Connecting people to employment opportunities

MSR Database
Technical User Manual

Technical User Manual

This document should provide you with everything you need to know about MSR's data management system. In addition to the information here, much of the code (available at github.com/mattmcd25/msrsite) is commented.

If this document fails to provide the information you need, don't hesitate to reach out (mjmcdonald@wpi.edu and rwwittenberg@wpi.edu). We will do our best to reply and answer any questions, but once we leave the country of Namibia, we don't plan on doing any more actual coding on this project.

1. Software

The database software is divided into two main parts - the client and the server. The server, written with Node.js and Express, responds to requests from the client, which is written using React.

Running `npm start` will begin three processes simultaneously:

- `npm run start-client`, which starts the development server and the React application on port 3000;
- `npm run start-server`, which starts the Express app on port 3005 with a proxy during development; and
- `npm run watch-css`, which rebuilds the .scss files into .css as needed.

Prior to deployment to production, `npm run build` must be run, which compiles the client and server into optimized files and builds the .scss into .css. Then, in production node environments, only the server is actually run, as it is configured to host the static assets of the client. A `heroku-postbuild` script is setup so that deploying to heroku automatically does this, but `npm run build` must be manually run when deploying to the server.

Server

The server itself is relatively simple, and composed of only five files. The server is written using Node.js and Express, along with the assistance of a few other libraries described below. Server.js is the main file which creates and launches the Express app, and the other four files (found in /src/api) represent the different actions that API calls can perform.

server.js

The server creates the Express app and configures it to launch on either the PORT environment variable or (by default) 3000. It then configures all of the API calls to make use of the actions defined in the other four server files, found in /src/api.

Upon being run, the server attempts to connect to the SQL database by calling the `connect()` general action. Upon success, the Express app begins listening on the preset port.

The API calls in this application make use of a few middlewares, some of which are from other libraries and some of which are our own.

- `bodyparser` is used to parse the request bodies.
- `@risingstack/protect` is used for basic SQL injection protection via regular expressions
- `checkTableID()` and `restrictTableID()` are used to ensure that only valid tables are requested from, to prevent SQL injections. Every tableID from every API call is checked against the list of allowed tables, which is obtained by querying the information scheme at launch. `restrictTableID` differs in that if the table being requested is found in the `TABLE_REBINDINGS` dictionary, it will instead return the results of querying the rebound table. This is primarily used to ensure that users with lower permission levels cannot access all of the data in the Member table.
- `authCheck()` and `validate()` are used to ensure that the user making API calls is signed in via Auth0, and that they have the appropriate permission level to do so.

In addition to the four main files, described below, there is a fifth section of actions, titled 'Restricted Actions'. These are used in the limited version of the application, and the only difference is that they make use of the restrictTableID middleware rather than checkTableID. These are the only API calls that do not require 'admin' user level, and therefore are the only ones a limited user has access to.

generalActions.js

The general actions consist of connecting and disconnecting from the database. These API calls are never actually used in the client side application, but are still available at `/api/connect` and `/api/disconnect` if needed. They are only called by the server, on initialization and on close.

queryActions.js

The query actions consist of everything relating to querying the database for information. The methods are:

- `selectAll`, available at `/api/select*/:table`, returns all in the table or view. If you request from the Member table, it overrides to return information from Member_Full (which is the same information with the columns in a different order) sorted descending by MSR member ID. This is used to populate the member table, load constants like Skills on launch, and as part of the advanced search process.

- `advancedQuery`, available at `/api/query/:table`, returns all information in the table or view that matches the condition specified by the body. For example, requesting query from Member with `{ MARITAL: 'Single' }` in the body would return all Single Members. This is primarily used in the advanced search process, but is also used to gather information for a specific member for the view/edit pages. When generating conditions, the equality operator is always used except in the following cases:
    - If the variable being compared is `LENGTH`, `YEAR`, or its name begins with `MIN`, then a `>=` operator is used instead.
    - If the variable being compared has a name beginning with `MAX`, then the `<=` operator is used instead.
- `getColumns`, available at `/api/colnames/:table`, returns the names of the columns of a table or view, as well as their data type (varchar, int, bit, etc) and the maximum length (if type is varchar). This is called on launch to get information about most of the tables, and this information is then primarily used for dynamically making forms (like the add member page) or data validation on edit screens.
- `getTables`, available at `/api/tabnames`, returns the names of all tables and views in the database. This is only called once, by the server at launch, and is used to verify that no API call has an invalid table name (to prevent SQL injections).
- `getFKs`, available at `/api/fks/:table`, returns a list of all of the tables that reference the specified table with a foreign key. This is only used when editing constants like Skills, and allows the 'force delete' option to be possible (IE, remove a skill from all places it is referenced and then delete it).

