RecordTop Records,
an E-Commerce Application

by

Justin M. Gruen

A report submitted to

The Department of Computer Science

Washington College

In Partial Fulfillment of the Senior Obligation for the Degree of

Bachelors of Arts

Major Subjects:

Computer Science

Approved by:

Dr. Kyle Wilson, Thesis Advisor

Dr. Shaun Ramsey, Department Chair

Contents

1 Introduction

1.1 Inspiration

I remember when I first arrived at Washington College. The grass was damp and the sky was gray from rain the night before. Despite that there was a definite excitement in the air as I hurried to my first class, excited for what lay ahead. Four years later as my journey from the schoolyard to the real world draws to a close that excitement I once felt had come back more prominent than ever. Countless possibilities lay ahead, and the world is my oyster, ready for me to build a pearl from. Unfortunately, though, from that excitement comes a paralyzing fear of indecision and worry. With so many possibilities, what if I make the wrong choices? One day, when I look back on everything I have done, how could I regret the least?

After a lot of introspection, I realized grown as I had through my four years, there was still a confidence missing; a confidence in my computer science abilities and a confidence in what I had learned. I have earned my degree, but am I worthy enough to be called a computer scientist? Do I have the knowledge and wherewithal necessary to prove both hirable to others and an accomplishment to myself? I was not sure how to go about doing so when it dawned on me my Senior Capstone was coming up, where I choose my own project and see it through. And so, with that motivation, I decided for my project that I would build a website from the ground up; my senior capstone idea was born.

1.2 Overview

As previously mentioned, the goal of this project was to create an e-commerce application that could be used by a small business. In this case, I decided to focus it for a business I made up for the project called RecordTop Records, a small business that manages and distributes vinyl records. Vinyl records are an older medium of sound storage stored in microgrooves

on the records that can then be played on a phonograph to produce music. Hence, while not often played, vinyl records are often sought after by hobbyists, collectors, and older generations alike. For reference, going forward I will alternate between the use of 'the store' and 'the website', but unless specified otherwise, they both refer to the website I made for the capstone project.

A website is often made up of two overall categories called the front-end and the back-end. Front end processes are logic run client-side that are focused on users. This can mean any type of user, or in the case of this website, a customer or business administrator. Although we will go more in-depth into it later, logic being run client-side means logic being loaded and ran on the user's computer. This can range from displaying the webpages necessary, to accessing databases, to even adding or subtracting data or displaying the right colors.

Back-end processes refer to the logic that occurs on a server and not seen by the users. For instance, Firebase, the service I use to store data, is a back-end service. I run calls to it from my front-end and the data is sent to their back-end server where it is processed. Data should not be processed and held in a database on the front-end for security reasons: running that data on a client's computer could allow the client to view and edit it, which would be a huge issue for multiple reasons, not to mention the egregious violation of trust should a user be allowed to view other user's personal information. In addition, back-ends are run constantly where front-end code is only running when the website is loaded. For my store this would not matter as much, but in others where they need to be constantly calculating or retrieving and updating data it can be.

It is worth nothing that although I structed my front-end and back-end a certain way, a lot of processes run client-side are not limited to only being able to be ran on the client. Although I access the firebase service from my front-end, I could also send the data to a back-end server of my own, and then have that send to the firebase service from there.

In my case though this would ultimately slow it down. I send and retrieve data from Firebase in almost every component of my website. All those requests already take up time and adding the middle step of sending it to my own back-end server first only adds more. It is much faster to just access it directly from my front-end. In addition, there are some cases where the functions available to me are greatly diminished by only using client-side logic. Stripe, the payment processing software I use, is accessed from a back-end server I created. While it has the ability to be ran from the client, because it is not as secure I would be locked out of much of the Stripe service; this includes the ability to use coupons, discounts, one-time payments, and more.

Regarding the functionality, I mentioned that the website is intended for two types of users: customers and administrators. For customers, my website provides functionality for:

a) Creating a user account with an email and password, a google account, or both.
b) Editing their account information as needed. This means linking or unlinking google accounts/email and pass accounts, changing their password within their account, or, if they forget their password and can't login, requesting an email to change it.
c) Browsing through available products by searching, filtering, or just perusing. Each product consists of a name, a price, a genre, and a picture.
d) And finally, purchasing the products themselves by adding them to a checkout cart and paying for them with real-world money.

Administrators, or those in charge from a business-end, can view and edit users and products. In the case of users, administrators are only allowed to change their registered names. Anything else would be a huge breach of privacy and violate the trust of the users. In the case of the products, these consists of the name of the record, its price, its genre, and a picture of it. Administrators can view this information as well as edit or delete any of the

products as needed. Unlike users, administrators have full control over products as the business needs to oversee creating and removing products from the site as needed.

2 User Stories

2.1 What are User Stories?

User stories, most used in the AGILE framework of software development, are a collection of thoughts from various potential user perspectives. The purpose of user stories is that, by articulating how users may desire to use our product for their own needs, we as developers gain a better understanding of a project's scope. According to Atlassian Agile Coach, it also

a) Keeps focus on the users, rather than functionality not needed,
b) Enables collaboration between team members as they decide how best to serve the user,
c) Drive creative solutions, encouraging developers to think critically and creatively how to solve tasks,
d) Create momentum and driving developers forward through the small goals completed within each user story. [1]

Generally, before user stories are created, a proposition story is generated intended to lead the thought process for all personas developed, in addition to a short one sentence description of each persona. Afterwards, each personal is more thoroughly fleshed out with a short description before finally assessing their needs regarding the software being developed.

2.2 Proposition Statement

For potential customers who are looking to purchase vinyl records, RecordTop Records is an e-commerce application that provides users with a way to easily search through and purchase records. Unlike other online vinyl record stores, our product was developed to look and function as cleanly as possible with the user in mind.

Potential Users

1. Jonathon Grapes the college sophomore
2. Dr. Steven Applebottom the collector
3. Max D Bailey, the business administrator

2.3 Hobbyist – Jonathon Grapes the College Sophomore

Jonathon Grapes is a 19-year-old college sophomore attending Washington College. He is a computer science major and swordplay enthusiast. After attending classes every day, he enjoys attending the various clubs offered around campus and, later in the evening, spending time with his friends and watching movies. Jonathon was also very close to his grandfather, who helped him found his love for older varieties of music and purchased Jonathon his own record player before he had gone off to college so he can still listen on his own. While he enjoys living in Chestertown, there are no record stores in the neighborhood and Jonathon is forced to look online to purchase new music.

	Problem Scenarios	**Value Proposition**
1	Needs a way easily browse through different records	Provide a filter to search through records
2	Needs to be able to easily pay for his records	Offer a simple checkout process

2.4 Collector – Dr. Steven Applebottom the Collector

Dr. Steven Applebottom is a 37-year-old neurosurgeon at The Johns Hopkins Hospital. Being a neurosurgeon is Dr. Applebottom's dream job and he would like to do it for many years, but its high stakes causes him a lot of stress. As a result, Dr. Applebottom keeps simple hobbies to help him focus on things besides his jobs; one of these hobbies is record collecting. Dr. Applebottom does not listen to the music but enjoys the prestige that comes with having so many records as well as the hunt of looking for new ones. Thankfully, his position as a neurosurgeon awards him enough money where he can enjoy this variety of hobby. Dr. Applebottom, in his search, especially enjoys looking through user listings or independent businesses to look for rarer or unique vinyl records that he would not otherwise.

	Problem Scenarios	**Value Proposition**
1	Would like to be able to purchase more than one record at a time	Offer a checkout cart that can manage multiple items
2	Needs records shipped carefully in excellent condition	Add a description of shipping precautions that we take
3	Would like to be able to easily keep track of purchases	Show a user their previously purchased items on their account page

2.5 Administrator – Max Daisy Bailey, the Business Administrator

Max D Bailey is 38, has three dogs, and graduated from college with a major in English. After publishing his first novel, he realized that the long hours were not for him and instead moved to the private business sector. Eventually, he found joy in a career of keeping track of company inventories. His company, and by proxy the customers, depends on him to maintain track of all the records they have available and update the website with their current stock as needed.

	Problem Scenarios	**Value Proposition**
1	Needs to keep track of item stock	Offer a table to view current items
2	Sometimes inputs incorrect information for items	Offer a way to edit items already placed into the table
3	Has to keep track of user count to ensure growth	Offer an administrative dashboard that could allow administrators to easily view numbers of users and items, along with other helpful information

3 Front-End, How Does it Work?

3.1 Libraries, Dependencies, and APIs

When writing code, there are two essential principles of programming that all programmers follow: don't be ashamed to Google that which you don't know, and if the code exists, don't feel like you have to reinvent the wheel. To focus on the second principle, it essentially means if you can, it is better to use someone else's code rather than spend hours creating your own version of the same thing. In the end, it could save hundreds of hours developing code not needed. This is the reason libraries were invented. Libraries in the context of programming are a collection of functions reusable by other programs. In this way I can use a behavior without having to implement it myself.

