please make sure that you will not overwrite any translations on transifex. This can be done in several ways, one of which is the following. Assuming you want to update the project called foo:

```
$ cd src/foo/
$ tx pull -a --force
$ git cola
```

In git cola, for each line you can decide on the newer version.

Then, in order to push changes from the foo project code to transifex, the transifex-client can be used like this:

```
$ cd src/foo/
$ tx push -a
```

In order to pull changes from transifex into the foo project's code, the transifex-client can be used like this:

```
$ cd src/foo/
$ tx pull -a
```

Note: The configuration for transifex-client is stored in `src/{package_name}/.tx/config`.

Warning: Transifex requires us to specify a "source language" (currently English). This has the benefit that translators do not need to handle technical strings. But it also has several disadvantages:

- The source language can not be translated on transifex.
- All translations will be lost when the string in the source language is changed.
- Downloaded translation files will contain all keys. Any key that does not have a translation will have the translation from the source language as a value.

An alternative could be to have a fake source language that simply has the technical strings themselves as translations and an exotic locale.

4. German Du/Sie

Adhocracy is currently used mostly in Germany, i.e. in German language. Unfortunately, there are two variants of German, a formal (Sie) and an informal (Du) one.

All translations should use the informal variant. When necessary, we use the script `bin/change_german_salutation` to convert informal translations to formal ones. Note that you will need to extend that script whenever the translation changes. The common workflow for this is: Iteratively run the script, check the output and add new rules until everything is fine.

Services

This section gives a brief introduction to the most important services.

adhConfig Provides access to the configuration. Basically identical with `/config.json` on the frontend.

adhHttp All HTTP communication should go through this services. Apart from caching it also contains abstractions for some API features such as *batch requests* or *OPTIONS requests*.

adhPermissions This service wraps `adhHttp.options()` and updates the result whenever the resource path changes.

adhTopLevelState Basic infrastructure for routing. You will need to use this service if you want to know things about the current route, e.g. which process you are in.

adhResourceArea Implements some more concrete aspects for routing. You will mostly use this to configure routes for individual resource types.

Project Specific

B-PLAN API

This document specifies the API used to a manage B-Plan processes in the a3 platform.

Process

The full process of creation and management of a B-Plan:

1. Create a B-Plan process
2. Edit an unpublished B-Plan
3. Get the HTML embed code and external URL to integrate the B-Plan
4. Make a B-Plan accessible
5. Edit a published B-Plan

Data fields

The following data needs to be provided to create a B-Plan:

- *organization*: The organization the B-Plan belongs to
- *bplan_number*: Number of the BPlan
- *bplan_name*: Could be the same as bplan_number
- *bplan_titile*: Could be the same as bplan_number
- *participation_kind*: Kind of participation, e.g. 'öffentliche Auslegung'
- *office_worker_email*: Email address to receive the B-Plan statements
- *short_description*: Teaser text
- *description*: Full description of the BPlan
- *external_picture_url*: External URL to the BPlan picture
- *picture_description*: Picture copyright notice
- *start_date*: Start time of the praticipation phase
- *end_date*: End of the participation phase, i.e. start time of the closed phase
- *external_url*: URL of the page where the BPlan process is embedded

Workflows

A B-Plan process transits the following workflows:

1. *draft*: Initial worflow state used for editing, the B-Plan is not public
2. *announce*: The B-Plan information is accessible, but no statements can be send
3. *participate*: B-Plan participation is active, statements can be issued
4. *closed*: The B-Plan is not accessible anymore

The transition from the *draft* state to the *announce* state has to be done via an API call. The further transitions to *participate* and *closed* are performed automatically by the a3 platform depending on the provided *start_date* and *end_date*.

API Calls

The following API calls are required to implement the process:

- login
- create a B-Plan process
- get the B-Plan workflow state
- make the B-Plan accessible
- get the B-Plan embed HTML snippet and external URL
- edit a B-Plan process

Initialization

For the example API calls an organisation "orga" is created. The organization for the B-Plan needs to exist beforehand in the a3 platform.

```
>>> from webtest import TestApp
>>> rest_url = 'http://localhost/api'
>>> app_router = getfixture('app_router')
>>> testapp = TestApp(app_router)
>>> resp = testapp.post_json(rest_url + '/login_username',
...                          {'name': 'admin', 'password': 'password'})
>>> admin_header = {'X-User-Token': resp.json['user_token']}
```

```
>>> data = {'content_type':
...                'adhocracy_core.resources.organisation.IOrganisation',
...          'data': {
...              'adhocracy_core.sheets.name.IName':
...                  {'name': 'orga'}
...          }}
>>> resp = testapp.post_json(rest_url + '/', data, headers=admin_header)
```

A working image url is needed to test referencing external images.

```
>>> import os
>>> import adhocracy_core
>>> httpserver = getfixture('httpserver')
>>> base_path = adhocracy_core.__path__[0]
>>> test_image_path = os.path.join(base_path, '../', 'docs', 'test_image.png')
>>> httpserver.serve_content(open(test_image_path, 'rb').read())
```

```
>>> httpserver.headers['Content-Type'] = 'image/png'
>>> test_image_url = httpserver.url
```

Login:

```
>>> data = {'name': 'god',
...         'password': 'password'}
>>> resp = testapp.post_json(rest_url + '/login_username', data)
>>> resp.status_code
200
>>> user_token = resp.json['user_token']
>>> auth_header = {'X-User-Token': user_token}
```

To login post the username and password. The 'user_token' from the response is used in a HTTP custom header in the following communication. The username here is just an example, please use your credentials.

Create a new bplan process:

```
>>> data = {'content_type': 'adhocracy_meinberlin.resources.bplan.IProcess',
...         'data': {
...             'adhocracy_core.sheets.name.IName':
...                 {'name': '1-23'},
...             'adhocracy_core.sheets.title.ITitle':
...                 {'title': 'Bplan 1-23'},
...             'adhocracy_meinberlin.sheets.bplan.IProcessSettings':
...                 {'plan_number': '1-23',
...                  'participation_kind': 'öffentliche Auslegung'},
...             'adhocracy_meinberlin.sheets.bplan.IProcessPrivateSettings':
...                 {'office_worker_email': 'moderator@bplan.de'},
...             'adhocracy_core.sheets.description.IDescription':
...                 {'description': 'Full description',
...                  'short_description':'Teaser text'},
...             'adhocracy_core.sheets.image.IImageReference':
...                 {'picture_description': 'copyright notice',
...                  'external_picture_url': test_image_url},
...             'adhocracy_core.sheets.workflow.IWorkflowAssignment':
...                 {'state_data':
...                  [{'name': 'participate', 'description': '',
...                    'start_date': '2016-03-01T12:00:09'},
...                   {'name': 'closed', 'description': '',
...                    'start_date': '2016-03-01T12:00:09'}]},
...             'adhocracy_core.sheets.embed.IEmbed':
...                 {'external_url': 'http://embedding-url.com'}
...             }}
>>> resp = testapp.post_json(rest_url + '/orga/', data, headers=auth_header)
>>> resp.status_code
200
```

The creation of a bplan consist of a post request containing all the required fields.

Get the workflow state:

```
>>> resp = testapp.get(rest_url + '/orga/1-23/', headers=auth_header)
>>> resp.status_code
200
>>> resp.json['data'] \
...     ['adhocracy_core.sheets.workflow.IWorkflowAssignment'] \
...     ['workflow_state']
'draft'
```

Perform a workflow state transition:

```
>>> data = {'content_type': 'adhocracy_meinberlin.resources.bplan.IProcess',
...         'data': {
...             'adhocracy_core.sheets.workflow.IWorkflowAssignment':
...                 {'workflow_state': 'announce'}
...             }}
>>> resp = testapp.put_json(rest_url + '/orga/1-23/', data, headers=auth_header)
>>> resp.status_code
200
>>> resp = testapp.get(rest_url + '/orga/1-23/', headers=auth_header)
>>> resp.status_code
200
>>> resp.json['data'] \
...     ['adhocracy_core.sheets.workflow.IWorkflowAssignment'] \
...     ['workflow_state']
'announce'
```

Get the HTML code snipped to embed the bplan and its external URL:

```
>>> resp = testapp.get(rest_url + '/orga/1-23/', headers=auth_header)
>>> resp.status_code
200
>>> embed_code = (resp.json['data'] \
...     ['adhocracy_core.sheets.embed.IEmbed'] \
...     ['embed_code'])
>>> print(embed_code)

<script src="http://localhost:6551/AdhocracySDK.js"></script>
<script> adhocracy.init('http://localhost:6551',
                        function(adhocracy) {adhocracy.embed('.adhocracy_marker');
                        });
</script>
<div class="adhocracy_marker"
     data-path="http://localhost/api/orga/1-23/"
     data-widget="mein-berlin-bplaene-proposal-embed"
     data-autoresize="false"
     data-locale="en"
     data-autourl="false"
     data-initial-url=""
     data-nocenter="true"
     data-noheader="true"
     style="height: 650px">
</div>
```

Edit a B-Plan process:

To edit a B-Plan the fields set in the initial post requests can be used.

E.g. Changing the description:

```
>>> data = {'content_type': 'adhocracy_meinberlin.resources.bplan.IProcess',
...         'data': {
...             'adhocracy_core.sheets.description.IDescription':
...                 {'description': 'Updated description'}
...             }}
>>> resp = testapp.put_json(rest_url + '/orga/1-23', data, headers=auth_header)
>>> resp.status_code
200
```

E.g. Changing the participation dates:

```
>>> data = {'content_type': 'adhocracy_meinberlin.resources.bplan.IProcess',
...         'data': {
...             'adhocracy_core.sheets.workflow.IWorkflowAssignment':
...                 {'state_data':
...                  [{'name': 'participate', 'description': 'test',
...                    'start_date': '2016-03-03T12:00:09'},
...                   {'name': 'closed', 'description': 'test',
...                    'start_date': '2016-05-01T12:00:09'}]}}}
>>> resp = testapp.put_json(rest_url + '/orga/1-23', data, headers=auth_header)
>>> resp.status_code
200
```

Legacy concepts

This documentation is kept for historical reasons.

