Northern Michigan University, Winter 2023

RetroFlix

NetFlix-Clone Website

Developed By Brenan Marenger

Introduction

I have always been intrigued by computers and they have played a significant role in my life. I feel grateful that I can turn a passionate hobby into a fulfilling career. Throughout my computer science learning experiences, I’ve taken interest in exploring the variety of areas within the field, how they're interconnected, and where I’d like to take my career. As I near graduation and start to think of career paths, I’ve found the most interest in the development of mobile and web applications. While applying to various development positions, I’ve noticed many crucial skills and tools that I lack proficiency in. The main objective of my project was to learn these industry standards and development techniques, as well as enhance my ability to quickly learn new systems. I also wanted to provide this project as a showcase of my capabilities to employers.

The project originated when I was working on a personal gift for my family of converting childhood VHS tapes to digital download. Having these assets at my disposal and knowing I wanted to develop an app, I decided to build a video streaming service. I chose Netflix to replicate due to popularity and the wide variety of complex features. Netflix is the most well-known streaming platform, and has an immense impact on the entertainment industry. They provide one of the most well-developed sites known for its accessibility, reliability, and clean designs. RetroFlix is my first full-stack web application where I recreate Netflix’s iconic features and designs using my own assets.

Taking on an application of this scale was very intimidating. Full-stack development is notoriously complex due to the variety of systems that simultaneously contribute to different areas of the project. Many companies tend to hire hundreds of developers so they can dedicate teams to various sections of the site. Although I had limited exposure to back-end development, I was eager to learn and dedicate time to make a successful project that would pass in the modern industry. Through plenty of research, communication, and critical thinking, I was able to enjoy the journey of building this project starting from scratch with only VHS tapes. This essay primarily discusses concepts with corresponding examples rather than each specific implementation. A better preview of the full code and visual designs can be found nicely documented on my GitHub provided in the resources section.

What does Retroflix offer?

Retroflix is a reactive video streaming platform, focused on providing a similar experience to Netflix. Smooth performance, proper security, plentiful features, and an appealing visual design are all priorities of the site. Initially, the user is prompted to log in or create a new account through an authentication system. After entering a successful email and password, the user will wait through a loading screen for the 'gallery' page to appear. At the top of this gallery page lies a featured film. This is a randomly selected video that plays a preview, briefly shows additional information, and contains many familiar controls.

Above the featured film lies the navigation bar where the user will be able to use a drop-down filtering menu, an animated search bar, and the ability to sign out. I've built the searching and filtering systems to work simultaneously with each other, allowing for any number of combinations. After using either of these systems, another 'Clear Filters' button will appear, resetting any filters previously applied. To help organize and track information, the user may use the query URLs that adjust to the correlating filters and searches.

Scrolling past the feature will ease you into the 'Top 5 Rating' system. This portion displays the top five most favorite videos across all accounts, displaying a large number to the corresponding rating position. If any thumbnail in the gallery is hovered, a small menu will appear displaying controls to play the video, show additional information, and the ability to favorite it.

The modal is one of my personal favorite features of the project due to its professional design and ease of use. On prompting, the background will dim, the query URL properly adjusts, and the modal will appear from a simple animation. Here the user will see a preview of the video and similar options to play and add it to favorites. Below this preview is all information pertaining to that video, such as its duration, genres, and a selection of videos relating to it.

'Your Favorites' and 'Watch Again' sections are unique to each user account. These segments work as expected, displaying the list of the users liked videos and a history of videos previously watched. In the 'History' section, the remaining duration of the video is indicated through a red bar, saving the exact spot that video was stopped and resuming when played again. Under this section, a variety of different categories are shown similarly, only including videos within each genre. Since all these sections vary in size, I developed my own adaptive carousel to hold them.

The last section of the gallery is the 'All Videos' portion where all videos the service has are displayed. I'm able to efficiently utilize this portion for all the searching and filtering systems. When the user selects 'Play' on a desired video, they will be brought to the 'Display' page. This screen will play the video providing a variety of familiar controls, which also have their own corresponding keybinds. My favorite is the AI generated captions. Above these controls is a timeline where the user can skip to specific areas of the video and also view the numerical duration remaining.

