**Goal:** to dissect the app, file by file, to convey how code within each file functions as well as how certain files/folders interconnect with one another

**Outcome:** A basic understanding of how this code for the app functions at the core

**Here is what the app does in general**

This app is very simple, consisting of three HTML pages in total (login, members, and sign-up). It illustrates how one might go about creating an app that has a login/sign-up feature requiring the user to input an email address and password. Whenever a new account is created, the following information is stored in a MySQL database: id, email, password, created\_at time, and updated\_at time.

A screenshot of a cell phone

Description automatically generated

As you may have noticed, the password is hashed. This is achieved via the use of passport and bcryptjs. The former serializes and deserializes the user whereas the latter hashes and unhashes the password. So, what exactly is hashing? Upon logging in, bcryptjs unhashes an already hashed password in the database to compare the unhashed password with the user input password. When creating an account, bcryptjs automatically hashes the user input password before it is stored in the database. How is this achieved? Via the use of hooking before creating a new user account which is a Sequelize method.

Hooks (also known as lifecycle events), are functions which are called before and after calls in sequelize are executed. For example, if you want to always set a value on a model before saving it, you can add a beforeUpdate hook.

**Its basic purpose is to execute user login/registration with authentication?**

Precisely. This app is essentially a skeleton that can be repurposed time and time again when building more complex or involved apps. That, and it illustrates how sequelize can be used in conjunction with mysql2. Why? To show how user information can be stored in a SQL database securely while simultaneously reducing the amount of boiler-plate code required. For example, if someone were to hack into this database they wouldn’t have access to user password information since it is stored in its hashed form. Considering security implications is of utmost importance throughout any app development process.

**Getting Started--config**

│

├── config

│   ├── middleware

│   │   └── isAuthenticated.js

│   ├── config.json

│   └── passport.js

│

**config/config.json**

A screen shot of a smart phone screen with text

Description automatically generated

The config.json file is where several instances are declared. These instances pertain to development, test, and production in this case. The development instance containing the passport\_demo database is used throughout the development process. The testing instance containing the database\_test database is used thereafter. The production instance is only used once one is certain of functionality and UI implications. The production instance is the live instance, or publicly accessible one, that users directly interact with. The dialect chosen for this program is MySQL but there are alternative dialects including but not limited to MSSQL, PostgreSQL, and SQLite. So what exactly does this file do? It creates instances for development, test, and production. These instances are used in conjunction with an ORM such as Sequelize.

**config/passport.js**

The passport.js file is located in the config folder. This file requires passport and passport-local. Passport is authentication middleware for node.js. Passport has one purpose and one purpose alone; authenticating requests. Passport achieves this via the use of plugins known as strategies. There are over 480 strategies found on <http://www.passportjs.org/packages/>. In this app, the passport-local npm is also declared as a dependency. This implies that this app utilizes a local strategy. But what is a local strategy? A local strategy is quite simple; it authenticates using username and password key-value pairs only. This local strategy is used in conjunction with express-session which keeps track of a users login status. This strategy requires a verify callback to accept user-input credentials.

Alternatively, the http-passport-bearer strategy, for example, is quite popular. It uses bearer tokens as opposed to express-session to authenticate user requests. Hence, session can be set to false when using http-passport-bearer. Moreover, npms such as chance that generate access tokens can be used in conjunction with http-passport-bearer which generates a new unique access token for each authenticated request once they are logged in. These tokens are stored in the database. This is but one way that this app could be built differently if one wished to monitor and log user requests more closely.

  This file uses a local strategy in conjunction with passport as discussed previously. It requires that an email value is provided for the user key and that a password value is provided for the password key. This file contains if/else loops that check the database to ensure that a user exists upon logging in and whether or not the password submitted corresponds to said username. Sequelize acts to serialize and deserialize the user across http requests to maintain authentication state. In summary, this file contains the logic for cross checking user input values against values contained in the database. The configure passport is exported at the end of this file.

**config/middleware/isAuthenticated.js**

 This file serves as middleware that restricts routes a user can visit if not logged in. If the user is not logged in, attempting to navigate to a restricted route redirects them to the login page.

**Models**

│

├── models

│   ├── index.js

│   └── user.js

│

**models/index.js**

The models/index.js file is generated via running the sequelize CLI. It collects all the models from the models directory and associated them if needed. The `const env = process.env.NODE\_ENV || ‘development’` line of code defines the config.json instance as the development instance. This config.json file can be converted to a .js file which would allow for the instance used to be dynamically updated. This file is essentially boiler-plate code for Sequelize. It connects sequelize models with corresponding databases (db). Ultimately, db is exported at the end of this file.

**models/user.js**

models/user.js utilizes the bcryptjs npm. Bcrypt is used for password hashing/unhashing which is achieved via salt. Salt is similar to the concept of nonce. Bcrypt uses a “beforeCreate” addHook method to automatically hash the user input password prior to account creation. This ensures that a user password is only stored in its hashed form in the database which is imperative in the context of security implications. This file also utilizes sequelize to create the user model which is simply username and password key-value pairs (local strategy). This file also contains logic to ensure that a password entered matches the password associated with the corresponding username in the database.

│

├── routes

│   ├── api-routes.js

│   └── html-routes.js

**Routes**

**routes/api-routes.js**

This file utilizes express (request, response) for post and get calls. These calls correspond to user login, user signup (registration), user logout, and acquiring user data. This file requires the use of passport as authentication middleware. If the user has valid login credentials they are able to access the members page. If the user fails to provide valid login credentials they will be sent an error. The response for acquiring user data contains the users email and corresponding id but not the password even in its hashed form.

**routes/html-routes.js**

This file requires the use of the path npm. Why? So that relative paths may be used to access HTML files (located in the public folder). The config/middleware.isAuthenticated.js file is required in this file to ensure that the user is logged in. If the user does indeed have an account they are directed to the members page. Using the isAuthenticated middleware, if a user inputs values that do not correspond with an existing account they are redirected to the registration page.

├── public

│   ├── js

│   │   ├── login.js

│   │   ├── members.js

│   │   └── signup.js

│   ├── stylesheets

│   │   └── style.css

│   ├── login.html

│   ├── members.html

│   └── signup.html

│

**Public**

**public/js/**

The three js files located in the js folder that is found within the overarching public folder contain jQuery elements and interact directly with the views aspect of the app (html files in this case). These js files contain logic that validate or reject user requests. The three file names are login, members, and sign up. This is where prevent defaults, request validation, and errors are handled. They can be thought of as “handlers”.

**public/x.html (x=login || members || signup)**

   These html files comprise the views portions of the app which, in other words, is the UI in its entirety. They are styled using bootstrap as well as via a style.css file. This style.css file is also located within the public folder (public/stylesheets/style.css).

**Server**

**server.js**

The server is where the PORT is defined, the express app is declared, and session is called. This is the main file declared in the package.json since without a server, be it local or remote, there is no live app. The server being functional is also integral throughout the development process since GET, POST, PUT, and DELETE express calls can be tested and trouble-shooted via the use of Postman (even if the user interface view aspect of the app has not yet been created). The default layout for the user interface (view) is also declared in this file in the form of requiring html-routes. The final lines of code in the server file sync the database and sequelize then launch the server via app.listen.

**Outlook**

Expanding on existing program

* Incorporate authentication tokens by using chance <https://www.npmjs.com/package/chance>
* This adds an additional layer of security as it accounts for logins and user actions by generating and storing unique access tokens each time

A screenshot of a cell phone

Description automatically generated

* The allowAccess function can be incorporated as middleware for express calls where .guid() = global unique id
* Can incorporate Regex logic into validation for email address input, for example

A screenshot of a cell phone

Description automatically generated

npm dependencies

* mysql2 -- <https://www.npmjs.com/package/mysql2>
* express -- <https://www.npmjs.com/package/express>
* express-session -- <https://www.npmjs.com/package/express-session>
* body-parser -- <https://www.npmjs.com/package/body-parser>
* nodemon -- <https://www.npmjs.com/package/nodemon>
* sequelize -- <https://www.npmjs.com/package/sequelize>
* passport -- <https://www/npmjs.com/package/passport>
* passport-local -- <https://www/npmjs.com/package/passport-local>
* bcryptjs -- <https://www.npmjs.com/package/bcryptjs>

Directory Structure

│

├── config

│   ├── middleware

│   │   └── isAuthenticated.js

│   ├── config.json

│   └── passport.js

│

├── models

│   ├── index.js

│   └── user.js

│

├── node\_modules

│

├── public

│   ├── js

│   │   ├── login.js

│   │   ├── members.js

│   │   └── signup.js

│   ├── stylesheets

│   │   └── style.css

│   ├── login.html

│   ├── members.html

│   └── signup.html

│

├── routes

│   ├── api-routes.js

│   └── html-routes.js

│

├── .gitignore

│

├── package-lock.json

│

├── package.json

│

├── README.md

│

└── server.js

That is my best attempt to try and explain how all the code ties together.