Concept: The Supergraph

Our Terminology

node Building blocks of Adhocracy participation processes. Examples: "document", "user", "likes", "vote", etc. Nodes can connect to other nodes using references (see below). They are implemented as python objects.

reference A reference connects a source node to a target node. References have a specific label, like: "contains", "has_author", etc. There a two basic types:

- `reference-to-one`: References which exist only once
- `reference-to-many`: References exists zero to many times

What constitutes a node and what constitutes a reference is a design decision made on the content design level.

It is often convenient to talk about nodes as `vertices` and references as `edges` in a graph.

References are implemented as python attributes containing object references. (The term "reference" exists both on the data model level and on the implementation level.) A reference can either connect to a target node, or to a container of target nodes (list, set, ...).

essence Some references are "essential" to a source node, and some are not. The essence of a node is the total of all nodes in the transitive hull of all essential references (i.e. all target nodes of essential references, and all targets of the essential references of those target nodes, and so on).

The concept of essence is important for change management and will be discussed in detail below. The idea is that if a node Y is in the essence of node X, and Y changes, X "naturally" changes with Y.

dependents The inverse essence of a node up to reflexivity: A node X is a dependent of Y if Y is in the essence of X, but not X itself.

content node A node that is self-contained, i.e. it has no outgoing references. (Content nodes are the leaves of the reference graph.)

follows Change management is implemented by `follows` edges between nodes. A node that changes in fact is copied into a new version that follows the previous version. `follows` edges are NOT references (neither on the design level nor on the implementation level).

`head` A node without outgoing `follows` edges

`fork` A node with more than one outgoing `follows` edges.

`merge` A node with more than one incomming `follows` edges.

`relation` A pattern of references and nodes that have a certain meaning. (See below for examples.)

Non-Mutability

Note: This section describes rules and properties that we define for adhocracy core. They are not enforced by the underlying db.

The properties contained in a node don't change after creation of the node. The same goes for properties of references. Also, created nodes and references don't ever get deleted.

The set of outgoing references from a node is not allowed to change. The set of incoming references can change. This also means that a reference from A to B implies that A is younger or equally old than B.

Some Intuition

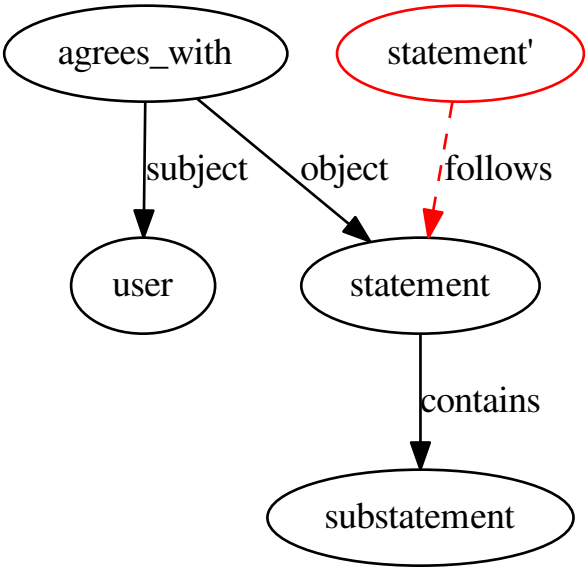
Imagine you have a node, transitively follow all its outgoing references and collect all the resulting nodes. This gives you the node's `essence`. Usually, this will result in a tree of nodes. A reference means (as defined above) that the referenced nodes are an "essential part" of the referencing node. So our tree of nodes is something like a deep-copy and recursively includes all the essential parts of our root node.

(Cycles using references are also allowed, so you might not get a tree, but a sub-graph. This sub-graph will still be a deep-copy in the described sense.)

Versioning

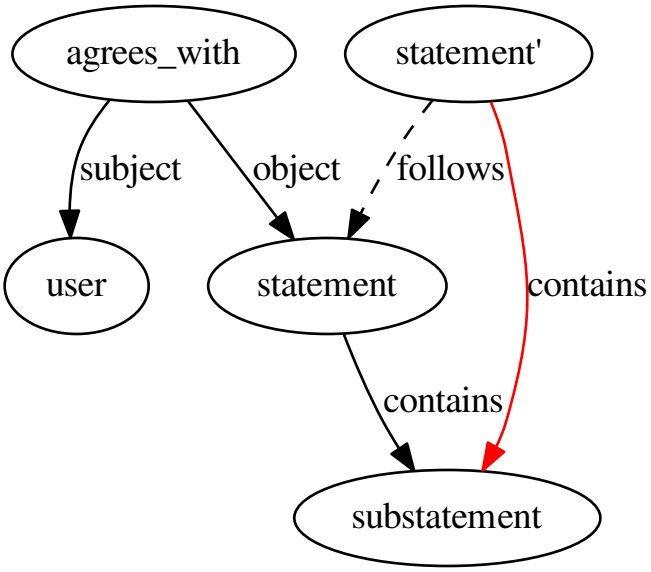
As existing nodes in the graph never change, every node modification creates a new node which is connected to the originating node with a `follows` relation. (We haven't decided how to implement this follows relation – it might be a reference or a node. In the following example graphs the `follows` relation is represented by a dashed arrow.)

Example 1.0:



The outgoing references will be copied automatically to point to the old referred nodes.

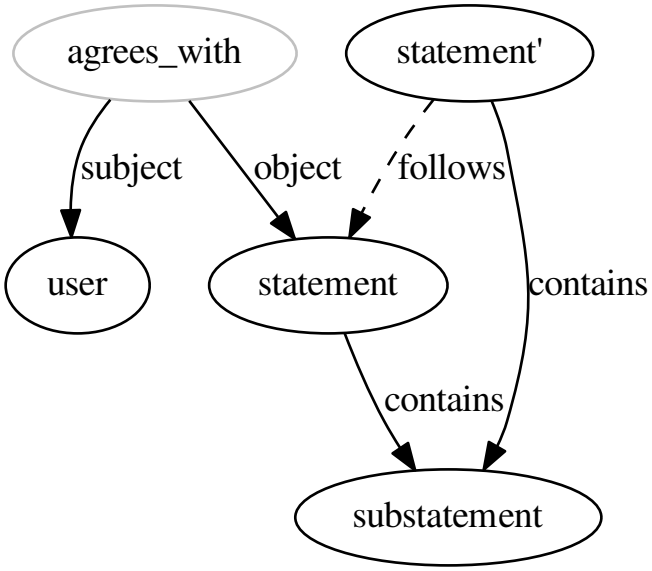
Example 1.1:



Incoming references have to be treated specially:

Nodes that are the `dependents` of the modified node are marked with a pending marker.

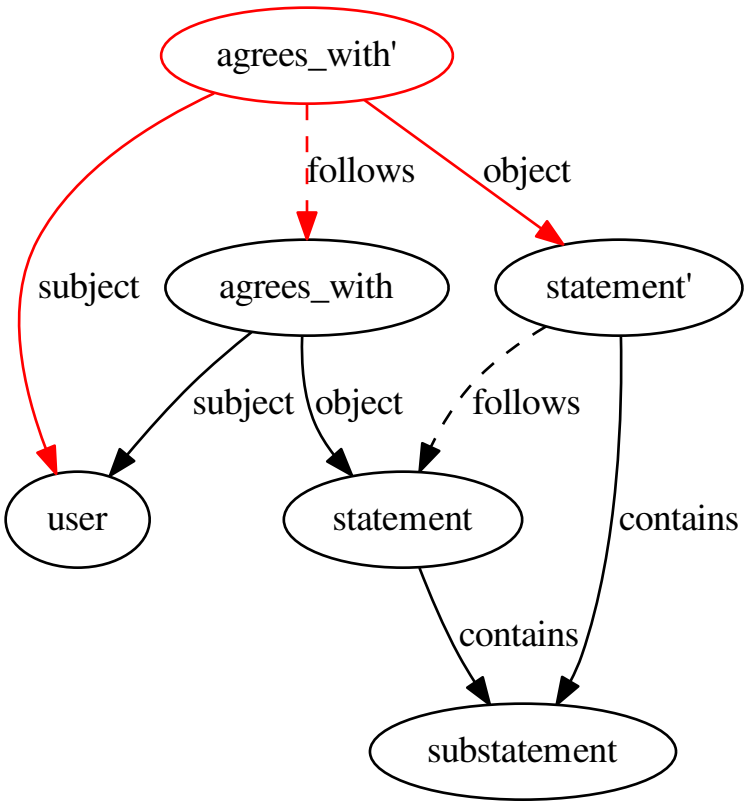
Example 1.2:



These nodes are notified and have three options:

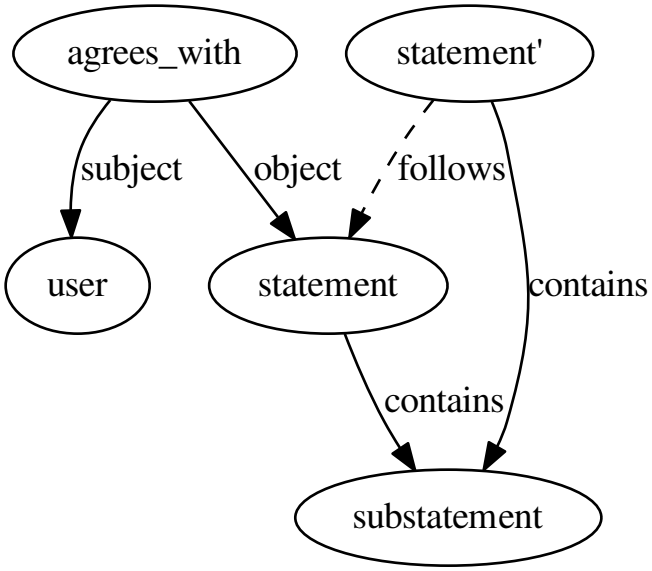
- They can confirm the changeset. This means they will be copied and their outgoing references will point to the new versions of the referred nodes. The old version will leave the pending state.

