Vorönn 2017

Leiðbeinandi: Guðný Ragna Jónsdóttir

Prófdómari: Torfi H. Leifsson

When in Iceland

Development Manual
Spring 2017

Andri Rafn Ágústsson

Ásgeir Frímannsson

Magnús Norðdahl

Skúli Þór Árnason

Table of Contents

1 Getting Up and Running

1.1 Preface

This manual aims to assist new team members to get integrated quickly into the development team. It also works as a reference source for seasoned members. It covers what you need to install on your machine to start working, what you need access to and how to get the project up and running along with style guides and class formats.

1.2 Prerequisites

1.2.1 Programs and dependencies

These are the programs and services you need to set up to run the development build for When in Iceland.

- A git client
  - i.e. GIT FOR WINDOWS
- Nodejs
  - LINK
  - Use the current version v7.*.*
  - This will give you acccess to npm (node package manager)
- A code editor
  - i.e. VISUAL STUDIO CODE
- Java SE Runtime Environment
  - LINK
  - Required for Elastic Search

- Elastic Search
    - LINK
- PostgreSQL
    - LINK
- RabbitMQ
    - LINK
    - In case of missing Erlang on Windows: LINK

1.3 Access Requirements

You will need access to the following

- The main github REPOSITORY
- Your own git branch for development, preferably something describing the feature you are working on.
    - A stable release version is kept on the **master branch**, this branch requires a code review by a second person before accepting a pull request.
    - A stable development version is kept on the **dev branch**, this branch does not require code review but we do expect you to make sure all tests pass before merging your code.

1.4 Installation Guide

This section assumes you are using a terminal and that you have finished all the steps above.

Once you have access to the repository run the following command to clone the project:

```
git clone https://github.com/Kayui/lokaverkefni
```

Enter the folder holding the project and run the following commands:

```
npm install
```

- This installs all dependencies specified in package.json

```
npm -g install gulp
```

- This installs gulp globally, is required to use gulp in the terminal

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

- This creates the .env file which holds environment variables required to run the application that we don't want exposed on github. This includes user and password combos for services such as postgresql, elastic search and more.
- Consult a fellow developer about how to set this file up.

Create two empty databases via postgreSQL corresponding to the fields defined in your .env file for MOCK_DB_NAME and DB_NAME and their passwords.

Start Elastic Search and RabbitMQ as explained on their respective sites for your platform.

From the root of the project run the `gulp mocha` command to run all tests.

if no obvious errors come up the install process was successful.

2 Code Style Guidelines

2.1 Server Side and Tests

Note that the majority of the rules specified below are enforced by **jshint**.

2.1.1 Indentation

- Indentation should always be four spaces.

2.1.2 Variable names

- Camel casing
  - Example: `variableName`
- Always use descriptive names
  - Example: `let newUserToAuthenticate = {name: 'Skúli'}`
- Constants
  - Nodejs require() constants use camelCasing
    - Example: `const moment = require('moment')`
  - All other constants use uppercase with underscores
    - Example: `const SOME_CONSTANT = 5`
- Model references
  - Example: `this.SomeModel = connection.SomeModel`
- Types
  - Always use `const` if the variable should not change during its lifetime and `let` instead of `var` when applicable

2.1.3 Class names

- Pascal casing
  - Example `AdventureService`

2.1.4 Functions

- Camel casing
  - Example: `someMethod(param) { /* do something */ }`
- Callbacks and bindings
  - Use ES6 arrow syntax where possible
    - Callback Example: `someMethod(param1, (param2) => { /* do something */ })`
    - Binding Example: `someMethod: (param) => { /* do something */ })`

Quotation marks

- Always use single quote first and double quote within if required
  - Example: `let someString = 'this is some sarcastic "text", right?'`

2.1.5 Semicolons

- Always end lines where required with a semicolon

2.1.6 File names

- Naming and file structure convention pattern: type/name/nameType
  - Service Example: `services/user/userService`
  - Model Example: `models/user/userModel`

2.2 Client Side

2.2.1 Indentation

- Indentation should always be four spaces.