How was it made?

Data Configuration

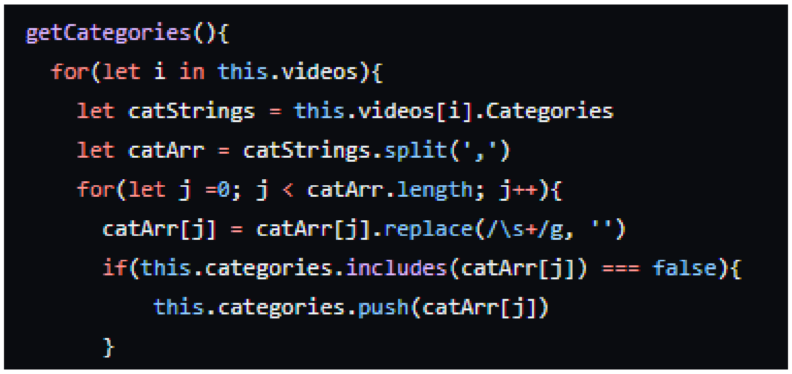
Data management had a major cascading effect as this project evolved. Prior to beginning the project, I had 45 two-hour videos to contend with. Developing a proper way to store and stream these videos was the first major issue I encountered. The first version of the project merely imported videos from a direct path on my harddrive to store them in a JavaScript variable. This wasn't a realistic way to distribute them and would never be used within the industry. The next idea I had was to stream them over a network and set up a buffer that would send video packets to the client. After further consideration and research, I decided to take a more typical industry approach by finding a service that would store and distribute these assets for me.

After trying different cloud-based storage methods, I decided on Amazon Web Services (AWS). I used their Simple Storage Service (S3) to create a 'Bucket' that holds 24 clips and their correlating thumbnails along with several other assets used throughout my website. These assets are then distributed through another AWS tool called Cloudfront. This service provides a delivery network that essentially allows me to retrieve any files from my bucket. I found the JSON data format to be a powerful tool for the storage and organization of my video objects. Each object contains its own set of data including these links from AWS and any additional information pertaining to that video.

Captions are a staple feature for modern video streaming services. Creating them for each video was a complex yet fun problem to solve. Traditionally to create these you would manually type them into a video tape text file (.vt) and set specific time frames. I found more efficient and easier ways to generate them using AI modals. In the end, I decided to use 'Whisper' from

OpenAI. To use this API I'd simply pass in a video file with optional parameters. It then detects what language and words are being used, generating an entire .vt file within minutes. Although I only generated the base-level English captions, Whisper can translate videos into almost any language at different intensity models with varying degrees of accuracy. To use this API, I had to configure Python, along with some dependencies like Pytorch. As a fun side-project, I ran these captions through OpenAI's Dall-E-2. This is a JavaScript API that generates images based on a text prompt. My idea was to use the context from the captions to generate thumbnails. Unfortunately, the images ended up being unrelated and irrelevant so I couldn't use them.

Finally, I refactored my app to automatically adjust for any future additions of data. Below is how I set up my interpretation of categories, to create a seamless transition when a new category is added.



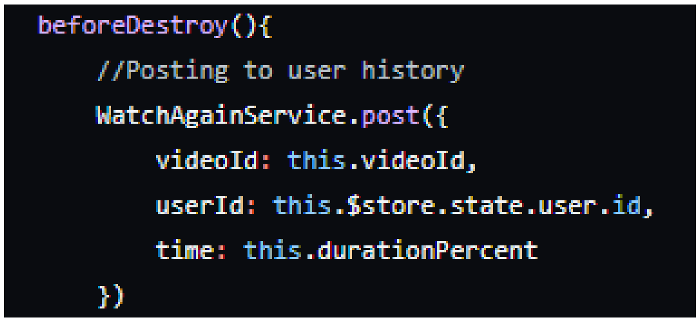
```
getCategories(){
  for(let i in this.videos){
    let catStrings = this.videos[i].Categories
    let catArr = catStrings.split(',')
    for(let j =0; j < catArr.length; j++){
      catArr[j] = catArr[j].replace(/\s+/g, '')
      if(this.categories.includes(catArr[j]) === false){
          this.categories.push(catArr[j])
      }
```

To put this into words, this function loops through each video object finding the list of comma separated categories. It then parses through each list and adds any new categories to a master list. This will update and create new components from within the gallery.

