Coding Standards

COS 301: REDIRECTION

2018

Team Members

Stephen Teichert - u16254661

Kyle Wood - u16087993

Russell Dutton - u16016612

Jeffrey Russell - u16010648

Justin Grenfell - u16028440

Byron Antak - u16039689

Contents

1 Introduction

As a team, we discovered a coding guide regarding JavaScript and Coding Standards in general that we liked (https://github.com/airbnb/javascript/blob/master/README.md) and as such the majority of our standards are derived from the document, with a few subtle changes

2 Coding Conventions

2.1 Naming Conventions

1. Be descriptive in your names. Avoid single letter names. Example:

```
function query() {
    return 5;
}
```

2. Use **camelCase** when naming functions, variables and instances of objects.
   Example:

```
var configManager = new ConfigurationManager();
```

3. Use **PascalCase** when defining classes
   Example:

```
class ConfigurationManager {}
```

4. Use **MACRO_CASE** for constants
   Example:

```
const MACRO_CASE = 'cool';
```

5. Do **not** use trailing or leading underscores.
   Example:

```
__this or this__ or even __worse__;
```

6. A base filename should exactly match the name of its default export
   Example:

```
import CheckBox from './CheckBox';
```

7. Acronyms and initialisms should always be all capitalized, or all lowercased (or use synonymous names such as textMessage for SMS)
   Example:

```
class HTTPResponseManager {
}
```

8. You may uppercase a constant only if it is exported or if it is a const, meaning you know for sure that it can not be reassigned. If the object is constant, only the object name captilised and not it's corressponding fields
   Example:

```
const IMPORTANT_DETAILS = { key: 'value' }
```

2.2 Formatting Conventions

1. Use soft tabs (space character) set to 4 spaces (this differs from the airbnb document we like the look of it more; it's easier to follow)
2. Separate operators with spaces
   Example:

```
let sum = x + y + z;
```

3. End files with a single newline character only
4. Use indentation when making long method chains (more than 2 method chains). Use a leading dot to emphasize that the line is a method call, not an entirely new statement
   Example:

```
$('#items')
    .find('.selected')
        .highlight()
        .end()
    .find('.open')
        .updateCount();
```

5. Leave a blank line after blocks and before the next statement.
   Example:

```
if (happy) {
    return 'YAY';
}
return 'SAD';
```

6. Do **NOT** pad your blocks with blank lines @BYRON!
   Example:

```
if (sad) {

    return 'BOO';

}
return '*smile*';
```

7. Do **NOT** add spaces inside parentheses or square brackets.
   Example:

```
if ( sad ) {
    return 'BOO';
}
```

8. Add spaces inside curly brackets.
   Example:

```
const IMPORANT_DETAILS = { key: 'value' };
```

9. Avoid code that runs across the page (roughly more than 80 character)
   Example solution (note leading operators from earlier point):

```
let shouldUpdate = jsonData
    && jsonData.meh
    && jsonData.derp.real
    && jsonData.derp.fake.really
    && jsonData.derp.fake.really.real
    && jsonData.derp.fake.really.giveUp.Yet;
```

10. Do **NOT** use leading commas
    Example solution (note leading operators from earlier point):

```
let arr = {
    Jeremy
    , Suzy
    , Candice
}
```

11. Leave an additional trailing comma (to make adding items easier later, also note babel and other transpilers remove 'em)
    Example:

```
let arr = {
    Jeremy,
    Suzy,
    Candice,
}
```

12. Add semi-colons after every line of code even if not required by JavaScript/TypeScript (for clarity and to prevent errors with JS's Automatic Semicolon Insertion algorithm)

2.3 Commenting conventions

1. Use /** ... */ for multi-line comments.
   Example:

```
/** this is a badly
    named array of people's names
    <insert long rant here>
*/
let arr = {
    Jeremy,
    Suzy,
    Candice,
}
```

2. Start all comments with a space to make it easier to read
   Example:

```
// this is a comment with a space:O
let code = "good";
```

3. Use // for single line comments. Place the comments above the line it refers to.
   Example:

```
// we will ignore the z dimension
let sum = x + y;
```

