Overview of BC Learning Network SMS2

Introduction

This guide is designed to be a cumulative overview of the SMS2 web application. SMS2 is a student management system which integrates with Moodle, a learning management system (LMS) employed by many school districts and universities around the world. SMS2 also integrates with MyEdBC, British Columbia's central student data repository.

This document will cover both business logic and programmatic logic of the application, and attempt to orient new developers to the project. If you have any questions while reading this document, feel free to contact the author at chris.james.foster@gmail.com.

This document is up-to-date as of: March 3rd, 2016

Table of Contents

Expected Knowledge

This documentation is written with two key expectations:

1. **You have basic familiarity with web development.** You have already used and have experience with technologies like Git, Javascript, CSS, HTML, and a server side language of some type. If you don't have basic prior experience with web development, you may find the following documentation difficult to read and may find it challenging to contribute to a project of this size.

2. **You are willing to learn the involved technologies.** This documentation is written as an overview of the concepts and technologies used within SMS, and assumes that you will learn any technologies you are not already familiar with. Each respective technology has better tutorials on their project pages

than we could write here, so this documentation will only describe each technology briefly and will primarily focus on their relevance to SMS. It will be very difficult to contribute to this project without understanding all of the technologies in this document.

Technologies in SMS

This section provides a brief overview of the technologies we use in SMS. If you are not familiar with any of these technologies, please read through their websites and appropriate guides or tutorials. You will need to be familiar with all of these technologies in order to develop any complex code in SMS.

Angular

AngularJS is a client side javascript framework for writing single page web applications (SPAs). The entire client side codebase for SMS is written in Angular, and you will need to know it to develop any frontend code. Angular is highly popular and supported by industry companies such as Google and Microsoft. Angular was primarily chosen over frameworks such as React or Ember due to the developer's prior experience with angular and the project's high level of maturity.

Angular provides structure and organization to the entire client side application, which has allowed SMS to grow to its current size without collecting substantial code cruft and will allow future expansion of the codebase. It also provides two way data binding which enables us to achieve a highly interactive application with substantially less code than traditional methods.

Gulp

Gulp is a build tool for running any build related tasks for the application. For SMS, this primarily involves minifying and concatenating the dozens of client-side javascript files into a single application file. It also manages the compilation of our SASS code into CSS (covered below). Lastly, it builds and generates our PDF templates for use at runtime. Gulp is covered more in the section titled "Building".

SASS / Bootstrap

SASS is an expansion of the traditional CSS language which provides substantially more power to developers. All of our CSS is a result of compiled SASS code. SASS adds traditional programming language constructs such as variables, loops, and conditionals. This allows us to integrate into SASS frameworks and write much more modular code.

We use BEM syntax and aim to write modular and reusable SASS in every way possible. SASS provides numerous extensions over CSS that allow us to hold a higher standard of code quality for our stylesheets.

Bootstrap is a highly popular CSS framework for building web applications. We use bootstrap as it saved us substantial time when approaching the design of the application, and it aids us with structuring and provides common helper classes. We use Bootstrap's SASS code directly and it's compiled alongside our own.

Elasticsearch

Elasticsearch is a database for advanced querying. Data is loaded from an authoritative data store, in our case RethinkDB, into Elasticsearch (they call this "indexing"). After data has been indexed, we can query Elasticsearch and it can return us data based on either fuzzy logic or complex queries. We primarily use elasticsearch to power the student search box, the course search boxes, and the student record search.

It can be challenging to ensure that data in Elasticsearch is kept in sync with the authoritative RethinkDB tables, and they can become out of sync in error conditions. Additionally, we currently do not have the fuzzy search optimized to the level such that is performs as we'd like. However, elasticsearch remains heavily useful to us for complex queries and text searches.

Node / Express

Node is the server side environment that we use to power the application. Node allows us to run javascript on the server, which gives us numerous benefits. Firstly, it does not require developers to learn a new language. Secondly, it allows us to develop the back-end and front-end with the same language. And thirdly, javascript on the server provides a nice working environment for dealing with JSON. Node's event-loop based architecture makes it's performance ideal for an IO server environment with low processing work such as ours.

