Jaffa Schemes

Mind Flayer's Atheneum – Dungeons and Dragons Storage Website

Samuel Anderson, Jeffrey Bringolf, Chase Lipari

Part A

Introduction - Jeffrey

Sam and Jeff both enjoy using character sheets on their computers when they are playing dungeons and dragons but hate having to use a pdf to keep track of their stats and health. An intuitive website would be nice since it would solve the problem of clunky pdf character sheets. Jeffrey specifically also really enjoys introducing the game to friends but found that making a character is often intimidating. A set of dropdown menus and fields to fill in would help fresh players feel less overwhelmed. Chase wanted to make an interactive, intuitive website and was excited by the idea of learning Dungeons and Dragons. We put our ideas and desires together and decided to make a custom online Dungeons and Dragons character sheet.

Since our primary desire is to replace clunky real-life or pdf character sheets with a more enjoyable experience, our goal for our interface is to prioritize intuitiveness and ease of use. It should be very intuitive for a user since one of the most annoying things in Dungeons and Dragons is when a person is wasting time fiddling around with sheets and looking up rules. A more streamlined character creation process would also be much easier for inexperienced players, since making a character is one of the more intimidating parts of the game.

We hope to have an intuitive website where you can login and see all your characters and custom spells Not only should it be a storage site, but it should also provide simple methods of performing common in game tasks such as, taking damage, receiving an item, adding spells to a spell book, short / long resting, etc.... Being able to interact with other users' characters, items and spells would be nice but is not a requirement for a successful website.

Some discussions arose about whether we should fill the database with default values before any user even logs in. It may not make sense for some tables, like the character or user table, but for others like the spells, race, class, etc... it makes sense to already have the Dungeons and Dragons values for all those tables already in the database for users to pick from. We decided that our database will be populated with data before any user interacts with it so that it is not entirely player-made content. Users will, however, be able to add their own custom spells, as well as obviously create their own characters.

Introduction authored by Jeffrey

Business Logic Requirements - Chase

Key Required Operations

In our application, we will have a way in which users may login and signup to their respective accounts. There will be a home screen for the user to browse and the user will be able to see all the spells, races, and classes from the player's handbook without logging in. Once the user is logged in, they will be able to view personal characters and spells which they can create themselves. The characters will have specific details that the user will be able to see as well as forms to edit each of the following: basic attributes, Hp, skills, saving throws, armor class, ability scores, owned items. The user will be able to any of the following dice (1d2, 1d6, 1d8, 1d10, 1d12, 1d20) with an area to input any bonuses to their roll. A corresponding details page must exist for each of the following elements: class, race, characters, and spells. In addition to a details page for a selected item, there will be pages to view a complete list of existing classes, races, characters, and spells.

Bullet point version:

- Home Screen
- Login / Signup
- Even when not logged in, can see all spells from player's handbook.
- Even when not logged in, can see all from races from the player's handbook.
- Even when not logged in, can see all from classes from the player's handbook.
- Once logged in, can view "My Characters"
- Once Logged in, can make a character.
- Once logged in, can add a homebrewed spell
- Once logged in, you can see "My spells"
- Focused Character must have all the character sheet details, as well as buttons to change specific things:
    - Shows all basic attributes from the character sheet and can all be changed
    - Button for HP change
    - Dice Rolls
    - Modifier beside dice roll
    - Skill Checks
    - Saving Throws
- A focused page must exist for each of the following:
    - Class
    - Race
    - Character
    - Spell
- A list page must exist for each of the following:
    - Classes
    - Races
    - Characters
    - Spells

Key required operations section authored by Chase – Edited by Jeffrey

Data Needs and Data Dependencies - Jeffrey

The following section will list the validity constraints of our database. It will also outline some consistent naming conventions that will be used when inserting into or updating the database.

Database naming consistency

We decided that any text values that will be compared, will be stored in lowercase. Things like names or descriptions do not need to be lowercase since they will never need to be compared. Things like spell names which will be compared to one another, must be stored in lowercase. We will determine in a case by case basic whether these fields will be capitalized when displayed to the user. Also, fields that will be similar amongst all records, like Spell target should be lowercased for consistency.

Below are the list of tables and their text fields that will be stored in lowercase in the database.

- Spell
  - Name
  - CastingTime
  - Range
  - Target
  - Duration
- Spell School
  - Name
- Class
  - Name
- Class Feature
  - Name
- Background
  - Name
- Background Feature
  - Name
- Owned Item
  - Name
- Race
  - Name
- Ethics
  - Name
- Morality
  - Name
- Skill
  - Name
- Ability
  - Name
- Racial Trait
  - Name

Data Needs

Our data serves the main purpose of allowing the user to create and store several dungeons and dragons characters. To achieve a certain amount of "automation" within the character sheet, we must make several database design decisions. We could easily have a column for each skill, saving throw, and ability score a character has but this would be inefficient for storage and would not be conducive to automation. We made several choices like having the Skill and Ability tables be their own tables rather than columns in a character which will allow us to calculate their values based on a character's proficiencies and bonuses. Storing these values in the character table itself would force the values to be more static and manually input by the user.