  Example 1.3:



- They can reject the changeset. This means, they will leave the pending state, but no new nodes nor references get created. The outgoing references of the formerly pending node will not change and point to old versions of nodes.

  Example 1.4:

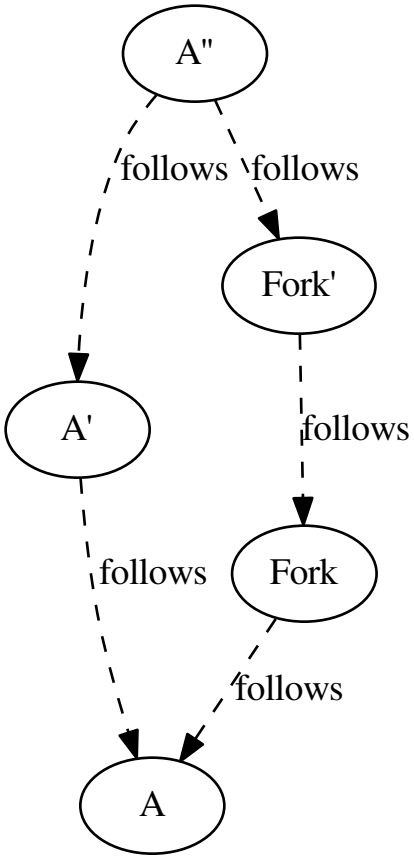


- They can do nothing and keep the pending state. At any later point in time a node can reject or confirm a changeset, probably triggered by some external event, e.g. user interaction.

Forking and merging

Modeling versioning in this manner also allows for forking and merging:

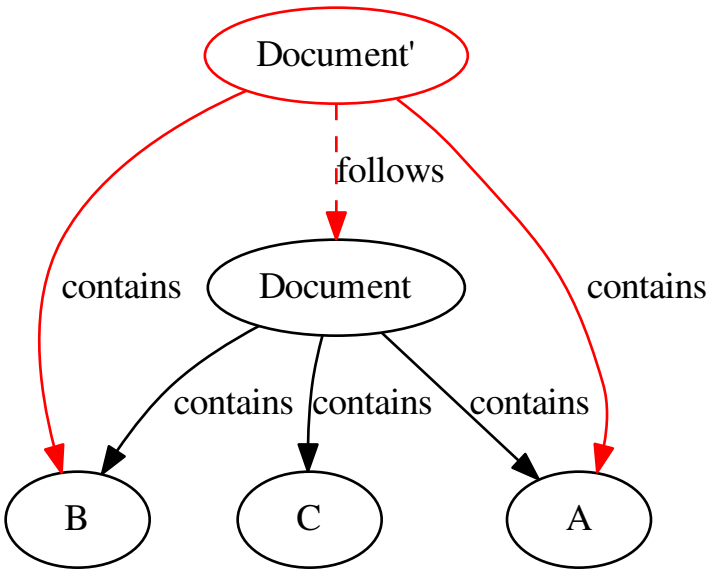
Example 2.0:



Deletion

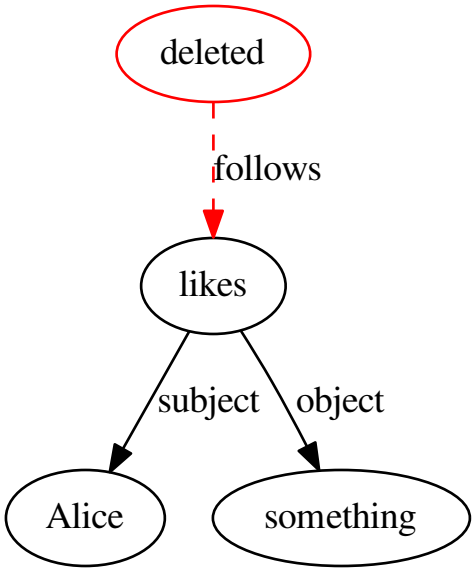
In many cases, deletion can be represented in the graph by modifying a referring node and remove some outgoing edges. It is not necessary to delete the referred node.

Example 3.0:



In other cases, it might be necessary to directly delete a node. For this case a special `deleted` node is introduced:

Example 3.1:



PROPOSAL: Not sure if this is already the intention, but it might be enough to have just one universal DELETED

node (or NULL node) in the whole graph. The DELETED node `follows` all nodes that have been deleted (multiple predecessors). Any node that has been deleted points to the DELETED node as its successor.

History manipulation

In some cases it might be necessary to modify or delete existing nodes and references directly, bypassing the versioning mechanism. This violates the non-mutability property and can be seen as a manipulation of the version history.

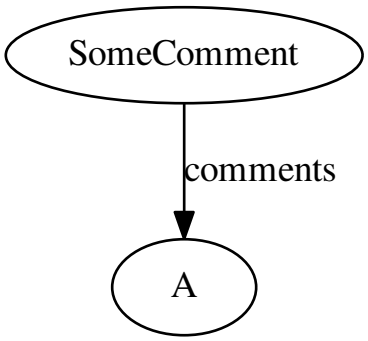
These manual modifications of the graph have to be done very carefully and could be considered as administrative tasks.

A typical example for such an administrative task is the real deletion of a node containing illegal content.

Relations

We defined relations as a pattern of nodes and references that have a specified meaning. Here is an example of a very simple relation:

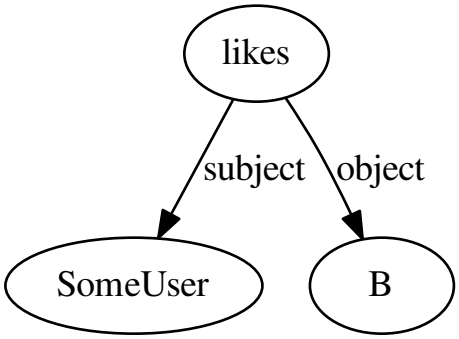
Example 5.0:



This `comments` relation captures the idea, that `SomeComment` comments on `A`. Also, the direction of the used reference implies, that `A` is an essential part of the comment.

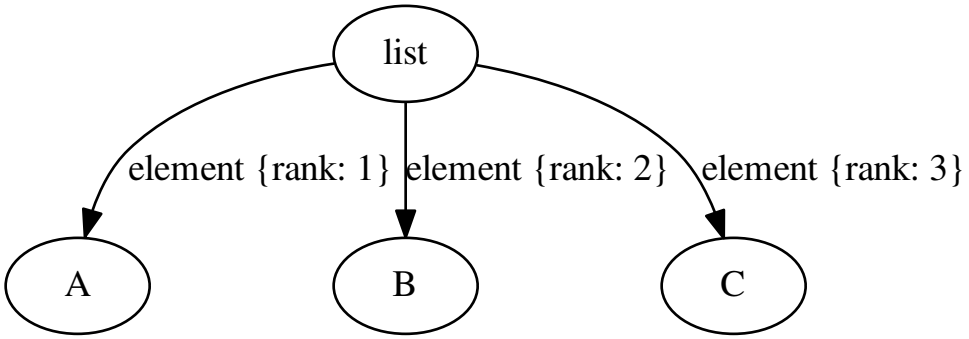
Here is another example of a slightly more complex relation:

Example 5.1:



This relation captures the fact, that `SomeUser` `likes` `B`. Again the directed references imply something about the nodes: `SomeUser` and `B` are essential parts of this `likes` node.

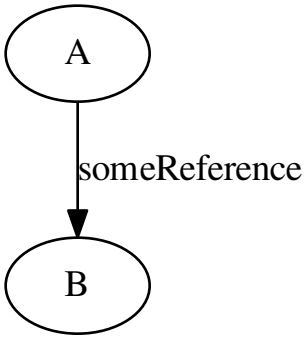
Here is how you could model a list:



The list relation allows you to store an ordered sequence of nodes. Again the direction of the used references implies that the elements are essential parts of the list.

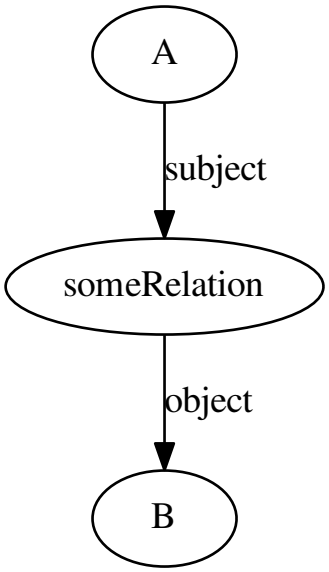
Modelling Data by Relations

The process of modelling your data is basically a process of defining relations. When defining a relation you always have to think about the direction of the used references. Here's a checklist that might help:



If you define a relation where `A` refers to `B` in some manner, then the following should hold:

- It makes sense that `B` is an essential part of `A`.
- A modification of B (creating a newer version `B'`) potentially leads to a newer version of `A` (`A'`) by triggering an update notification. The class of `A` should know how to handle such an update notification: immediate automatic confirmation, immediate automatic rejection or keeping the pending state and taking means to gather a manual decision.
- No other nodes want to refer to the reference itself. If you want to be able to refer to something, you have to model it as a node. If you want to refer to the relation between `A` and `B` in our example, you have to add an additional node:



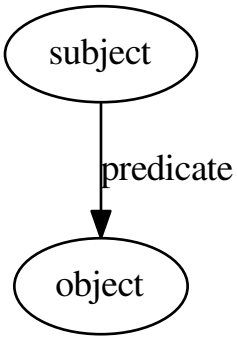
This way you still retain the idea that `B` is an essential part of `A`.