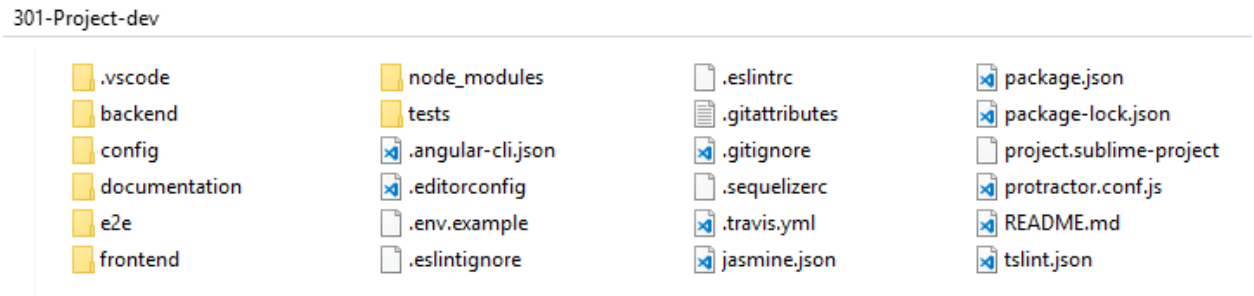
4. Prefix your comments with FIXME or TODO to help other developers quickly understand that there is a problem that needs to be revisited
   Example:

```
class Calculator extends Abacus {
    constructor() {
        super();
        // FIXME: shouldn't use a global here
        total = 0;
    }
}
```

5. Use // TODO: to annotate solutions to problems identified by // FIXME
   Example:

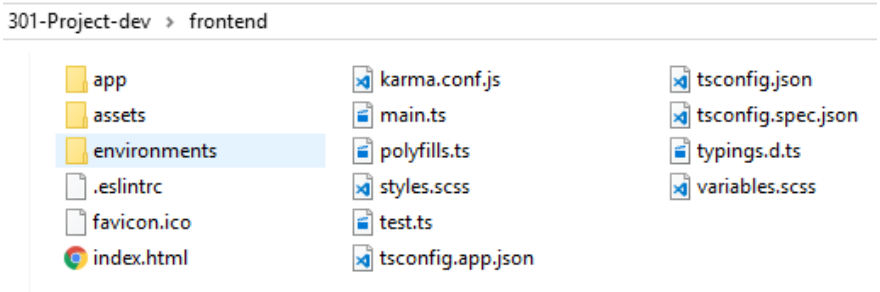
```
class Calculator extends Abacus {
    constructor() {
        super();

        // TODO: total should be configurable by an options param
        this.total = 0;
    }
}
```

3 File Structure



As shown in the image above the structure has clear separation between all parts of the system, Mainly the frontend and the backend. The config folder is shared by both backend and frontend systems, and therefore has a shared root. All of the system tests and documentation are stored in the tests and documentation folders respectively.

The backend folder contains everything that runs on the backend. The controllers and LUA folders contains everything that is required for the LUA code generation, the database folder contains everything database related, the static folder is for information and images that need to be served by the API.



Any files within the frontend root folder are required by angular and apply to the frontend as a whole. All Angular components (hence web pages are found in subsequent folders with the name of the component and all related code.

4 Code review process

Included in our automated tests on github are linter's which are configured to ensure the files in the branch match with the standards as stated in this document above. These linter's are run along with the tests whenever something is pushed to our repository. If the linter fails the file, the tests will fail. Neither the master nor the dev branch can be directly accessed through a merge or push request. In order to push or merge with the dev branch the member must create their own branch related to an issue created, they must then submit a pull request to the dev branch. Once the pull request has been created 2 other members must review the code and any changes made and ensure it abide by our coding standards, they then approve the pull request.

We have implemented Karma (https://karma-runner.github.io/2.0/index.html) as the angular frontend test system.
We are using Jasmine (https://jasmine.github.io/2.0/introduction.html) to test our express backend API server.

We are using Travis (https://travis-ci.org/) to automatically test everything that goes through our github repository. When anything in any of the branches is changed Travis runs all tests to ensure everything is working as it should.