Our database will be quite extensive due to the sheer quantity of information required for a character sheet. We were mildly concerned about the storage requirements as the number of characters increases so we made some decisions to prioritise memory. For example, the separation of Spell School from the Spell table allows us to save storage space. Rather than each spell containing a text value, it contains only an integer. This is important since our data serves the purpose of populating characters and character sheets, but also serves the purpose of allowing plenty of users to make a vast number of characters.

Data Dependencies

Our database design is filled with data dependencies. Due to this structure, several tables need to be created and populated before others. This section will outline the order in which tables should be created to prevent any dependency issues. The tables will be grouped together, with all the tables within each group being interchangeable in the order they're created and populated. In other words, no two tables within a group are required to be created in any particular order. For each table, the table that it is dependent on will be indicated after it in italics. For example, table Foo relies on table Bar: Foo *(bar)*

Group 1

- Spell School
- Background
- Class
- User
- Race
- Ethics
- Morality
- Ability

Group 2

- Spell *(spell school)*
- Class Feature *(class)*
- Background Feature *(background)*
- Character *(background, class, user, race, ethics, morality)*
- Racial Trait *(race)*
- Skill *(ability)*

Group 3

- Class Permitted Spell *(class, spell)*

- Known Spell *(character, spell)*
- Saving Throw Proficiency *(character, ability)*
- Skill Proficiency *(character, skill)*
- Skill Expertise *(character, skill)*
- Owned Item *(character)*
- Saving Throw Bonus *(character, ability)*
- Ability Score *(character, ability)*

Validity Constraints

Each of the following lists will cover a separate table in the database and will describe what constitutes valid or invalid data for that specific table. See the Entity Relationship diagram later in this document for the full database design.

The Character table has the following validity constraints:

- The id will be set to one greater than the current largest id in the table.
- The user id must be an integer and will be set upon creating the character and will not be changed after being set. It must reference an existing id in the user table.
- The Race id must be an integer and must reference an existing id in the Race table.
- The Class id must be an integer and must reference an existing id in the class table.
- The Ethics id must be an integer and must reference an existing id in the Ethics table.
- The Morality id must be an integer and must reference an existing id in the Morality table.
- The Background id must be an integer and must reference an existing id in the Background table.
- Name must be a non-empty string.
- MaxHP must be an integer greater than 0.
- Current HP must be an integer with no constraints. A character can go below 0 hit points when damaged and can exceed their max hit points with temporary hit points.
- Level must be an integer between 1 and 20, inclusive.
- Armor Class must be an integer. No constraint on the actual value since we are allowing homebrewed spells which could potentially allow weird things to happen with Armor Class.
- Speed must be a non-negative integer. Negative speed does not make sense even with the potential for weird spell effects.
- Initiative must be an integer. No restraint on the value for the same reason as Armor Class
- Experience must be a positive integer.
- If a user is deleted, all the characters with the user id of the user deleted must also be removed from the database.

The Spell table has the following validity constraints:

- The id will be set to one greater than the current largest id in the table.

- The school id must be an integer reference an existing id in the spell school table.
- The user id references an existing id in the User table, if it is a player's handbook spell it references id 0 which will be the default user.
- The spell level must be an integer between 0 and 9, inclusive.
- The description must be a string and cannot be empty.
- The spell name must be a string, cannot be empty and cannot contain numbers. By convention, Dungeons and Dragons spells do not contain numbers.
- Casting time must be a string and cannot be empty.
- Range must be a string and cannot be empty.
- Target must be a string and cannot be empty.
- Verbal must be a Boolean value, indicating whether a spell requires verbal components.
- Somatic must be a Boolean value, indicating whether a spell requires somatic components.
- Material must be a Boolean value, indicating whether a spell requires material components.
- Materials will be null if Material is false, indicating no materials are required for the spell.
- Materials must be a string and cannot be empty if Material is true.
- Duration must be a string and cannot be empty.
- Damage must be a non-empty string, or it can be null if the spell has some other effect than damage. This way, the user will never be shown an empty string for damage.
- If a user is deleted, all the spells in this table with the user id of the deleted user must also be removed from the database.

The Spell School table has the following validity constraints:

- The Id will be set to one greater than the current largest id in the table.
- The Name must be a non-empty string.

The User table has the following validity constraints

- The Id will be set to one greater than the current largest id in the table.
- Usernames are case sensitive, meaning two users can have the same username if one of the characters differs in capitalization (Ex. Jeffrey vs. JeFFrey)
- Passwords must meet the requirements set by the default behaviour of isStrongPassword in the validator npm module. (Contains at least 1 lower case letter, 1 capitalized letter, 1 special character, 1 number, and contains at least 8 characters.)
- Passwords must also be encrypted before being stored.

The Class-Permitted Spell table has the following validity constraints:

- The Class id must be an integer which is an existing id in the class table.
- The Spell id must be an integer which is an existing id in the spell table.
- If a spell is deleted, all records in this table containing that spell's id must also be deleted.