updateActions.js

The update actions consist of everything related to updating or changing the information in the database. The methods are:

- `insert`, available at `/api/insert/:table`, inserts the information specified in the body into the table. For example, calling `insert/skill` with `{ NAME:'Test', DESC:'None' }` would insert a new Skill with name Test and description None.
- `update`, available at `/api/update/:table`, updates the table according to the body. For this api call, an extra argument (of type object/dictionary) must be added to the body called `PK`, which the content of is then used to generate the condition for the update statement. For example, to set the name of member with ID 1 to Matt, you would call `update/member` with `{ FIRSTNAME:'Matt', PK:{ ID: 1 } }` in the body.
- `Delete`, available at `/api/delete/:table`, deletes all entries from the table that match the condition generated from the request body. For example, calling `delete/site` with `{ ABBR:'R' }` would delete the site with code R from the table.

When generating conditions, equality is used for everything. This means that a request body like `{ LENGTH: 0.5 }` will only match entries with a length of *exactly* 0.5, unlike the query behavior. This is done to ensure the correct entry is modified when updating or deleting.

serverAuth.js

Server Auth contains the methods required for the server to authenticate the client. The main two methods export.getUsers, exports.getLevel, exports.getPermissions, exports.updateUser, Run a variety of calls that interact with the auth0 management api, and client api. In general, the serverAuth file caches all users and their permissions since the last run of the server. When a user authenticates for the first time in a while, serverAuth first needs the api token to access the client api. ServerAuth uses the getSysToken method in the file to get this token using the auth0 management api with the management id and secret. In order to get the users permissions, the server will parse the user id of the individual trying to access the website from the access token saved on their browser during login. The server then calls the auth0 client api with the api token and user id to access all of the users information, but most importantly, the app metadata where the users permissions are stored. All static auth0 keys are stored in the file authConstants.

Client

The client makes up the majority of the entire application. It is designed primarily using React, making use of a variety of other libraries such as React Router v4 for routing and React Material Design for visual elements. In the folder structure, it is broken down into four main sections:

- Components, found in `/src/components`, are the generic elements making up most of the pages.
- Pages, found in `/src/components/pages`, are the actual pages displayed by the React Router.
- Displays, found in `/src/components/displays`, are the components used to display data on the view, edit, and query pages.
- Other core/miscellaneous files, found in `/src` or `/src/data`, are the actual Routers, the core application (index.js), and miscellaneous helpers used throughout the code.

Core Application

Index.js

This is the starting point of the application, as well as the central point for constants or global data for the application. On start, it simply renders the `LockedApp` component into the DOM. Also stored in this file is a variety of information:

- `HEADERS`, `CONSTANTS`, and `FKS` store information from the database that is loaded in once by initialize. These are effectively constants, and never change again.

- `WORKSTATUS`, `WORKTYPE`, `STATUS`, `MARITAL`, and `GENDER` store all of the options for these five fields (as they are restricted to only specific options).
- `TODAY` is used to ensure that code referencing the current date does not have issues based on the time changing.
- `searchResult` is used to store the result of the most recent advanced search, so it is still displayed if a user clicks on a member on the result page and hits back.
- `auth_level` is the authorization level of the user, used to filter information based on the permission level.

The `initialize` method is used to load all of the constant information from the database at launch. The other four methods simply interact with the other variables.

LockedApp.js

The locked app is an extremely simple router. By default, it goes to the login page, but also contains a route for the callback page used by Auth0.

