Spring 2018
Instructor: Birgir Kristmannsson

Cats

Development Guide

Andri Karel Júlíusson
Sigurður Marteinn Lárusson
Skúli Arnarsson
Smári Björn Gunnarsson

Table of Contents

1 Getting Started

1.1 Introduction

The purpose of this guide is to assist developers that have just started working on the project in getting it up and running and also in understanding the structure of it. This can of course be useful as a reference point for other developers that have been working on the project longer. The guide also contains coding rules and style guides.

1.2 Prerequisites

These are the dependencies you need to set up on your computer for the current development build of the Cats system. This can be set up on any operating system but a **Unix-based** system is recommended.

- A **git** client
  - For example this one
- **Node.js** run-time environment
  - Can be downloaded here
  - Use version 8.*.* (version 8.11.1 recommended)
- **npm** package manager
  - Is part of **Node.js** installation so should be installed already
  - Can be downloaded here (If for some reason it is not installed)
- A code editor
  - For example Visual Studio Code
- **PostgreSQL**
  - Download here

1.3 Access Requirements

You will need access to:

- The **GitHub** repository of the system.
- Access to the corresponding account on **CircleCI** (To monitor the status of the latest deployed release, this is possibly not necessary for all individuals working on the system)

1.4 Installation Guide

This section will detail how to install the system after all the prerequisites have been set up and all access requirements have been fulfilled. In this section it is assumed that you are using a terminal.
To install the system, navigate to your desired directory and follow these steps:

To clone the project into that directory:

```
git clone git@github.com:svennidal/jokula-v18.git
```

Switch to the dev branch and pull again:

```
git checkout dev
git pull origin dev
```

To install all dependencies for the project run the following command in the directory where package.json is located (in the root of the newly cloned directory):

```
npm install
```

To create the **.env** file that contains the environment variables for the system, run the following command:

```
cp .env-example .env
```

Create a user in Postgres. See this for reference.
Create one empty database using **PostgreSQL** (we recommend naming it **jokula**):

```
psql -U [yourusername]       //Login to psql
create database jokula;      //Create the database
\q                           //Log out of psql
```

Open the **.env** file you created and fill out the corresponding variables. This includes:

- **DB_USER**: your postgres username
- **DB_PASS**: your postgres password
- **DB_HOST**: the name of the database you created

Navigate to the `/src/config` folder and run the following command:

```
cp database-example.js database.js
```

Open the newly created **database.js** file and fill out these variables:

- **host**: localhost
- **user**: your postgres username
- **password**: your postgres password
- **database**: the name of your database

These config files are not stored on GitHub, only locally.

1.5 Running the system

Now there are only several things left to get the system running

To create the database tables run the following command:

```
npm run migrate
```

To seed the database with mock data run the following command:

```
npm run seed
```

Finally, to run the system open up two terminals and run the following commands:

Client in one terminal:

```
npm run dev
```

Server in the other terminal:

```
npm run server
```

Navigate to **localhost:8001** and you should see the login screen for the system.

You can login using our admin account:

Username: **admin@admin.is**

Password: **admin**

2 Code Style Guidelines

2.1 Indentation

- Indentation should always be 1 tab

2.2 Variable names

- Camel casing in server and client
  - Example: `variableName`
- Use descriptive names
  - Example: `let projectToUpdate = project`
- Constants should be uppercase
  - Example: `const CONSTANT_NUMBER = 42`
- Types
  - Always use `const` if the variable should not change and `let` when appropriate

2.3 Function names

- Camel casing in server and client
- Use descriptive names
- Use ES6 arrow syntax where possible (and feasible)
  - Callback example:

```
someMethod(parameter, (x) => {/* do something with x */})
```

2.4 File names

- Server
  - Files in server should use Camel casing
    - Examples: `projectService.js, employeeRoute.js`
  - Folders in server should be lowercase only
- Client

- Vue files should use Pascal casing
  - Examples: `CreateCustomer.vue, DivisionTable.vue`
  - Other files should use Camel casing
- Folders that are used to categorize different components and views should use Pascal casing
  - Example: `components/UserComponents/UserComponent.vue`

2.5 CSS/Less

- All styles that could possibly be reused should be placed in the global **styles.less** file
- In the global **styles.less** all styles should have corresponding block comments indicating which element they affect
- When writing **less** code the hierarchy of the HTML document should be followed where possible
  - Example:

```
<div class="container">
      <div class="inner"></div>
</div>

.container{
      .inner{
      }
}
```

- All class names and identifiers should use Kebab casing
  - Example: `name-of-class`
- All variables that can be reused should be stored in the global **styles.less** file
  - Example: `@color_blue: #185291`

2.6 Other

- Quotation marks
  - Always use single quotes if possible, use double quotes if quotes are required within a string. Example:

```
let str = "this is my string 'and this is the string within'"
```

  - Use back-ticks (` `) and template literals when combining things into strings
    - Example: `` `User number: ${user.number}` ``
    - Further reading on template literals can be found here
  - For Pug template code (used in Vue components) it is preferable to use double quotes
- Semicolons
  - Always use semicolons to end statements (even though you think you don't have to)

3 Helpful Commands

3.1 Running the application

To run the server of the system

```
npm run server
```

To run the client of the system

```
npm run client
```

3.2 Bookshelf.js

To run the client of the system

```
npm run createmodel
```

Create a new Bookshelf migration

```
npm run createmigration [nameofmigration]
```

Create a new database migration

```
npm run createmigration
```

To run all outstanding migrations

```
npm run migrate
```

To undo the last batch of migrations

```
npm run rollbackmigration
```

To create a new seed file

```
npm run createseed
```

To rollback a single migration then migrate to the latest migration and apply all seed files:

```
npm run resetdata
```

To purge the database and run all the seeders

```
npm run seed
```

3.3 Unit Testing

To seed the database and run all unit tests

```
npm run seed
```

To run tests without first seeding the database(use of this script is discouraged)

```
npm run testonly
```

3.4 Code Coverage

Runs seed script then runs all unit tests and then generates a code coverage report(viewable through the coverage folder). The output buffer is filled with coverage report information. Use **'npm run seedtest'** for viewing tests

```
npm run test
```

To run a code coverage report and generate the output

```
npm run report
```

3.5 Documentation

To generate API documentation (viewable in the documentation folder)

```
npm run document
```

4 Server Structure

This section will describe the structure of the server side of the system. The server is split into several parts. Each part of the server will now be explained in a separate subsection. The server resides in the `/src` folder.