Front-end Development

To develop the graphical interface of my project, I decided to utilize Vue. This is one of the most popular open-source, front-end JavaScript frameworks used for efficiently creating user interfaces, components, and state management systems. I had a brief introduction to Vue in a course and decided to choose it over other frameworks due to the beginner friendly syntax and variety of useful tools created within the community. It follows a Model-View-ViewModel pattern (MVVM) meaning it acts as a two-way binder with the view and the model layers in an application, making reactive web pages. The model layer is referred to as the 'state' of data or logic that can be uniquely changed. The user interacts with this through the view layer, an enhanced HTML template to interface and display current states of data.

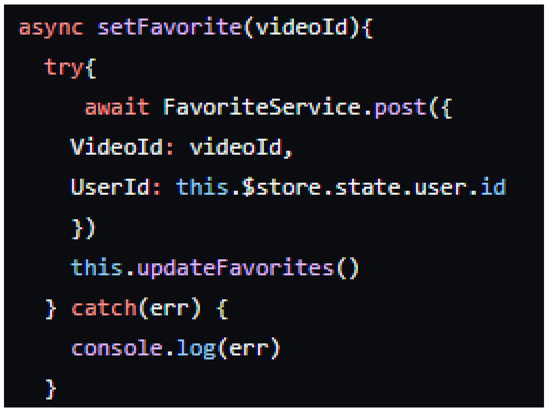
I did the majority of my website's functionalities within specific 'Lifecycle Hooks' and methods. Lifecycle hooks are the stages that each component, or section of the site, goes through while attaching to or detaching from the page. These hooks include created, mounted, before mounted, unmounted, updated, destroyed, and before destroy. Lifecycle hooks are a crucial part in my project since elements are put together and taken down in a strict order, triggering functions during specific cycles. Here is an example of when I use 'beforeDestroy' so the call occurs when leaving the video player page, making a post to the user's history.



```
beforeDestroy(){
    //Posting to user history
    WatchAgainService.post({
        videoId: this.videoId,
        userId: this.$store.state.user.id,
        time: this.durationPercent
    })
```

I pass 'data' variables with the 'this.' prefix referring to this current instance of that variable within the app, sending the user's id, video, and duration watched as parameters. Using the 'destroy' hook here failed to work since the component's data had already been removed before this post was made, leaving out crucial information. Similar interactions will affect hooks such as 'created' versus 'mounted' versus 'before mounted', especially when I request and display assets to the page. Other hooks I use include, ‘Computed’, ‘Updated’, and ‘Watch’. I found the ‘Watch’ hook to be particularly useful for systems like updating query URLs or pausing the featured video while the modal is active, triggering a function when a specified data variable is changed.

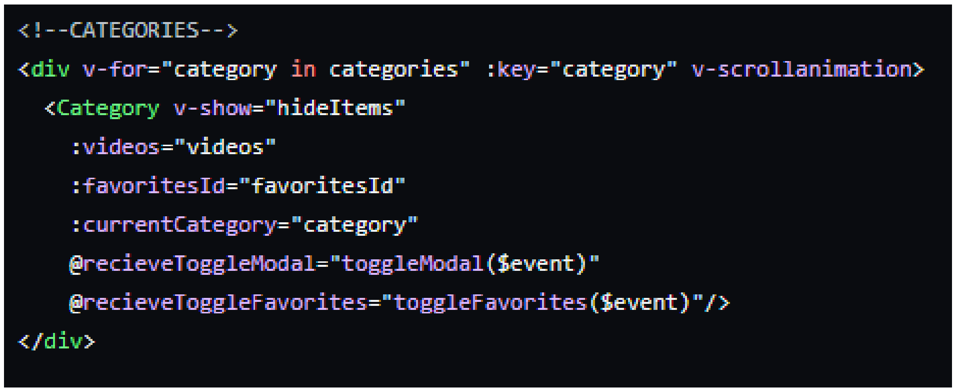
‘Methods’ allow me to create functions that alter the view and data states, typically using vanilla JavaScript within them. They can be triggered in a variety of ways usually from the HTML clicks, keypresses, or mouse over events. The HTML syntax is written like this: @click=”methodName(params)”. I set up the keyboard hotkeys, using a similar strategy. Below shows an example of a method that adds a favorite to my database from an on-click listener attached to a button. Notice I also call another function, updateFavorites, within this method.



```
async setFavorite(videoId){
  try{
     await FavoriteService.post({
    VideoId: videoId,
    UserId: this.$store.state.user.id
    })
    this.updateFavorites()
  } catch(err) {
    console.log(err)
  }
```