2.2.2 Variable names

- Camel casing
  - Example: `variableName`
- Always use descriptive names
  - Example: `let newUserToAuthenticate = {name: 'Skúli'}`
- Constants
  - Nodejs require() constants use camelCasing
    - Example: `const moment = require('moment')`
  - All other constants use uppercase with underscores
    - Example: `const SOME_CONSTANT = 5`
- Model references
  - Example: `this.SomeModel = connection.SomeModel`
- Types
  - Always use `const` if the variable should not change during its lifetime and `let` instead of `var` when applicable

2.2.3 Functions

- Camel casing
  - Example: `someMethod(param) { /* do something */ }`

- Callbacks and bindings
    - Use ES6 arrow syntax where possible
        - Callback Example: `someMethod(param1, (param2) => { /* do something */ })`
        - Binding Example: `someMethod: (param) => { /* do something */ })`
    - Caveat:
        - In the case of Vue do not use ES6 arrow syntax on instance properties or callbakcs. Doing so will result in `this` not pointing to the Vue instance. Fall back to the old function syntax in these cases.

Quotation marks

- Always use single quote first and double quote within if required
    - Example: `let someString = 'this is some sarcastic "text", right?'`

2.2.4 Semicolons

- Always end lines where required with a semicolon

2.2.5 File names

- Vue naming and file structure convention pattern:
    - Folder names are fairly forgiving, livereload is set up to find any `*.vue` file within `client/app/vue` and its sub folders, and you can require as needed.
    - Always capitalize route and component file names
        - Example: `vue/routes/Adventure.vue`
        - Example: `vue/components/Navbar.vue`

3 Gulp Helper Commands

3.1 Running the Application

gulp serve

- Starts the application locally at localhost:8080 by default or using the port declared in your .env file. Livereload will automatically track your changes and restart as required.

3.2 Testing and Documentation

gulp mocha

- Starts a test server on port 1337 and then runs all services and api tests.

gulp mocha --api

- Runs all api tests

gulp mocha --services

- Runs all service tests

gulp mocha –file filename --api

- Starts a test server on port 1337 and then runs all api tests for the file specified

gulp mocha –file filename --services

- Runs all services tests for the file specified

Note: All mocha tests also generate a code coverage report via Istanbul accessible at /coverage/index.html (open the file directly) while also printing out basic info in the console.

gulp document

- Converts comments above service methods into *.md files which document methods available. These are stored within /documentation/services/
- Converts comments above api routes into a single file called index.thml which is stored within /documentation/api/

3.3 Database Manipulation

gulp db:rebuild

- Rebuilds the postgreSQL production database to fit the sequelized model definitions as specified within /server/models/
- Note: Removes all data that exists in the production database

gulp db:seed

- Seeds the PostgreSQL production database with mostly random test data as specified within /server/models/ and then seeds it with data
- Modify the data generation in /server/database/config/seeder.js
- Note: Does not remove existing data, may fail if conflicting data exists

gulp db:rebuild:seed

- Rebuilds the postgreSQL production database to fit the sequelized model definitions as specified within /server/models/ and then seeds it with data
- Modify the data generation in /server/database/config/seeder.js
- Note: Removes all data that exists in the production database

Note: All db commands require the .env environment variable APP_ENV set as development, this should never be used in production as all data will be corrupted or lost. All of the database

manipulation commands should ideally be replaced with native migration and seed methods in the sequelized command line interface, consider that a future to-do.

4 Server Layout

4.1 Routing

4.1.1 HTTP verbs

router.get when declaring data fetching routes

router.post when declaring data insertion routes

router.put when declaring data update routes

router.delete when declaring data deletion routes

4.1.2 Commenting routes

All routing methods are automatically documented using APIDoc and should be commented according to their standards.

4.1.3 Status codes

Use the following status codes as described when returning data from the api to the client:

200: Request get success

201: Data insertion successful

204: No content, delete was successful

400: Bad request, format of the request was invalid

401: Unauthorized

403: Forbidden

500: Server error

4.1.4 Web routes

All web routes are delegated to Vue on the client side via one route that always points to /index and another that handles not found requests.

4.2 Service Layer

Stored within /server/services

These are the classes your API routes will delegate work to and fetch data from.

