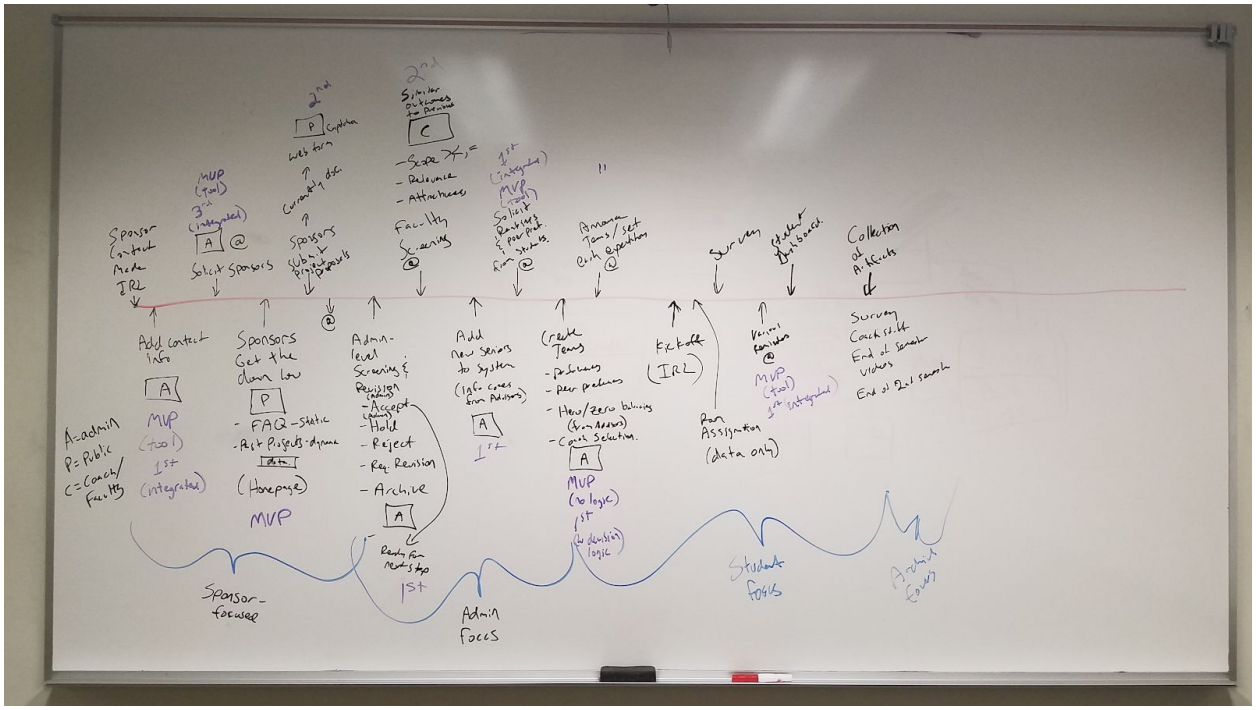
Senior Project Web Util Reference

Tom Amaral
6/23/2020

Contents

Overview

This document serves as a high level reference for the Senior Project Web Utility. The intention of the SPWU is to provide an all-encompassing digital management system for the Software Engineering Department's Senior Project program. This includes the entire lifecycle, from soliciting and managing project sponsors, to the selection and curation of proposals, to tracking and handling student progress throughout the project block, and more. Below is an image of the rough outline of the aforementioned lifecycle that was written during the initial design of the SPWU.



Getting Set Up For Development

The SPWU itself is originally designed in Node.js/Express.js on the backend, SQLite for the data layer, and a traditional HTML/CSS/Javascript front-end utilizing Semantic UI. It was developed using VSCode and Firefox as the two main dev tools.

Web Server Overview

The entire SPWU project is hosted on the SE Department's servers. It can be accessed by SSH at the address `seniorproject.se.rit.edu` on the default SSH port. An account must be created on that VM in order to access the server. The server is a Linux machine running Ubuntu server. As of the time of writing, the project is entirely stored under the directory ~/txa2269/site. There are shell scripts in place to start and stop the Node.js server.

