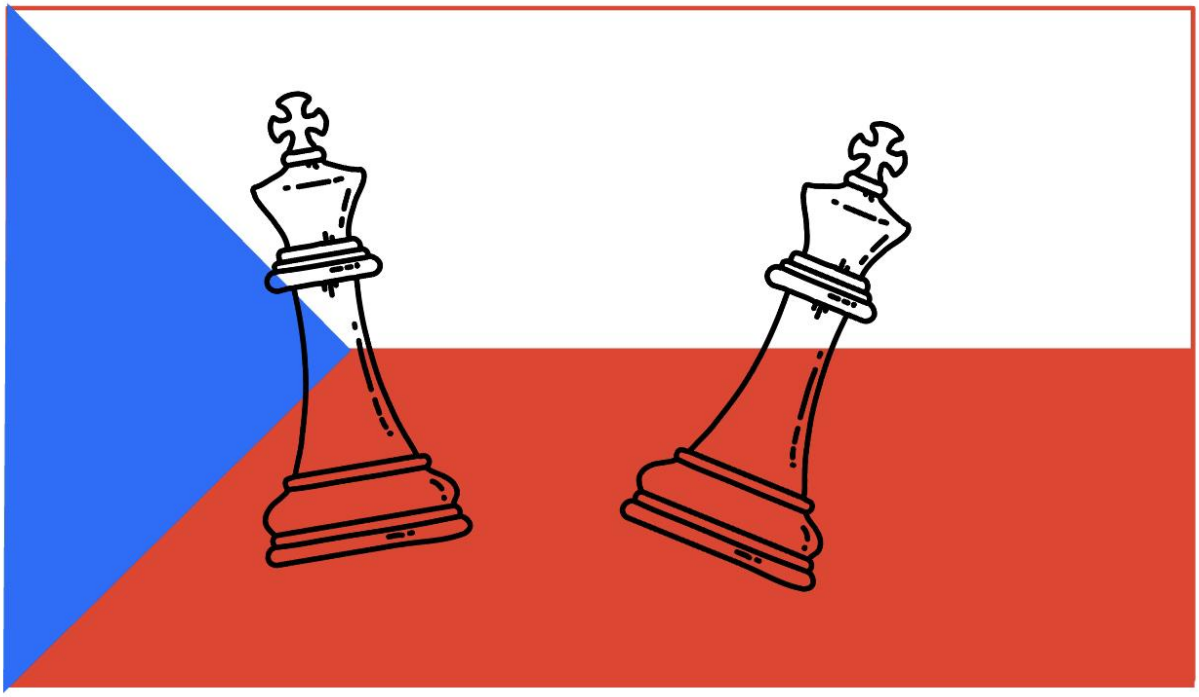


Nudge RPG Website
Technical Specifications

Cam Chrissis
Zara Masino
Jason Demers

Tech spec touchups

1. Go through the diagrams and update the colors for any components that have changed during the last weeks of development in terms of database reading and writing.
2. One more read through for spelling errors, formatting, and correctness
3. Hyperlinking

Introduction

Dungeons and Dragons, more commonly referred to as D&D, is a fantasy based tabletop RPG where players create characters who go through adventures and combat according to the ruleset. It originates from tactical war games but differs with character creation and the addition of more fantasy elements. While D&D has more focus on combat and rules, some other games have taken a different approach and leaned into the creative and narrative side of RPGs. These games involve the wildly popular game Fate, as well as the up and coming game NUDGE, which has been in the works for 5 years by Christopher Stuetzle, Associate Professor and Chair and Department of Computer and Data Sciences at Merrimack College.

Professor Stuetzle wants a website that will be used alongside NUDGE, providing an easy way for users to create and reference their characters and also coordinate campaigns with their friends. The website will need a strong focus on sustainability and flexibility so that the website can be built on and changed as the game evolves over time. Our group Czech Mates, consisting of Cam Chrissis, Jason Demers, and Zara Masino, has been tasked with designing and building this website for Professor Stuetzle.

Executive Summary

This website will allow users to log into their accounts through Google. There are two types of accounts: user and administrator.The user will have the ability to search for and follow other players, with mutual followers being friends. A user will be able to create characters through filling out character sheets. These character sheets are saved and kept for later reference. They will be able to edit these characters afterwards, delete them, or copy them. Users will also interact with worlds. They can create and invite friends to their own world, as well as send scheduling reminders. They can view the information of characters in the campaign as well as remove them as long as they are acting as the GM in that world. Admin will have additional abilities with an administrator page. They will be able to view and manage the database of users through this page.

External Libraries

Name	Description
bootstrap	**Website styling**
csvtojson	**Converting CSV files to JSON objects**
Firebase	**Database and authentication**

react	**General framework**
react-bootstrap-typeahead	**Typeaheads with specific styling**
react-dom	**DOM and server rendering**
react-router-dom	**Page routing**
react-scripts	**Creation of a react app**
react-to-pdf	**Save react element as PDF**
@emailjs/browser	**Send automated emails**

To install these dependencies you require npm and node.js. Navigate to the [...Czechmates/czechmates] file in the terminal and run [npm install].

API Calls

Google Authentication Through Firebase

In the login page of the website, located at …Czechmates/czechmates/pages/index.js, we use the Google Authentication that is provided as a service through Firebase. We use the popup version and we retain the user's display name, email, and uid.

EmailJS

The manage and add world popups of the website, located at …Czechmates/czechmates/components/ManageWorldPopup/index.js and …Czechmates/czechmates/components/AddWorldPopup/index.js respectively, use EmailJs in order to send emails to potential members of a world. We use the free version of the email template and email service provided by EmailJs.

Design Choices

Firebase

Firebase is a collection of cloud based development tools for building, deploying, and scaling apps. In this website, we use two of the tools offered by Firebase: Realtime Database and Authentication. We chose to use Firebase Realtime Database because it is structured as a JSON tree, which is compatible with the frequency in which we wanted to use JSON objects in

our project. We also chose it for its real time update ability, which meant that the user's information would always be up to date. We chose to use Firebase Authentication because it was easy to integrate with our database, and most of the setup was identical to the database setup.