Using libraries, I can devote more time to the unique aspects of my project. These libraries when used in code can come in the form of dependencies, libraries needed for the code to work, or as APIs, code used to access features in a service such as another application or server. In my website both dependencies and APIs were used in its development in both the front-end and back-end. The following sections describe the six libraries I used.

3.3.1 React

React is a front-end library used in building user interfaces through reusable pieces of code called components. Components follow an object-oriented style of programming, meaning that developers can create larger components that are made up of smaller components. In addition, React allows for one-way data flow from parent to child components through a props object that represents the data being passed. Each component can maintain their own state, or dynamic data, and their own methods. In addition, React offers the use of JSX, a JavaScript syntax extension that makes it easier to work with data and HTML objects in React components.

3.3.2 Material-UI

Material UI is a React framework that provides custom components in the form of unique components such as Drawers or Tables, or more intuitive versions of already existing HTML objects such as Buttons and TextFields. Material-UI components also provide the benefits of a styled-components style CSS and are self-supporting (they don't require a global style-sheets).

3.3.3 Firebase

The Firebase library is an API to the back-end platform Firebase, developed by Google, that make it easier for developers to create mobile and web applications. Specifically, for my project, I use Google's Firebase email authorization system, the NoSQL Firebase Realtime Database, and Firebase storage to hold images. Together I can efficiently store, update, and retrieve user and product data at will.

3.3.4 Recompose

Recompose is a precursor to React hooks that makes working with higher order components easier to use. Higher-order components are functions that take in a component and return a new updated one (such as with a new prop as I will explain in the next section). In the case of my project, I use its compose() utility function to easily run a component through multiple higher-order components at once.

3.3.5 Stripe Checkout

The Stripe API is responsible for accessing the functionality of the back-end service Stripe. Stripe is responsible for processing purchases and charging user's a debit or credit card. Stripe Checkout, the library I am using, is a simpler version of the full Stripe API that allows quicker setup of Stripe while maintaining most of its functionality.

3.3.6 Axios

Axios is a promise-based HTTP client for the browser and Node.js that essentially acts as a mailman between the front-end and back-end of an application, whether that be a user-made backend or a back-end service. In the case of this project, I use it to submit product and customer information to the backend which the Stripe API then sends to the Stripe service.

3.2 Firebase

The most central piece to the front-end of the website is the Firebase class. By creating a Firebase object with the methods to access it inside, I can easily and safely access the Firebase service. The constructor of the Firebase object contains the reference to access each Firebase service I need: the Email Authentication Provider, the Authentication, the Database, the Storage, and the Google Authentication Provider.

```
class Firebase {
  constructor() {
      app.initializeApp(config)

      this.emailAuthProvider = app.auth.EmailAuthProvider;
      this.auth = app.auth();
      this.db = app.database();
      this.stor = app.storage();

      this.googleProvider = new app.auth.GoogleAuthProvider();
  }
```

Figure 1: Firebase Class Constructor

I send this object to each component that needs to access the Firebase through React context and higher-order components. React contexts are intended to simplify sending props to multiple children. Normally, you would have to pass the props down from parent to children all the way down. Using contexts, I can create a Firebase Context using `const FirebaseContext = React.createContext(null)`, export it to the top parent component, and call the provider as shown in the image below:

```
ReactDOM.render(
    <FirebaseContext.Provider value={new Firebase()}>
      <App />
    </FirebaseContext.Provider>,
  document.getElementById('root')
);
```

Figure 2: Firebase Context Provider

Next, higher-order components in React are functions that take a component and returns a new component. In this case, I have one called withFirebase that takes in a component as a prop, calls the context consumer now containing the Firebase, and creates a new component with the firebase props passed to it as shown below:

```
export const withFirebase = Component => props => (
    <FirebaseContext.Consumer>
        {firebase => <Component {...props} firebase={firebase} />}
    </FirebaseContext.Consumer>
);
```

Figure 3: Firebase Context Consumer

Now, if I need a component to access the Firebase, I can simply import and call `newcomponent = withFirebase(component)` and retrieve the Firebase object as a regular prop. Passing through both the methods allows me to easily pass the Firebase object only to classes that need it.