Git

The SPWU is tracked by Git version control, which is also hosted on the SE server. There are two branches, master and dev. Master is not for development, as it is intended to be the production ready project code. The changes pushed to master will be the files that are actually hosted and available via the public URL (`seniorproject.se.rit.edu`). For development, the dev branch must be used, as the master branch cannot be checked out due to the way the Git repository is configured. It is important to note that you may have to place an SSH key for your machine on the SE server so that you are authorized to access the git repo through SSH. To pull the branch (after gaining access to the server), one can initialize an empty repo locally and call

```
$ git remote add origin seniorproject.se.rit.edu
$ git pull origin dev
```

To push to the dev branch,

```
$ git push origin dev
```

To merge the dev changes to master, on the server call

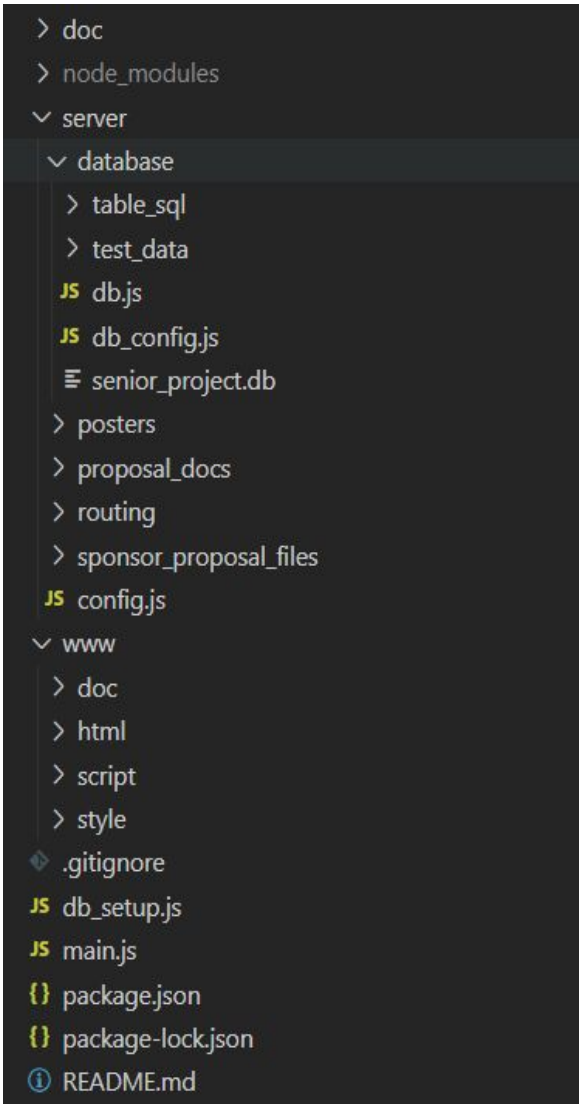
```
$ git merge dev
```

It is important to note that if you ever add any package dependencies, you must run

```
$ npm install
```

on the /site directory on the SE server to fetch the new packages.

Project Structure



Displayed to the left is an image of the project directory structure. Individual files will be covered in their respective sections later on. Starting from the top, we have the /doc folder. This holds a basic readme. /node_modules contains all the packages and modules necessary for Node.js to function. The /server directory contains all the backend code and resources. /database contains any resources pertaining to database interactions. The /table_sql directory contains .sql files that define the schema of the database. /test_data contains some .sql files with inserts containing some test data that I created. The /posters directory contains the image files referenced by the projects in the database for the posters each team must create. /proposal_docs is an empty directory that is filled with a pdf version of the sponsor proposal form data submitted by potential sponsors on the front-end. /routing contains the express.js routes necessary for handling REST calls and serving html. /sponsor_proposal_files is a directory for dumping file attachments from proposal form submission.

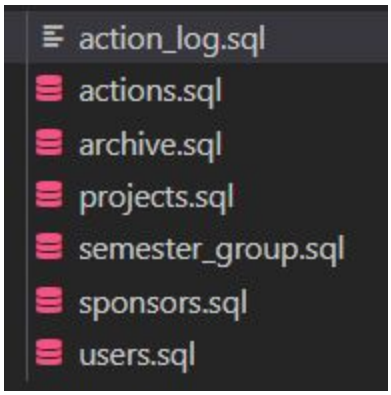
Moving to the client side, /www holds all public facing resources. /doc is for any public facing documents. /html holds all html files. /script contains all javascript files. /style contains all css files.

Running & Debugging

Main.js is the entry point of the application. To start the Node.js server, one can use either
`$ node main.js  OR $ nodemon main.js`
Nodemon is a developer tool that automatically relaunches the server after changes have been made to the project. Very handy. Throughout this project, I was using Firefox in Private Browsing mode to avoid any caching weirdness.