- If a class is deleted, all records in this table containing that class's id must also be deleted.

The Known Spell table has the following validity constraints:

- The Spell Id must be an integer which is an existing id in the Spell table.
- The Character Id must be an integer which is an existing id in the character table.
- If a spell is deleted, all records in this table which contain the id of the deleted spell must also be removed.
- If a character is deleted, all records in this table which contain the id of the deleted character must also be removed.

The Class table has the following validity constraints:

- The id will be set to one greater than the current largest id in the table.
- The Name must be a non-empty string.
- The Hit Die must be a non-empty string.

The Class Feature table has the following validity constraints:

- The Class id must be an integer, and which is an existing id in the class table.
- The Name must be a non-empty string.
- The description must be a non-empty string.
- The Level must be an integer between 1 and 20, inclusive.
- If a class is deleted, all records in this table with a class id equal to the id of the deleted class must also be removed.

The Background table has the following validity constraints:

A. The Id will be set to one greater than the current largest id in the table.
B. The Name must be a non-empty string.

The Background Feature table has the following validity constraints:

- The Background id must be an integer which is an existing id in the background table.
- The Name must be a non-empty string.
- The Description must be a non-empty string.
- If a background is deleted, every record in this table with a background id equal to the id of the deleted background should be removed.

The Saving Throw Proficiency table has the following validity constraints:

- The Character Id must be an integer which is an existing id in the character table.
- The Ability Id must be an integer which is an existing id in the ability table.

The Skill Proficiency table has the following validity constraints:

- The Character ID must be an integer which is an existing id in the character table.

- The Skill ID must be an integer which is an existing id in the skill table.
- If a character is deleted, all rows in the skill proficiency table with a character id equal to the id of the deleted character must be removed.
- If a Skill is deleted, all rows in the skill proficiency table with a skill id equal to the id of the deleted skill must be removed.

The Skill Expertise table has the following validity constraints:

- The Character ID must be an integer which is an existing id in the character table.
- The Skill ID must be an integer which is an existing id in the skill table.
- If a character is deleted, all rows in the skill expertise table with a character id equal to the id of the deleted character must be removed.
- If a Skill is deleted, all rows in the skill expertise table with a skill id equal to the id of the deleted skill must be removed.

The Owned Item table has the following validity constraints:

- The Character Id must be an integer which is an existing id in the character table.
- The Name must be a non-empty string.
- The Count must be a positive integer.
- If a character is deleted, every row in the Owned Item table with a character id equal to the id of the deleted character must be removed.

The Race table has the following validity constraints:

- The Id will be set to one greater than the current largest id in the table.
- The Name must be a non-empty string.

The Ethics table has the following validity constraints:

- The Id will be set to one greater than the current largest id in the table.
- The Name must be a non-empty string.

The Morality table has the following validity constraints:

- The Id will be set to one greater than the current largest id in the table.
- The Name must be a non-empty string.

The Saving Throw Bonus table has the following constraints:

- The Ability Id must be an integer which is an existing id in the ability table.
- The Character Id must be an integer which is an existing id in the character table.
- The Bonus must be a non-zero integer. A zero would imply no bonus, which is a waste of data since an absence of a row implies the same thing.
- If an ability is deleted, every row in the saving throw bonus table with an ability id equal to the id of the deleted ability must be removed.
- If a character is deleted, every row in the saving throw bonus table with a character id equal to the id of the deleted character must be removed.

- If an existing saving throw bonus's bonus value is changed to zero, the table row should be deleted instead.

The Skill table has the following validity constraints:

- The Id will be set to one greater than the current largest id in the table.
- The Ability Id must be an integer which is an existing id in the ability table.
- The Name must be a non-empty string.
- If an ability is deleted, each row in the skill table with an ability id equal to the id of the ability deleted must be removed.

The Ability table has the following validity constraints:

- The Id will be set to one greater than the current largest id in the table.
- The Name must be a non-empty string.

The Racial Trait table has the following validity constraints:

1. The Race Id must be an integer which is an existing id in the race table.
2. The Name must be a non-empty string.
3. The Description must be a non-empty string.

The Ability Score table has the following validity constraints:

- The Ability Id must be an integer which is an existing id in the ability table.
- The Character Id must be an integer which is an existing id in the character table.
- The Score must be an integer. No restrictions are placed on the value since it allows custom spells which enhance or reduce ability scores.
- If an ability is deleted, each row in the ability score table which an ability id equal to the id of the deleted ability must be removed.
- If a character is deleted, each row in the ability score table which an ability id equal to the id of the deleted character must be removed.

Data needs and data dependencies section authored by Jeffrey

Part B

Coding Standards - Samuel

We have set up a set of rules that we will follow for our standards of coding. From comments, to naming, and types of errors to be thrown.

Documentation

All the methods and classes must be properly documented to generate a proper *JsDoc* from our documentation.