3.3 Administration

Once a user registers and verifiers their account with administrator privileges, they can access the Administration component of the website. Here is where the administrator can manage users and products. Using the on() function from the Firebase library, it will pull a list of current entries as well as keep a connection open to the database so that when an entry is added or edited, the table will automatically update to show the new information. On the following page is an image of the user table:

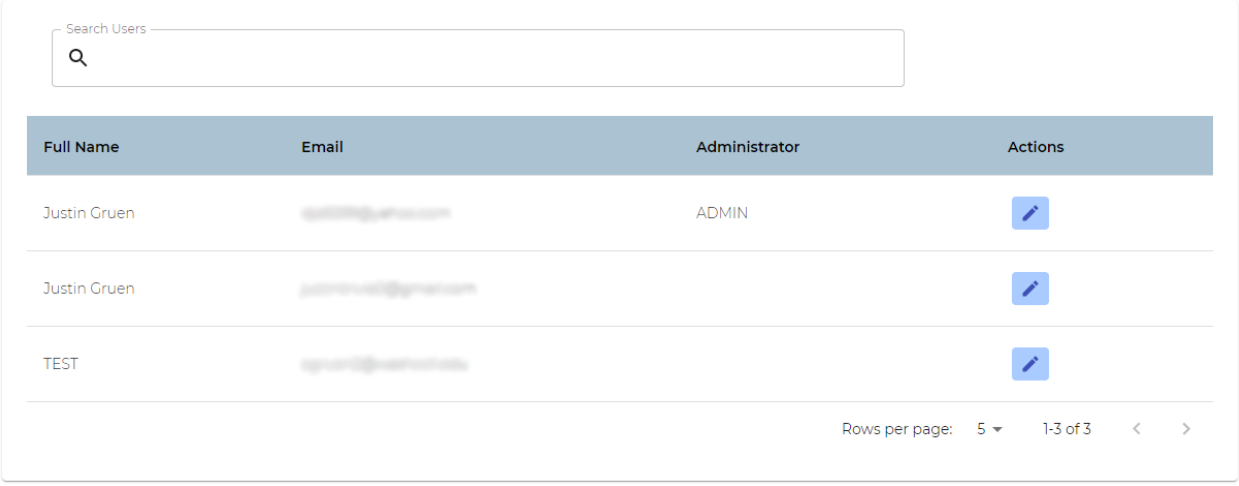

Figure 4: User Table

The table only allows the administrator to edit the user's name, does not allow them to view or edit the user's password, and does not allow users to be deleted. Anything else would be a large breach of trust and privacy. The product table, on the other hand, is more complex as it also uses the Firebase storage service in addition to providing the administrator with more abilities to efficiently manage inventory. It allows the administrator to not only add or delete entries, but also edit any information about the products.

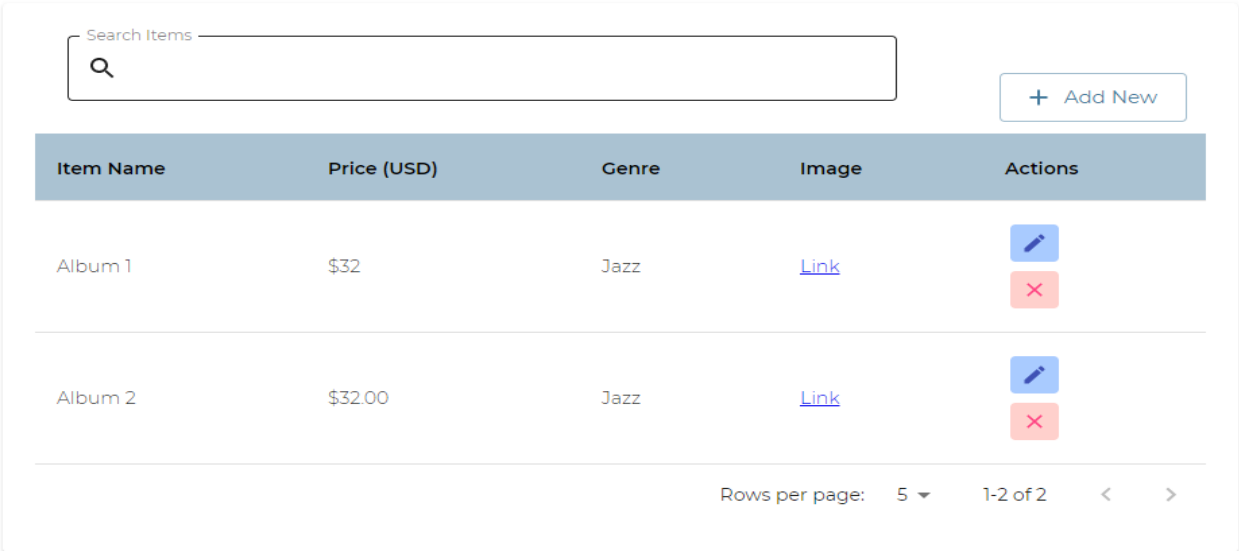

Figure 5: Product Table

In a more in-depth explanation, below are the popups for adding and updating entries, respectively:

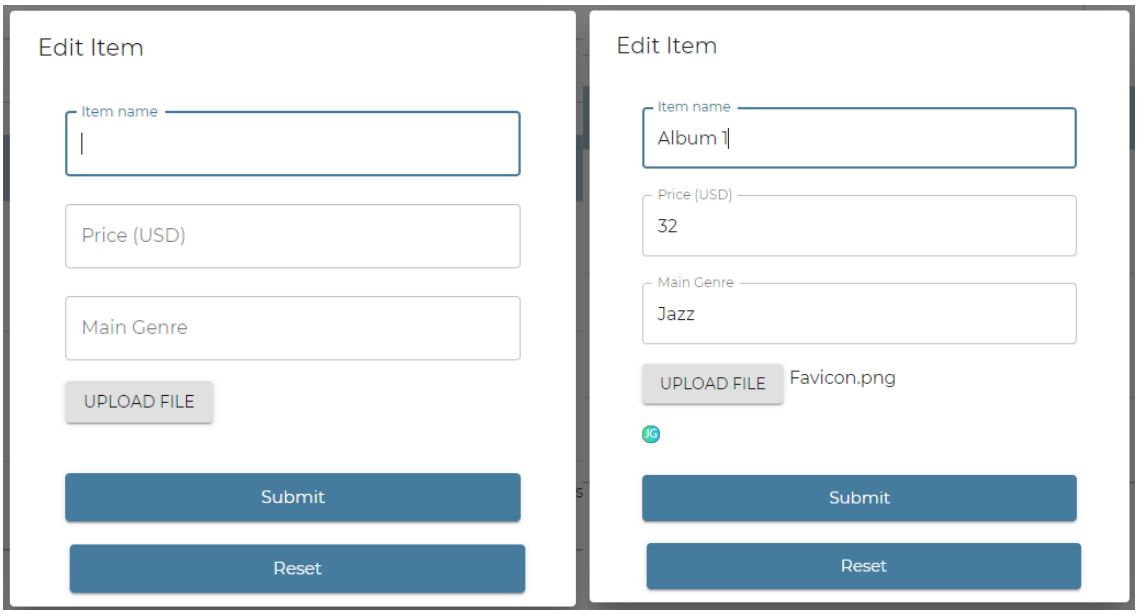

Figure 6 & 7: Item Add Pop-up & Item Update Pop-up, respectively

The table maintains a state holding whether the user clicked "Add New" or the edit button. This way, when a user clicks the add button, the pop-up will display as empty as shown on the left. If a user is editing information, however, it tells the popup to instead display information of the item being edited as shown on the right. In addition, it also tells the form whether to add a new item by using the push() function or update() function.

Originally, the form will display the image from the image url found in the entry. When a user changes the image to something else, I use JavaScript's FileReader() object to display a preview of the image, as it has no regular link it can be displayed from. I am unable to provide a full path instead to the image object due to security reasons; the browser cannot access the file system.

After an entry is done having its information added or edited, the website runs the respective process. Shown below is how new items are added:

```
firebase.prodimage(state.image.name).put(state.image)
.then(() => {
  firebase.prodimage(state.image.name).getDownloadURL()
  .then(url => {
    firebase.items().push({
      itemname: state.itemname,
      price: state.price,
      genre: state.genre,
      image: state.image.name,
      imageurl: url
    })
  })
})
```

Figure 8: Firebase Product Upload Process

To be able to also upload a link for the image to the product entry, it uploads the image to the Firebase storage and retrieves a link first before finally pushing all the information it has to the database.

Updating an entry is similar except that to keep the storage tidy the code first checks whether the current image held in the state is the same as the image being uploaded. If it is, then the image is unchanged and it can just update the other information. If not, it runs a check to see whether any other entries are currently using the image before deleting it. A similar check is ran when deleting a product entry. Afterwards, it calls the same process for adding an item except it calls the update() method instead of push(). The total code for updating an entry is shown on the following page:

```
if(imageChange) {
  firebase.items()
    .orderByChild('image')
    .equalTo(originalName)
    .on("value", snapshot => {
      if (!snapshot.exists())
        firebase.prodimage(originalName).delete()
      })

  firebase.prodimage(state.image.name).put(state.image)
  .then(() => {
    firebase.prodimage(state.image.name).getDownloadURL()
    .then(url => {
      firebase.items().push({
        itemname: state.itemname,
        price: state.price,
        genre: state.genre,
        image: state.image.name,
        imageurl: url
      })
    })
  })
} else {
  firebase
  .item(state.uid)
  .update({
    itemname: state.itemname,
    price: state.price,
    genre: state.genre,
    image: state.image.name,
    imageurl: state.imageurl
  })
}
```