- Look out for reference cycles. If you define relations that make reference cycles very likely, you should reconsider your modelling. The supergraph allows reference cycles, but they certainly smell bad. (See *conjoined_nodes*.)

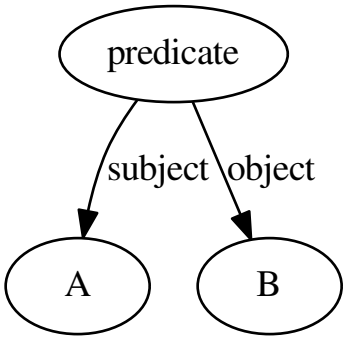
Note: Nodes and relations are the means you have to model your data. Don't fall back on simple vertices (not nodes) or simple edges (not relations) for this.

A Common Pitfall

If you model binary relations (something along the lines of "subject predicate object"), it's tempting to model the predicate as a single reference:

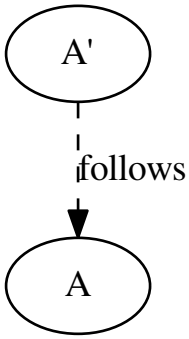


However make sure this is really what you want: Is `object` an essential part of `subject`? If not, you have to change this to:

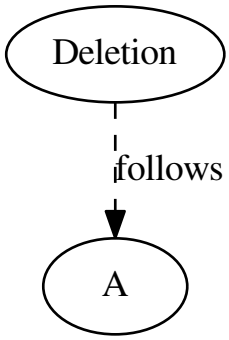


A non-exhaustive list of relations

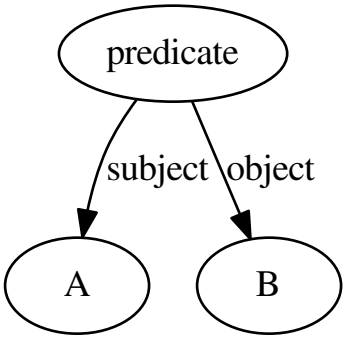
`Follows` This is the relation used to connect nodes to its predecessor or predecessors. This might be modelled like this (we are still undecided on this):



`Deletions` Node deletion is realized as a unary relation connected to the deleted node.



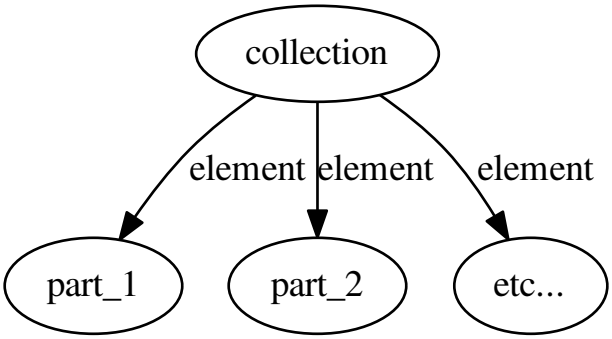
`Predicates` Predicates are classical subject-predicate-object relations (also called binary relations), expressible as a verb.



Example: `comments`

`Collections` Collections contain parts.

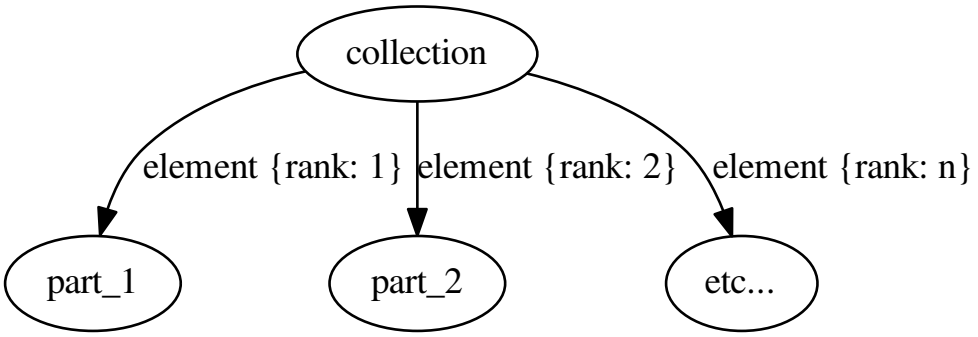
Implemented as a list vertex with references-to-many to parts



Example: `Set`, `List`

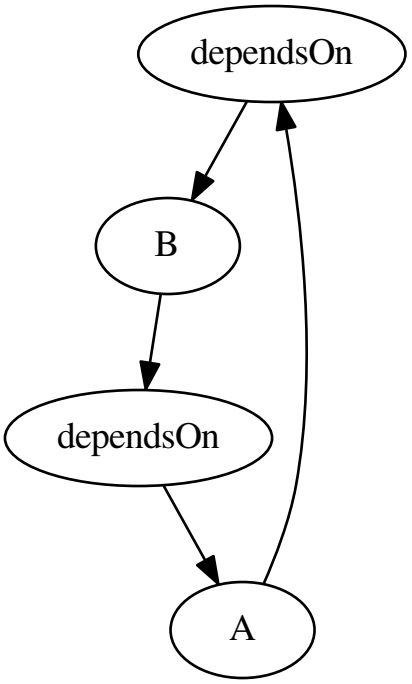
`Lists` Ordered collections.

Implemented as a collection with ranked edges.



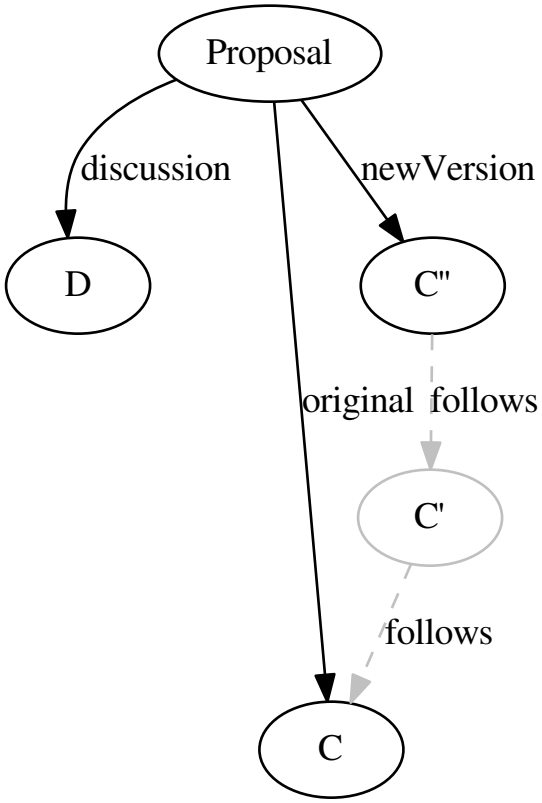
Example: `Document`

`Conjoined Nodes` Nodes which essentially belong to each other. Once one node is updated, the other node has to be updated too and vice versa - the nodes are synchronised. This can be achieved through cyclic subgraphs.



Possible examples: Translations, Binational treaties.

`More complex relations` Example: Some discussion leads to a set of (proposed) changes.



Implementation Notes

This paragraph is a summary of the data structure discussions on Fri 2013-07-19 and before. The later sections are obsolete to a varying extent.

Nodes are implemented as python objects, references as attributes. In addition to the attributes, there is a method:

```
refs(): { <attr> : <node> }
```

that returns a dictionary mapping python strings containing attribute names to the resp. reference target nodes. This is interesting because not all attributes of the node object are references.

The dependents (inverse references, i.e. only direct dependents) are represented by a method:

```
deps(): { <node> : { <interface> : [ <attr> ] } }
```

that returns a dictionary mapping nodes to dictionaries, which in turn map interfaces to lists of reference names (references are implemented as attributes containing python references).

This way, it is easy to ask an object which other objects are referencing it.

Alternatively dependents could be implemented as:

```
deps(): [ (<node>, <interface>, <attr>) ]
```

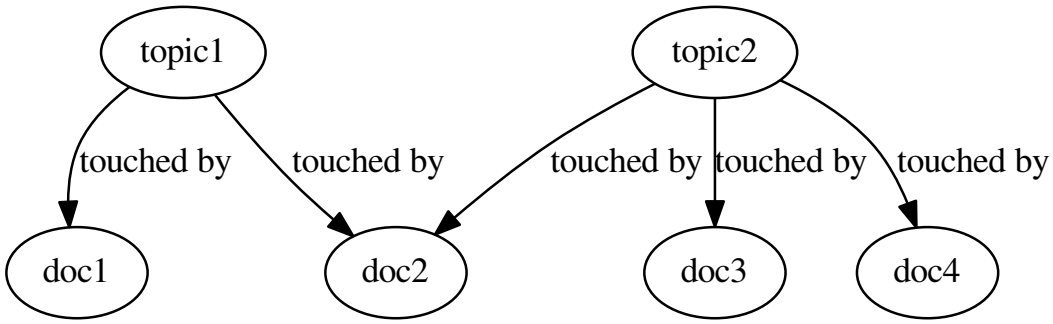
There should probably also be transitive hulls for references and dependents, e.g. `trans_refs()` and `trans_deps()`, which can be implemented easily in terms of the above methods. (XXX: is it more pythonic to say "function" instead of "method"?)

Change management is modelled by nodes being copied into `follows` nodes. There is a number of meaningful and desirable ways in which references can react to changes in referenced and dependent nodes.

If a reference is essential, the target must notify the source of the reference. The source then has three options:

- create a new version itself, keep the old reference unchanged, and update the reference in the new version to point to the new version of the target. Example: if a paragraph in a document has been updated, the document should be considered updated as well.
- ask the user what to do about the change. Example: If a user "likes" a node, and the node changes, the user should be able to decide whether she also likes the new version, or only the previous version.
- ignore the change, keep the reference pointed to the old version of the target, and do nothing. Example: Change suggestions: a user wants to express that she would support a proposal if some changes are made. This change suggestion refers to one version of the proposal and shouldn't be updated to newer versions.