After trying to make the entire application in one set of Javascript, HTML, and CSS files, I quickly found this to be very unorganized and the root of many issues. A prominent feature of Vue is their composable API and component system. This allowed me to reuse and separate sections of the app utilizing object-oriented designs. These component files condense the HTML, CSS, and JavaScript onto one, singular .vue file/object, allowing for the creation of distinct scopes and far better organization. At runtime, the app is built from one singleton Vue object, constructing the site and only rendering different components when needed.

Some of the components I made include my login screen, loading page, navigation bar, category sections, carousel, video display page, and the main gallery. Due to this separation I frequently had to transfer data between components. A ‘Prop’ is the parent component’s data sent down to a child component while an ‘Emit’ sends data from the child back up to the parent through method-like calls. Here is how I populate my category components within the main gallery.

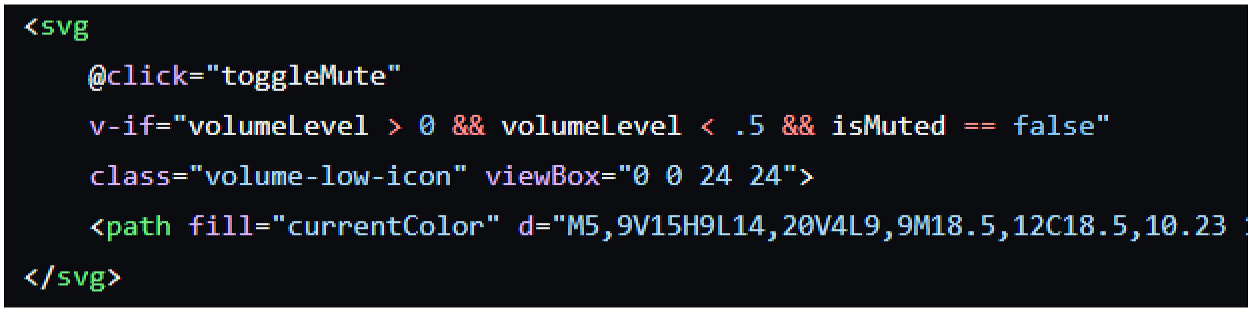


```
<!--CATEGORIES-->
<div v-for="category in categories" :key="category" v-scrollanimation>
  <Category v-show="hideItems"
    :videos="videos"
    :favoritesId="favoritesId"
    :currentCategory="category"
    @recieveToggleModal="toggleModal($event)"
    @recieveToggleFavorites="toggleFavorites($event)"/>
</div>
```

This element utilizes the v-for directive to make a loop, creating a new ‘Category’ component for each category in an array. Each instance is passed that category, a list of videos, and the user's favorites as props, using the ‘:symbol’ syntax. The emit comes from within each category's method body, written: ‘this.$emit(‘recieveToggleModal’, params)’. They are then received by the parent with: ‘@recieveToggleModal’ and handled by the method call ‘toggleModal($event), with ‘event’ being the stored parameters.