App.js

App is the core router for the application. Its render method returns a router that switches on the different paths and wraps the resulting page in the `Layout` component. Rather than using the component routing type, we use the render function type. This allows us to use our helper method `componentWithRefs` to add extra properties to all components. These extra properties are references to the methods of the App:

- `setTitle` sets the title on the top of the navigation drawer
- `setActions` sets the buttons on the top left of the navigation drawer
- `toast` is used to create a small snackbar popup on the bottom of the application
- `popup` and `dismissPopup` are used to control the popup windows

The App also contains a `dismissToast` method, but this is not included in all components as toasts are configured to automatically dismiss themselves after a certain amount of time.

LimitedApp.js

The limited app is the same as the App, except all of the administrative pages (manage, add, edit) have had their paths removed. This could definitely be refactored to use the `isAdmin` method and be part of App, but we ran out of time.

Pages

AdminPage.js

The component making up the Administrator Settings page, found at `/manage`. The actual code for this page is extremely simple; the list of options are dynamically generated by the `settingCards` function at the top of the file, which is effectively a constant that makes use of props. Each of the dictionaries returned by the `settingCards` function is then turned into an `ExpandingCard` containing the children specified.

In the case of the four constants (Skills, Languages, Sites, and Certificates), the card contains a `ConstantTableElement`. In the case of the admin account settings, the `AccountManagerElement` is used. More information on these can be found in the Displays section.

EditMemberPage.js

The edit member page is, by far, the grossest file in this entire application. I apologize greatly for it, and if we had more time, a great deal would have gone into refactoring it. With that being said, I will attempt to explain is as much as possible to make your life easier. Starting at the top:

- `defaultFor` method is used to return the default value for when a new work, placement, certificate, or training is created. This information should probably be dynamically generated according to the database columns, and is definitely a potential place for improvement.
- `updateActions` method resets the actions in the header bar of the application according to the current state of the application. The only thing that ever changes (and the only reason this function exists) is to change the save button to be disabled when the user enters something invalid, and give it an updated list of issues for its tooltip.
- `componentDidUpdate` gets called whenever the state of the component just changed. In this method, we go through and check each important piece of the state to see if it is invalid. First, it uses the dataLengthIssues method to ensure none of the values are too long to be inserted. Then, it ensures that the member site is not empty, as this would cause a foreign key error. Lastly, it makes sure every certificate has a type, and there are no duplicates. If any of the ‘issues’ returned by these checks are different than last time the component updated, it updates the save button accordingly.
- `componentDidMount` is used to do all of the initial loading of the member’s current information. It begins by trying to get the member information by ID. If the specified member doesn’t exist, it returns home; otherwise, it loads the rest of the member’s information, sets the title and action buttons, and sets loading to false.
- `past` and `getAndSave` are simple helper methods used to help save two copies of all information obtained from the database (ex: state.mem and state.pastMem).

- `saveChanges` and `saveJobs` are the bulk of this class. The code itself should be (relatively?) self explanatory, but the basics is that it uses the `difference` and `intersection` helper methods. Find the difference of past - current returns anything that was removed by the user; the difference of current - past returns anything that was added. The intersection is anything that was not removed or added (ie, existed before and stuck around), and these are all checked for changes to the other fields within them. There is a decent amount of repeated code in this method, and I'm pretty embarrassed by it to be completely honest, but we were pressed for time and it does the job.
- `updateMember`, `updateItem`, `updateCert`, and `setLangs` are the `onChange` methods passed into the `EditMemberDisplay`. They update the corresponding piece of the state accordingly.
- `setSkills` and `setItemSkills` are used to change the lists of skills associated with the member or a job/training.
- `addLang`, `addItem`, and `addCert` are the `onClick` methods for the add buttons in the display.
- Similarly, `removeLang`, `removeItem`, `removeCert`, and `removeClicked` are the `onClick` methods for the remove buttons. `removeClicked` then calls `removeMember` (which, in turn, calls `removeJobs`) to do the actual deleting of the entire member if the user confirms the action.
- `render` just returns the `EditMemberDisplay` with all of the data and handlers if loading is false, otherwise a react-md `CircularProgress` tracker.