If a reference is not essential, things get more complicated. The source node will still be notified of any change in any target (it always is for all references), but it has more freedom of choice in what to do, and with that comes more confusion. Example:



If topics (in wikimedia-speak: categories) are modelled this way, neither of the options of essiential references are desirable, because we would always create a new follower node of any topic that touches any document that has a new version. We either want to reference only the head of each document, and always update all references whenever documents are updated, or we want to reference all versions in the history of the document. (If we only reference heads, then what happens if somebody keeps badges or comments or whatnot on the old version, refusing to update? Then the old document, still referenced by the comment, falls out of the topic category. Hum. I think topic references would need to be copied, not moved. This would cause a lot of references. Perhaps references should be modelled the other way round, not as "touched by", but as "touches". But I digress.)

But if we simply keep track of the head of each document, what happens with forks? In a naive implementation, only the head created earliest would keep the topic, and all forks would miss it, because the node from which they fork would have passed on the reference to the follower already.

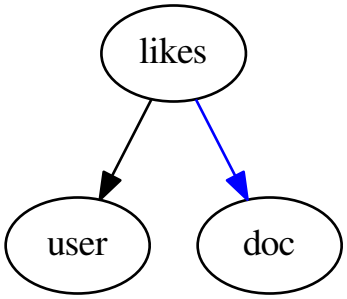
Disallowing target node forks may be sometimes an option, but in this case it is not. So there has to be another notification event: If a node is forked (has one or more followers already, and gets another one), all follower nodes are traversed, and all dependents of those nodes are notified of the fork.

The dependents can then decide what to do. In the topic model above, the topic node has to visit the new head and reference it as well, without killing the old reference. In other cases, it may raise an exception and thereby disallow forks in target nodes.

This means that some node types are forkable and others are not. Nodes therefore need an attribute:

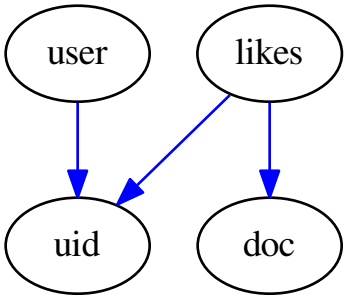
```
forkable : bool
```

Because essential edges guarantee immutability of target nodes, they are to be preferred over non-essential nodes when modelling application data. The following model:



(Essential egdes are blue.)

has a non-essential edge, i.e. the clear update rules of essentiality do not apply when the user updates her email address. The following model gets by with only essential edges:



XXX: Isn't change management of graph data structures a problem that somebody has figured out on a theoretical level yet?

Concept: Modelling a Simple Use-Case with The Supergraph

1 create participation process and content Superuser Father has an Instance Hive. He adds an a participation project "Homestuff" to discuss proposals. He creates an proposal "dishwash table" and allows other users

to access the proposal.

2a user disagrees and comments - user statement about content A user Alice looks at an existing proposal. She states her disagreement with the proposal (using a "disagree" button). She justifies her disagreement with a short text.

2a user agrees User Carl looks at everything, and annotates the proposal with an agreement (using an "agree" button).

3 user seconds disagreement - user statement about statement User Bob looks at the proposal, sees Alice's reaction and states that he seconds both her disagreement and the justifying text.

interfaces that are inherited from

INode: deps() : { <node> : { <interface> : [<attr>] } } refs() : { <attr> : <node> }

IAssessment(INode):

@essence uid : string

@essence object : INode

IAssessable(INode):

@not_essence assessments : [IAssessment]

concrete interfaces

IUser(INode):

name : str

uid : str

@not_essence user_assessments : [IAssessment]

IProposal(INode, IAssessable):

@essence title : str

@essence content : string

IDisagreement(IAssessment, IAssessable): (uid : str) (object : IProposal)

@essence rationale : string

IAgreement(IAssessment, IAssessable): (uid : str) (object : IProposal)

@essence rationale : string

ISeconds(IAssessment): (uid : string) (object : IAssessment)

where to put everything

IPool(INode): @not_essence contents : set(INode)

IProposalPool(IPool): (contents : set(IProposal))

IAssessmentPool(IPool): (contents : set(IAssessment))

IMyParticipationProcess(IPool): @not_essence assesments : IAssessmentPool @not_essence proposals : IProposalPool

IUserPool(IPool): (contents : set(IUser))

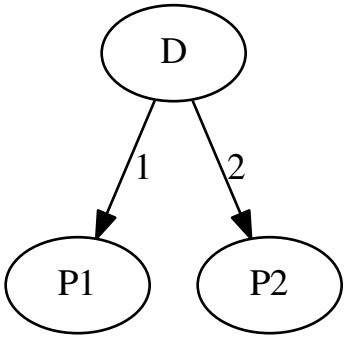
IInstance(IPool): @not_essence contents: set(IPool) @not_essence users : IUserPool

Concept: Simulating Patches with The Supergraph

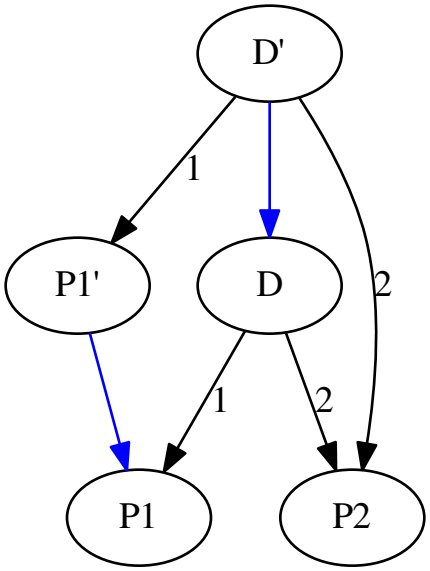
I think, we can simulate the patch ideas and interface while sticking to model everything **not** as patches but as document versions in the supergraph. We need:

- to divide data into smaller structured parts (but we wanted to do that anyway),
- intelligently consider `follows`-edges.

Imagine, you have a Document with two paragraphs:

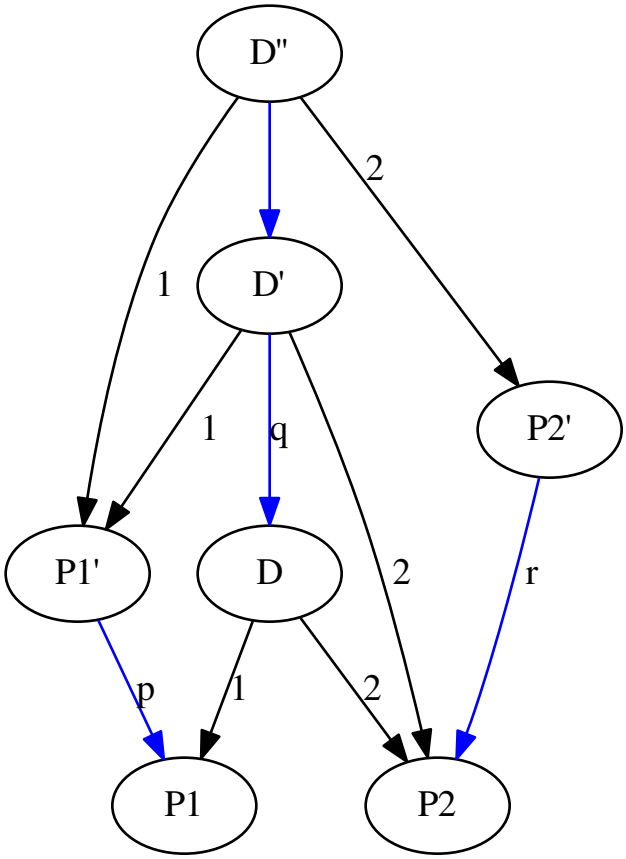


If someone creates a new version of `P1` we get a new version of D (by essence propagation):

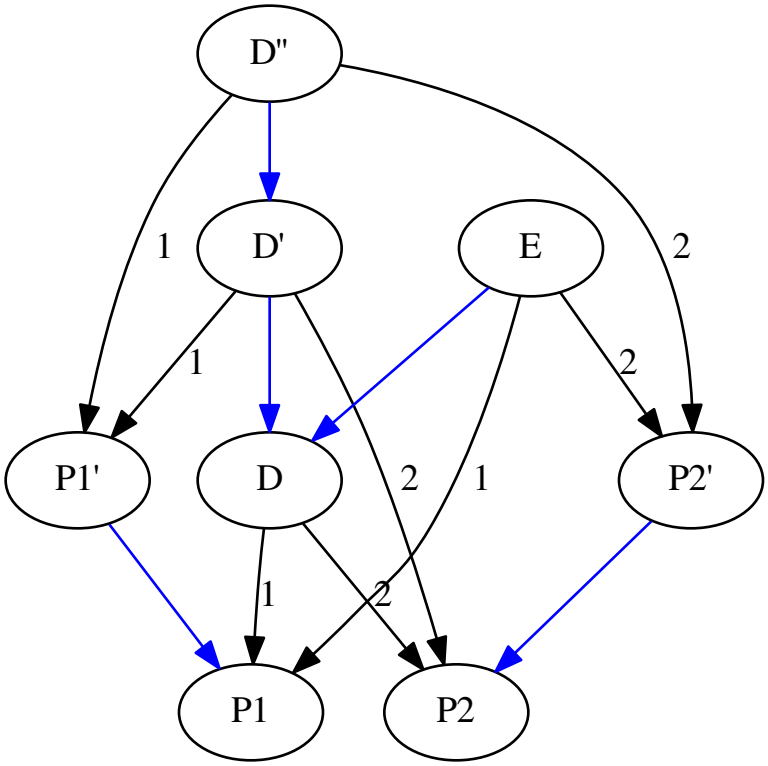


(`follows`-edges are blue.)