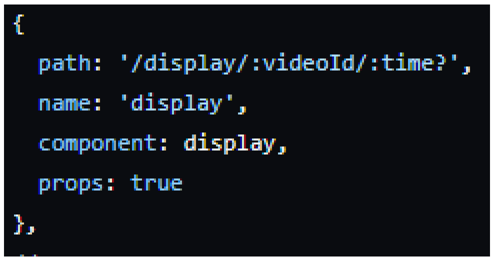
Components are capable of making elements easily reusable, using a builder pattern. I designed my own carousel skeleton component using this strategy, saving hundreds of repeated lines of code. It is responsible for building nested sections such as ‘Category’, ‘Favorites’, and ‘Watch Again’. From the gallery, an array of pertaining videos is sent to each of these component sections, like ‘Watch Again’. Their job is then delegated to the carousel component where it will be dynamically created with a basic structure. The strategy pattern is used here to decipher between different sections and style each carousel uniquely. I set up reactive classes for section specific features, like the ‘Watch Again’ progress bar. All carousels come with side scroll buttons, interactive index tabs, and a consistent format. The number of tabs adjusts in real-time based on the width of the screen and number of videos currently visible from that section.

Vue directives are an effective and efficient tool within the HTML portion. These refer to attributes like v-for, v-if, v-once, v-modal, and v-show. They are a way to set up bindings, add conditionals, and manipulate the DOM. I was able to design my own directive, ‘v-scrollanimation’ for an easy screen observer tool. I can simply add this attribute to any element tag and it will trigger a function when that element comes into view, applying some preset CSS classes, like fading in. These conditional directives are used to hide, remove, and toggle different elements. Here is one of the three ranges within the volume slider showing the low-volume icon and hiding the other two when conditions are met.

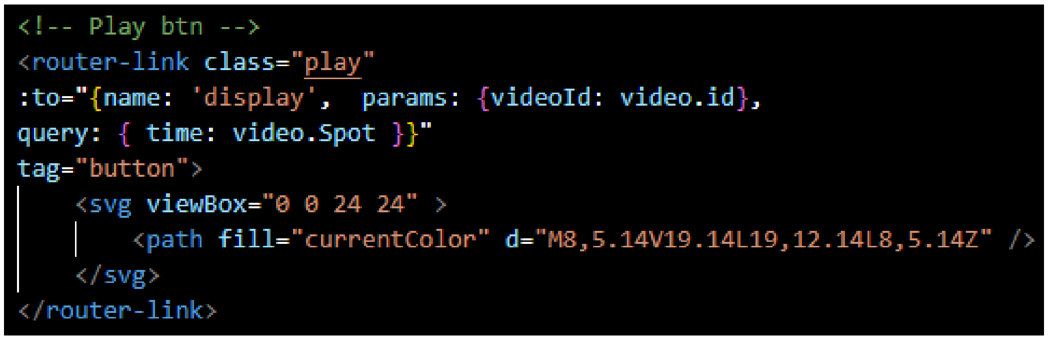


```
<svg
    @click="toggleMute"
    v-if="volumeLevel > 0 && volumeLevel < .5 && isMuted == false"
    class="volume-low-icon" viewBox="0 0 24 24">
    <path fill="currentColor" d="M5,9V15H9L14,20V4L9,9M18.5,12C18.5,10.23
</svg>
```

I use many libraries and add-ons throughout the front-end portion. These primarily include Vue Router, Axios, VueUse, Webpack, and Vuex. An important tool to navigate through my different pages is Vue Router. I set up ‘routes’ to various pages in the app, passing props and parameters between them. Here are examples of how I route to the display page of a video. The URL includes the parameters for the videos id and an optional time stamp for resuming history. Which is a similar method I use to create query URLs in different parts of the app. The HTML image shows how I can trigger the route navigation from a SVG button.



```
{
  path: '/display/:videoId/:time?',
  name: 'display',
  component: display,
  props: true
},
```

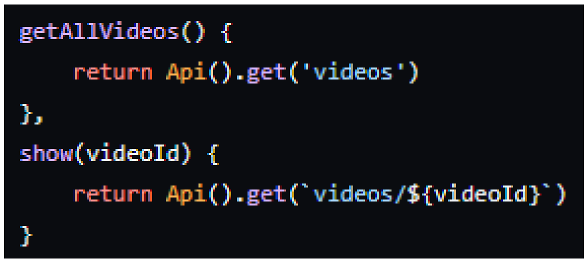


```
<!-- Play btn -->
<router-link class="play"
:to="{name: 'display',  params: {videoId: video.id},
query: { time: video.Spot }}"
tag="button">
    <svg viewBox="0 0 24 24" >
        <path fill="currentColor" d="M8,5.14V19.14L19,12.14L8,5.14Z" />
    </svg>
</router-link>
```

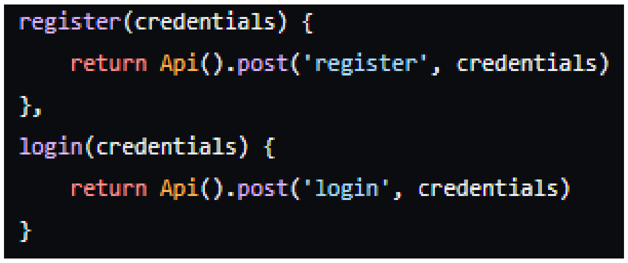
Vue Router is also responsible for preventing bypassing users. I set up a beforeEach function to check if the user has a login token before routing to any pages, properly redirecting back to the login page when necessary.

Vuex is a state management pattern and library where I store information such as the token, userId, and 'isLoading' state. I use concepts such as state, mutations, and actions that allow me to create, update, and retrieve these variables within each session. I can access and change these state variables throughout my program with calls like 'this.$store.state.user.id' or 'store.commit('setLoading', true)'. These variables are stored into the user's own local storage, allowing for easy retrieval and debugging.

Axios is a commonly used tool for routing the communication between the client and server, expanding the controller layer and connecting my API. This is where all my 'services' and back-end calls remain, passing additional information through URL queries and parameters. Here are a few examples of different Axios calls to the server, I will discuss how they are handled in a later section.



```
getAllVideos() {
    return Api().get('videos')
},
show(videoId) {
    return Api().get(`videos/${videoId}`)
}
```



```
register(credentials) {
    return Api().post('register', credentials)
},
login(credentials) {
    return Api().post('login', credentials)
}
```

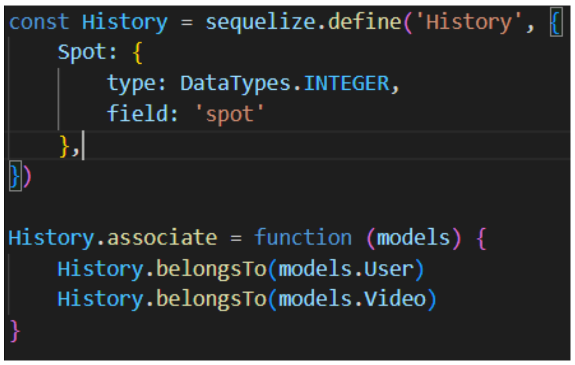
The last part of front-end development contains Vue’s HTML and CSS styles where elements are formed for the view layer. Within the HTML template, I’m able to create elements dynamically using reactive attributes, directives, classes, and data. Within this I use Vue’s built-in transitions for simple animations and set up complex layouts to display data and handle events.

For styling I settled with plain CSS as I had some issues properly configuring Tailwind, Vuetify, and SCSS with my earlier version of Vue. Styling was a big learning curve and more difficult than anticipated. I experimented through Chrome DevTools and used tutorials which are provided in the resource section below. Reusing and utilizing components, like the carousel, better organized CSS by having scoped styling sheets to separate the different sections of the page, removing any interference. I learned how to use SVGs to efficiently create button icons throughout the site. They create simple vector images and animations based on mathematical formulas, removing the need to import files or retrieve assets. Similar to this, I learned techniques like ‘Lazy Loading’ that only render elements currently needed or visible to the user. My videos were taking far too long to load even while they were not playing. My solution for this was to make thumbnail images for each video and use ‘Skeleton Loading’ placeholders before the assets load in.