Figure 9: Firebase Product Update Process

3.4 Product Catalog

The catalog component is responsible for displaying all the product entries in the database. I call attention to it to discuss how each product is displayed. In CSS, a single grid is a two-dimensional layout system for HTML objects to be displayed in a row or column. Creating nested grids, or grids within grids, allows for rows and columns. For example, a row of columns. Shown below is a visual example of the difference between the two:

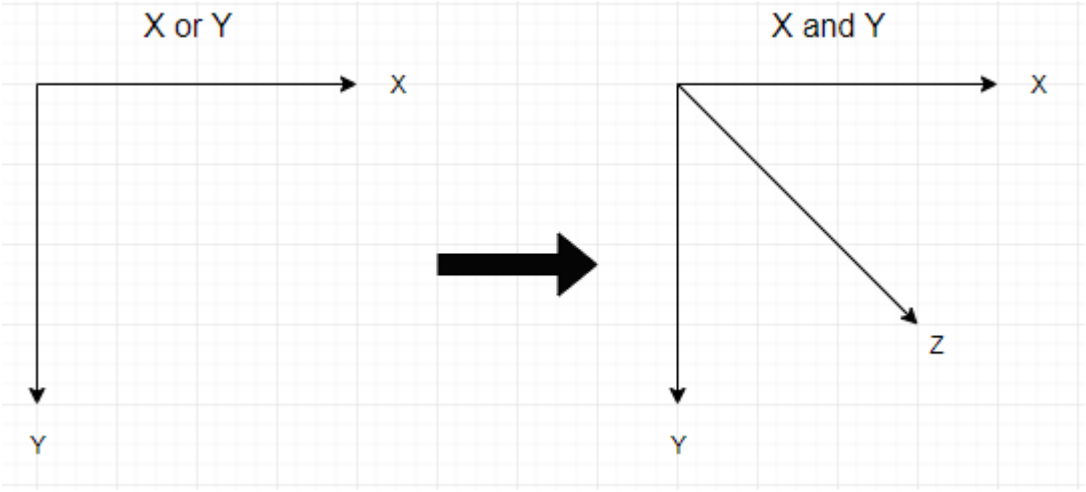

Figure 10: Grids vs Nested Grids

Using this method comes with one caveat though; I can only load data into one column at a time. This means that if I loaded 6 items, it would be displayed like this:

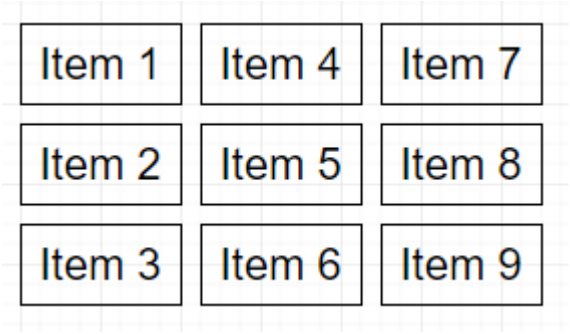

Figure 11: Original Nested Grid Solution Results

Not an issue when a user does not care about the order, but it bothers me as the developer. To remedy this, I divide the initial array list into an array matrix, as shown on the follow page:

```
let cols = [[],[],[]]   // There's got to be a better way of doing this
let col = 0
itemsList.forEach(item => {
  cols[col].push(item)
  col++
  if (col === 3) col = 0
});
```

Figure 12: Current Nested Grid Product Solution

Essentially I place an item into the first array, then the second into the second array, then the third into the fourth array, and then it loops back so the fourth item goes back into the first array, and it ends up looking like how we'd expect it to like this:

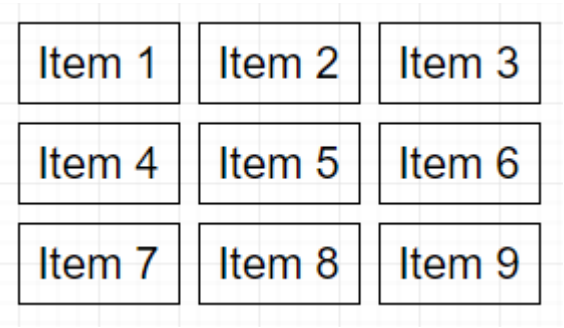

Figure 13: Current Nested Grid Solution Results

Finally, sometimes either or both last two columns may contain less items than the first. Because grids are meant to be dynamic, the shorter columns will end up spacing the items in a different way, like the following diagram:

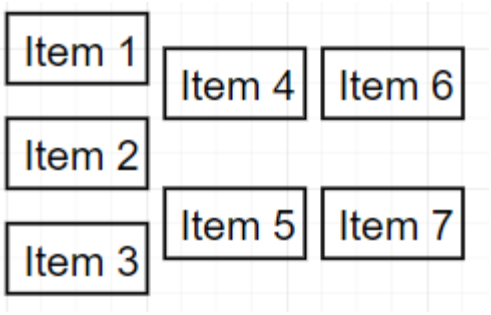

Figure 14: Differently Sized Grids

To fix this, I check if either column has less items than the first. If so, I push a null item onto that column's list so it is consistent with the first column. Then once the website displays the grid, if it sees an object is empty, it will instead create an empty grid item, so it ends up looking like how the user would expect it to.

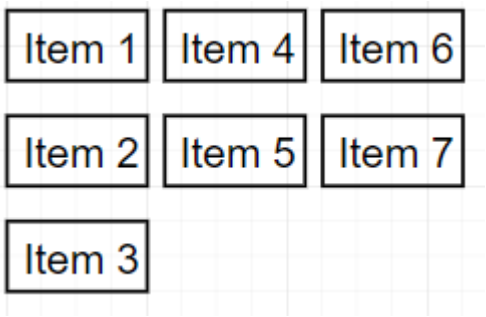

Figure 15: Corrected Grid with Different Sizes

3.5 Checkout & Stripe Back-End

The last major piece important to the website is the checkout process. As previously described, the store uses the Stripe Checkout library to charge a user's debit or credit card. This takes place over both the front-end and the back-end. In the front-end, once a user clicks the "Add to Cart" button beneath the product they want, the website sends the data to their user entry in Firebase. I use Firebase to hold cart items so it can persist between sessions

should a user decide to come back to the website later. The downside to this unfortunately means that unless they delete their account or remove the items from the cart themselves, they will always be in there which can be annoying.

After the user decides they are finished adding cart items, clicking the checkout button retrieves the items from the user's database entry and opens the Stripe Checkout card-entry form. The information entered, along with the calculated total cost, is sent to the server back-end through Axios by using `Axios.post()`. Axios is a library that can transfer data between my front-end client and back-end server. After the back-end receives the data it creates a charge and sends the data using `stripe.charges.create()` where Stripe can automatically process the request and charge the user the amount owed.

4 Reflection

4.1 Challenges

Throughout the course of the project, I faced many difficulties as I learned to build a complete storefront website from the ground up. Likely the greatest of these challenges was learning to scope and not overextend. I found it extremely easy think of a cool functionality I could include and begin doing so, only to later wonder whether it was necessary and ultimately take it out. Learning to visualize the end goal of my project was an obstacle that challenged me until the very end. For instance, as previously explained, the administration section is made up of a section for users and another for products. To be able to easily switch between the two, I thought it would be cool to implement a nice looking swipeable tabs component that I had found. However, after spending a couple of hours with it I realized it did not make any sense.

I had previously decided that the administration component would be intended only for use on a computer, therefore if I used the swipe feature I would have to spend even more time fixing the component's design to work on mobile, as that is the medium it is intended for. In addition, the bar holding the tabs did not fit in with the style of the rest of the page. It would look nice by itself maybe, but with another bar already above it, the navigation bar, it looked sorely out of place. However, these challenges were helpful. It was in situations like this I was forced to sit and think about the product and truly decide how I wanted it to ultimately be shaped.

Another major challenge I was faced with was deciding which checkout API to use. Although I used the Stripe API as previously mentioned, for almost a week I was stuck between Stripe and another called SnipCart. Personally, I thought Stripe looked and functioned better from a user's perspective, but it required me to spend time creating a backend server as it required. SnipCart, on the other hand, had an easy-to-implement

functionality as well as could automatically handle multiple items in a checkout cart, but ultimately in my opinion looked worse. It took nearly a week of research and indecision before ultimately deciding to use the Stripe API. While SnipCart was easier to implement and could already handle multiple items, Stripe is an industry standard and I decided that the extra time would be worth the knowledge gained.

4.2 Future Ideas

While the project itself is finished, should I come back to this someday, there are plenty of directions that I can take it in. The first of these would be to add more functionalities directed towards customers, such as customer reviews, a list of previously purchase products, and a list of recommended products on their account component. Customers may find these helpful if they are looking for similar products to purchase but searching by musician or genre aren't sufficient. Perhaps I could filter this recommended list by what other people who purchased similar items would also purchase, or perhaps I could implement a mini quiz asking the user questions about their interests and from there generate a list.

Another idea I could implement is expanding product information. As of now it only allows to a product name, a price, a main genre, and a picture. Not only do albums often have multiple genres, but it is set up where administrators must type in the genre. If they misspell the genre and are unaware, the disc will not show up when customers or administrators filter by the genre, and the mistake may go unnoticed. To these extents, implementing a dropdown with set genres allows to multiple genres to be selected would be more helpful. In addition, as previously explained, the reason administrators can freely type in the genre in the first place is because I may not be able to account for them all should I create a dropdown. Instead, maybe at the bottom of the dropdown or elsewhere I could provide an option for the administrator to add a new genre. This would easily allow the developer, myself, to not need to account for all possible genres; they can be added as needed.

Similarly, on the topic of item information expansion, putting a stock counter would also be helpful. This way it makes it easier to keep track of the number of records in inventory without the need for a third-party software, as well as make it easier to notify customers when inventory is low. It could even be set up for administrators to be able to set a point where inventory is low per item, and the store will automatically create a warning beneath an item in the catalog as well as email myself or a business manager, for example, that the inventory is below the certain point.

Next, I would like to use cookies or other local storage options to allow customers to purchase products without needing to be signed in. Currently, I maintain user's current cart items in the Firebase storage within each user. This allows me to hold a user's cart items even after they log out, but it means the user needs to be signed in so it knows which user is looking to purchase which items. Else, I currently have no way implemented to hold the cart items list. Instead, utilizing some form of local storage would allow even unsigned-in users to purchase records.

Finally, even though the website loads relatively fast, and each component loads near-instantaneously, I think it would be a good idea to convert from a React application to a Next application. While React is CSR, or client-side routing, Next builds on React and has SSR, or server-side rendering. SSR is another method of rendering websites that preloads the page, whereas React normally requires webpages to be loaded and rendered client-side. Upsides to SSR include better search engine optimization with Google due to how they scan websites and accomplish their indexing, as well as smoother loading, but also means a slight delay between components as each one is loaded. You might argue that CSR, in the case of my website, works extremely well because there is essentially no loading between switching components (this does not refer to loading accomplished when data is pulled from Firebase). However, in my opinion, it feels unnatural. Even though users prefer the shortest time possible, I feel that the quick loading makes it feel almost too fast, and not like a real website.

For instance, the catalog page loads product pieces, and then loads the image afterwards giving it an unpolished feel. SSR would allow the page to be fully loaded before showing it on the user's screen.

References

[1] Rehkopf, Max. “User Stories: Examples and Template.” *Atlassian Agile Coach*, 2020, www.atlassian.com/agile/project-management/user-stories.

[2] Weiruch, Robin. “A Firebase in React Tutorial for Beginners [2019].” *RWieruch*, 26 Nov. 2018, www.robinwieruch.de/complete-firebase-authentication-react-tutorial.

[3] Weiruch, Robin. “React Firebase Authorization with Roles.” *RWieruch*, 26 Nov. 2018, https://www.robinwieruch.de/react-firebase-authorization-roles-permissions.

[4] Weiruch, Robin. “React Firebase Auth Persistence with Local Storage.” *RWieruch*, 27 Nov. 2018, https://www.robinwieruch.de/react-firebase-auth-persistence.

[5] Weiruch, Robin. “Social Logins in Firebase React: Google, Facebook, Twitter.” *RWieruch*, 02 Dec. 2018, https://www.robinwieruch.de/react-firebase-social-login.

[6] Weiruch, Robin. “How to link Social Logins with Firebase in React.” *RWieruch*, 06 Dec. 2018, https://www.robinwieruch.de/react-firebase-link-social-logins.

[7] Weiruch, Robin. “Email Verification with Firebase in React.” *RWieruch*, 20 Dec. 2018, https://www.robinwieruch.de/react-firebase-email-verification.

[8] Weiruch, Robin. “How to use React Router with Firebase.” *RWieruch*, 01 Jan. 2019, https://www.robinwieruch.de/react-firebase-router.

[9] Weiruch, Robin. “How to use Firebase Realtime Database in React.” *RWieruch*, 16 Jan. 2019, https://www.robinwieruch.de/react-firebase-realtime-database.

[10] Weiruch, Robin. “Accept Stripe Payments with React and Express.” *RWieruch*, 20 Jan. 2017, https://www.robinwieruch.de/react-express-stripe-payment.

[11] CodAffection. “Complete React Material UI Tutorial Introduction.” *Youtube*, commentary by Shamseer, 16 Nov. 2020, https://youtu.be/m-2_gb_3L7Q.