Database Structure

Table Schema



The table names within the database are listed in the image to the left. action_log is a logging table that contains the rows that describe a user's submission for an action (more on actions in client-side reference). actions table describes the attributes of an action. archive is a table that contains rows with project information of projects that have been completed and are no longer active. semester_group is a simple table that outlines necessary information to describe the two semesters that make up a project block. sponsors is a table to store sponsor information, entered by an administrator. users defines the table that contains all user information for those involved with Senior Project in one way or another.

```
CREATE TABLE action_log (
    action_log_id       INTEGER PRIMARY KEY AUTOINCREMENT, -- internal id for this row
    action_template INTEGER NOT NULL,                 -- the id of the template action this is formatted on
    system_id       INTEGER NOT NULL,                 -- system ID of executor
    project         INTEGER NOT NULL,
    form_data       BLOB,
    files           BLOB,
    FOREIGN KEY (action_template) REFERENCES actions(action_id),
    FOREIGN KEY (system_id) REFERENCES users(system_id),
    FOREIGN KEY (project) REFERENCES projects(project_id)
);
```

```
CREATE TABLE actions (
    action_id       INTEGER PRIMARY KEY AUTOINCREMENT,
    semester        INTEGER NOT NULL,
    action_title    TEXT,    -- The title of the action
    action_target   TEXT,    -- individual, coach, team, admin
    is_null         INTEGER, -- 1 for true, used for calendar events, no-op events, etc
    short_desc      TEXT,
    start_date      TEXT,
    due_date        TEXT,
    page_html       TEXT,    -- HTML for the page of
    FOREIGN KEY (semester) REFERENCES semester_group(semester_id)
);
```

```sql
CREATE TABLE archive (
    archive_id      INTEGER PRIMARY KEY AUTOINCREMENT,
    project_id      INTEGER,
    priority        INTEGER, -- how high a project should be displayed on clientside
    title           TEXT,
    team_name       TEXT,
    members         TEXT,
    sponsor         TEXT,
    coach           TEXT,
    poster_thumb    TEXT,    -- path to poster thumbnail
    poster_full     TEXT,    -- path to full resolution poster image
    synopsis        TEXT,
    video           TEXT,    -- path to project video (if any)
    FOREIGN KEY (project_id) REFERENCES projects(project_id)
);
```

```sql
CREATE TABLE projects (
    project_id              INTEGER primary key autoincrement,
    status                  text,   -- Submitted, Needs Revision, Future Project, Candidate, In Progress, Complete
    display_name            text unique,
    title                   text unique,
    organization            text,
    primary_contact         text,
    contact_email           text,
    contact_phone           text,
    attachments             text,
    background_info         text,
    project_description     text,
    project_scope           text,
    project_challenges      text,
    constraints_assumptions text,
    sponsor_provided_resources  text,
    project_search_keywords text,
    sponsor_deliverables    text,
    proprietary_info        text,
    sponsor_avail_checked   int,
    sponsor_alternate_time  text,
    project_agreements_checked  int,
    assignment_of_rights    text,
    team_name               text,
    poster                  text,
    video                   text,
    website                 text,
    synopsis                text,
    sponsor                 integer,
    coach1                  text,
    coach2                  text,
    semester                integer,
    FOREIGN KEY (semester) REFERENCES semester_group(semester_id),
    FOREIGN KEY (sponsor) REFERENCES sponsors(sponsor_id),
    FOREIGN KEY (coach1) REFERENCES users(system_id),
    FOREIGN KEY (coach2) REFERENCES users(system_id)
);
```

```sql
CREATE TABLE semester_group (
    semester_id INTEGER PRIMARY KEY AUTOINCREMENT,
    name        TEXT UNIQUE NOT NULL,
    dept        TEXT,
    start_date  TEXT,
    end_date    TEXT
);
```

```sql
CREATE TABLE sponsors (
    sponsor_id   INTEGER PRIMARY KEY AUTOINCREMENT,
    fname       TEXT,
    lname       TEXT,
    company     TEXT,
    division    TEXT,
    email       TEXT,
    phone       TEXT,
    association TEXT,
    type        TEXT,
    notes       TEXT
);
```

```sql
CREATE TABLE users (
    system_id   TEXT PRIMARY KEY NOT NULL UNIQUE,
    fname       TEXT,
    lname       TEXT,
    email       TEXT,
    type        TEXT,
    semester_group  INTEGER,
    project         INTEGER,
    FOREIGN KEY (semester_group) REFERENCES semester_group(semester_id),
    FOREIGN KEY (project) REFERENCEs project(project_id)
);
```

Creating & Managing Test Data

A utility script exists for rapidly deploying the database schema and any test data in the /table_sql and /test_data directories. In the top level of the project, there is a script called db_setup.js. Calling the script will reset the table schema and insert any test data available. Be sure to not call the script on the production system, as **it will wipe out any rows in the database**

Client-Side Overview

As of the time of writing, there are 2 public facing web pages accessible from the front page. The front page itself, and the sponsor page. From the sponsor page, one can get to the proposal submission page, and the submission result pages (success/ failure). The only existing private page is the admin page.