Express is a common web framework for node. While node manages the execution environment for our server-side javascript, express handles things such as routing, middleware, parameters, request parsing, and so on. All of the API and request handling code is structured within express.

RethinkDB

RethinkDB is a NoSQL database with a powerful query language and support for SQL-like joins. We use RethinkDB as the main data store for all of the application's data and to also store cached copies of exported data from Moodle. RethinkDB's extremely powerful query language and join system allows for very flexible and advanced queries, but misuse can result in slow performance which is currently problematic for a few parts of the application. In some areas of the application, we use Thinky, which is an ORM for RethinkDB. Due to limitations with Thinky, we've slowly been moving code away from it and mostly use it to represent standard models such as the Student and Enrolment models.

Redis

Redis is a very fast key-value store. We don't interact with redis much within the application itself, and instead it is utilized by our session management plugin. Whenever someone logs into SMS, the session is stored within redis. This allows us to retain user logins between restarts of the application and scale the number of web processes to more than one (if necessary).

Docker

Docker is a container-management application. A 'container' is similar to a lightweight virtual machine. It has tools for managing sets of containers, scaling containers to large numbers, and storing containers on their public infrastructure. The docker components we expect you to be familiar with are the docker engine and docker-compose.

Nginx

SMS itself isn't directly aware of nginx, but we use nginx in our recommended docker-compose setup. In production, **SMS should be hosted behind an SSL reverse proxy**. SMS does not support SSL directly, so you'll have to use a terminating proxy such as nginx to do this. Additionally, scaling your installation with docker-compose becomes significantly easier if you have a tool like nginx to direct requests to SMS containers (although for most districts this is not a concern).

Docker Structure

Please see the installation document for an explanation of how we store the application and it's dependencies inside docker.

Package Manager

Our project is almost entirely javascript, so we use the node package manager (npm). Npm's configuration is primarily stored in a single file, `package.json`. This file manages all of the dependencies used within SMS and describes most of the application tasks it supports. For an introduction on getting the application running with SMS, see the project's README. Dependencies should *never be checked into the repository*.

Application Structure

This section will break down the organization of the project at the filesystem level and the layout used by SMS.

/ (root)

The project's root contains very high level files for SMS. The server's main entry point, `server.js`is present here. Additionally, the `package.json`, `gulpfile.js`, and `Dockerfile` all describe the contents, dependencies, and build process of SMS. We also store a `schools.csv` file which is a hardcoded list of all schools in British Columbia, and a `achievements.csv` that we generate as part of the application tasks is stored here.

/client/

The client directory contains all template and javascript files related to the front-end of the application. It's broken down into a number of separate applications: `login/`, `parent/`,`registration/`, `staff/`, and `student/`. Additionally, a `common/` directory is present for components that are shared between numerous applications. When a user is not logged in, the server will redirect them to the login application. After they've logged in, they'll be redirected to whichever application fits their user type. The registration application is an extra module that is accessed by students who want to sign up to SMS and register for courses.
Note that the parent application and student application are currently empty, they have no content but were originally planned for development.

/dist/

The `/dist/` directory contains compiled files generated by the build process. The client will load it's source files from this directory. The PDF and email templates are also stored here. Until you've ran the build process, you likely won't see anything in this directory or it may not be present.

/node_modules/

The `/node_modules/` directory is where npm stores the dependencies of the project. While we reference this directory a few times in the source code, and `require()` calls will load files from here, you should never edit the contents of this directory. You won't see anything in this directory until you've run `npm install`.