IndexPage.js

The main page of the application. This is one of the simplest pages, simply loading all of the member information from the database and displaying it in a `MemberTable`. Unlike some of the other `MemberTable`s in the application, this one provides an `onRefreshClick` method, which then in turn causes the refresh button to be included.

Layout.js

This displays the overall layout of the application, that is present in all pages. It consists of a react-md `NavigationDrawer`, displaying the nav items as specified in the `Navigation` component. In addition to whatever children are passed into the layout by the router, the layout also includes a react-md `DialogContainer`, used to show popups to the user, and a react-md `Snackbar`, used to show small informational messages (called 'toasts') along the bottom of the screen.

LoginPage.js

The login page is the true start point of the application, and the first page rendered by ReactDOM. In its `componentWillMount` method, it loads the web fonts used by the app. In its `render` method, it renders a basic page with a login button, unless the user is already logged in, in which case it redirects to the `LaunchScreen` component.

MemberPage.js

Similarly to the `EditMemberPage`, the view member page begins by loading all of the relevant information of a member (unless the ID is invalid). It then passes all of this information into a `MemberDisplay` component.

NewMemberPage.js

The new member page simply displays an editable `PropListCard` generated from the Member table headers with all of its fields empty by default. When the add button is clicked, it simply inserts into the Member table according to the form data. This includes simple data validation for the length of the inputs and the fact that Site cannot be empty.

NotFoundPage.js

This page is currently unused, but was previously a catch-all for invalid URLs.

QueryPage.js

The query page is also relatively dense, although it is much better designed than the edit page. Starting from the top of the file:

- `componentDidMount` tries to load the saved search from the client. If there is one, it switches to display mode and displays the previous search. Otherwise, it switches to query mode.
- `clear` clears all the search fields in all cards back to the initial state. For the most part, this just consists of clearing the state; however, due to the fact that the autocompletes are not controlled, they must be cleared using refs.
- `setQueryMode` switches to query mode by clearing the fields, changing the title and action buttons, and clearing the stored search.
- `setDisplayMode` switches to display mode by clearing the fields, changing the title and action buttons, and saving the new search result. In addition, it generates a title based on the condition that was used and sets that to be the title of the member table.
- `search` does the actual searching of the database. It makes use of the `propSearch` and `jobSearch` helper methods to create a promise that queries the corresponding table for each of the different search criteria. It then takes the results of these promises and continually finds the intersection of the current `matchingIDs` (which starts as all IDs) and the result of the search query, until the final list contains only the IDs of members who match *all* criteria. These IDs are then used to filter the list of all members, and the resulting list of matches is used to set display mode.

- `update`, `setLang`, `addLang`, and `removeLang` are the change listeners for the `QueryDisplay`.
- `render` returns the `QueryDisplay` when in query mode, and a `MemberTable` when in display mode.

TrainingPage.js

The training page is used to add a training session to multiple members at once. It is extremely similar to the new member page, the main difference being the addition of two `ChipListElement`s for the lists of members that passed and failed the training. In the `handleSubmit` method, we iterate through the list of success members and the list of fail members, and add the training to each of them with `SUCCEEDED` set to true or false accordingly.

UnauthorizedPage.js

The unauthorized page is displayed to new users who have just created an account and have no permissions yet. It displays a simple message, inviting users to email MSR asking for permissions. This is only shown to Auth0 users with the 'newUser' or an unknown user level.

Displays

AccountManagerElement.js

The component used to manage the account permission levels. Visually, it consists of a save button and an `exclusive CheckTableElement`. The `levelFrom` and `updateRadios` functions are simple helpers that convert the state of the radio buttons into a text user level, and ensure that only one button is clicked at a time, respectively. When saving, it compares the past state of each user with their new state, and uses the Auth0 API calls to save it if necessary.

Cards.js

This file contains a few components that are simple wrappers for the other display elements, putting each one inside of a `BlankCard` with a title, actions, children, and footer.

CheckTableElement.js

A component representing a table, where each row has a list of true/false properties. If the `edit` property is specified, it makes it editable with checkboxes. If `edit` and `exclusive` are both specified, it uses radio buttons instead of checkboxes. If `acData` is specified, it adds an autocomplete at the bottom to add more entries to the list. If `tips` are specified, each row of the table will have a tooltip. This component is purely functional and has no state. This component was designed with the intention of it only being used for languages, so I cut a few corners and it isn't the best design.

ChipListElement.js

This component represents a list of chips, with an autocomplete to add more and buttons to remove them in `edit` mode. If `tips` are specified, each chip will have a tooltip. This is a pure component, and the added methods in the class only exist to make it easier to update the list.

ConstantTableElement.js

This component represents the table displaying and allowing the editing of the constants like skills or languages. It is paginated, and includes add and save buttons at the top, with an X next to each row to delete. Clicking on a field in the table allows it to be edited. This component is entirely self contained and separate from the admin page, and has its own state.

Because this is designed to only be used with constants, a table name and primary key are specified in props rather than data. In the save method, the current value of the data is compared to the global constant, and any changes necessary are made to the database before storing the new list as the global constant.

This element should be relatively straightforward, and mirrors the layout of the edit page (to a much lesser scale); the only complicated part comes in the deleting. When the user clicks the save button, if deleting of a row fails due to it being referenced elsewhere in the database the `catchDel` method is called. This gives the user the option to cancel or force delete, where force delete goes through every table that references this constant with a foreign key and deletes all occurrences of this value.

EditMemberDisplay.js

This component creates all of the other display element cards corresponding to the member information passed in. First is a `PropListCard` for the member data, followed by a `PropAndChipCard` for each work/placement/training, a `PropListCard` for each certificate, a `CheckTableCard` for languages, a `ChipListCard` for other skills, and lastly a `BlankCard` containing the add and delete buttons.

This is probably the single least flexible part of the application, along with the `EditMemberPage` and the other `MemberDisplay`, and could be greatly improved with the use of the global `FKS` variable and some refactoring time.

ExpandingCard.js

This is a very simple component representing an expandable react-md `Card` with title, actions, an icon, and children.

MemberDisplay.js

This component mirrors the `EditMemberDisplay`, except all the components are *not* editable, and the list of skills comes first. Once again, this could be dramatically improved and simplified with some refactoring.

PropListElement.js

This is the most commonly used component in the displays, and shows a list of properties. In normal mode, it simply displays the properties with the help of the `DisplayUtils` formatting. In edit mode, it uses a variety of react-md components to make the information easy to control:

- If `acData` is specified and there are more than 20 options, an `Autocomplete` is used.
- If `acData` is specified and there are less than 20 options, a `SelectField` (drop down menu) is used.
- If the data type in the database is a boolean, a `Checkbox` is used.
- If the data type in the database is a date, a `DatePicker` (calendar popup) is used.
- Otherwise, a simple `TextField` is displayed, with text validation generated based on the field's type and max character length in the database.

This component has no state, but does keep track of refs to the autocompletes, as these are still uncontrolled components and there was no other way to clear them when invalid data is entered/when the clear button is pressed.

QueryDisplay.js

Similarly to the `AdminPage`, the query display is a list of dynamically generated `ExpandingCard`s according to the `searchCards` function. It generates elements almost exactly as it would on the edit page, except each element begins empty instead of having a default value.

Other Components

Callback.js

The callback page used by Auth0 to store the access and ID tokens.

IssueButton.js

A button that dynamically changes itself based on the list of ‘issues’ passed in. If there are issues in the list, it returns a label (styled to look like a disabled button) instead of a button, with a tooltip according to the issue. This is only necessary because disabled buttons do not show their tooltips, but I wanted to inform the user of *why* the button was disabled instead of leaving them to figure it out for themselves. This element looks a little ugly at times, as it is styled to look like a disabled raised button even if the enabled button is flat, but this is a minor complaint.

LaunchScreen.js

The launch screen simply displays the MSR logo and a loading circle while initialization goes on in the background. If the user is not logged in (generally when their access token expires), it clears the tokens and sends them back to the login page. Otherwise, it requests their user level from the server and stores it in the client, initializing accordingly. If the user is a ‘creator’ or ‘admin’, it renders the full `App` component into the DOM, replacing the `LockedApp`. Similarly, if they are a ‘user’, it renders the `LimitedApp`, and if they are a ‘newUser’ or another unknown level, it renders the `UnauthorizedPage`.

MemberTable.js

Displays a paginated, filterable table of members. A list of members is passed in as a property, which is then filtered into `match` based on the input in the text box (handled by `updateInputValue`), and then sliced into a `display` to show one page at a time. This class does all of the heavy lifting of filtering the members down to a list, which is then actually rendered by the `MemberTableBody`.

MemberTableBody.js

If the `loaded` property is set to true, this component renders the actual data table of members, including pagination and a ‘No results’ method if the list is empty. This is a stateless pure component, and the information to display is handled by the `MemberTable` itself.

MemberTableHeader.js

This is another stateless pure component, and it simply renders the header of the table. It includes a title, a separate section to display information about the filter condition (on the query page), the filtering search bar, a download as CSV button, and a refresh button (if the `onRefreshClick` function is defined).

MemberTableRow.js

Represents one row of the MemberTable. Each row is wrapped in a route, so that clicking on a row in the table redirects you to the information page for that member.

Navigation.js

Returns a `div` containing a list of the ‘nav items’, which are generated by the `NAV_ITEMS` function and vary depending on whether `isAdmin` is true. If the nav item specifies a `to` attribute, it is wrapped in a link; otherwise, it is given an `onClick` handler function.

Tooltip.js

A simple convenience component that allows anything to be easily wrapped in a react-md tooltip on the fly, without needing to call `injectTooltip`.

Miscellaneous

databaseManager

The `databaseManager` is the main point of connection from the client to the server. It consists of three main sections:

- The internal functions are used to make other actions possible. This include things like making get/post/patch/delete requests, checking the status of an HTTP response and throwing an error if necessary, and cleaning the returned JSON into a nicer format.
- The exported functions are the building blocks of the application, and correspond almost one-to-one with the API calls listed in server.js (so look there for more information!). The main thing to note here is that if any of the `get` requests are made and the user is *not* an admin, the `databaseManager` makes use of the limited version of the API calls.
- The exported helpers make use of the original functions to make other common operations easier. Currently, the only functions here are used to get properties of a member by ID.

Utils

A collection of generic utility functions that operate on lists, dictionaries, and strings. The majority of these should be self explanatory, but the more obscure ones are:

- `makeDict` makes a dictionary from a list where each value in the list is a key with a value of an empty string. This is used to generate empty `PropListElements` from the headers of a table.

- `filterObj` removes all key/value pairs from an object where the value is either an empty string, false, undefined, or an empty object. This is used to ignore values a user didn't specify in the query page.
- `dictFromList` goes through a list of objects and converts it to a dictionary by accessing the specified key of the object and using that as the key in the dictionary. This is mostly used to convert information from the database into a friendlier format, since we already know there will be no duplicate keys.
- `uniteRoutes` is used when the primary key of a constant (like skill or language name) is changed to ensure that the right constant is the one being updated in the database. Without this, the constant table would think that renaming a skill was the same as deleting the old and adding a new one, which would obviously cause issues if the skill was referenced anywhere.

DisplayUtils

This contains a bunch of tools for formatting data to look better. It is primarily based around the giant `formats` dictionary at the top, which maps the name of a database column to a friendlier version of the name and occasionally a function to format the value. All of the non-exported functions with 'pretty' in the name are used as formatters for values, and the exported 'pretty' functions make use of this format object. The other functions in the file include:

- `textValidation` is used to generate `size` and `type` constraints for a text field based on the type of a column in the database.
- `dataLengthIssues` compares the size of a field against its contents to ensure nothing is too long and returns 'issue' objects accordingly.
- `issueTip` converts the 'issue' objects generated elsewhere into a friendly format to give to the user as a tooltip.
- `doubleDate` takes a dictionary (usually generated by `dictFromList` or `makeDict`) and, if there is a key for a date, it changes it so instead there are two: one with `MIN` and one with `MAX` before its name.
- `displayTitle` uses the other formatters to convert the state of the query page into a title for the member table.

AuthConstants

Auth constants contains the keys necessary to authenticate the client to to the server these constants are used by both serverAuth and AuthMan.

AuthMan

The AuthMan file is used by the client application to handle all necessary client side authorization actions.

- `setAccessToken` stores the users accessToken to their browser

- `setIdToken` stores the users id_token to their browser
- `getAccessToken` gets the user’s access_token from their browser
- `getIdToken` get the user’s idToken from the browser
- `clearIdToken` clears the id_token from the browser
- `clearAccessToken` clears the access_token from the browser
- `getUserId` decodes the jwt and returns the user id from the access_token
- `getParamerterByName` parses the url after call back an returns the specified token to be stored on the browser
- `isLoggedOn` checks to see if the user is logged on
- `isExpired` checks to see if the user’s tokens are expired
- `getTokenExpirationDate` gets the expiration date of the user’s tokens
- `login` calls the auth0 client side log in screen
- `logout` clears bth tokens and redirects user back to locked app.

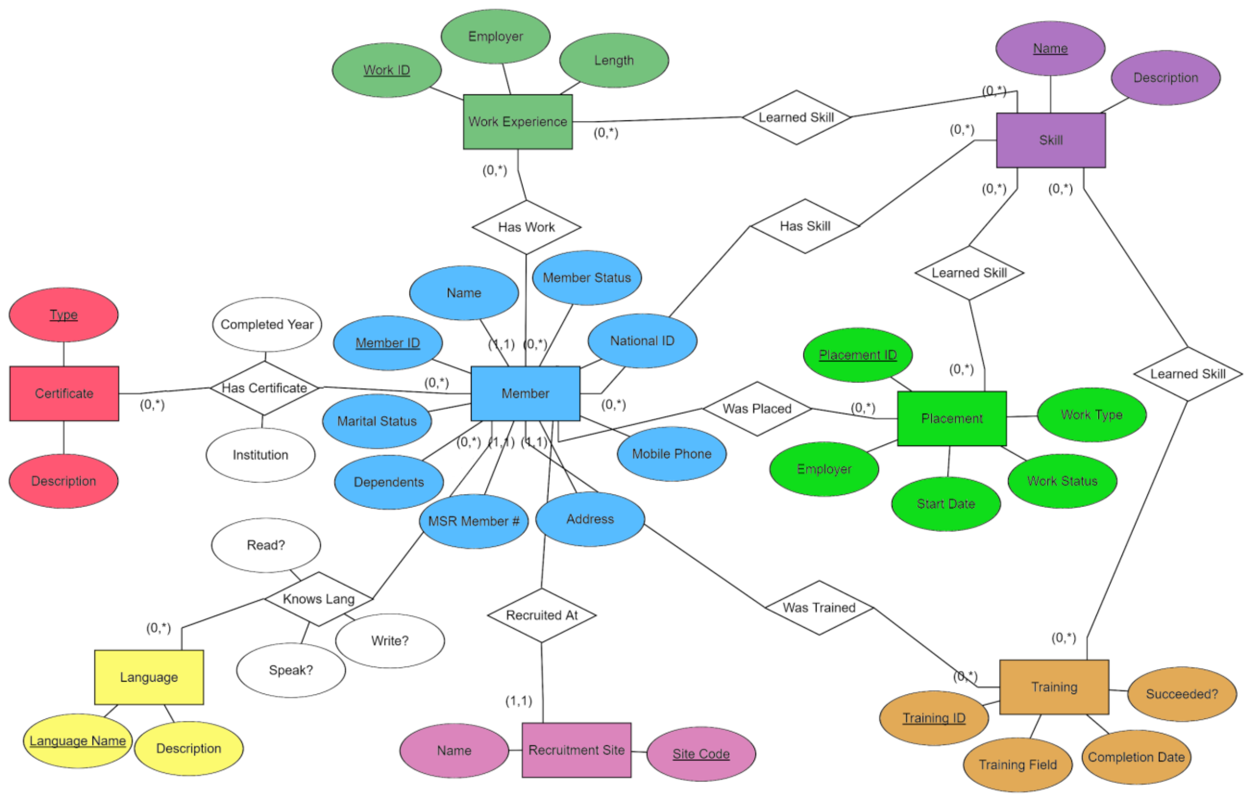
Database

The database itself should be relatively self explanatory, and is easiest to view and understand with a tool like DataGrip. The credentials are:

```
const config = {
   user: 'msrtest',
   password: 'msr2018!',
   server: 'den1.mssql4.gear.host',
   database: 'msrtest'
};
```

The Member table is the center of everything, connected to Work, Placement, Training, Has_Skill, Has_Cert, and Know_Lang. Certificate, Language, Skill, and Site represent the four constants controllable from the admin page.