Public Facing Functionality

The current implementation supports the dynamic displaying of high priority archived projects on the front page, with information blurb and synopsis. Additionally, the sponsor page supplies a small FAQ and a link to the proposal form. The proposal form can be considered functionally complete, and the backend is equipped to store the form information into the database, along with a generated pdf version, and any file attachments. The form is validated on both client and server side.

Admin Dashboard

The admin dashboard is currently capable of displaying all active semester blocks, with their respective teams, and the actions required of those teams for their given semester group. Clicking on an action opens a tooltip with some basic information, and a button to open the action's own page html. this will most likely contain a form and some basic text, depending on the action. **Actions are any department deliverable or action required by the team to complete the academic requirements of senior project.** On submission, the action page redirects to a submission success page and then routes the user back to the admin page. As of right now, the form is submitted to the back end, but there is no implemented functionality to handle the form submission, other than a redirect to a submission successful page.
In addition to the dashboard, there is also a proposal tab with rudimentary display of proposal form pdfs and attachments.

Back-End Architecture

Database Handler

Inside db.js lies wrapper functions for sqlite actions. The most important is the query() method, which takes a prepared sql statement string and an array of params, and executes it against the database. Any values are returned in a promise.

```
/**
 * Takes a sql statement and corresponding array of values and executes it.
 * @param {String} sql The sql query to execute. Use prepared statements.
 * @param {Array} values corresponding values, if any, to prepare into the query
 * @returns {Promise} The resulting rows, if any, of the query. For operations such as insert, it will be empty.
*/
query(sql, values = []) {
    return new Promise((resolve, reject) => {
        this.openReadWrite();
        if (this.seniorProjectsDB) {
            this.seniorProjectsDB.all(sql, values, (err, rows) => {

                if (err)
                    reject(err);

                else
                    resolve(rows);
            });
        }
    });
}
```

db_config.js contains some string mappings to variable names, but was never fully realized as a config, due to time constraints on development.

Endpoint Architecture & Routing

All HTTP routing can be found in server/routing. index.js contains the endpoints for simple page serving, while db_routes.js contains all of the current business logic. Of the routes displayed below, the top four are stubs. /getProposalPdfNames and /getProposalPdf handle the displaying of the proposal form submission pdfs on the admin dashboard. The same goes for /getProposalAttachmentNames and /getProposalAttachment. As you might notice, there is an extra property on these four routes, CONFIG.authAdmin. This is a basic authentication header function that checks if the user is authorized. The method is located in config.js, and is mostly a stub intended to be replaced by the SE department's authentication API.

Suggestion: Modify the /admin endpoint to dynamically serve a dashboard suited for the level of user attempting to login i.e. send a student to the student version of the dashboard, etc.

/submitProposal is the route responsible for handling the submission of sponsor proposals. An example of server side validation using express-validator is present in the function parameter of the route handler.

/getActiveTimelines currently calls a helper method to construct a JSON object containing all the active timelines of project teams in progress, their actions, and action status, among other information. The helper method, calculateActiveTimelines() contains a very long and seemingly complex query (but it's really not) that joins all the necessary tables and does a manual conversion to JSON string output for the actions attached to a timeline.

/submitAction is a stub of a route that handles the submission from the form of an action.

```
// Routes
db_router.get('/selectAllSponsorInfo', (req, res) => {···
});

db_router.get('/selectAllStudentInfo', (req, res) => {···
});

db_router.get('/selectAllCoachInfo', (req, res) => {···
});

db_router.get('/selectExemplary', (req, res) => {···
});

/**
 * Responds with a list of links to pdf versions of proposal forms
 */
db_router.get('/getProposalPdfNames', CONFIG.authAdmin, (req, res) => {···
});

db_router.get('/getProposalPdf', CONFIG.authAdmin, (req, res) => {···
});

db_router.get('/getProposalAttachmentNames', CONFIG.authAdmin, (req, res) => {···
});

db_router.get('/getProposalAttachment', CONFIG.authAdmin, (req, res) => {···
});

//#endregion

db_router.post('/submitProposal', [···
],
async (req, res) => {···
});


db_router.get('/getPoster', (req, res) => {···
});


db_router.get('/getActiveTimelines', CONFIG.authAdmin, (req, res) => {···
});

db_router.get('/getTeamTimeline', CONFIG.authAdmin, (req, res) => {···
});

db_router.post('/submitAction', [···
], (req, res) => {···
})
```

Further Notes & Advice

- Break out db_routes.js into a more single-responsibility type endpoint architecture
- If the SE department's login API does not provide a level of user, check the system id against the user table to find out user auth level
- Always be wary of SQL injection, don't trust user input, etc
- try to find an open source front end module for interacting with the database from the client side, don't try to do it from scratch
- A lot of my decisions may come off as questionable, or even wrong. Don't worry, your time will come to face similar situations (: