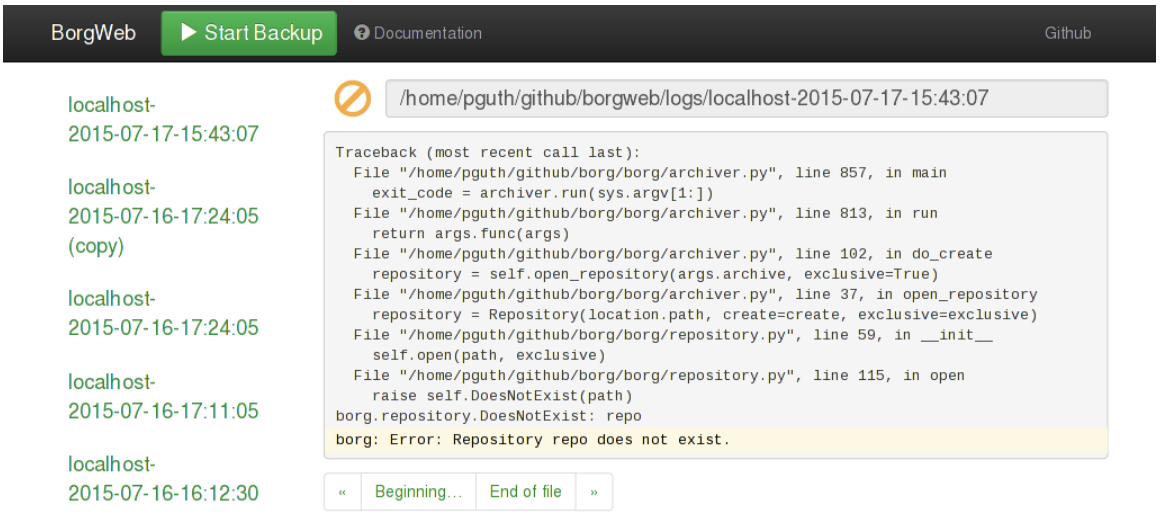
Borgweb frontend development

July 22, 2015 | 0 Comments | Tags: technology, wedv, berichtsheft, development, coding, open source, JS

Having written Peertransfer last semester I now had the chance to try myself at another JS heavy-ish project: Borgweb a log file viewer for Borgbackup.



Borgwebs JS talks to a RESTful API to:

1. Get a list of log files (on the left).
2. Display the contents of log files.
3. Paginate between chunks of the log file (only request the part that is beeing displayed).
4. Display miscellaneous information about the log file.
    - Highlight erronous lines.
    - Indicate overall state of the backup that produced the log.
5. React to the height of the browser window and display only a suitable amount of lines.
6. React to height changes and re-render.

The first version I wrote was very complicated code-wise and it was quite hard and awkward to debug or even introduce new features. Having all code in one big file did not help. There was some aversion in our company

to use JS code structuring tools like Gulp and Browserify maybe because we underestimated the complexity these few features already introduce (for a beginner like me).

So a few discussion of algorithms later I decided to rewrite and use a more modular way of structuring my code and also tried hard to keep functions short.

Starting to use ES6 constructs I needed to transpile and decided to use Babelify. It's one of two contenders in that space but is the one that is community based. Building on previous attempts I could reuse/improve upon old build chain scripts and pretty much stayed with this version:

```
var lib = require('./gulpfile.lib.js')
...
var files = {
  js: './index.js',
  jsBndl: '../borgweb/static/bundle.js' }
...
gulp.task ('watch', function () {
  newBuildLog(); browserSync.init({ proxy: 'localhost:5000',
open: false })
  lib.generateBundle(files.js, files.jsBndl, true)

  gulp.watch([files.js, 'src/**/*.js'], function (ev) {
    if (ev.type == 'changed') {
      newBuildLog()
      log("Changed: " + ev.path)
      var ret = lib.generateBundle(files.js, files.jsBndl)
      setTimeout(function () { browserSync.reload() }, 150)
// todo
      return ret } }) })
```

What's neat for me about this is, that it watches all me used JS files and continuously rebuilds the `bundle.js` that is actually used by the browser. BrowserSync handles reloading the website when files changed. Since it spawns a local webserver I could use another machine remotely open the website via SSH (almost like shown here) and have BrowserSync handle the

reloading automatically - second monitor without having to plug cables *yay*.

When glueing code together I liked to keep it verbose so I had a separate export section at the end of every module like so:

```
module.exports = {
  noBackupRunning: noBackupRunning,
  pollBackupStatus: pollBackupStatus,
  ...
}
```

And also a section in my `index.js` where I made the functions globally available that needed to be callable from the frontend:

```
window.startBackup = borg.startBackup
window.switchToLog = logViewer.switchToLog
window.nextPage = logViewer.nextPage
...
```

Of course there are still open issues on Github. But we have a first working version and already have a PIP package.

So lessons learned here:

1. Write short functions.
2. Structure using modules.
3. Automatize if possible.