A (mostly complete) ERD for the database is below:



2. Hardware

Currently, MSR’s Data management system is running on Gearhost and Heroku. Gearhost is serving the database, and herokuapp is serving both the server side and client side applications.

We do also have a server, running Windows Server 2008 Enterprise, that is available and planned to be used for this purpose. However, due to the short length of our project and the fact that we didn’t get the server until our second to last day, we weren’t able to get anything set up beyond installing our application on it.

Heroku URL: msrna.herokuapp.com

Server IP: 41.182.21.206 (external) or 192.168.178.87 (from within MSR’s network)
Username: Administrator
Password: P@ssw0rd

3. Potential Areas for Improvement

Move off Gearhost

The database is currently still hosted on Gearhost, even though we have a server of our own! Making this change would *most likely* require switching to MySQL due to licensing/costs, which would then require changing the API calls to use MySQL syntax.

Finish Server Configuration

As mentioned above, the server is not fully setup and fully implementing the hardware is definitely a spot for potential work. Running off Heroku and Gearhost is less than ideal, and while the time and storage capabilities have been sufficient for now, they may limit potential growth. Something like this would most likely include the changes mentioned above to move off Gearhost, ensuring the server has failsafes to keep the application running as much as possible, and configuring a subdomain of msr.org.na to point to the server's IP address. Windows Server 2008 is also already 10 years old (at the time of writing this), and I can't imagine it's going to get any better, so finding a new server OS would also be nice if possible.

Refactor

Specifically, the `EditMemberPage` is the worst. However, there is also plenty of room for improvement on the `EditMemberDisplay`, `MemberDisplay`, and `QueryDisplay`. Ideally, these would be modified to be dynamically generated, rather than me manually saying render skills then render work etc. This way, if a new table that referenced member ID as a foreign key was ever added, it would automatically appear on these pages!

Expandable Database

If the front end is actually updated to dynamically generate the views, then the next logical step is to add the ability to expand the database. This would require another section in the admin page, and would basically be a UI that allows them to design a table themselves. In addition, the ability to modify existing tables by adding new columns or changing data types would be beneficial. I haven't put too much other thought into this, but if a feature like this was implemented it could greatly increase the lifespan of the application, as MSR could add new fields or tables as their needs change.

Preset Trainings

MSR holds a lot of the same trainings, so it would be nice to add some preset options to the batch add training page. This could easily be hardcoded, but ideally the presets would be configurable via another section on the admin page and (probably) another table in the database.

Communication & Revision History

Add some way of keeping track of who spoke with what members when, and who edits what members when. This would involve more interaction with the Auth0 accounts, and probably another table or two in the database.

Attachments

Some way of adding attachments, like resumes, CVs, or even just member profile pictures, would be nice, but this is not something we have laid any groundwork for.

Reminders

A complex reminder system, where the employees of MSR can set up ‘rules’ (ex. When a member has been active for 2 years, when a certificate expires) that the system would then remind them for. This would also require more pages, more tables in the database, and a lot more work than it’s worth.

4. Account Information

Below is all of the account information that may be needed to maintain the application. Please be careful with where this information goes!

Email Account

Used as the sign in for all other services. May be required for verification purposes.
Username: msrnadatabase@gmail.com
Password: Khomasdal80!

Heroku Account

Used to deploy to the herokuapp.

Username: msrnadatabase@gmail.com
Password: Khomasdal80!

Auth0 Account

Used to record user profiles and manage authentication.
Username: msrnadatabase@gmail.com
Password: Khomasdal80!

Github Account

Used on server to pull from repository, or on heroku to deploy.
Username: msrnadatabase@gmail.com
Password: Khomasdal80!

Gearhost Account

Used to host the MSSQL database for free.
Username: msrnadatabase@gmail.com
Password: Khomasdal80!