/server/

This directory contains all of the server-side code. Any templates that are rendered on the server and all of the route definitions and their related code are stored here. At the root of this directory, the `app.js` file defines the main route configuration and middleware setup. All other files in the directory represent more generic modules that are used by one or more routes at different points.
The API routes themselves are stored in the applicable directories: `login/`, `parent/`,`registration/`, `staff/`, and `student/`. These routes are used by the client application via a REST API. We use express 3 subrouters to structure the API, and the subdirectories in each folder match the API route. So for example, the API endpoint at `/api/staff/lms/teachers` can be found at `/server/staff/lms/teachers.js`.
The `external/` directory also contains routes that render the HTML to start the client-side applications in the user's browser. The user will visit these pages, which starts the application that can then interact with the appropriate REST API endpoints.
The `models/` directory contains the database models represented in RethinkDB. This involves a few helper methods attached to each model as well.
The `templates/` directory contains the source for the client-side application launch pages rendered by the server and both the email/PDF templates which are compiled during the build process.

/styles/

This directory contains all of the SASS code written in the application. The SASS code in the`sass/` subdirectory will be compiled by the Gulp build process and stored in the `compiled/`subdirectory, from which it will be served to the client. All files that are not `main.scss` should start with an underscore so they are not built separately by the SASS compiler.
In the `sass/` subdirectory the `main.scss` file is the actual file that is compiled and served to the client. This file exclusively contains `@import` statements for the dependencies, generics, extensions, and components. We also define project-

wide configuration variables in this file. If you add a new SASS file you'll need to update this file to see it included in the CSS output.
The `generic/` directory is used to store SASS components that are highly flexible and self contained. They could potentially be used in other projects since they are not dependent on any design or business logic in SMS. In an ideal scenario, you want to write code that can be stored in `generic/`.
The `extensions/` directory is used to add extensions to already imported code. The `main.scss`file includes numerous frameworks, and the most notable one is Bootstrap. In a few places we need to alter or extend Bootstrap related code, so this can be done in `extensions/`.
The `components/` directory is for self-contained components that are not modular enough to be used in another project. These are design elements and sets of classes specifically designed to implement a component in SMS. A good example of a component would be the SMS student feed (`_student-feed.scss`), since it is specific to SMS and the classes are not likely to be reusable in another project. Something such as a class for centering elements however (`_center.scss`), would go into `generic/`.

/task/

This directory is for storing one-time application tasks to be ran. While the server itself is started and continually listens, the developer tasks perform a single operation and then exit. The tasks are primarily used to setup a new installation, import data into the application, reindex the data in RethinkDB or Elasticsearch, or sync with Moodle or MyEdBC.

Many of the operations triggered by developer tasks are also triggered regularly by the application, so the developer task simply makes a one time call to the worker script.

Migration tasks are also stored in the `migration/` subdirectory. Database migrations all *must*follow the format of `UNIXTIME-what-it-does.js`. For example: `1448061918-add-course-mapping.js`. On startup, SMS will read this directory and automatically run any database migrations necessary by calling the exported `run` function on that module. The migration, if successful, only runs once.
Note: All of the tasks perform a single operation, which means that some tasks may cause an index to become invalid or out of date. To fix this, you'll have to run a reindex on whatever data is out of date. For this reason, these tasks are for development or setup purposes only and you shouldn't run any of them in production.

All of the tasks should have a comment at the top of the file describing what the task is for.

/worker/

This directory contains operations that SMS runs regularly. At startup, each of these worker processes is started. They all maintain their own timers and perform the operation automatically on each interval. These are used for things such as syncing with Moodle, loading the list of schools, generating the `achievments.csv` file, and so on.
Some scripts in the `task/` directory call these worker scripts as part of a developer task, so many of the workers are written in ways that allows them to be called either once or on an interval.
All of the workers should have a comment at the top of the file describing what the task is for.

Github Workflow

Development

We use standard Github fork-and-branching workflow. The standard Github flow requires that you create a new branch for your changes, create commits on that branch, and use a *pull request*to merge the changes in.

In addition to Github flow, all branches should be created on forks on the SMS repository. *You should not create branches or commits directly on the SMS repository*. Making pull requests with forks is covered in Github's documentation. Forks mean that even people without write access to the main repository can work on SMS, and it helps to avoid making Git mistakes on the main repository.

If you don't have write access and you make a pull request, someone who has write access to the main repository will need to review your work and accept it.