A service class should have the following format:

```
'use strict';
Class SomeNewService {
	constructor() {
	// this ensures that we use the correct database during testing
	if (processor.env.TEST) {
		this.setConnection(require('../../database/mockDatabase.js'));
	}
	else {
		this.setConnection(require('../../database/database.js'));
	}

	// Set the active database connection and make references to models
	setConnection(connection) {
		this.Connection = connection;
		this.SomeModel = connection.SomeModel;
	}

	// The comment block below is required for automatic api documentation
	/**
	* A description of the method
	* @name someMethod
	* @param {type} Description of a parameter
	* @returns {type} Description of the return value
	*/
	someMethod() {
		// We do some work that promises some response to the api
		return new Promise((resolve, reject) => {
			// Do some work in the datalayer
			this.SomeModel.someMethod().then((data) => {
				// success! We send it to the api
				resolve(data);
			}).catch((reason) => {
				// fail! We notify the api
				reject('description of what went wrong', reason);
			});
		}
	}
}
module.exports = new SomeNewService();
```

4.3 Data Layer

Stored within /server/models

These are the classes your service layer will delegate work to and fetch data from. These classes define your database models and the relations between them. They also handle work delegated by the service layer such as data fetching, inserting, updating and deleting.

A model class should have the following format:

```
‘use strict’;
const Sequelize = require(‘sequelize’);


Class ModelName {
        constructor() {}
        define() {
                this.ModelName = this.connection.define(‘modelName’, {
                        // Here we use sequelize syntax for defining tables
                        // Take a good look at the definitions for more details
                        someColumn: {
                                Type: Sequelize.TEXT,
                                allowNull: true
                        }
                }, { paranoid: true } // this adds a soft delete feature to the table

                // here we have multiple options for defining table relations
                // hasOne (1:1), hasMany (1:m) (
                // belongsTo (1:1), belongsToMany (n:m)
                // Take a good look at associations for more details


                // example use:
                this.hasMany = [
                        { ‘model’: ‘SomeOtherModel’ },
                        { ‘model’: ‘YetAnotherModel’ }
                ];
        }


        setConnection(connection) {
                this.connection = connection;
                this.define();
        }


        findAll() {
                // example of fetching all rows from the database
                // including all associations
                // Take a good look at querying for more details
                return this.ModelName.findAll({
                        Include: [ { all: true } ]
                });
        }
}
module.exports = new ModelName();
```

4.4 Testing the Service Layer

Stored within /server/tests/services/

These are the unit tests that test the service classes.

Each service has its own test file /server/tests/services/nameOfServices/test.js

and a mock db seed file /server/tests/services/nameOfServices/data.js

A unit test class should have the following format:

```
'use strict';

const db = require('../../../database/database.js')
const assert = require('assert');
const someService = require('../../../services/some/someService.js');

// here you write the name of the service you are testing
describe('#### NameOfServiceClassBeingTested', () {

        before((done) => {Being
                // Sync will create all the needed tables (force: tears them down first)
                mockConnection.connection.sync({ force: true }).then(() => {
                        // this will seed the database with test data
                        require('../serviceName/data.js')(db).then((result) => {
                                done();
                        });
                });
        });

        // Below is an example of a unit test
        it('Here is a short description of what is being tested', (done) => {
                // Arrange data
                let someObject = {...}
                // Act
                serviceToTest.functionToTest().then((data) => {
                        // Assert
                        assert.equal(data.id, 1);
                        // done() muser be called after all assertions
                        done();
                });
        });

});
```

4.5 Testing the API Layer

Stored within /server/tests/api

These are the unit tests that test the API classes.

Each API has its own test file /server/tests/API /nameOfAPI/test.js

A super test class should have the following format:

```
'use strict';
const app = require('../../../app');
const faker = require('faker');
const request = require('supertest');
const assert = require('assert');
const dotenv = require('dotenv');

// here you initialize all agents/users that are tested
const client = request.agent(app);
const anon = request.agent(app);
const admin = request.agent(app);
const seller = request.agent(app);


// here you write the name of the API you are testing
describe('#### NameOfServiceClassBeingTested', () {

        before((done) => {Being
                // Sync will create all the needed tables (force: tears them down first)
                mockConnection.connection.sync({ force: true }).then(() => {
                        // here you can initialize all data that must exist in the db before testing
                });
        });

        // Below is an example of a super test
        it('Logging in as admin', (done) => {
                // What agent is acting
                admin
                        // Type of request and URL
                        .post('/api/auth/login')
                        // Body of request
                        .send({ contactEmail: 'admin@wii.is', password: 'something' })
                        // Status code expected
                        .expect(200)
                        // What to do with the data
                        .end((err, res) => {
                                if (err) { done(err); return; }
                                // Assert
                                assert.equal(res.body, 1);
                                done();
                        });
        });

});
```

