Plat/al Documentation

Release 2.0

Polytechnique.org dev team

November 15, 2016

Contents

This project holds the second generation of plat/al, Polytechnique.org's PLATform for ALumni.

It provides the following features for alumni associations:

- Full directory, including advanced search functionalities
- Managing forlife email addresses, including complex setups (secondary domains, IMAP, GoogleApps)
- Event organization tools, including online payment
- And much more.

Contents:

CHAPTER 1

Contributing

1.1 Resources

Source code	https://github.com/Polytechnique-org/platal2
Documentation	http://platal.readthedocs.org/
Milestones	https://github.com/Polytechnique-org/platal2/milestones
Issues	https://github.com/Polytechnique-org/platal2/issues
Help	`#platal` on chat.freenode.net

1.2 Development process

Dev

- Bugfixes go directly on `master`
- Small features start with an issue describing the problem
- Begin the patchset with a piece of design documentation (might be a comment in the file)
- Big features *must* start through a design doc in the `docs/` folder
- Don't forget automated tests
- Always add a ChangeLog entry

Deployment

- Merge `master` into `prod`
- Add the release date to the ChangeLog
- Bump version numbers on the `prod` branch
- Tag that commit
- Build the packages (`make dist`) from that tag
- Merge back `prod` into `master`

Hotfixes Things break after a release?

- Add the fix on the `prod` branch
- Merge back `prod` into `master`

1.3 Coding guidelines

A consistent codebase is important for readability. The coding guidelines should be used to *improve* readability; watch this talk for a few examples.

Philosophy: **The Zen of Python**

> Beautiful is better than ugly.
> Explicit is better than implicit.
>
> Simple is better than complex.
> Complex is better than complicated.
>
> Flat is better than nested.
> Sparse is better than dense.
> Readability counts.
>
> Special cases aren't special enough to break the rules.
> Although practicality beats purity.
>
> Errors should never pass silently.
> Unless explicitly silenced.
>
> In the face of ambiguity, refuse the temptation to guess.
> There should be one – and preferably only one – obvious way to do it.
> Although that way may not be obvious at first unless you're Dutch.
>
> Now is better than never.
> Although never is often better than *right* now.
>
> If the implementation is hard to explain, it's a bad idea.
> If the implementation is easy to explain, it may be a good idea.
>
> Namespaces are one honking great idea – let's do more of those!

Python Use Django's style guide

Javascript Use AirBnB's coding rules

CHAPTER 2

Testing

CHAPTER 3

Software stack

3.1 Languages

The Plat/al project uses:

- Python3.4
- Javascript (ES5)
- bash (not sh)
- GNU Make

3.2 Application components

Plat/al v2's stack is split in two parts:

Backend Developed with Django, this component provides a REST-like API over the internal directory.

The API is built upon django-rest-framework; see the API documentation for details.

Frontend The user-facing website is a full javascript, single-page application.

It fetches data from the API, renders it, handles HTML events, and sends updates to the backend.

3.3 Front-end development

This project uses modern engineering methods for front-end development.

This means that:

- External libraries are loaded from NPM or from Bower
- The required versions are expressed in either `package.json` (for development and server-side dependencies) and `bower.json` (for "in-browser" dependencies, both JS and CSS)
- The build pipeline (concatenation, minification, optimization) might use Gulp

The build process includes:

1. Loading development dependencies from NPM through `npm install`
2. Loading in-browser dependencies from Bower through `bower install`

3. Running JS compilation from project code and libraries, through `gulp build`

3.4 Cache busting

For bandwidth efficiency, all assets (JS, CSS, images) **SHOULD** be served with a long cache duration (i.e cache forever). This is possible if JS/CSS/image filenames include a hash of their content (e.g. `platal.a244de499.js`)

This is performed using `gulp-rev` as a last build step.

3.5 Components

3.5.1 Backend

The backend provides at least some features, including:

- The admin console
- A Rest-like API, used by dynamic parts of the website
- A simple `selftest` page, where admins can check that all features are working properly

3.5.2 Frontend

The front-end is based on a full-javascript, single-page application.