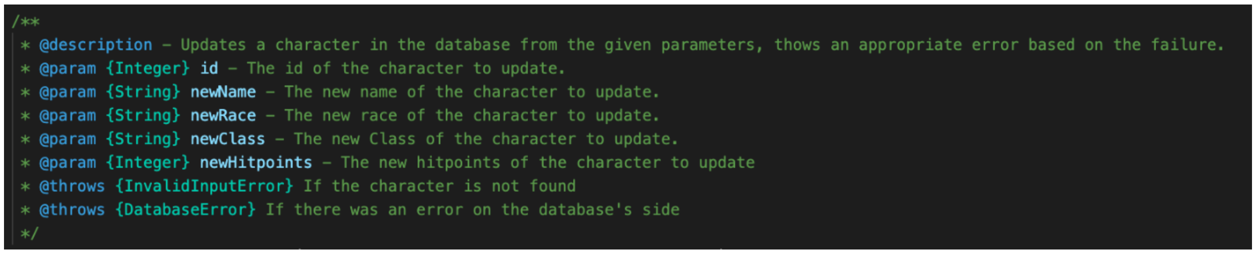
These summaries are to be put before each method and class. The summary must be concise but long enough to properly describe how the method works so we can see when hovering over the method exactly what it expects and what is should return, if anything. Some method summaries, especially the model, will be highly detailed, describing step by step how the method works.

After the description, all the parameters will be explained with their types explicitly stated (ex. Integer, String, Double, etc.) even though JavaScript is dynamically typed.

The exceptions thrown must also be stated in the documentation in order to properly catch them when using the methods from other modules of the code.

If the method is expected to return a value, then this must be written down as well. The return must be explicitly typed (String, Integer, Character, Spell, etc.) and it must be marked if it should ever return a null value if something fails in the method.

All in all, a properly documented method will look something like this:

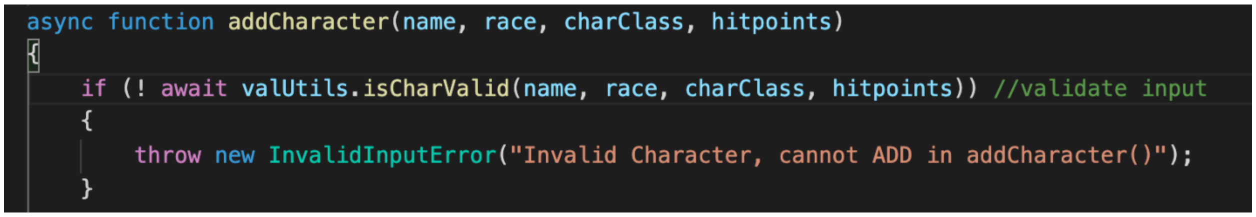


```
/**
 * @description - Updates a character in the database from the given parameters, thows an appropriate error based on the failure.
 * @param {Integer} id - The id of the character to update.
 * @param {String} newName - The new name of the character to update.
 * @param {String} newRace - The new race of the character to update.
 * @param {String} newClass - The new Class of the character to update.
 * @param {Integer} newHitpoints - The new hitpoints of the character to update
 * @throws {InvalidInputError} If the character is not found
 * @throws {DatabaseError} If there was an error on the database's side
 */
```

Comments

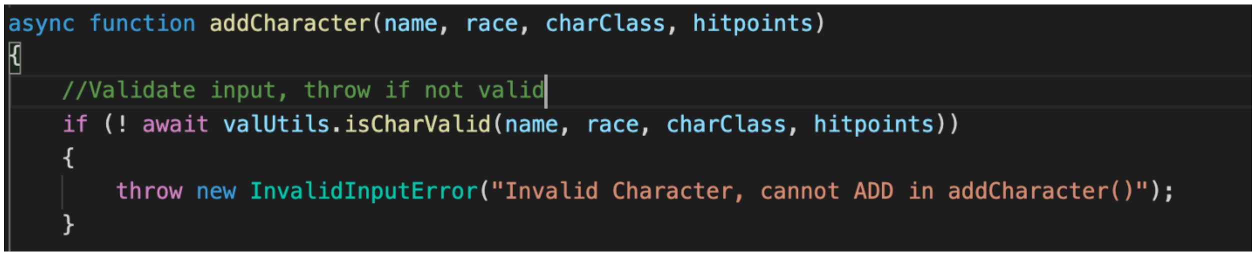
In methods, comments are allowed and encouraged if done properly. Comments must not be inline and must be relevant to the next few lines of code. By breaking down a part of the code into a single comment, the code will be more readable and will help while debugging.

Example of bad comment:



```
async function addCharacter(name, race, charClass, hitpoints)
{
    if (! await valUtils.isCharValid(name, race, charClass, hitpoints)) //validate input
    {
        throw new InvalidInputError("Invalid Character, cannot ADD in addCharacter()");
    }
```

The inline (//validate input is in the wrong spot and makes it less easy to follow along in the code). A better and good example can be seen here:



```
async function addCharacter(name, race, charClass, hitpoints)
{
    //Validate input, throw if not valid
    if (! await valUtils.isCharValid(name, race, charClass, hitpoints))
    {
        throw new InvalidInputError("Invalid Character, cannot ADD in addCharacter()");
    }
```

Variable Naming
For our variable naming we decided to keep it simple and go with the usual *JavaScript* naming conventions. Class names will be PascalCase. Methods, variables (*let, var, and const)* will be *camelCase*. Any constants that are used as a typical constant declared at the top of the code will be *CAPITALIZED* (ex. MAX_HP = 99).

Errors
An error class will be created to have them all accessible from each of the modules of our project. To keep them uniform they will each have a similar constructor in order to relay as much information as possible to the module or method catching the error and it will also help with the server logs.

No generic errors will be thrown from the *JavaScript* built in *Error* class.

The constructor will take in parameter for the module name, method name, and error message. The class will then put all these together to create an error message to relay. An example error message when an add fails because of Invalid Input:

`(CharacterModel) – addCharacter: Name must be alpha`

Data Storage
For our convention on data storage see 'Database naming consistency' in 'Part A' of this document. All our decisions are detailed in this section.

Interface Design - Samuel

Color Palette
The design we decided to go with is a modern flat UI design. Based on the document written on Material.io we decided on some background colors as well as some foreground colors to make things like buttons and important text pop.

Our interface inspiration is how do we really think people who play dungeons and dragons want a website to look and feel. Nobody wants to be blinded by a bright white light because a web page is in light mode when they're in their parent's dark basement playing dungeons and dragons.

Our palette is as follows:

Background
#313431 ■- To have the dark mode we envisioned we started our background with a dark shade of grey.

Background Elevations
#484A48 ■- This color is used to give a sense of height for sections of the website that need to be focused such as forms, character info, etc.

#5F615F ■- This color is used to give an even greater sense of height for sections of the website that need have even more emphasis. Used for form input fields.

Button/Pop color

#7FB3C8 ■- We decided to go with a color that is not too pastel-like but also looked good on top of our background colors.

#3E505B ■- The hover color of the button/pop color. It goes darker to emphasise the action of hovering, it also changes the text color from dark to light.

Navbar/footer/tables

#212529 ■- The table design is darker to make it seem more 'static' than an 'elevated' part of the website such as a form. It is a very dark shade of blue to go with the overall color scheme.

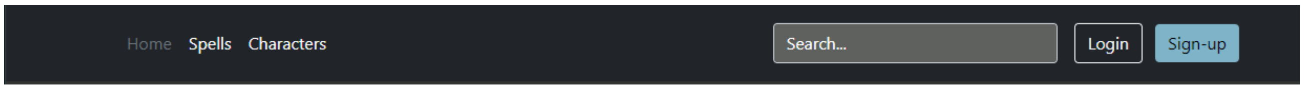
Text

#FFFFFF □ - The majority of our text is white to have a good contrast between the background and foreground colors.

Partials

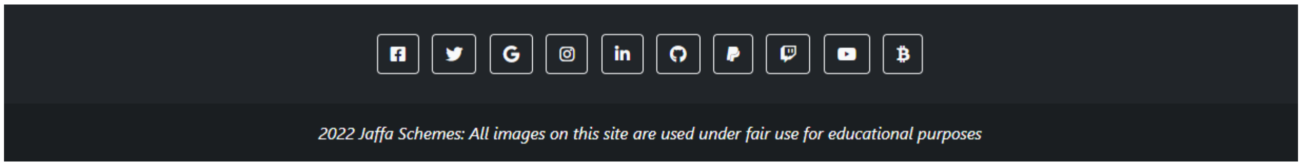
For the different pages on the website to have a similar 'feel' we decided to have a static header and footer. It will look the same on every page.

Header



We decided on a simple dark bootstrap header that would look great with our background color as well as not be too distracting so the user can focus on the more colorful page that will be rendered. The final product will have more tabs for other features such as classes, races, 'my characters' but the overall look and feel will stay the same.

