Initiation Documentation Version 2

Main Files:

1. CSS files
   1. **PromoStyle.css**
   2. **searchingStyle.css** (Used only by search\_demo.html).
   3. **newSearch.css** (Used only by search.html).
2. HTML files
   1. **Project\_Initiation\_home.html**
   2. **ProjectForm.html**
   3. **PromoForm.html**
   4. **ProToProj.html**
   5. **search\_demo.html**
   6. **search.html**
   7. **AddBilling.html**
   8. **rolodex.html**
   9. **close.html**
   10. **editMe.html**
   11. **thanks.html**
3. Corresponding HTML to JavaScript files:
   1. **[NONE]**
   2. **ProjectVal.js**
   3. **ProjectPro.js**
   4. **ProToProj.js**
   5. **search.js**
   6. **newSearch.js**
   7. **home.js**
   8. **rolodex.js**
   9. **close.js**
   10. **editMe.js**
   11. **[NONE]**
4. Server files:
   1. SHNserver.js
   2. SHNserver2.js
   3. package.json
   4. package-lock.json
   5. Node-modules (directory containing library files and directories)
5. Files needed for Microsoft Sign In:
   1. authConfig.js
   2. authPopup.js
   3. graph.js
   4. graphConfig.js
   5. ui.js
6. Other files:
   1. favicon.ico
   2. cert.pem
   3. key.pem

These files are best kept within the same directory. The exception here is to keep server files, its required libraries, and the certificate files together. And all other client-side files can be kept in another directory. If the above files need to be separated out more for whatever reason, then their paths will need to be updated in other files accordingly.

Server Files**:**

All server files run using **Node.js**. Therefore, the scripts run on the server will require **package.json** and a **node\_modules** directory containing all the library data needed to run within the scripts. When adding new libraries to our toolset, these files and folders will get created and/or updated when installing them via command line with “npm install [Optional scope] [package\_name]” if Node.js and chocolatey are installed on that machine.

**SHNserver.js** Runs using Node.js.

Libraries:

* Node-adodb
* Express
* https
* Cors
* Body-parser
* Pdfkit-table
* pdfkit
* Fs
* Fs-extra
* Nodemailer
* Win-permissions-js

**SHNserver2.js** is a continued version of SHNserver.js. It contains the same libraries as SHNserver.js but it runs on port 3001 instead of 3000. Although I could’ve shoved more APIs into SHNserver.js, I chose to split it so that SHNserver.js doesn’t become too ridiculously long over time.

Project Initiation Home

**Project\_Initiation\_home.html** is the homepage of the initiation site. It contains all the links needed to access every site.

Website Styles

**PromoStyle.css, newSearch.css, and searchingStyle.css** are the styles used for the websites. PromoStyle.css is used for every website to give them SHN color scheme. newSearch and searchingStyle were made for the websites designed to return database results.

Initiation Forms

Each form has a CSS file and their corresponding JavaScript file linked to its page. Much of the user interaction is controlled by JavaScript by inserting data and making API calls. For as long as the user stays active and doesn’t refresh their page, user inputs are stored in a series of global variables stored at the top of each JavaScript file.

Sample from ProjectVal.js:

let projTitle = ''; // required

let projMgr = 0; // required

let qaqc = 0; // required

let teamMem = new Array(); // required

let startDate; // required

let endDate = ''; // required

let projLoc = ''; // required

let latitude = ''; // required

let longitude = ''; // required

let Projkeywords = new Array(); // required

let keywordString = '';

let otherKeys = new Array();

let shnOffice = -1; // required

...

The JavaScript files can be described as a series of function calls meant to save user data, validate user input, insert new page content into a container, fill data with previous user data, etc..