It is based upon the following components:

- UI: React.js
- Event/rendering loop: Flux
- Ajax wrapper: *to be defined*
- Event management: *to be defined*

3.5.3 Request handling

Using a split stack makes debug more complex; here is a short description of what happens:

A user hitting https://www.polytechnique.org/ receives a simple HTML document that:

- Declares where version-specific JS/CSS/... files should be loaded
- Declares where the API is located
- Starts rendering the DOM through React.js

In details, the request is handled as follows:

1. The HTTP server (might be Django) receives the request
2. It generates a static HTML page, including the related JS/CSS files and the URL to the API
3. The browser parses this HTML page, and loads the JS/CSS
4. The JS app sends a few queries to the API to fetch more information (e.g the user's data)
5. The JS app updates the DOM, triggering a HTML re-render

6. When the user provides input, the JS app reacts by querying the API for the requested data
7. Based on that new data, the JS app updates the DOM again, triggering another HTML rendering.

3.6 Remote services

The site enables a user to configure other services, e.g:

- Mailing lists
- Newsgroups
- Email redirection accounts

In order to provide an easily reusable platform, those services should expose a standardized configuration interface, using a simple HTTP/JSON API.

This ensures that all components can be upgraded without breaking the user configuration pages.

CHAPTER 4

REST API

CHAPTER 5

Migration from plat/al 1

For a smooth transition from the first version of plat/al, some rules are required.

Coexistence

- The new site **MUST** reuse most URLs from the old site
- The Apache config can be used to switch sets of URLs to the new site

Database

- The old MySQL database will be kept as the main storage engine
- An additional database can be used for specific needs of the new stack
- Old tables can be altered (e.g adding an `auto_increment` field) for greater compatibility

Authentication

- Users **SHALL** authenticate only once during a session, even when switching versions
- The password (for strong authentication with sensitive tasks) **MAY** be required twice, as few pages require this for common usage
- Option 1: Use `authgroupex` from the new site to the old
- Option 2: The new site reads the PHP sessions — but this might break logout

Design

- Skin choice isn't available on the new site
- Enforce the `NewDefault` skin for the old site once finished
- Make sure the menu / header / footer have the same look between both sites

CHAPTER 6

Archived design discussions

For future reference, this document lists a few elements that led to the current choice of technology, stack, ...

6.1 Front-end development

Two approaches are possible for frontend development:

- All pages are rendered through Django templates, with some dynamic JS code for some pages
- Django only serves the API, all pages are rendered in the browser in JS

6.1.1 Django templates

With this approach, a user's request to https://www.polytechnique.org/ is handled as follows:

1. The Django server receives the request
2. It finds the correct `View` through its routing rules
3. The `View` builds a context, chooses the HTML template, and provides both to the template rendering engine
4. Django sends back the rendered HTML template to the browser
5. The browser parses the HTML and presents

If the page is dynamic (e.g search, profile):

6. The browser loads the related Javascript files
7. The JS stack generates a HTML DOM section based on its state and (optional) API queries
8. When the user provides input, the JS stack queries the API
9. Based on the new data, the JS stack updates the DOM, triggering a HTML re-render

Pros

- Closer to plat/al 1 design
- Easy to write a simple page
- The first page loads faster

Cons

- Complex handling of mixups between Django HTML templates and React HTML templates

- Behavior of the pages is split between two codebases
- Each page loads the whole skeleton

6.1.2 React.js rendering

With this approach, a user hitting https://www.polytechnique.org/ receives a simple HTML document that:

- Declares where version-specific JS/CSS/... files should be loaded
- Declares where the API is located
- Starts rendering the DOM through React.js

In details, the request is handled as follows:

1. The Django (or Express) server receives the request
2. It generates a static HTML page, including the related JS/CSS files and the URL to the API
3. The browser parses this HTML page, and loads the JS/CSS
4. The JS app sends a few queries to the API to fetch more information (e.g the user's data)
5. The JS app updates the DOM, triggering a HTML re-render
6. When the user provides input, the JS app reacts by querying the API for the requested data
7. Based on that new data, the JS app updates the DOM again, triggering another HTML rendering.

On subsequent pages, only new *data* is loaded; the whole codebase was loaded along the first page.

Pros

- Clean split of roles: Django provides the data, React provides the UI
- A single codebase for the whole UI
- Pages feel more dynamic
- Comes with great browser-based testing tools
- Rate-limiting searches is easier, since it's a single entrypoint in the API

Cons

- Unusual paradigm and language
- Adding a new page might be more complex (to be confirmed)
- The whole app must load before the user sees anything (can be avoided with e.g http://skitjs.com)

CHAPTER 7

ChangeLog

7.1 Next version

First version; mostly docs and development environment.

P