React

React is a library that allows you to build a component based user interface with dynamic updates. We decided to use React so that we could take advantage of the prebuilt UI, as well as to leverage the component based setup in order to organize and modularize our code. It also paired well with Firebase Realtime Database because of the real time nature of Firebase and the dynamic updates provided by React.

EmailJS

EmailJS is a service that allows you to send emails on the client side. We needed to send automatic invite emails, and we wanted to incorporate it in the frontend because of how we have worked with React. By using EmailJS, we can directly trigger the email and pass it the appropriate content right in the component that it is relevant to. There is a limit of 200 free emails per month.

In order to use the email functionality, you must:

```
const actualParams = {
                  member: value, //username of recipient
                  code: worldCode, //the code of the world to be joined
                  sender: senderName, //username of the World Owner
                  memberEmail: memberEmail //the email of the recipient
                };
emailjs.send('service_5lol5zu', 'template_fumphwg',
                actualParams, 'nCCc6oQB6cd4n0ljl')
                    .then((result) => {
                        alert('email sent successfully');
                    }
```

Printing Popup

When the user wants to print their character sheet as a PDF, we provide them with a preview of the character sheet before it is actually printed where they have to confirm that they really wish to print. This is spurred by two major factors. The printing library that we use, react-to-pdf, takes a react component and creates a PDF from it. This was ideal for us since we are using react and the character sheet is a component. The caveat is that the component has to be rendered before it can be printed, and we were printing from an alternative page. This popup was a workaround so that the component could be rendered before the print function was

called. The second reason was that since the PDF is a translation of the component, it is helpful to the user to see what the PDF will look like before they print it, in case they want to change the sizing.

Friend Functionality

In this website, users do not send friend requests in order to become friends. Instead, users are automatically made into friends when they mutually follow each other. This was at the recommendation of the client so that we would not have to manage a friend request system. It also made it easier to handle the interactions between users since we have total control over the friending actions.

The Full Character Sheet

In the sub character page of this website for a given character, there is a tab that displays all of the other information in the subcharacter sheet condensed into one component that is viewed only. This was done at the request of the client so that users could use it as a condensed reference of their entire character while playing. This view is also available in the world information so that other players can use it as reference as well.

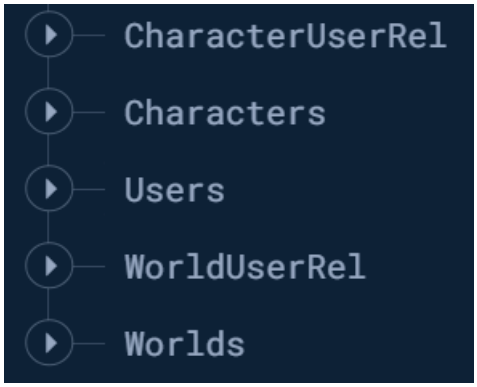
Developer Resources

In order to interact with the database and authentication for this project, go to this link. You must be signed in with the NUDGE admin account or you will be unable to access it. You are able to delete, add, and modify the information in the database.

In order to interact with emailJS, go to this link. You must be signed in with the NUDGE admin account to access and edit this. Here you can change the template for the world invite emails, or change other settings related to the account.

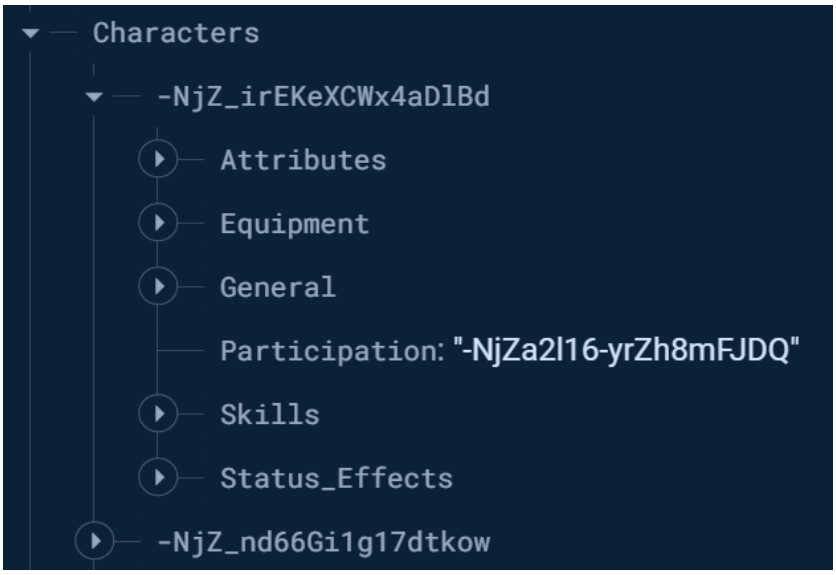
Anyone can clone the project from our gitHub repository.

Database



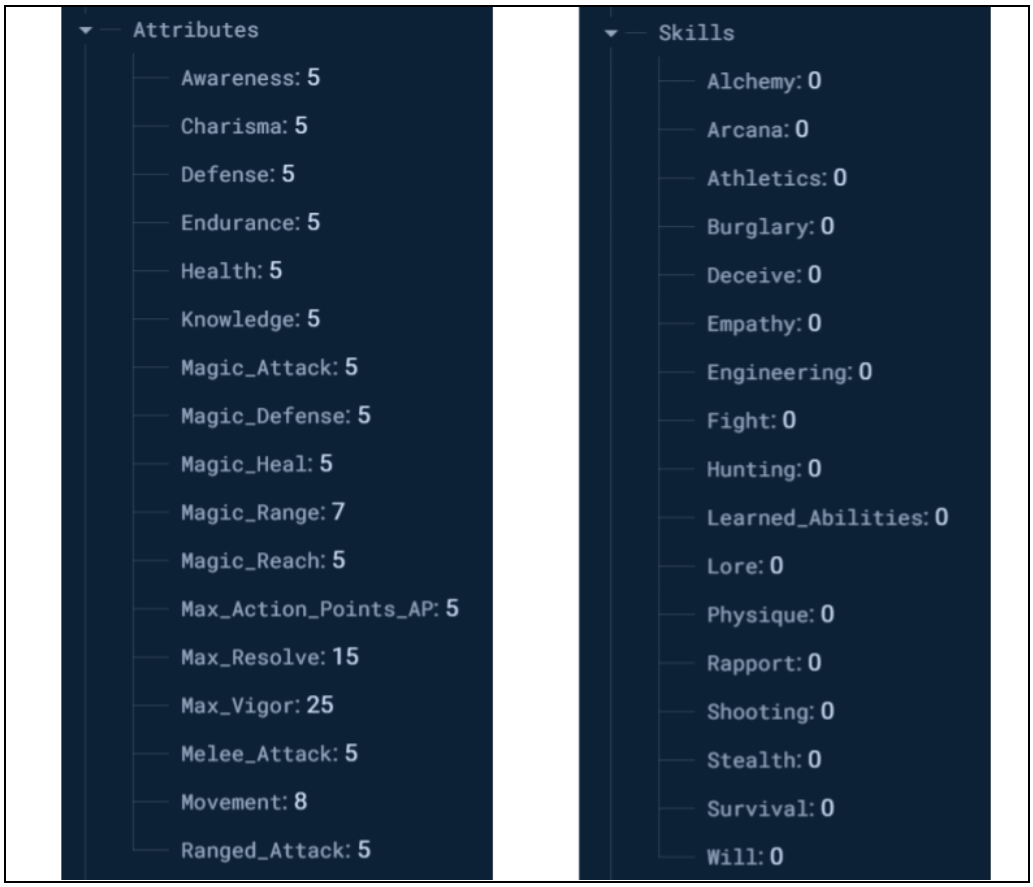
Our database contains five collections stored in Firebase's realtime database. Character and User relationships, Characters, Users, World and User relationships, and Worlds.

Characters:



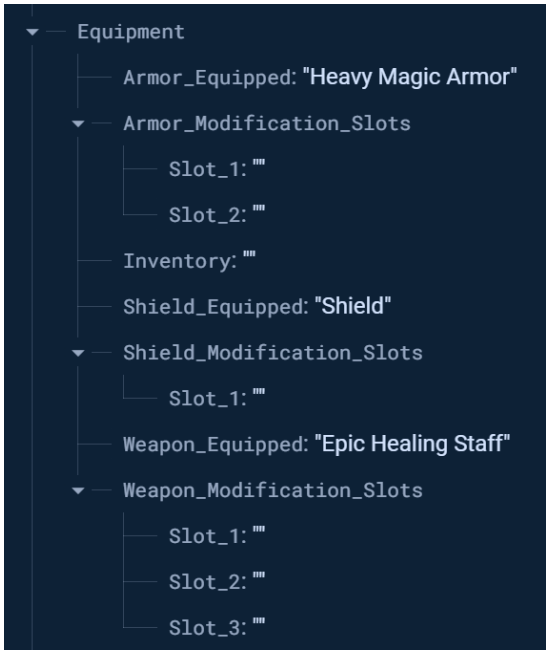
The character section stores the information about every character in the database. Each one is linked to a unique key and has u[to 6 subsections: Attributes, Equipment, General, Participation, Skills, and Status Effects. Participation is optional and is only present if the character is a member of a world.

Attributes and Skills:



The skills and attributes are two subsections under the character information in the database. These values are stored as integers except for the learned abilities, which is a string With the exception of a few, attributes default to 5. Attributes are calculated using formulas which use the skills, level, and attributes of the character. The attributes that default to something other than 0 either use another attribute or the character level in their calculations, while the defaults of 5 just use skills. The skills default to 0 until changed by the user manually in the site.

Equipment:

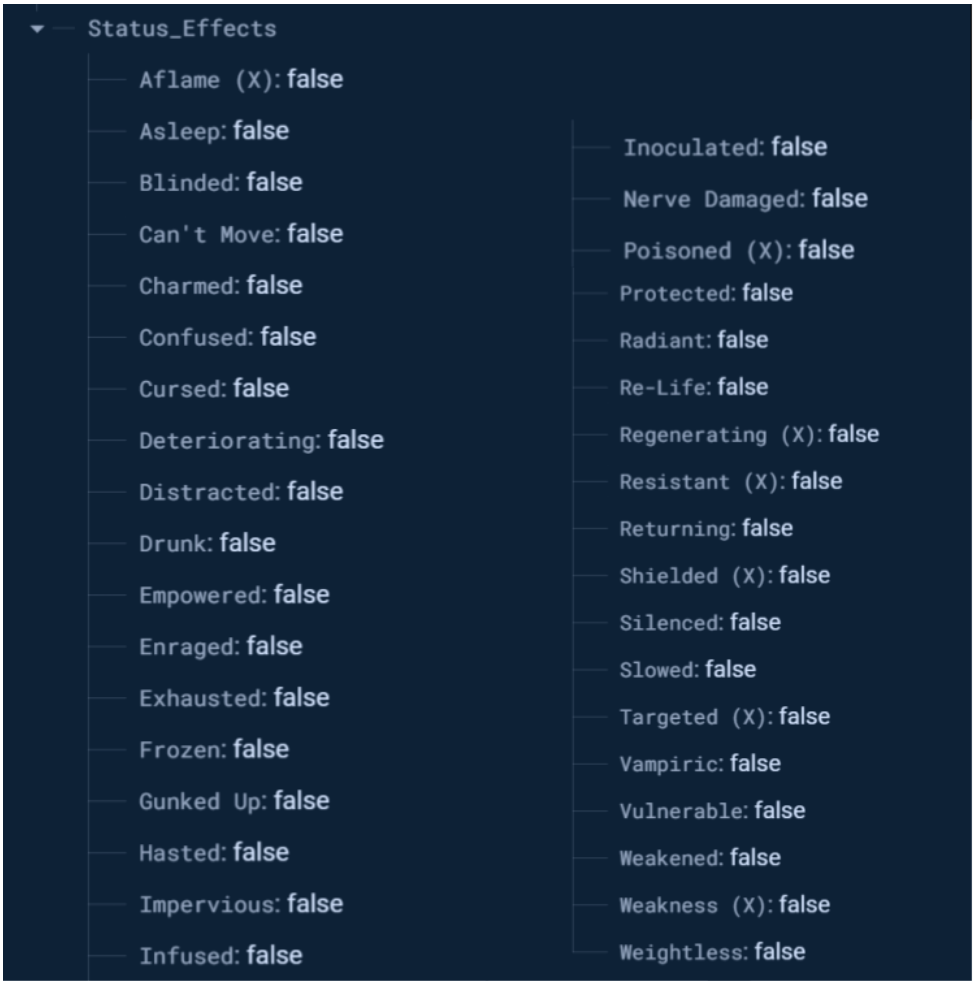


The equipment section stores everything that a specific character currently has equipped. Also stored is the character's inventory, and modification slots for their armor, shield, and weapon.

General

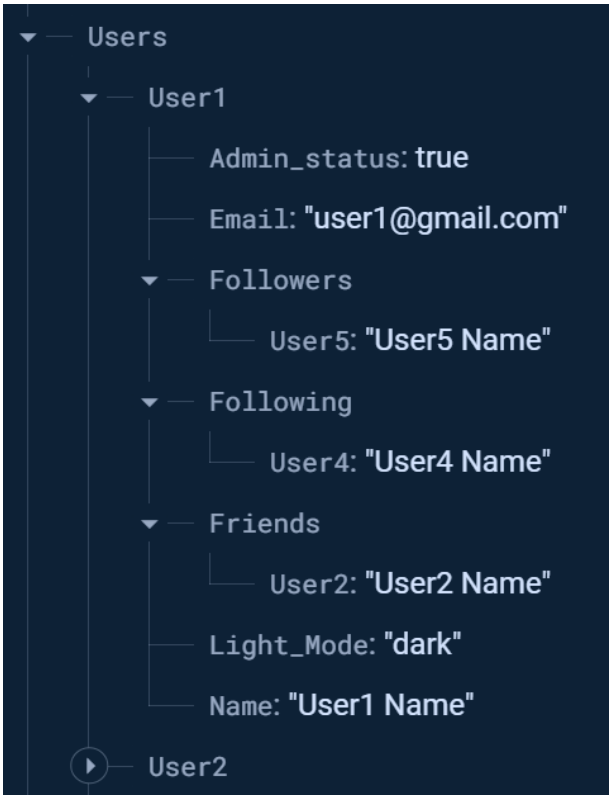
The general section stores the general information for a character. Each character in the database has a general section. All of these values are stored as strings except for the level, which is stored as an integer.

Status Effects



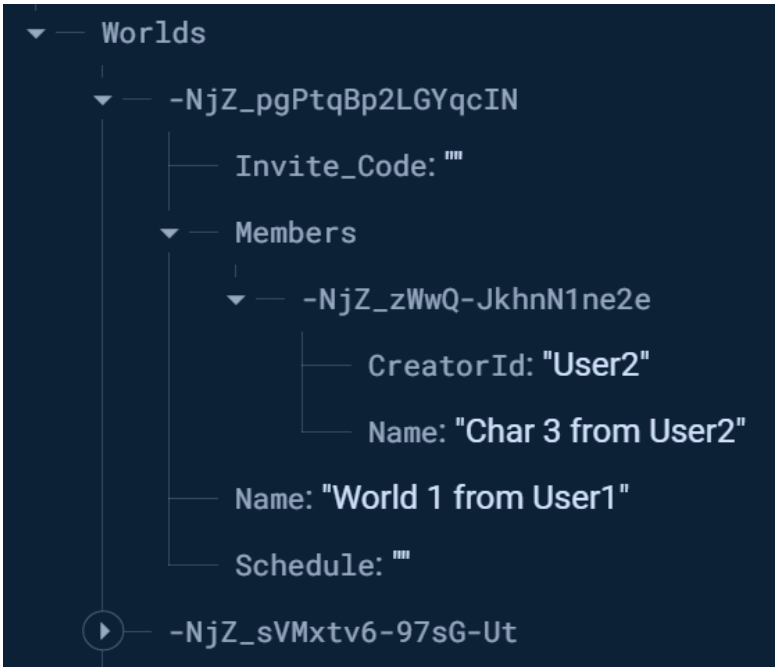
The Status effects of a character are stored as booleans. The Boolean represents if the player is currently afflicted with an effect. These may cause printing to fail on a Firefox browser due to the way they are rendered in the sheet view.

Users



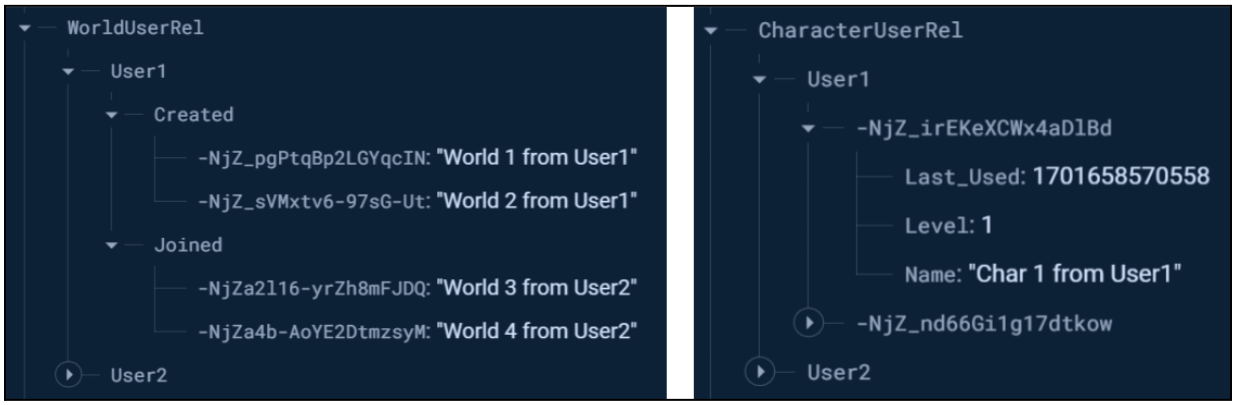
The Users section contains all the information that is directly related to the user and their account, including their name, email, admin status, and their social relations with other users. Each user has a friends, following, and followers list, which stores the ids and names of the users in each one. This is used to update user information when a change is made in one user that requires changes to be made in another user to accurately reflect the current information in the database, such as a user changing their name.

Worlds



The Worlds section stores all of the important information for displaying and editing worlds. All of the words are stored in one area in the database so that they can be easily referenced with just their key. Members stores all of the members of a world as objects, tracking the characterId as the object key, and the creatorId and character name in the value. This is used to display the members and view character information from the world popups.

Relations (World and Character)



Because of the nature of Firebase Realtime Database, we decided to denormalize our data with the help of these two relationship sections.

The World User Relationship **(pictured left)** records the relationships between a user and the worlds they create as well as the worlds that they have joined. We most frequently need

to reference worlds in the context of a known user, so we have them nested underneath the userId. Each user has two subsections:

1. Created worlds
2. Joined worlds

Creates stores the ids and names for all of the worlds that the user has created, while Joined stores the ids and names for all of the worlds that the user has joined. This section is most frequently used to display all of a user's worlds on their world page, but it is also used in order to update information in the appropriate worlds when a user's user information or character information is relevant to that world. The worlds are split into created and joined both for distinguishing actions that the user can make on them depending on ownership, and also when the user sorts their worlds in the world page.

The Character User Relationship **(pictured right)** records the relationships between a user and their characters. Similar to the World User Relationship, this section of the database is most frequently used to display a user's characters on their character page. Each user has their characters stored as objects, with the characterId used as the key. We store the name, level, and the epoch date of the last time that character was used. These are used for when the user wants to sort their characters by these elements, and the name is also used for display.

Database Actions

This section dives into the code that actually reads and writes to the database. It will explain 3 examples from the code, one each for onValue, get, and update. This will allow future developers to understand the choices made on why we use these functions where we do. For additional information on firebase realtime database functions, click here.

Update

```
158     // create the database reference
159     const userRef = ref(db);
160     // create an object to store the paths and the values to update in those paths
161     const updates = {};
162     // create a path through the database using the userID and socialID
163     // then set the content to the new value (new info or null to remove)
164     updates[`Users/${action.userId}/Followers/${action.socialId}`] = action.content;
165     updates[`Users/${action.socialId}/Following/${action.userId}`] = action.userName;
166     updates[`Users/${action.userId}/Friends/${action.socialId}`] = null;
167     updates[`Users/${action.socialId}/Friends/${action.userId}`] = null;
168     // send the updates to the database
169     update(userRef, updates);
```

Update is the firebase method that allows you to change, add, and remove entries in the database. First you get a reference to the database (line 159). The database will have been imported from the file that stores your database credentials. Then you create a new object, in this case called updates, in order to make multiple changes at once and therefore reduce the burden on your database. In this specific example, the user is removing a friend, so we must update 4 places in the database:

1. The current user's follower list (line 164)
2. The former friend's following list (line 165)
3. The current users friend list (line 166)
4. The former friend's friend list (line 167)

We are removing them from each other's friends list and adding them to their following or followers list instead to represent that the following is no longer mutual. We do this by creating a new property for the updates object. The key represents the path through the database to the place that we would like to change, and the value is the information that we want to update it with. Setting it to null will remove it from the database. The final step is to run the update function by passing in the reference to the database as well as the updates object (line 169).

Get

```
64      // get the member information using the worldID to create the path
65      get(child(worldRef, `Worlds/${action.worldId}/Members`)).then((snapshot) => {
66        // if the world has members
67        if (snapshot.exists()) {
68          // loop through the members and set their participation to null
69          for (const [key, value] of Object.entries(snapshot.val())) {
70            updates[`Characters/${key}/Participation`] = null;
71            updates[`WorldUserRel/${value.CreatorId}/Joined/${action.worldId}`] = null;
72          }
```

Get is the firebase method used to read from the database, returning a "snapshot" that has the current value in the database at the point in time when the call is made. Similar to Update, you use a reference to the database. For get, you also provide the path for where you would like to read from (line 65). A good practice is to check if the call returned any information. You can use snapshot.exists(), which returns true if the value at the queries location is not null, meaning that it has some information for you to manipulate (line 67). When the snapshot is returned, you can access the value with snapshot.val(), which is in object form. In this example, we loop through snapshot.val() and update the characters and user world relations appropriately when a world is deleted (lines 69-72). If you wanted to use the key of the snapshot, then you would use snapshot.key, and there is a plethora of other information stored in the snapshot, we just don't use much else of it in this project.

onValue

```
47      // check that the worldID is not undefined
48      if (worldId !== undefined) {
49        // use the database and the worldID to create the reference path
50        const worldRef = ref(db, 'Worlds/' + worldId);
51        // call onValue to load all of the world information
52        onValue(worldRef, (snapshot) => {
53          // set the result to the useState variable worldInfo
54          setWorldInfo(snapshot.val());
55        });
56
```

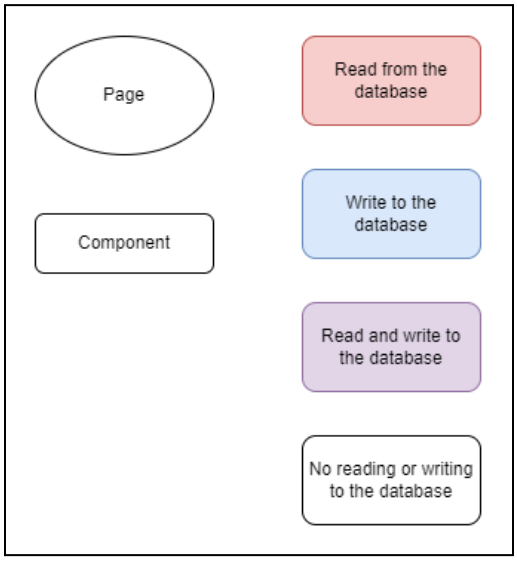
onValue is another firebase function that reads from the database, similar to get, but it also attaches a listener to the location that it reads from. This means that whenever that value changes in the database, the information in your site will also be updated. Here is another way

to make a reference, where we again pass in the database but also the specific path that we wish to use (line 50). You call onValue with this reference and you are returned a snapshot, similar to get (line 52). In this example, we are just setting a useState to the result of the snapshot (line 54). This is a standard practice that we employ throughout the code, which leverages the auto updating nature of onValue in coordination with the rerendering ability of React when a useState is changed. These updating styles in combination allow our website to update with the most recent information automatically.

Information Flow

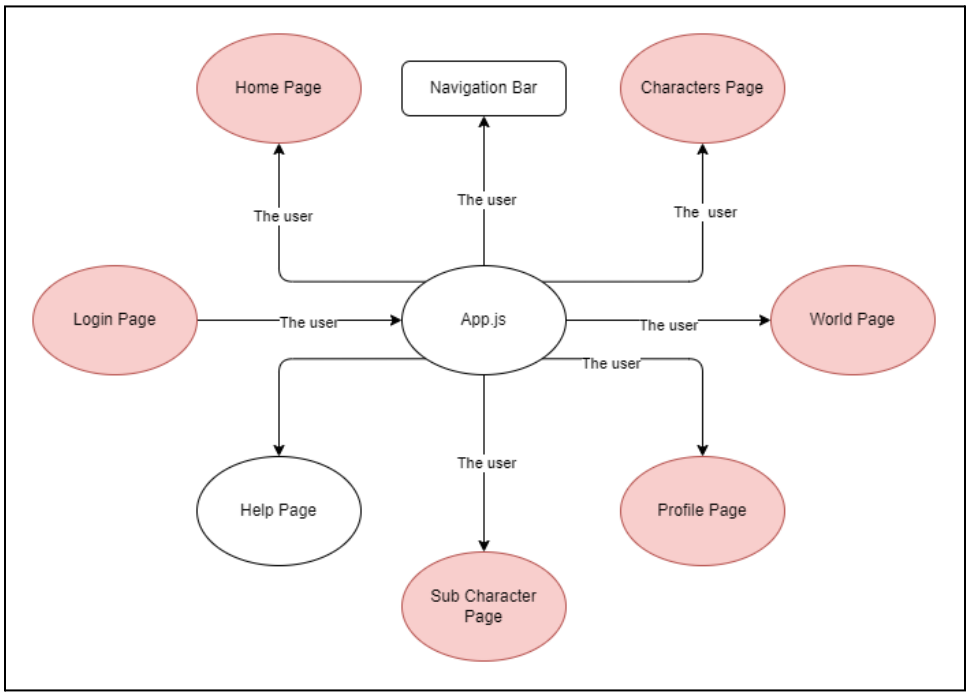
This section details the component and page structure of the website, as well as where the database is incorporated throughout. It does not detail every single piece of information that is passed but it approximates it so you can get a general understanding of how the components are connected, what information each component is given, and where the database actions are taken.

Key



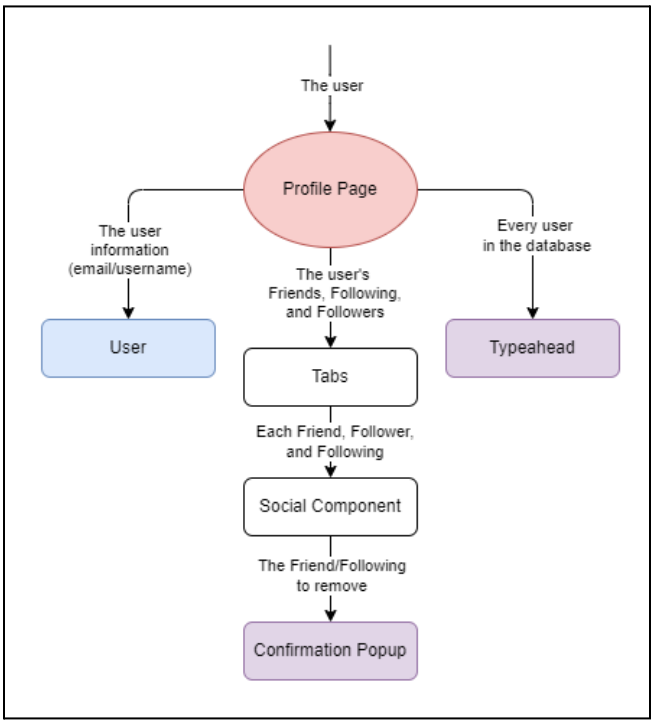
This key is to help with understanding the diagrams below. The left side is a key for the shapes and the right is the key for the colors.