Now while looking at `D'` someone modifies `P2`. Once again we get a new version `D''`:



If you now look at `D` and its essence you have three `follows`-edges to consider, labelled `p`, `q` and `r`. `r` is the interesting one here. It should be no problem to build an interface that allows you to pull in `P2'` into `D` and thereby creating a new version of `D` (called `E`) that didn't exist before:

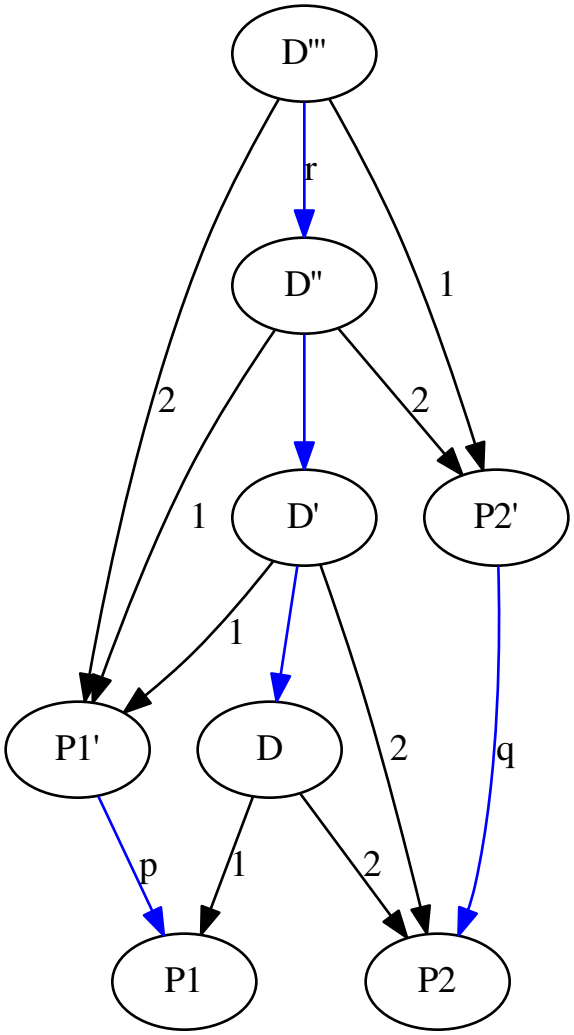


This version `E` implicitly existed as a possibility once `P2'` was created. It can be created ephemerally to be looked at in an interface and it can be brought into existence (in the supergraph) if someone considers `E` relevant.

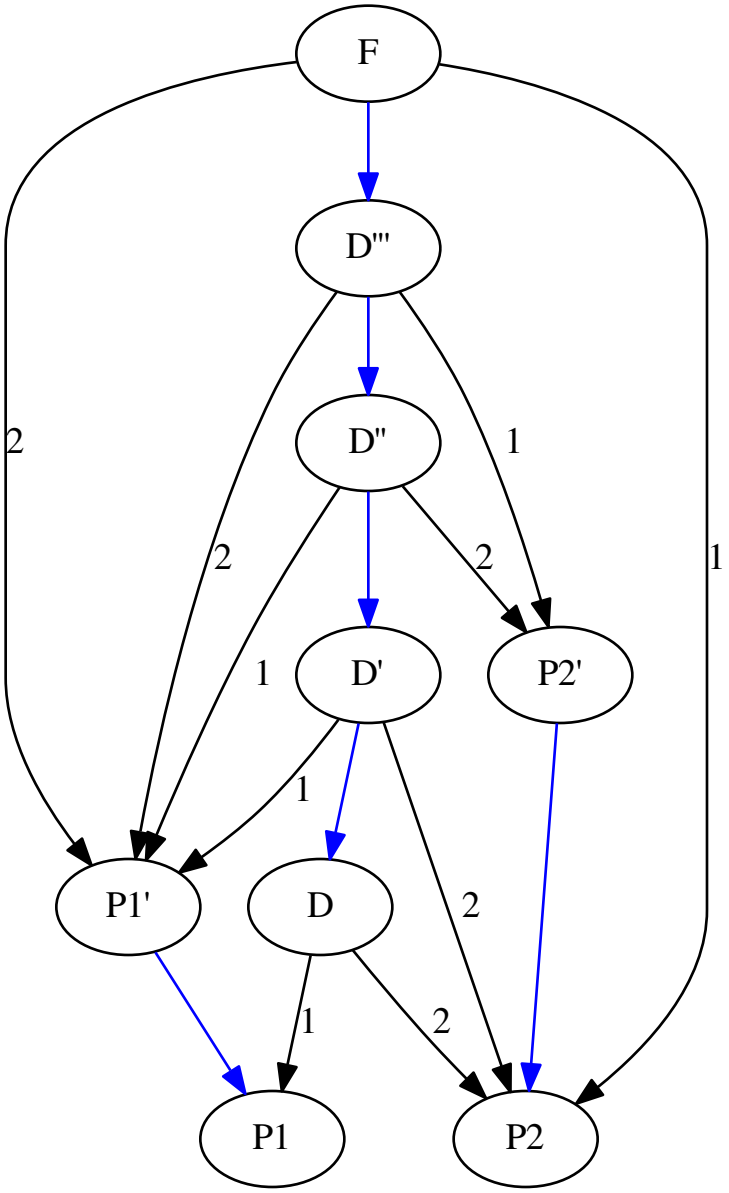
Isn't that great?

Now here is something even darcs cannot do (in one text file):

Imagine someone changes the order of the paragraphs in `D''`, (`E` is removed for clarity):



We look at `D'''`, its essence and all the `follows`-edges, again labelled `p`, `q` and `r`. Reverting `r` is trivial and would just revert the change and lead to `D''` which already exists. But an interface could allow you to try out a version where the paragraphs' order is changed, but `P2'` (for example) is reverted to `P2` (via `q`, creating `F`):



Concept: The Supergraph - Summary 2013-11-12

chronologisch

- **vor 12 monaten:**
  - essenzkanten und sonst nicht viel
  - pseudocode-algorithmus gibt's in irgendeinem etherpad

- **jüngere geschichte:**
  - komplexeres / flexibleres konzept
  - references (think json: "{content-type: ..., path: ...}")
  - wenn objekt aktualisiert wird, müssen eingehende referenzen benachrichtigt werden.
  - welche event handler gibt es? -> hängt vom typ ab.

zum aktuellen verständnis:

- dokument zeigt auf absätze.
- watchlist: wenn sich was updatet, gibt's eine email.
- category: either reference all DAGs or all versions. (latter case: some versions fall into a category, some don't.)
- zentraler use-case für essenz-ding: dokumente können aufgesplittet werden, aber es gibt trotzdem eindeutige versionen. essenz-kanten will man vielleicht als eine variante von handlern.
- vorschlag joscha: auf property-sheet-ebene essenzeigenschaft definieren: ein property-sheet ist entweder essentiell oder nicht. das macht es vielleicht einfacher, einen essenzbaum eines objekts zu bauen.
- zyklen sind grundsätzlich erlaubt.
- batch-updates sind eine optimierung einer folge von http posts / puts, haben aber die gleiche semantik. ausnahme: event propagation findet nur einmal pro batchlauf statt. (vielleicht kann man beweisen, dass das die gleiche semantik ist? nicht im strengen sinn, weil weniger knoten angelegt werden: man will für mehrerer änderungen in einem batch nur eine neue version der beteiligten objekte anlegen.)
- essenzkanten können keine zyklen bilden, weil objekten in einem essenzbaum nicht destruktiv kanten wachsen können.
- einfache muster:
  - objekt hat lineare history, und ein objekt referenziert immer den head.
  - objekt referenziert immer genau eine version
  - objektreferenz wird immer dem user gegeben, wenn das referenzierte objekt ein update bekommt. der user muss entscheiden.
  - user A ist auf watchlist von user B. wenn user B eine neue email-adresse bekommt, will user A eine nachricht bekommen, aber nichts entscheiden: die referenz auf der watchlist wird mitgezogen. (das ist ein komisches beispiel, weil man eigentlich eine nicht-versionierte user-id watchen möchte und nicht das user-metadata-objekt, aber vielleicht gibt es irgendwo ein besseres beispiel für das selbe muster.)
- propagation muss für jeden schritt ausprogrammiert werden: objekt X schickt update events an referierendes objekt Y; objekt Y entscheidet, ob es an objekt Z, das objekt Y referiert, ein eigenes event schickt.
- generell wird es spannend zu sehen, wie teuer die event-lawinen werden.
- es gibt pyramid-events: (event-typ, geändertes objekt, interface des geänderten objekts). diese events kann man subscriben. der event-typ enthält (von hand programmiert) die information über die natur der änderung (z.b. welche attribute etc.).
- dieses system kann man benutzen, um events über die referenzkanten zu propagieren. die frage ist, ob man über den referenz-event-propagation-mechanismus alles abdecken kann, oder ob es noch andere eventhandler geben soll.

versionables

lineare objekthistorie

bedingungen:

- kein merge (immer höchstens ein vorgänger)
- kein fork (voränger muss immer aktueller head sein)
- eine version ohne vorgänger darf nur einmal auf dem leeren DAG angelegt werden.

api ist gleich wie bei den anderen versionables. (dag-versionables und linear-versionbles sind spezialisierungen voneinander.)

implementierungsfrage / UI-frage: wie lockt man den head, falls mehrere user bearbeiten? natürlicher default: fehlermeldung bei konflikt. alles, was eleganter und mächtiger ist, will man vielleicht auf DAGs bauen, nicht auf linearen versionen.

variants

fork-graph, mit der einschränkung (im UI), dass nur von master gebrancht werden kann.

varianten sind immer varianten einer norm. man kann sie man gegeneinander diffen.