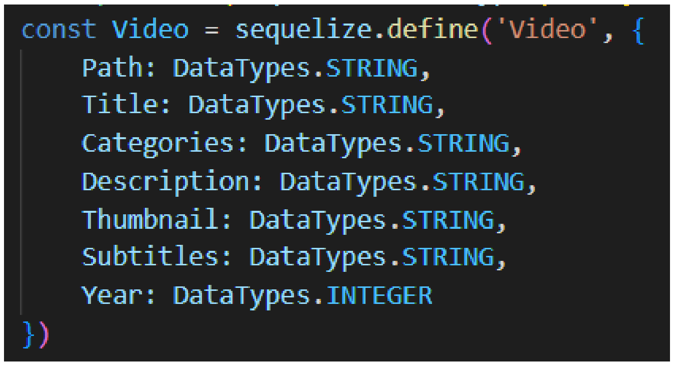
Back-end Development

Although creating a back-end server wasn't in my initial plan, I'm glad I incorporated one into the project. The server is primarily responsible for providing end-points that interact with my SQLite database. I also seed the data of my videos and enforce an authentication system. To develop this I opted to use Node and Express, as I found this commonly used in Vue apps. I was able to efficiently learn and adapt my project through code examples and documentations that I provide in my resources section. In a Model-View-Controller app, this portion of the project is referred to as the Model, where the controllers communicate to and the database is held. Optimally, I would refractor the server to do more data organization before sending a response back to the client, focusing on a facade structure.

The tables I use in my database include 'User', 'History', 'Favorites', and 'Videos'. This was my first experience with databases so setting up tables and migrating data took time to learn. For simplifying interaction, I used an object-relational mapping (ORM) tool, Sequelize. With this I'm able to create associations that set relationships between tables. For example, favorites are simply a 'belongsTo' association between a user and a video. Here are some examples of schemas I created for my history and video tables.



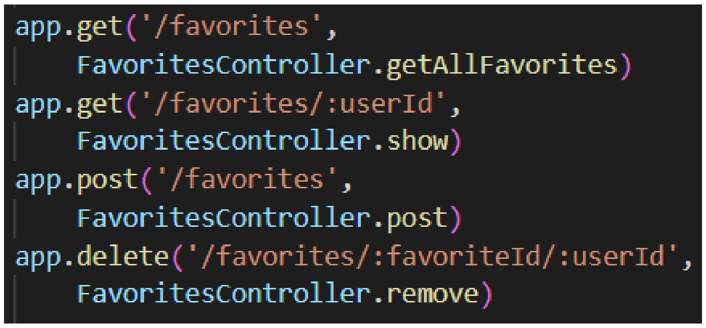
```
const History = sequelize.define('History', {
    Spot: {
        type: DataTypes.INTEGER,
        field: 'spot'
    },
})

History.associate = function (models) {
    History.belongsTo(models.User)
    History.belongsTo(models.Video)
}
```



```
const Video = sequelize.define('Video', {
    Path: DataTypes.STRING,
    Title: DataTypes.STRING,
    Categories: DataTypes.STRING,
    Description: DataTypes.STRING,
    Thumbnail: DataTypes.STRING,
    Subtitles: DataTypes.STRING,
    Year: DataTypes.INTEGER
})
```

The 'History' table is similar to 'Favorites' except the additional field created at that association, the 'Spot' in the video. I theory crafted a 'Profile' table for this project. I was planning to allow each account to make multiple profiles, similar to most streaming platforms. I'd make a new 'hasMany' association between a user and their profiles, shifting around the other associations appropriately. I didn't discover 'DB Browser for SQLite' until further in my project development but this software would have helped implement systems like this. It provides an interactable UI for the tables in a database which help to directly change data and configures.

To connect the front-end Axios requests to the server, I set up a 'Router' on my server to receive and handle them. This program listens for methods such as 'get', 'post', and 'delete' and passes along additional information through parameters and queries as shown below.

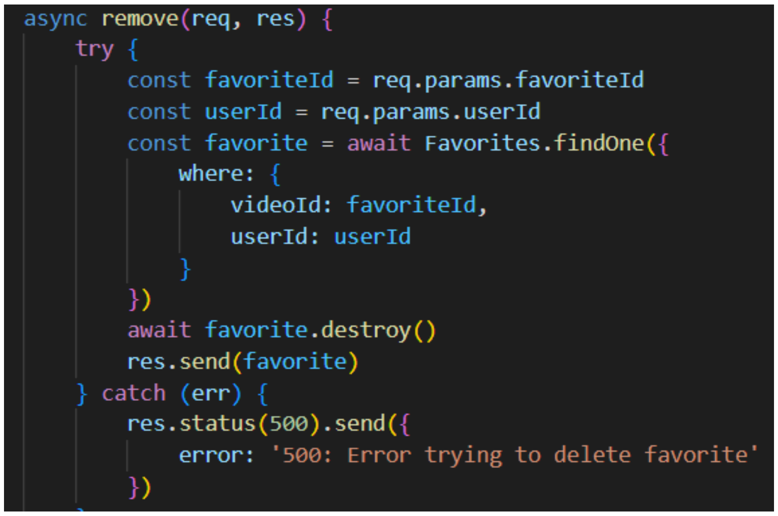


```
app.get('/favorites',
    FavoritesController.getAllFavorites)
app.get('/favorites/:userId',
    FavoritesController.show)
app.post('/favorites',
    FavoritesController.post)
app.delete('/favorites/:favoriteId/:userId',
    FavoritesController.remove)
```