4.1 Routes (`/src/routes`)

Each separate routes file is responsible for one part of the data being fetched from the server. For example we have one `employeeRoute`, one `projectRoute` and so on.

The purpose of each route file is to determine which service to use to retrieve the information(or store the information) the client wants, use that service and return the results back to the client.

The routes are accessed through the app.js file which is the entry point to the server.

A service should have the following format:

```
/**
 * @apiGroup someAPIGroup
 * @apiName The name of the route
 * @apiDescription A description of the route
 * @api {get} /someroute
 * @apiSuccess {Array} The return value of the route
 */
router.get('/', function(req,res,nex) {
    someService.findAll().then((result) => {
        res.send(result);
    }).catch((reason) => {
        res.send(reason);
    });
});
```

4.2 Services (/src/services)

Each service is responsible for one part of the data being fetched and manipulated for the server.

A service should have the following format:

```
class MyService {
    /**
     * A comment describing the service
     * @returns {Array} The array the service returns
     */
    someMethod() {
        return new Promise((resolve, reject) => {
            someModel.fetchAll({
            }).then((model) => {
                resolve(model);
            }).catch((reason) => {
                reject(reason);
            });
        });
    }
```

4.3 Tests (/src/tests)

Each test file is responsible for testing different parts of the server. These are the tests that are executed when the test command is run. The tests show whether the server is returning the correct results and updating the data in the right manner.

A test file should have the following format:

```
describe('#### TestAPI, function (){
    describe('** GET /test **', () => {
        it('1. Describe what the test should do, (done) => {
            admin.get('/test).end((err, res) => {
                assert.equal(res.status, 200); //Assert that it works
                assert.equal(res.body.length,4);//Another assert
                done();
            });
        });
    });
});
```

4.4 Models (/src/models)

The models are simple objects representing individual database rows. They also specifies any relations to other models.

```
const someModels = bookshelf.Model.extend({
    tableName: 'tableName',
    idAttribute: 'idAttributeInTable',
    hasTimestamps: true,
    softDelete: true,
    otherModels: function() {
        return this.hasMany('otherModel');
    }
});

module.exports = bookshelf.model('someModels', someModels);
```

4.5 Utilities (/src/utils)

The server has several utilities. Utilities are simple functions that are used widely in the server and do not belong to any particular part of it.

The utilities are the following:

- `checkRole(roles)`
    - A middleware function. Takes as parameters an array of roles and checks if the user sending the request has any of those roles. If the user does have any of the roles it permits the request. Otherwise it informs the user it is Unauthorized for the action.
- `checkAuthenticated()`
    - A middleware function. Checks if the user is logged in.
- `createHash(password)`
    - Creates a hash from a password string that is passed in as a parameter.

5 Client Structure

This section will describe the structure of the client side of the system. The client is split into several parts. Each part of the server will now be explained in a separate subsection.

5.1 Routing (/client/router/index.js)

All routing is handled by Vue on the client side. This is done in `/client/router/index.js` and declared in `/client/main.js.`

Adding new routes in `/client/router/index.js` is simple, simply import the component that you want to render on that route and add the following to the routes array:

```
{
      path: '/some-path',
      name: ,'NameOfView',
      component: ImportedComponent,
      beforeEnter(to, from, next) {
        assertAuthenticated(to, from, next);
      }
},
```

The `beforeEnter` is a method that allows you to control what happens before the route is entered. In the example above it is used to assert whether the sender of the request is logged in.

5.2 Views (/client/views)

The views for the client side are contained in `/client/views` folder. The views represent a page in the application and they are categorized in folders by their responsibility (employees, projects etc.) Views that are not categorized(login, settings etc.) simply reside in the root of the views folder.

5.3 Components (/client/components)

The components for the client side are contained in `/client/components` folder. The components represent visual objects within the views. These can for example be tables, modals, navbars, table items and so on. These are also categorized in folders by their purpose (Modals, Dashboards etc.)

5.4 Utilities (/client/utils)

The utilities are stored within `/client/utils.`

The client has several utilities. Utilities are simple functions that are used widely in the client and do not belong to any particular part of it.

The utilities are the following:

- Validators:
  - `emailValidator(email)`
    - Takes a string as a parameter and verifies that it is a valid email.
  - `phoneValidator(phone)`
    - Takes a string as a parameter and verifies that it is a valid phone number
- Date formats:
  - Reside in the `dateFormats.js` file
  - Simple functions to convert `datetime` string into different formats
- Image converter
  - Takes a binary value of an image and converts it to a data string that can be set as a `src` for `img` tags in **HTML**
- **Vuex** store
  - The **Vuex** store for the applications resides in the store.js file.
  - Further reading on **Vuex** can be found here

5.5 Assets (/client/assets)

Resides in the /client/assets folder. The assets are simply all the resources needed for the application: images, fonts, and stylesheets.