5 Client Layout

5.1 Routing

As mentioned before, all web routing is handled by Vue on the client side.

This is done within /client/app/vue/routes and declared in /client/app/vue/main.js

5.1.1 Defining web routes

Web routes are handled in individual files that represent pages, to declare a new web route simply add a new block within the routes array in /client/app/vue/main.js in the following format:

```
// create this file first within client/app/routes
import RouteVariableName  from ‘./routes/RouteFileName.vue;
{
	path: ‘/url-path’,
	component: RouteVariableName,
	name: ‘nameForRoute’,
	beforeEnter: (to, from, next) => {
		permit(to, from, next, [‘roleToPermitAccess’]);
	}
}
```

path:

name:

beforeEnter with **permit** is an abstraction around the navigation guards present in vue router, this allows you to permit only access to specific user roles as an array of role names. Omit this field if your route should be available for all users including those not signed in.

Available roles: **‘client’**, **‘seller’** and **‘admin’**.

5.1.2 The Vue route file

A *.vue file comes in the following format:

```
<template lang=”pug”>
// all html is written in pug empowered by vue
#some-page-container
        // this repeats the paragraph element for every string within the array
        p(v-for=”item in someArrayOfString”) {{ item }}
        // this displays an external component and passes in a property
        SomeExternalComponent(:someProperty=”’nice’”)
</template>

<script>
Import SomeExternalComponent from ‘../components/SomeExternalComponent.vue’;
export default {
        // this gives you access to the logged in user (is null if not logged in)
        props: [‘user’],
        // this registers a component with the instance, allowing for use
        components: { SomeExternalComponent },
        // define required variables here
        data () {
                return {
                        someArrayOfString: []
                }
        },
        Methods: { // define required methods here
                someMethod: function () {}
        },
        created: function () {
                // this method is called automatically when the component is started
                // this is the perfect place to call an api resource to fetch data
                // you want to display for the user
                axios.get(‘/api/some-resource’).then((response) => {
                        // this will take the result of the api call
                        // and bind it to an instance variable defined within data()
                        this.someArrayOfString = response.data;
                }
        }
}
</script>

<style lang=”less”>
// all css is written in less
// make sure this is correct relative to your file
// this is a link to the global less file holding premade variables
@import ‘../globals.less’;
</style>
```

External components are written in the same manner and stored within /client/app/vue/components.

For more information visit the VUE DOCUMENTATION.

5.2 CSS

5.2.1 Bootstrap

The Bootstrap framework is used for the frontend css. For more information, visit the BOOTSTRAP DOCUMENTATION.

5.2.2 Less css

Less is used to fuel all css development. To make things even easier and to keep a design consistency between pages several constants have been declared within /client/app/vue/globals.less:

```
// global colors
@wii-purple, @wii-dark-gray, @wii-light-gray, @wii-light-gray-2,
@wii-white, @wii-blue, @wii-red, @wii-green, @wii-yellow

// variables
@wii-nav-height, @wii-subnav-height, @wii-searchnav-height
```

It is our aim to reuse as much css a possible. All global scope css classes shall be defined in /client/app/vue/App.vue, so before you declare a new class, be sure that you cannot reuse what is already declared. If you want to specify css only used by a single template, be sure to contain it within the id of the template:

```
<style lang="less">
@import './globals.less';
#template-id {
        //some specific css
}
</style>
```

Several components have also been pre-defined within /client/app/vue/App.vue:

```
a.wii-button
button.wii-button
.wii-modal
.wii-title
.media-tiny-profile
.media-small-profile
.media-medium-profile
```

For more information visit the LESS DOCUMENTATION.