* starter(res) runs the necessary code to load the page. As of the second build of the PPI, this function began to be used by all forms to maintain the use of Microsoft’s login across multiple pages. The call to this function can be found in the handleResponse(response) function located in the file **authPopup.js**.
* getPage(currPage) gets page data, fills previous user input (if any), and makes an API call for any needed data for that page. currPage is the parameter for a page number.
* fillPage(newPage) fills the page with any previously inserted data. newPage is the parameter for a page number.
* reqField(currPage) validates user input before going into the next page. This will call the getPage function if inputs pass the validation. If not, the user will be prompted with an alert from their browser to fix fields.
* saveChoices(currPage) Saves user inputs for the specified currPage (except for page 1, which is done by reqField).
* page1(), page2(), page3(), page4(), page5(), page6(), etc., or nextPage(num) return the HTML skeleton for the need individual pages.
* searchKeywords(wiio), saveCheck(), and fillCheck() interact with each other to search the keywords on the initiation page. The idea is that each time the user inputs and deletes from the search field, the checked boxes get saved before refreshing the list of keywords to get matching results. Previously checked boxes will stay filled as the user interacts with the search functionality.
* getUsers(num) makes fetch statements to grab a selection of values needed by the user. This includes getting current employees, keywords, and profile codes from the database.
  + https://<host\_address>/ gets the employees.
  + https://<host\_address>/1 gets the keywords.
  + https://<host\_address>/2 gets the profile codes.
* Other field-specific functions:
  + expandWhy() Inserts field upon a “Yes” selection for a service agreement.
  + customAmount() Inserts field upon a “Enter Amount:” selection for the retainer.
  + limit(element) Limits characters allowed in a number field.
  + getCheckbox(group, id, value, label) Inserts HTML syntax needed for the checkbox elements to format properly.
  + getNumberField(label, newID, value, step, min, max, required) Inserts HTML syntax needed for the number fields to format properly.
  + getTextField(label, newID, value, required) Inserts HTML syntax needed for the text fields to format properly.
* preparePost() or submitBilling() runs when the user finally clicks the “Submit” button on the review page. It formats data conforming to the project database’s requirements, and it formats the data so it can be sent as a JSON object via HTTP request. This function uses the following formatting functions:
  + format(myString) inserts the escape (\) to any backslashes, double quotes, and single quotes.
  + formatMultiline(myString) inserts the escape (\) before single quotes.
  + teamString(memberArray) creates a string from an array by inserting a separator “ || ” between each element.
* Information gets sent to the host. On success, it should return a JSON object with an attribute called status. Ex: {“Status”:”023016”}. The forms send to the following API addresses:
  + ProjectForm → https://<host\_address>/result
  + PromoForm → https://<host\_address>/promo
  + ProToProj → https://<host\_address>/ProjPromo
  + AddBilling → https://<host\_address>/submitBill (returns “Success” instead of a project number).

Other from functionalities:

* Billing Group:
  + Before initializing a billing group, the user is prompted to search by project number. The search is sent to https://<host\_address>/billMe to retrieve the results. On successful submission, the function resultString(jsonRes) will process the returned data. This function takes the JSON object as a parameter. It will create the HTML syntax to display the results, and it will add the results to a JavaScript Map object, so that the information can be easily retrieved upon user selection.
  + When user selects the project to add a billing group to, the function billForm(proj) will take what had been stored in the Map objects to parse and fill the first page with the needed information. It will also grab manager info at https://<host\_address>/mgrs.
* Promo to Project
  + The user needs to search the database for a promo number without a corresponding Project ID at https://<host\_address>/searchPromos.
  + On selection, startForm(myPromo) will take the sent JSON and fill the criteria to the global variables. The function getKeysByName() will also run by sending an API request to get the key IDs for form selection usage.
  + The data will be sent to https://<host\_address>/ProjPromo upon submission to update the promo entry with project information.

Search Functionalities

There are two different pages for the search website. Both of which use the searchingStyle.css file, because PromoStyle.css sometimes made the search results too unreadable. general\_search.html uses searchAll.js as its JavaScript. The request searches by the following: Project ID, Promo ID, First Name, Last Name, Project Location, Keywords, Client Company, Profile Code, and Description of service.

Clicking “Search” invokes the function search() by sending to https://<host\_address>/search. Results are stored in Map objects. This happens so that billing groups can be grouped together under the same project listing, since the database was designed to duplicate a project row each time a billing group is added.

Update: redacting general\_search.html to replace it with search.html. This new search page is designed to fit the formatting needs of build 2.

The page also includes a link to the Advanced search page, which is the second search page called search\_demo.html. This page gives a series of fields the user may search by. Besides there being specific fields to search by, the structure is like the simpler search page and will search for corresponding projects by sending to https://<host\_address>/info.

Closing Projects:

This is run using close.html along with close.js in the same directory. This page allows searching for similar project IDs. Clicking search invokes findProjects() to search for projects with similar IDs and are currently open. The results are formatted using function resultString(json), which takes the json response as an argument and returns a stringified HTML table. That table is set to the inside of the paragraph element with ID “results”. Apart from the HTML contains a “Close” button. This will invoke function closeProject(ID), where ID is pre-set within the results with the corresponding Project ID. Running this function will prompt an alert on your browser with “Are you sure you want to close this project? Project ID: [Project ID].” Only when the user clicks yes will it send info to our SHNserver2.js script to close the project in the database and move the corresponding folders.