Footer

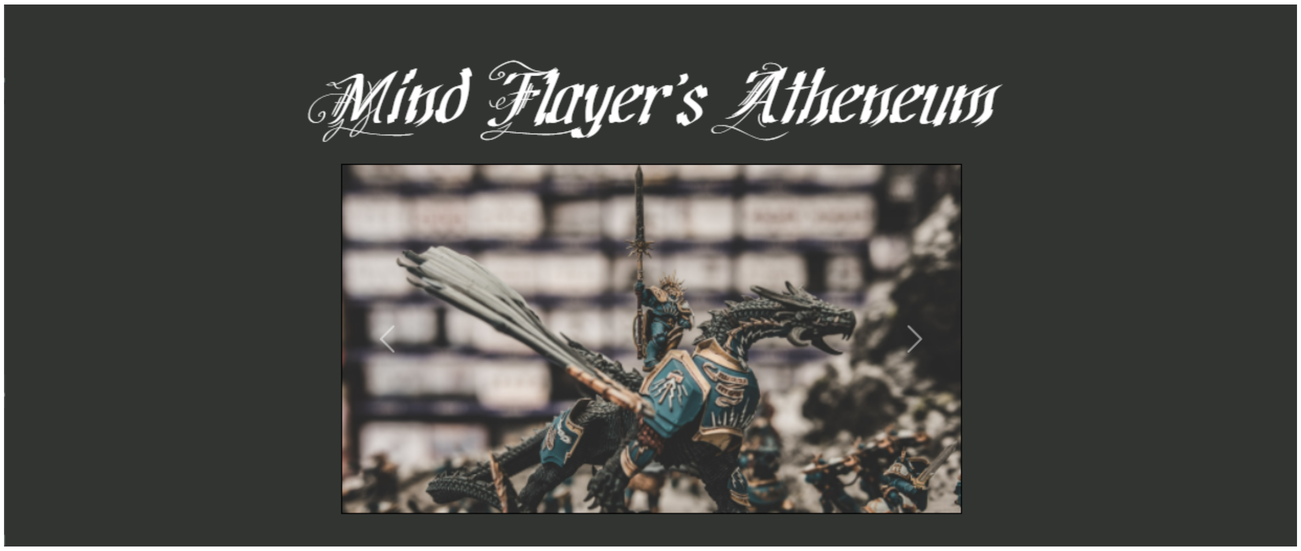


2022 Jaffa Schemes: All images on this site are used under fair use for educational purposes

Akin to the header, the footer is a simple design with our team's name as well as buttons for socials/other links that might be of use. We will put relevant icons here such as some good dungeon and dragon resources such as roll20, dnd5e wiki, reddit, etc.

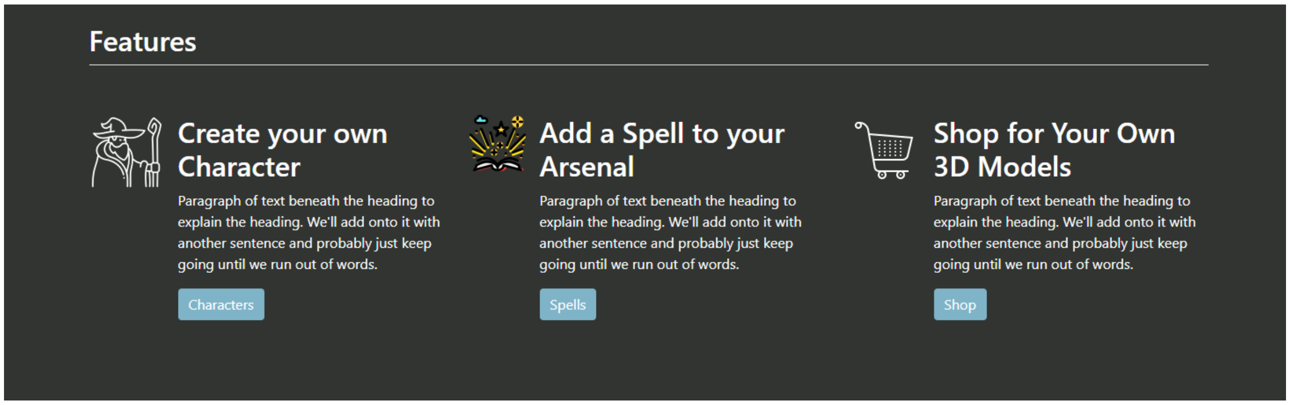
Home page

Hero



Thinking ahead, we decided to have a carousel of images to showcase some of our website features. The images are temporary for now (although we might end up using some/all of them). The carousel has an image darkening effect on hover which could allow us to add clickable buttons to bring the user to different parts of our website.

Features



Features

Create your own Character

Paragraph of text beneath the heading to explain the heading. We'll add onto it with another sentence and probably just keep going until we run out of words.

Characters

Add a Spell to your Arsenal

Paragraph of text beneath the heading to explain the heading. We'll add onto it with another sentence and probably just keep going until we run out of words.

Spells

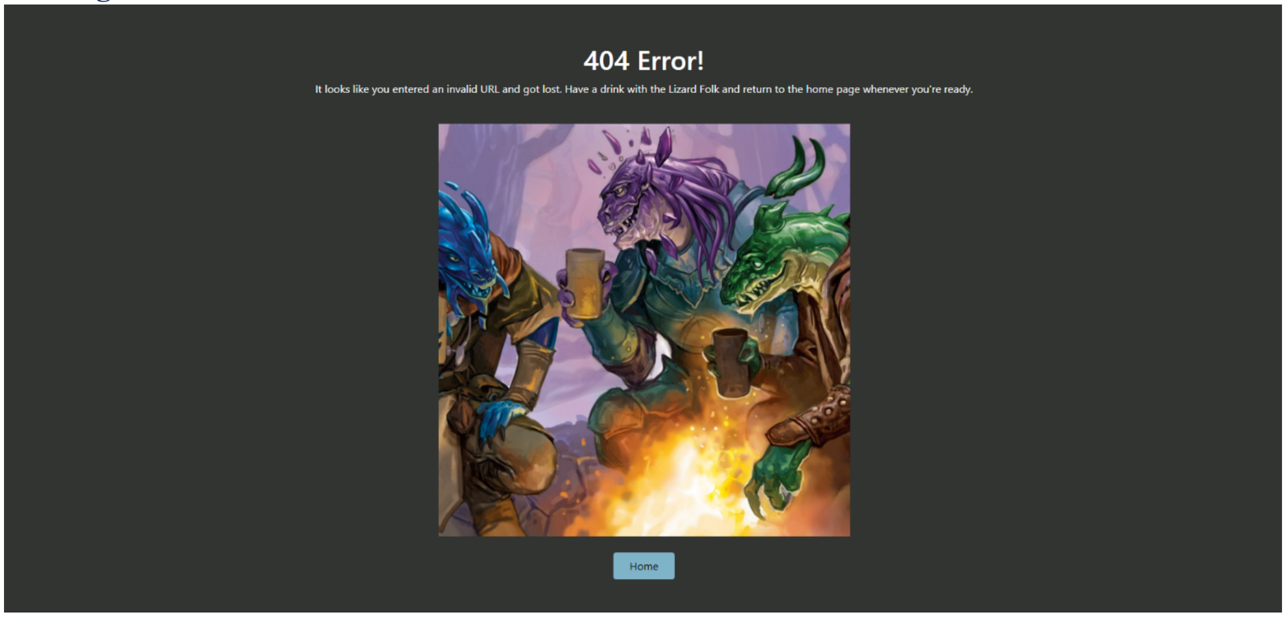
Shop for Your Own 3D Models

Paragraph of text beneath the heading to explain the heading. We'll add onto it with another sentence and probably just keep going until we run out of words.

Shop

The features section describes the main features of our website in more detail to convey to the user what they can do. We wanted something sleek but with a little bit of flair. To do so we added the small icons next to each of the sections. Our features might be different once all is said and done but the look will be similar to what is shown above.

404 Page


404 Error!

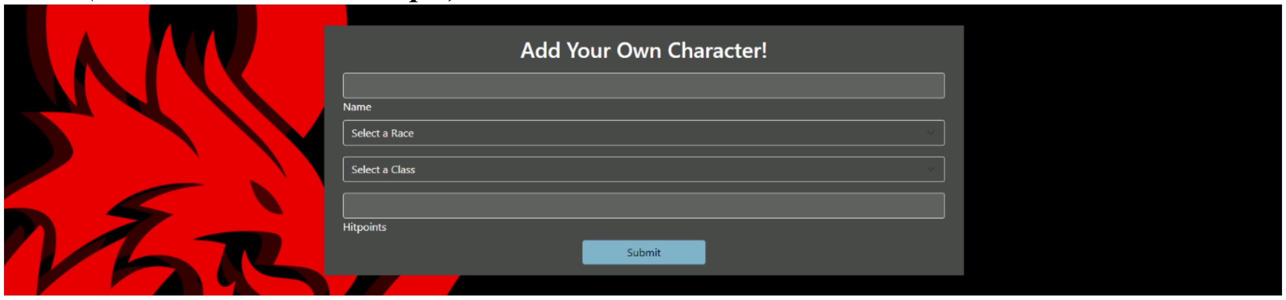
It looks like you entered an invalid URL and got lost. Have a drink with the Lizard Folk and return to the home page whenever you're ready.

We wanted our 404-error page to be humorous. Everyone knows the iconic 'error 404' so we decided to add some spice and put an image with some accompanying funny text on the page.

Lists Pages

For other pages we decided to have a simple layout with most of the information on a single page, so that the user isn't redirected to different pages and having to get re-acquainted with the layout of a page.

Form (Characters in this example)


Add Your Own Character!

Name

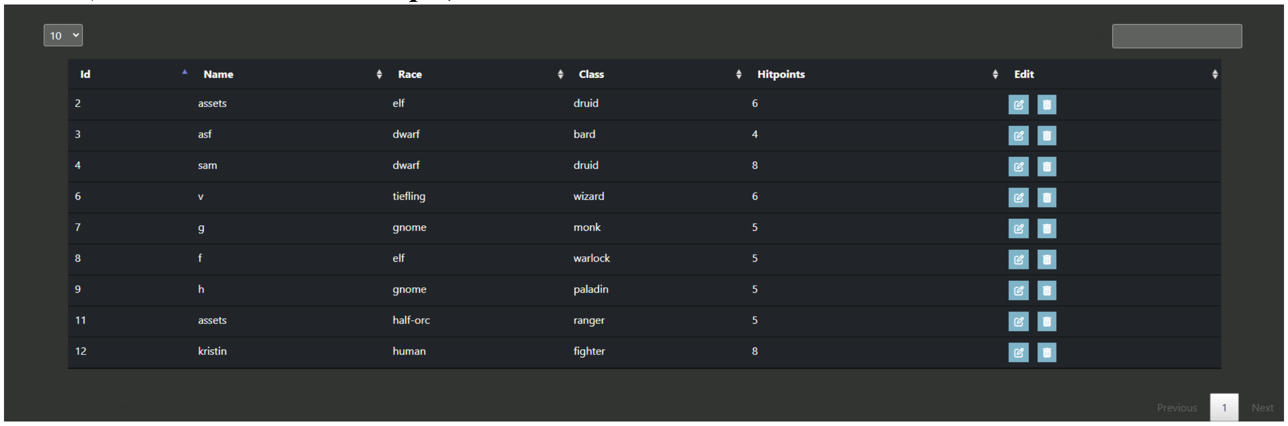
Select a Race

Select a Class

Hitpoints

The form has all the inputs needed in a div with a different color in order to accentuate the elevation we are trying to convey. The forms (especially for character) will have a lot more inputs. Html form validation is also in place in order to have even less possibilities of failure in the model module.

