DARIAH Contrib Tool - Deployment

2016-10-18

Dirk Roorda

Server

`tclarin11.dans.knaw.nl`

Url

`dariah-beta.dans.knaw.nl`

Database

Mongodb via pymongo.

Web-app

We use Bottle, a Python3 micro framework to route urls to functions that perform requests and return responses.
The webserver is httpd (Apache). Bottle connects to it through mod_wsgi (take care to use a version that speaks Python3).
See **Prerequisites** below.

The connection is defined in the default config file (for contents, see *default_example.conf* in the github repo):

```
/etc/httpd/config.d/ :
    default.conf
    shib.conf
    ...
```

Location

```
/opt
    shibboleth
    web-apps
        dariah
            server
                app.py:  routes and controllers
                data.py: json data from mongodb
                default_example.conf: example config file for Apache
httpd server
                serve.py: wsgi entry-point for apache
                serve.sh: local development server
            static  (static files, css, javascript, fonts, etc)
                css: stylesheets built from src
                js: javascript built from src
                favicons
```

```
                images
                fonts
                docs
                    deploy.pdf: notes on deploying this web app
                tools (These files are not active in the web scenarios,
except for documentation.
                        They are helpers to prepare the data for the
app.)
                    update.sh: script to deploy updates of the web
app. Pulls code from the github repo, restarts httpd.
                    from_filemaker.ipynb: Jupyter notebook for
legacy data conversion
                    dump.sh: dump the mongodb as set of bson files
                    load.sh: load a set of bson files into mongodb
                    compose_countries: tool to tweak a map of
European countries,
                        result in /client/src/js/helpers/
europe.geo.js
            client
                node_modules: javascript dependencies
                package.json: npm config file
                README.md: short description for humans
                gulpfile.babel.js: config file for gulp, the build tool
                gulp_dev.sh: script for development builds
                gulp_prod.sh: script for production builds
                index.html: html entry-point for the client side app
                src
                    css
                        *.scss, *.css (plain CSS and SASS
stylesheets)
                    js
                        components
                            *.jsx: client-side code in jsx
                        helpers
                            *.js: client-side code and data in js
                        main.jsx: client-side entry-point for the
javascript
                    css
```

Prerequisites for the server

Python can be installed by means of the package manager:
We assume httpd (Apache) is already installed, and Mongodb likewise.

```
yum install python34
```

On a strict system, like SELinux, you can install Python3 and the extra modules needed by means of **yum install ...**

However, some of these modules end up in the Python2 framework, so I had to use **pip3**. On a strict system, you have to build pip3 first! On SELinux, this worked

```
sudo yum install python34-setuptools
sudo easy_install-3.4 pip
```

Then you can say

```
sudo pip3 install pymongo, bson, bottle
```

In order to run python3 in the webserver, I followed the mod_wsgi guide.
As preliminaries I had to install devel versions of apache and python3 first

```
yum install httpd-devel
yum install python34-devel
```

Then I downloaded the mod_wsgi source code (version 4.5.7), untarred it, and configured it with whatever python3 I found on the path.

```
cd mod_wsgi-4.5.7
./configure --with-python=/bin/python3
```

Then

```
make
sudo make install
```

After this, httpd works with python3.
The website runs with SELinux enforced, and also the updating process works.

Develop environment

Prerequisites

Javascript

- Install `nodejs` from the download page. Then install all javascript dependencies in one go by executing

```
cd /path/to/dariah/client
npm install
```

- Now you can perform builds, by saying, in the same directory

```
./gulp_dev.sh
```

or

```
./gulp_prod.sh
```

Python

- Install python3.x.y from the download page. Then install additional modules by means of *pip3*:

```
pip3 install pymongo bson bottle
```

- Now you can run the development server by saying, in the same directory

```
./serve.sh
```

The client application is a react component. The source code is in `client/src`.
It is a set of `components` in *jsx*, (a react enhancement of Javascript), and the javascript itself is ES6.
There are also a few auxiliary functions in `helpers`, all in plain ES6.
Most of the styling is defined in the JSX, but there are a few CSS style files, either in SASS, or in plain CSS.
For example, we use the open source mapping library *leaflet*, which comes with a plain style file.

The *jsx* and *js* of components and helpers will be bundled with other javascript sources from `node_modules`.
Javascript from other sources, such as *leaflet*, resides in `static/js` and will be included directly by the main html file `index.html` .

The build tool is *gulp*. It will browserify the javascript, apply babel transformations from ES6 and JSX to browser compatibele Javascript. It will bundle, minify, and provide source maps. I have configured two tasks: for developent and for production.
The development task skips minification, and continues to watch for changes, so that when you work, builds will happen whenever you save source files.
The production task runs once, and does perform minification.

The *react* components live and work client side. The server side tasks are performed by Bottle. On your local server, you can just invoke `server/serve.sh`, which starts a small webserver that listens to localhost on port 8000. The app source code resides in `app.py` and other `*.py` files imported by it. Whenever you save a python source file, the server reloads itself.
The module `app.py` defines routes and associates functions to be executed for those routes. These functions take a request, and turn it into a response.
The functions in `data.py` are a subset: they query the mongodb and return json data.