A2: varianten einer norm sind relevant im kontext eines proposals und können dort bewertet werden. ein beteiligungsprozess entscheidet, welche variante zur originalversion der nächsten version werden. gibt's jetzt auch bei absatzweisem kommentieren.

speziell für varianten: wir brauchen ein tag, das den head markiert, so dass die erzeugung von varianten durch forks den head nicht verschiebt. werden tags von paragraphs an die enthaltenden proposals vererbt? wie?

zwei modi: "edit" (tag weiterschleifen), "fork" (neues tag anlegen).

sönke findet spannend: ich habe mehrere versionen, die möglicherweise inhaltlich auch gar nicht konkurrieren, und will jetzt in einem demokratischen prozess einen merge daraus erstellen. prozesse determinieren wann welches tag wohin verschoben wird. (achtung! UI *einfach* halten!)

es gibt update-metadaten wie z.b. "typo" und "inhalt". ein like auf einem objekt kann man als user so konfigurieren, dass es typo-updates automatisch liked, bei inhalt-updates aber auf der alten version sitzen bleibt. (das ist ein beispiel für eine viel allgemeinere klasse von anforderungen. die lösung sollte möglichst flexibel sein. oft will man einen moderator haben, dem man vertraut, die metadaten zu pflegen und dabei nicht zu lügen.)

Random Thoughts

object hierarchy

must be optional: when posting a new document into a pool, the server does not consider the structure of the path of the pool, but the type of the supergraph node denoted by the path. this way, a user can decide to install an a3 instance that has transparent rest paths or not, and someone can implement a new backend that does not support transparent paths. rationale: there is a trade-off between data security requirements ("information must not be leaked through URLs") and usability requirements ("URLs must be informative"). users should be able to make different decisions.

authorisation management: a server implementation MAY require an object hierarchy for authorization management purposes, but the protocol MUST be independent of this in the sense that it must be possible to implement a trusting server that that allows full read/write access to the complete supergraph for everybody under unstructured paths.

lazy / bulk loading

should there be rest machinery for pulling supergraph nodes partially? no: keeping nodes atomic reduces protocol complexity enormously. it is in the responsibility of the data model designer that nodes are never getting too large. if there is a list of outgoing edges in a node and that list is growing too big, the model designer can choose to replace the list by a reference, and move the actual list into the list node that reference points to.

bulk get / post: yes, we want that.

should the supergraph structure be kept intact on the client side? yes! if the client wants to submit an update, it needs to know where nodes end and edges start, and cannot have a molten pile of nodes in one json object. (the following client implementation approach is flawed: supergraph node references are path strings, and following a references is implemented by ajax-getting that path and replacing the path by it in the object. this would give the client enough information to render the GUI, but when the nodes are to be posted back to the server because they have changed, the client has no way of knowing which attributes were originally internal to one node, and which were edges.)

marker vs. property sheet interfaces

property sheet interfaces are called e.g. "adhocracy_core.propertysheet.ILikeable", marker interfaces e.g. "adhocracy_core.content.IProposal".

backend implementation: the goal is that schemas should be defined in a maximally concise way. colander schema, management view, rest view, etc. are generated automatically, but not magically from that concise representation.

concise schemas can have the following forms:

- "propsheet1 = {field1: type1, field2: type2, ...}".
- "marker1 = [propsheet1, propsheet5, propsheet2, ...]"
- inheritance for markers: "markerx = markera + markerb + {propsheet12}".
- inheritance for propsheets: "propsheet12 = propsheet1 + {field18: type18}".

some of this is provided by colander. ideally, the backend could be made much less redundant by using colander more masterly.

structure of the json objects communicated over the rest api

since substance d is used as backend platform, the data model follows zope concepts and conventions. but there is more.

usually, a rest resource corresponds to a supergraph node (there may be other resources). it has the following structure:

{ content-type: ..., path: ..., data: ... }

content-type is a string that contains the type stored under data. it may be "adhocracy_core.content.I*" or the typeof-string-representation of a javascript primitive type. in this case, the data attribute will contain a json literal, not an object or a list of objects. (what about 'Object'? 'Function'?)

path is a string that is the key of the resource.

data is everything else. it may be missing, in which case an ajax call will be triggered by the client if and when necessary, along the lines of:

var x = { content-type: ..., path: ... };

// force: x.data = $.ajax('GET', x.path).data;

once the data attribute is retrieved, it consists of an object that has one attribute for each property sheet interface implemented by the resource. the value of each property sheet interface attribute depends entirely on the data model, except that it may contain further lazily fetchable resources.

note now that the content-type may NOT be "adhocracy_core.propertysheet.I*": a resource (as it's usually a supergraph node) implements a marker interface, and contains all data required by all property sheet interfaces subsumed in that marker interface. marker interfaces can be viewed as property sheet interface sets. since property sheet interfaces retrieval is never delayed like retrieval of referenced resources, there is no need to wrap them in an object containing a content-type, a path, and a data attribute.

"content-type" is a reserved keyword in this api because of the special way it can be used to control selecting more resources for transport over the rest api (in either direction).

the meta attribute from the prototype will go away. some of the information it contains can move to the data section of resource objects, some is unnecessary, and content-type and path are already taken care of.

dynamic content

a resource / supergraph node is a python object. the last chapter explains how to send attributes of those python objects to the client. what about the methods?

the rest api is indifferent towards where the json object in the GET response is coming from (specifically whether it is from a database lookup or some on-demand computation). for the client, there is therefore no difference between an attribute and a method: in both cases, some property sheet interface attribute contains an attribute with a content object missing the data attribute as value. in the first case, if the path is called, a lookup will take place; in the second, a python method will be called.

only that's not true. there are at least two differences: methods can have arguments, but attributes can't. and there is no way of telling in general whether invoking a method twice yields the same return value.

rest apis do not provide any mechanism for sending functions or function calls to the server for evaluation, so there is no way of passing arguments to a method to be invoked. (of course, there are many obvious ways: the arguments just need to be encoded in the url somehow. but that would be rest-ish, not rest.)

Badges

This is a summary of the "badge concept" user story. The concepts are not yet implemented in Adhocracy 3.

Adhocracy 2 knows a variety of badges and similar entities. The aim is to streamline these entities into more consistent or possibly independent concepts in Adhocracy 3. Some aspects shall not be implemented in A3.

This document can be replaced once the concepts are ready in A3.

Badge features in Adhocracy 2

The various badges, categories and tags can be categorized as the following:

Allowed targets What can be badged?

Scope Do badges exist globally or locally only?

Exclusivity Can only one badge out of a certain group be assigned to a target object?

Hierarchy Can badges be structured hierarchically?

Dedicated pages Does this badge have a dedicated page?

Create permission Who may create badges / choose available badges in a given context?

Assign permission Who may assign badges to targets?

View permission Who may view badges?

Color If shown as normal badge, what color should it have?

Icon If shown as thumbnail, which icon should be shown?

Visibility Should it be shown at all in listings?

Impact Effect on mixed list sortings

Implicit user role If a user has this badged, which additional role should she have? (we should drop this)

Voteable Allow users to vote whether the assignment applies (tags in Adhocracy 1)

Behaviour Assign a certain behaviour to a badge (research project at HHU)

Badges, categories and tags in A2

Some remarkable aspects of badge-like entities in A2:

Default

- Non-exclusive
- Non-hierarchical
- No dedicated pages
- No icon
- No image
- have colour
- Can be created and assigned by moderators
- All badges can exist globally and additonally per instance

Categories

- Can be assigned by normal users
- Exclusive
- Hierarchical
- Have dedicated pages
- Have an image

Thumbnail badges

- Have an icon

User badges

- Can have optional user role assignment

Tags

- Can be created and assigned by normal users
- Have no color

Requirements for badges in A3

(this is incomplete)

- Allow to define which badges can be assigned to a certain resource with which rules (see features above).
- It must be possible to freely define multiple available badge groups.
- Allow to define badges globally and locally. It should be possible to restrict and extend the available badges locally.
- NTH: Badges can be connected through a common taxonomy, i.e. if a local process wants to call a badge *Umweltpolitik* it can be connected to a global badge *Umwelt*.
- All badges shall be indexed and can be used in pool queries.

Example

Some proposal resource might be badged as the following:

- Badge group *decision_state*:
  - available badges "beschlossen", "abgelehnt"
  - exclusive
  - creatable by admins (not really necessary, because hardcoded)
  - assignable by moderators
- Badge group *topic*:
  - creatable by moderators
  - assignable by users
  - hierarchical
  - non-exclusive

Code Review Process

Preface

This document describes a light-weight code review process that can be used with git alone and no other tools.

Note: Code reviews are currently done with GitHub.

TODO: add good practice how to do code review

Status of this document

This document should be read as request for comments (RFC). It will be used for a trial period of two sprints (starting from 2014-06-10); in the sprint starting on 2014-07-21, a decision will be made on which parts of this document will remain valid.

Requirements

This section is a (hopefully complete) list of all requirements of the a3 development team as of 2014-06-02, in arbitrary order. As some of the requirements are inconsistent, the following sections necessarily constitute a compromise (and not necessarily the optimum in any metric).

- low-footprint, trivial to adopt.
- no need to adjust work habits to yet another new application software / UI.
- offline use (no need having IP connectivity while working).
- git repo contains all review history in the resp. branches (to the extend those branches have not been deleted).
- allow for synchronous review (talk the branch through together on the same physical display).
- allow for asynchronous review (pass comments and little fractional changes back and forth between reviewer and reviewee through something as convenient as email or a web page).
- passing a branch back and forth between reviewer and reviewee during the review process should be trivial.
- the reviewer can make changes (e.g. small typos) herself, not only ask the reviewee to do them. (all changes by the reviewer of course need to be double-checked by the reviewee.)
- comments can be attached to - the branch - lines in the full diff - individual commits - lines in commit diffs
- review comments can contain links into web / other code locations / other commits / ...
- review comments and code should be separated, e.g. in a file called `REVIEW.txt` in the root directory of the repository that can be easily removed before the merge.
- review comments should be contained in the code as comments, probably in a special mark-up form that can be pruned automatically before the merge.
- github-style pull requests
- email notifications for
  - branches ready for review
  - passing a branch back and forth between reviewer and reviewee.

  emails should contain context and links.
- allow to rebase a branch (or a clone of the branch) during the review process.