The SMS repository has two main branches to be aware of: `master` and `dev`. Master represents the current stable version of the application, and dev represents the current development branch. Since forking is used, there shouldn't be any other branches on the main repository. All pull requests from forks are made to the `dev` branch.

Releases

For instructions on creating a release, see the releases document.

For details on steps 5 and 6 in the release instructions, see the section on 'Build process' below.

Build process

There are two types of 'building' in SMS:

1. Instructions for building the application are covered in the README.
2. Building a Docker image is described in the section on Docker. Building the docker image will indirectly build the application itself (using the instructions in the `Dockerfile`).

User Viewpoints

This section describes the various users that will use SMS and the current features available to them.

- **Admin** - An admin will have complete access to the application. They are redirected to the staff application upon login, and have full access to all of the features inside the staff application such as the admin section and course section.
- **Clerical** - A clerical user has mid-level access to the application. They are redirected to the staff application upon login, but cannot perform any administrative activities. Clerical users will primarily be able to manage students in SMS, rather than any SMS-level settings.
- **Counsellor** - A counsellor has low-level access to the application. They are redirected to the staff application upon login, but have view-only access to any data. They are able to leave comments on students, but cannot perform any other activities. They do not see any SMS-level settings, like the clerical users.
- **Student** - A student is a student who is currently enrolled in a school that is managed by SMS. The student application in SMS is currently empty and cannot be used to do anything, and students cannot login to SMS like staff can. The registration application, however, is intended for students and is their primary interaction with SMS. They can register for an account and signup for courses through the registration application.
- **Parent** - A parent is the guardian of a student currently enrolled in a school that is managed by SMS. Parents have no access to the system at this point, cannot login, and the parent application contains nothing at this point.

External Integrations

This explains the external systems that SMS integrates with. It does not include the components that are expected to be available within a Docker instance, such as RethinkDB, Redis, Elasticsearch, and Nginx.

Moodle

Moodle is the learning management system that SMS administrates. The setup for Moodle is extensive and covered in our installation document. Within the codebase, we often refer to Moodle internally as the 'LMS'. In user-facing text, we refer to it as 'Moodle'. In the future, SMS may be expanded to other learning management systems. The interface points between SMS and Moodle are:

- `server/lms.js` - Defines numerous functions used within different API routes for real-time access to the LMS.
- `worker/updateCourses.js` - Syncs the list of courses in Moodle with SMS's internal cache of courses
- `worker/updateEnrolments.js` - Syncs all students' course data cache (grades, progress, etc..) with the current data in Moodle.

MyEdBC

MyEdBC is British Columbia's province-wide repository of student data. We do not interface with MyEdBC directly, but we instruct clerical users to perform actions in MyEdBC and we work with the data exported from MyEdBC.

Districts must generate a regular export of all their data in MyEdBC and store the file in a directory accessible to SMS. This is detailed in the installation document. SMS will sync with this data file, and update any students who have had their information changed. Linking between SMS student and MyEdBC student accounts is done manually by a clerical user, with recommendations provided by SMS. The `updateRecords` worker handles the syncing with the data file. Once a student is linked with a MyEdBC record, we consider that the authoritative copy of their data and don't allow anyone to edit that students information within SMS anymore (it should instead be done in MyEdBC). Whenever a student achieves the 5% completion in a course or finishes a course, we ask the clerical user to update the student's status in MyEdBC. We currently have no way to check that this is actually done so they sign off on it using a checkbox interface in SMS.

It is possible to enable or disable MyEdBC integration using an environment variable, since not all districts are using MyEdBC. In this case we don't show any MyEdBC related data and always allow manual editing of student data in SMS. It's important to consider if your code needs to behave differently based on MyEdBC status of an installation and write code to handle that.

Email

We integrate with an external email server to send mail. This is configured through an environment variable, which can be specified in the docker-compose setup. Our email usage is fairly standard, but we don't provide this ourselves and expect it to be configured to send to any domain such as outlook or gmail.

You cannot run SMS without an email installation, as users will receive their passwords via email and other important information is sent that way.