Page Connections



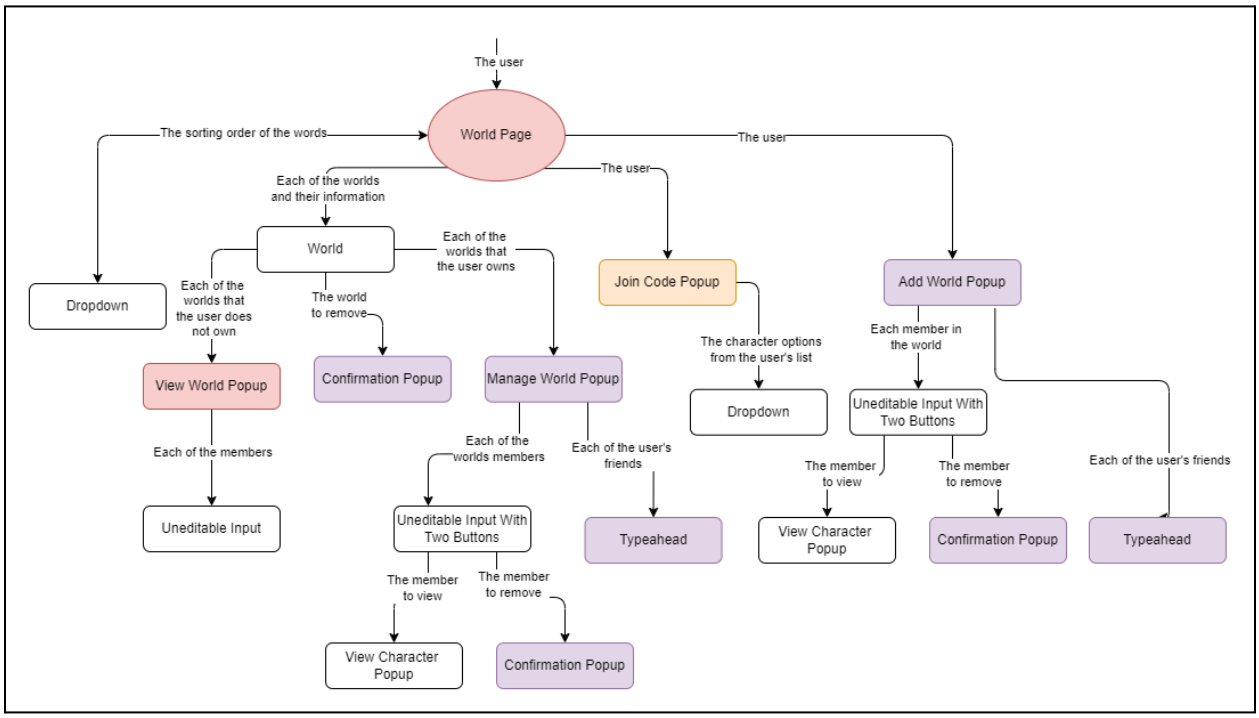
This diagram shows the information that is passed between the pages. All of these pages are initially created in the App.js file. Note that the login page sends the username to App.js. This is because there is no user until someone has logged in. Once the login has been successfully completed, the App.js file creates the other pages and the navigation bar, which all read information from the database.

Profile Page



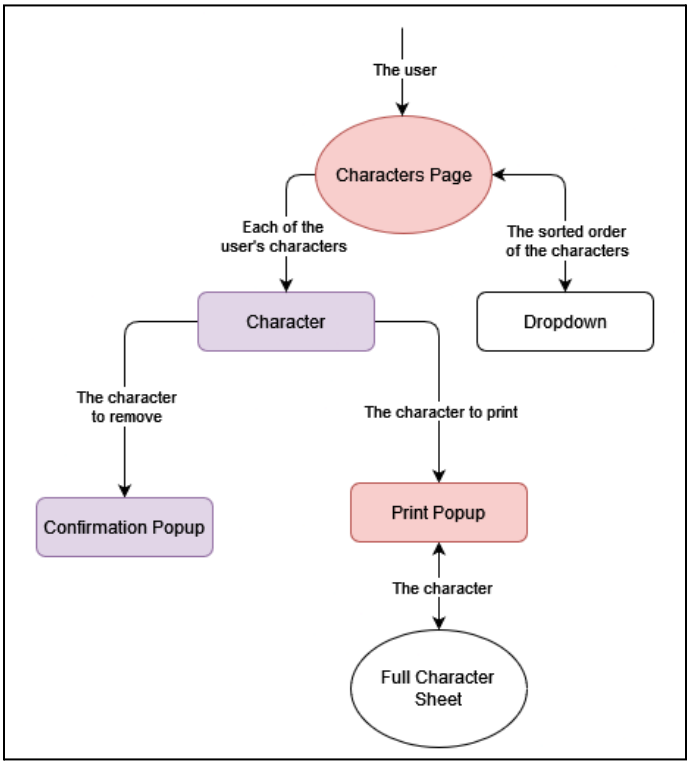
The profile page is given the user information, and it gets the user's information from the database. The User components make up the name and email of the user. The User component for the name writes to the database when the user changes their username. The typeahead, which allows the user to select another user in the database and follow them, has to read and write to the database in order to determine if the person that the user follows should be moved to the friend list, as well as to actually add the new friend or follower to the current user's appropriate list.