The router then directs each request to a corresponding 'Controller' method. This is where it eventually communicates with the database. I found the software, Postman, useful for testing and validating these endpoints as well as populating my database with users and videos.

Creating a user session requires a few different libraries and systems. I first check the user's input with my authentication policy and a schema package, Joi, enforcing a proper email and password. If the credentials are acceptable, I create and send back a JSON Web Token (JWT). This acts as a key for the user to consistently interact with my site and to validate their account login status. Before storing sensitive information into the database, like a password, I hash it with Bcrypt. This randomly transforms the value into an unpredictable string. I also added a 'Salt' factor that puts additional characters into the value before hashing for even better security. To expand on this, I would implement the library, Passport. This is a middleware security system used to authenticate requests made to the server making sure the right user is sending them.

When retrieving specific sets of information from my database, I use a ‘Where’ statement with arguments that extract only relevant data. Below I provide an example of how I implemented removing a favorite. I pass parameters for the video and the user id to get the location in the database, remove it, and then send back the result to the client.



```
async remove(req, res) {
    try {
        const favoriteId = req.params.favoriteId
        const userId = req.params.userId
        const favorite = await Favorites.findOne({
            where: {
                videoId: favoriteId,
                userId: userId
            }
        })
        await favorite.destroy()
        res.send(favorite)
    } catch (err) {
        res.status(500).send({
            error: '500: Error trying to delete favorite'
        })
```

Conclusion

My goal was ultimately met; I feel confident in my ability to develop and learn complex web applications from scratch. This project took over two months of consistent work and about 2500 lines of my own code, excluding CSS and JSON files. While developing this app I decided to simultaneously learn Git which allowed me to organize the project in an accurate industry setting. I created an extensive ‘README’ in Markdown where I explain the app’s features, tools, and provide an in-action preview. This project has greatly increased my traction with employers. In fact, it helped me land my first internship halfway through development and I have another one planned for this summer. In my current internship as a front-end Vue developer, I use all of these same concepts and tools but at an even higher level. It has allowed me to work effectively, help co-workers, and provide useful insights to the team.

Having experienced the full process of this project has also given me a better understanding of web development as a whole. If I were to recreate an app of this level again, I would adjust some of the tools and techniques to further optimize the project. Some particular tools like Vue 3, TypeScript, SCSS, Vitest, and Firebase would simplify and extend the capabilities of this project. Concepts like mixins, unit testing, object patterns, and different development methodologies would further enhance the development experience.

In working through this project, I followed my initial proposal fairly well. The only changes I made were to remove features I didn't find to be valuable learning experiences and to add other more challenging systems. Overall, I'm very happy with my journey in developing RetroFlix. I am grateful for all the help I received in making this project into what it is today. This project allowed me to push myself and learn more about the industry I'm interested in. I found a greater appreciation for application development and look forward to growing in my professional career.

Additional information, code, and previews can be found at:

github.com/BrenanMarenger/Capstone-Project

Resources

Video Player Styling

github.com/CodingGarden/css-challenges/tree/master/netflix-video-player

Backend Server Guides

github.com/codyseibert/tab-tracker

github.com/sequelize/express-example

SVG Icon Library

https://fonts.google.com/icons

Useful Documentations

https://sequelize.org/

https://vueuse.org/

https://vuex.vuejs.org/

https://vuejs.org/

https://router.vuejs.org/

https://expressjs.com/

https://github.com/openai/whisper