Tool Candidates

Should we decide in the future to use software on top of git, this is an incomplete list of options:

- bugseverwhere
- gerrit
- gitissues
- reviewboard
- phabricator

Code Review

Code review happens on personalized branches. Merging a story branch into master happens right after the merge of the last necessary personalized branch, so no review process is needed there.

The merge of a story branch should be done by two persons, but this is not a strong rule.

All changes and comments that the reviewer makes are either made directly in the code (see Section 'Markup language' below), or in a file called `REVIEW.txt` located in the working copy root. Reviewer and reviewee should agree on which option is preferred for what.

Synchronous Process

0. The author has completed a personalized branch for review.
1. The author chooses a reviewer and contacts her in person or by any means preferred by both.

   All documentation of the pull request must be contained in the commit log (short and long commit messages). Any documentation to the PR as a whole is appended to the commit log in an empty commit (`git commit --allow-empty`).
2. The reviewer checks out the branch to be reviewed, and makes changes and comments in the working copy.
3. Reviewer and author go through the comments in person.
4. Once all comments and changes have been agreed on, one or more additional commits are made by the author or by author and reviewer in pair programming mode.
5. The branch is merged into its base branch.

Asynchronous Process

0. The author has completed a personalized branch for review.
1. *(create pull request)* (PR) The author sends an email to a3-dev with subject `[PR] bloo (audience)`, where `bloo` is the name of the branch and `audience` is a description of possible reviewers (e.g. names or the name of the subsystem).

   All documentation of the pull request must be contained in the commit log (see synchronous process). The commit log (or the last commit) may be contained in the email body.
2. *(assign pull request)* A reviewer sends a response to the PR on a3-dev with subject `Re: [PR] ...` and an optional message in the body (e.g. "I'll do the review tomorrow"). If several reviewers respond simultaneously, they resolve the conflict outside this process.
3. The reviewer checks out the branch to be reviewed, makes any changes and comments in the working copy, and adds them to the branch in one or more commits. The short commit messages must start with `[R]` for review.
4. *(merge)* If there are no more review comments or changes, the reviewer merges the branch into its base. The branch must not be merged until all review comments are resolved.
5. *(re-assign)* If there are changes, the reviewer sends a response to the PR to a3-dev. Body may be empty or contain the commit log. At this point, reviewer and author change roles, and the author becomes the reviewee. Proceed at step 3.

Recipes

As above, first do something like:

```
git checkout branch-to-be-reviewed
export BRANCHPOINT=...  (see above)
```

To see which files have changed:

```
git diff $BRANCHPOINT --stat
```

If file paths are shortened you might want to specify a width like this:

```
git diff $BRANCHPOINT --stat=3000
```

To see all changes in a branch in one diff:

```
git diff $BRANCHPOINT
```

To see all changes to an individual file:

```
git diff $BRANCHPOINT -- <path>
```

To see all changes, organised by commits and enriched with commit messages:

```
git whatchanged -p $BRANCHPOINT..
```

To get a richer interface you can pipe the output of all of these commands into tig

Markup language

The file `REVIEW.txt` may contain any free text. (A format for what is in there may emerge in the future; there may also be tools in the future to process it.) For example it may be useful to add commit lines that can be interpreted by tig (see https://github.com/jonas/tig/issues/299).

The reviewer may make any changes to the code, including comments, in the hope that the author will like them and keep them in the final branch HEAD.

In addition, the reviewer may make specially marked comments that the author needs to process. These comments must match the regex:

```
^# REVIEW: .*
```

Depending on the language of the file under review, the `#` must be replaced by the respective comment lexeme (`#` for python and yaml, `//` for javascript, typescript and SCSS, `<!--` for html (with the extra `-->` at the end), `..` for rst, and so on).

Further lines may be added after this. Those just need to match `^#  .*` or corresponding. Note the space in both the first and all following lines.

Debates may emerge as author and reviewer realize they disagree. In that case, the comment answering a `REVIEW` comment may start after an empty line with:

```
^# REVIEW[mf]: .*
```

where `mf` is the developer shortcut of the developer that adds the comment. While this information may also be available from `git blame` it is convenient to have it right there.

During the review phase, `REVIEW` comments may either be removed manually or transformed into helpful comments to be imported into the base branch.

Dos and Don'ts

A branch must not be merged as long as `REVIEW` comments remain.

`FIXMEs` are discouraged in master. For now, they are allowed, but we should find a more fancy bug tracking approach. (redmine?)

FIXME[cs]: Personally, I mostly use FIXME for "this works as is, but it is a hack/inelegant/inefficient, so if we could find a better solution that would be great", NOT for bugs. For bugs and things that really need to be resolved to make the code function as it's supposed to, I use TODO and ensure that all TODOs are indeed handled and deleted before merging into master.

FIXME[mf]: `git notes --help` may be relevant, but I haven't looked at it yet.

FIXME[nd]: we want the commit hook to work on staged copy, not working copy. (where should we move this point? i don't think it belongs here.)

FIXME[mf]: line numbers! we want code line numbers everywhere! can git do line numbers in every line in diff?

FIXME[tb]: following things might be useful additions:

- **what should/must be done before creating a pull request**
  - **only one feature per pull request**
    - only include changes that are really needed; do refactoring in a separate pull request
    - small fixes and library updates should be done in or near master, not inside of larger feature branches. This allows everyone to profit sooner. In cases where the fix/update would have been done in multiple branches, this also avoids merge conflicts.
  - be prepared to explain every single change.

Changelog

0.0 (unreleased)

- rest api backend prototype [joka]

Roadmap

End of September 2016

Features

- process navigation
- SDI for managing processes and users

Middle of November 2016

Goal

A3 will develop into a product for civic participation. We want to offer city administrations and other political institutions the first version of the platform adhocracy.de, which lets them run idea collection processes with or without maps. It should get them interested in the topic of e-participation and trying it for their processes.

KPIs

- we can offer an initial set up of a new process and organization with medium effort
- all development debt is mapped as far as possible (#debt)

Features

- idea collection process with or without a map
- private and public processes
- initiators can invite new users
- users can edit their own account information
- initiators can generate embed snippets for processes
- concept for newsfeed / event-stream
- concept for notifications (users can follow processes and proposals in processes)

End of December 2016

Goal

Adhocracy should be optimized to allow processes to be administered by initiators. It also should optimize platform features that allow users and organizations to feel more at home.

KPIs

- we can offer an initial set up of a new process and organization with little effort
- At least two persons feel comfortable configuring the adhocracy backend (#busfactor)
- we have a written strategy for onboarding of new developers (#busfactor)
- we have a strategy to tackle identified critical development debt on the go (#debt)

Features

- initiator interface to edit process info and users
- improved user profile - avatars - description - link to social media
- organization pages with organization info (short description, link, logo) to collect processes by an organization
- new process type: polls with discussion
- simple stats on landing page and for processes (number of users, number of comments, number of votes)
- better embedding

- single-sign on with popular identity providers (e.g. Google, Twitter, Facebook)
- follow processes or organizations with notifications
- allow platform/organization/process initiators to communicate with participants of the platform/organization/process (email/newsletter)
- user dashboard where users can see the content they created and follow
- simple subscription management (email/notification) for users

Middle of February 2017

Goal

- Adhocracy provides an attractive environment that allows users to easily setup a simple participation process.
- More complex processes can be setup easily without help from developers.
- We have a dedicated concept for trainings in adhocracy and other activities related to online participation.

KPIs

- we can offer an initial set up of a new process and organization with tiny effort
- Over 80% of all identified adhocracy development debt has been tackled (#debt)
- at least two people are able to independently further develop backend and frontend (#busfactor)

Features

- initiator interface for processes and organizations
- new process type: participatory budgeting
- better discovery of processes (recommended processes, featured processes)
- search function for comments and contents
- initiators can edit permissions in all processes
- more filtering options for processes (e.g. most activity)

Constraints/challenges which we identified

- Busfactor (at least two people should be able to operate/fix/further develop backend and frontend at any time (#busfactor)
- development debt (#debt)

Glossary

ACM An Access Control Matrix defines the rights of a list of principals. An ACM crosses principals with permissions. At the intersection of a principal and a permission there is an action. The action can be either `pyramid.security.Allow`, `pyramid.security.Deny` or `None`. None is a default value and does not grant any right.

DAG Versions of one resource that build a directed acyclic graph.

group A set of users. Can be mapped to permission *role*.

groupid Unique id of one *group*: “group:<name>”.

local role A *role* mapped to a *principal* within a local context and all his children.

post_pool A normal or *service* `adhocracy_core.interfaces.IPool` that serves as the common place to post resources of a special type for a given context. If *resource sheet* field with backreferences sets a `adhocracy_core.schema.PostPool` field, the referencing resources can only be postet at the *post_pool*. This assumes that a post_pool exists in the lineage of the referenced resources. If a *resource sheet* field with references sets this, the referenced resource type can only be posted to *post_pool*.

principal A principal is a string representing a *userid*, *groupid*, or *roleid*. It is provided by an authentication policy. For more information about the permission system read User Registration and Login.

role A set of permissions that can be mapped to *principal*

roleid Unique id of one permission *role*: “role:<name>”.

service A resource marked as *service*. Services may provide special rest api end points and helper methods. You can find them by their name with `adhocracy_core.interfaces.IPool.find_service()`. The *service* has to be in lineage or a child of a lineage pool for a given context.

userid The unique id for one userique id of one *group*: “group:<name>”.

CHAPTER 3

Indices and tables

- *Glossary*
- genindex
- modindex
- search