World Page



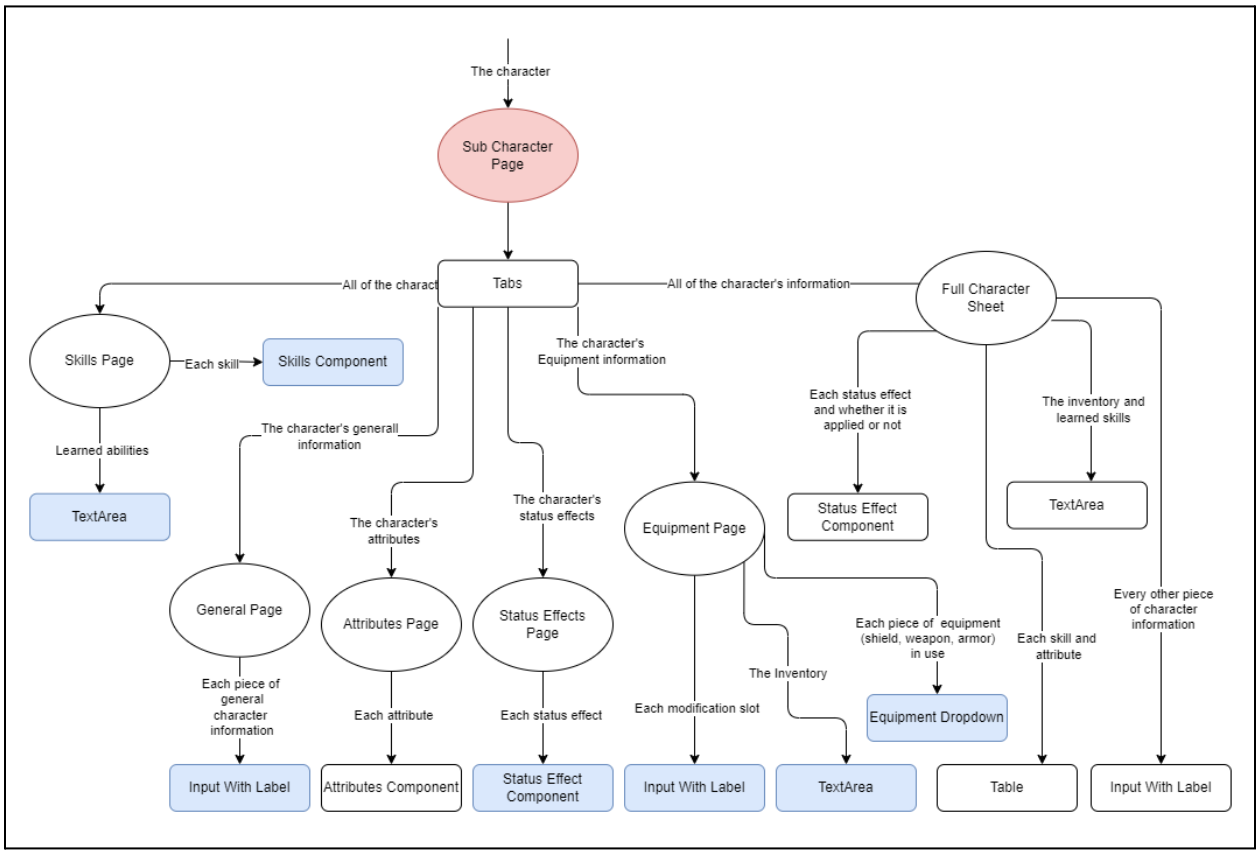
The world page initially gets the current user and then reads in the world information from the database. The pop ups do most of the reading and writing in this page. The view popup only reads the information from the database because the user cannot edit the information in someone else's world. The confirmation popups deal with removals of worlds and members of worlds, as well as when a user leaves another world. All of them require reading and writing in order to update different sections of the database whenever an action is taken. The typeahead functions similarly to the typeahead in the profile page, but it only displays the user's friends.

Character Page



The character page is given the user and then it reads all of the user's character information from the database in order to display them. The print popup reads the character information for a specific character and then passes it into the full character sheet.

Sub Character Page



The subcharacter page is used to display the information for a particular character, so it reads in the information for a character when it is loaded up. It gets all of the information and passes it into the other components separately depending on what they require (i.e. the equipment page only gets the equipment information). Because of the use of onValue to do this initial reading, and the fact that the character information does not influence anything other than the character, this section is predominantly either free of database actions or has write only. The attributes and full sheet pages are read only so they do not have any write actions. The smaller components are the most likely to contain write functionality, such as the inputs with labels and the text areas. When these components are edited, they update the database in the appropriate sections.

Glossary

- **GM** - Game Master. The GM controls all aspects of the game besides a player character's chosen actions.
- **Campaign** - A series of individual experiences that take place within a role playing game.
- **World** - A setting in which a campaign for a given role playing game takes place.
- **RPG** - A type of game in which players assume the roles of characters in a fictional world.
- **Party** - A group of users sharing an RPG world. Players must be following each other in order to join a party.
- **Friends** - Players who mutually follow each other on the platform

.

Stuff for us to use while we create this document

- The technical specification's role is to be a document you can hand to future developers so that they can read the document and be familiar enough to
- Goals of the tech. Spec.
    - A technical reference for those working on the current project
    - An internal check to see if the requirements specification is met
    - A guide for future groups who are performing maintenance or adding features
- You should include the following sections:
    - General overview/executive summary
    - Libraries and other external entities necessary to be installed/built to work on the project, including where to find them
        - **Got a few of them below, not all of them tho**
    - State diagram/flowchart of program flow
        - Modular break down of each state in the diagram, including any external library or API calls required
            - **Not sure what we should do for this, just grab from functional spec, touch it up and add the API calls maybe?**
        - Description of any algorithms used
            - **None?**
        - Explanation and justification of any design choices regarding the module
            - IDK man so many things, what does it even mean by module, maybe just treat this as a like, major design choice section
            - Firebase - JSON objects, realtime abilities
            - We chose react - ease of UI, modular setup
            - We have a popup for the printing - lets us use the print library
            - We use EmailJS - doesn't require server to send emails
            - Login page instead of home first - no info to display if you aren't logged in
    - Any hardware or non-functional requirements (such as databases)
        - Where these objects are utilized in the project
            - Everywhere
        - How they are interacted with from inside the code
            - Read, Write, Both
        - This might/should include information flow diagram
            - Yayyy diagram time
        - **I think that I covered these in the information flow diagram and the firebase function section**
    - Any other material you believe another group would benefit from having
    - Glossary

I liked the explanation of the directories that would be cool to have
Also startup instructions would be good too

Referencing files would be good when explaining things so people can go to them to look directly

External Libraries

I grabbed these right from our package-lock.json file. I don't think we technically need all of them or even what some of them are but for any I know (or am pretty sure) we use I grabbed the links

(I only included the green ones for now, we can add the others later).

Libraries:

- node.js https://nodejs.org/en
- @testing-library/jest-dom
- @testing-library/react
- @testing-library/user-event
- Bootstrap https://www.npmjs.com/package/bootstrap
- csvtojson https://www.npmjs.com/package/csvtojson?activeTab=readme
- Firebase https://firebase.google.com/docs/web/setup
- React https://www.npmjs.com/package/react
- React-bootstrap-typeahead https://www.npmjs.com/package/react-bootstrap-typeahead
- React-bootstrap-validation
- React-dom https://www.npmjs.com/package/react-dom
- React-icons
- React-router-dom https://www.npmjs.com/package/react-router-dom
- React-scripts https://www.npmjs.com/package/react-scripts
- react-to-pdf https://www.npmjs.com/package/react-to-pdf
- Styled-components
- Web-vitals
- @emailjs/browser