Editing:

The page editMe.html is only meant to be accessed from the search.html page, because it requires that session storage contains the json of the selected project/promo information. Only search.html will save this info into the user’s session when the user clicks “Edit.” Otherwise, an error will be thrown, and nothing will load.

I programmed the file editMe.js to allow for editing either Projects or Promos. Depending on which one it is, the corresponding layout to the Initiation forms will be loaded. When the user switches pages, the previous entries will load into the inputs. The exception to this is dependent on the layout of the inputs from within the PPI database.

Currently, the server and the client side JavaScript’s will format team members to be entered as a string of Employee IDs separated by commas (i.e., 23,69,42,), and Keywords are inserted as a string of the literal keywords names separated by “ || ” (i.e., “CalTrans || Volcanic || myCustomKeyword“). Older entries don’t always conform to the same format, and therefore may not prefill those kinds of entries. Another thing that can happen is when dropdown selections appear as blank, which happens because an employee is no longer active, a database value didn’t match any of the defined options, or the datatype of the database value isn’t compatible with the user input.

Once the user finishes their edits, preparePost() will run to edit, format, and send to the “/updater” API in SHNserver2.js to update the corresponding Project/Promo entries. If successful, the editMe.html page will load thanks.html to thank the user and give them the option to edit more entries or go back to the home page.

Server Scripts:

**SHNserver.js** and **SHNserver2.js** are our two main server scripts, and they both conform to the same kind of format.

**SHNserver.js** hosts most of the custom-built APIs run using the library called express.

Shortened version of program:

const app = express();

const https = require(‘https’);

const cors = require('cors');

const bodyParser = require('body-parser');

var jsonParser = bodyParser.json();

app.use(cors());

// Certificates

const options = {

key: fs.readFileSync(\_\_dirname + '\\key.pem'),

cert: fs.readFileSync(\_\_dirname + '\\cert.pem')

}

… // Other libraries

app.get('/', (req, res) => {

… // Return employee information.

});

app.get('/1', (req, res) => {

… // Return keywords.

});

app.get('/2', (req, res) => {

… // Return profile codes.

});

app.post('/result', jsonParser, (req, res) => {

… // Creates project in database and folder system.

});

app.post('/promo', jsonParser, (req, res) => {

… // Creates Promo in database and folder system.

});

app.post('/keyName', jsonParser, (req, res) => {

… // Returns IDs by keyword name.

});

app.post('/ProjPromo', jsonParser, (req, res) => {

… // Creates project from promo.

});

app.post('/info', jsonParser, (req, res) => {

… // Searches database and returns results (Advanced).

});

app.post('/search', jsonParser, (req, res) => {

… // Searches database and returns results.

});

app.post('/billMe', jsonParser, (req, res) => {

… // Searches projects for creating billing group.

});

app.post('/mgrs’, jsonParser, (req, res) => {

… // Returns employees apart of a project.

});

app.post('/submitBill', jsonParser, (req, res) => {

… // Creates billing group for project.

});

app.post('/searchPromos', jsonParser, (req, res) => {

… // Search for promos for Promo to Project.

});

app.post('/rolodex', jsonParser, (req, res) => {

… // Search client contact information.

});

app.post('/contacts, jsonParser, (req, res) => {

… // Update client contact information.

});

// OTHER HELPER FUNCTIONS (See “Helper Functions”).

…

// Specifies and opens port number to use.

https.createServer(options, app, function (req, res) {

res.statusCode = 200;

}).listen(3000);

Every app.get and app.post function above is an API waiting for a request, listening through port 3000. Whichever server hosts the script, the URL to access each of these APIs are the server’s hostname at port 3000, along with the appended pathing for the corresponding API. For example, for a server with a hostname of “e-hv-ppi.shn-engr.com” that runs this script, keywords can be accessed at https:// e-hv-ppi.shn-engr.com:3000/1. An IP address can be used in place of the hostname instead, but the hostname is often more preferred.

jsonParser is what helps the API read any JSON objects the client has sent to it, especially in a post request. Finally, (req, res) means request and response. req is used to read data sent via http request from the user, and res is used to help send back a response to the user.

Within each of these functions are the operations required to process and/or return the data needed for the user. The APIs that initiate new Promos, Projects, and Billing Groups rely on the existing project folders to create the new project and promo IDs. It does so by reading the names of the existing projects or promos, and it finds the latest existing initiation number to generate the next new number.

After an API figures out what the next ID will be, it does the following in order:

1. Finds what the next ID will be based on directory folder names.
2. Creates and/or updates a database entry.
3. Creates the directory for the project, promo, or billing group.
4. Creates and fills PDF with the input information, and places it in its corresponding folder.
5. Creates other needed directories within the initiated folder, and changes windows permissions for the PUBS folder.
6. Emails required personnel with the PDF.
7. Returns request back to the user form.

If an error occurs within the program’s steps, the steps executed previously remain existent and any following next steps don’t get executed, and the user may need to inform us if they receive an error message. If not an error message, then it’s possible that their data has been entered into the database, but an error occurred during either folder creations or email.

Helper Functions:

These are functions whose operations are generally used across multiple initiations, and the APIs will call them during their execution. The following functions are currently being used by the APIs:

* removeEscapeQuote(SQLFormat) – For SQL formatting purposes on MS Access, any user input text with a single quote (‘) within it needed to have another leading quote immediately after it. This is MS Access’s way of escaping text containing single quotes. Otherwise, MS Access would throw an error.
* createDirectories(root) – Creates the directories needed within a newly created project, promo, and billing group. It also changes the permissions for the PUBS folder to the following:
  + Administrator and Admin Staff: GA - General All
  + Domain Users:
    - GR - General Read
    - X - Execute/traverse
  + Marketing (Modify permissions)
    - RC - Read control (read permissions)
    - GR – General Read
    - GE – General Execute
    - GW – General Write
    - RD - Read data/list directory.
    - WD - Write data/add file
    - AD - Append data/add subdirectory
    - REA - Read extended attributes
    - X - Execute/traverse
    - RA - Read attributes
    - WA - Write attributes
* emailPersonel(pdfName, pathway, message, mailList, subjectLine) – Given the required parameters, this function will email everyone on the mailing list with the PDF with the initiation’s specified subject line and message.
* getAdmin(office, isArcata) – Depending which office the user specifies for their initiation, corresponding personnel will need to be informed via email too. This function returns an array of emails based on office location.
* getDir(id) – Gets the path of which office the initiation folders need to be placed inside of.
* removeSpace(myString) – For every space that exists in each string, it gets replaced with a dash (-). This is often used for adding the titles to the initiation folders, and it’ll help keep the folders accessible if a system error occurs.

Database Structure:

The database is structured the same way as the old intranet’s database. In my opinion, this was one of the most frustrating aspects of building the new Initiation system. In normal database practice, there needs to be separate tables within the same DBMS for whenever there are fields that involve an arbitrary number of entries. Instead, everything was shoved into one database table called “Projects.” The issue to point out here is that this Projects table could’ve been easily structured far better than this original one. Plus, there was no previous documentation found on how and why the database was structured the way it is now.

When making use of this database, the goal was to build it in a way so that SHN employees can still make use of older data that had been previously entered. To achieve this goal, I needed to know how it was previously structured, and to work with it as best as I can. Below are implementation rules we’d noticed and are currently implemented:

* Projects and Promos are inserted both as individual entries. For entries that have both a Project and Promo number, Project number take highest precedence amongst the Promo number. That’s why when someone converts a Promo to a Project, the entries Projectid field gets updated rather than inserting a new entry.
* When a billing group is added to a project, a duplicate project entry is created with the updated Billing group number and Billing group title.
  + Some older Billing group entries don’t include previous project data. So naturally, we assume that the first occurring entry of the selected project has all the project’s information.
  + A single project can have as many billing groups as it needs, but that project cannot have more than one billing group containing the same group number.
* The database is controlled by Microsoft Access. In Access, most of the fields are of data type “Long text” (String) except for Latitude, longitude, and prevailing wage. Latitude and Longitude are type Number, and Prevailing wage is of type Short Text.
  + Previously, the initiation forms allowed latitudes and longitudes to be in both decimal and deg/min/sec format. If someone submitted info in ged/min/sec format, it won’t get inserted into the database, but it does get printed to the PDF. This functionality eventually changed during the testing of build 2, when everyone was suddenly telling me that it needs to be only decimals.
  + Short text is like long text, except the number of characters are far more limited.
  + In previous times, I’ve tried changing the data types of specific column fields, but it won’t allow me due to space constraints. For example, ProjectMgr is type Long Text, but that text is often the ID of the staff member from the “Contacts” database table. To make it easier to perform inner joins in SQL statements, I wanted to change ProjectMgr’s data type to type Number. Unfortunately, I couldn't make this happen.