Table (Characters in this example)



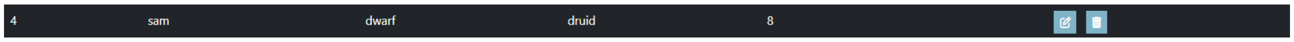
10

Id	Name	Race	Class	Hitpoints	Edit
2	assets	elf	druid	6	
3	asf	dwarf	bard	4	
4	sam	dwarf	druid	8	
6	v	tiefling	wizard	6	
7	g	gnome	monk	5	
8	f	elf	warlock	5	
9	h	gnome	paladin	5	
11	assets	half-orc	ranger	5	
12	kristin	human	fighter	8	

Previous 1 Next

Our tables will have a darker background than the root background to give a sense of importance and separation. These tables will have different data (from characts to spells, classes, and monsters). There will be a filter box as well as sorting based on the column in the table.

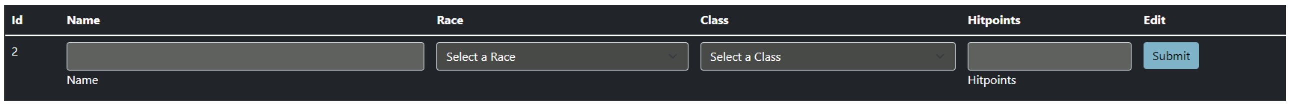
Table Row



4	sam	dwarf	druid	8	

Every Character in the database is a row in the table. The name (in this case is ’sam’) is clickable and will take you to another page (/characters/:id (:id in this case is 4)). There are also 2 buttons in the Edit column. The edit button, , is clickable and will replace the row in the table with an editable form that sends a ’put’ request when it is submitted. The delete button, , will send a ’delete’ request with the id of the character to delete them from the database.

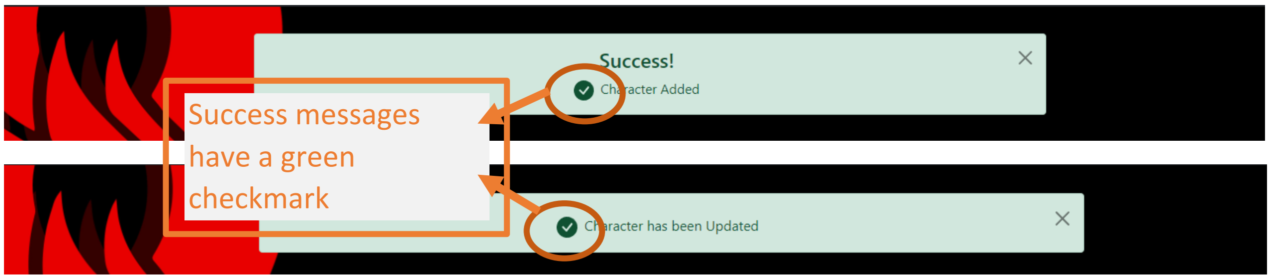
Table Row Edit Mode



Id	Name	Race	Class	Hitpoints	Edit
2	Name	Select a Race	Select a Class	Hitpoints	Submit

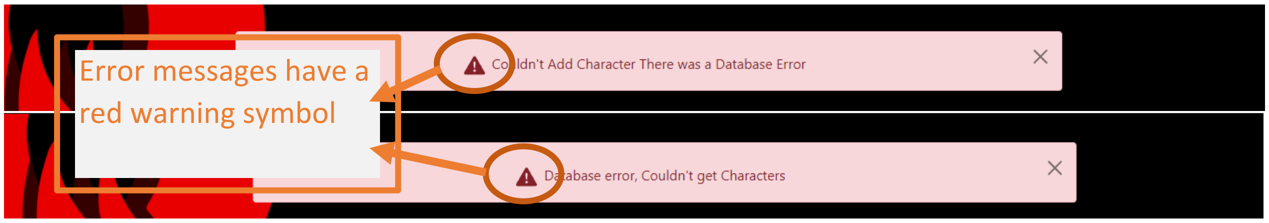
Once the edit button is clicked, the matching row will go into edit mode and a submit button will be there instead of the 2 buttons. Once the form is filled out, the object with the matching Id will be updated to match new information added by the user. The submit button will send a ‘put’ request in order to update the database with the new information.

Alerts



Success!
Character Added

Character has been Updated



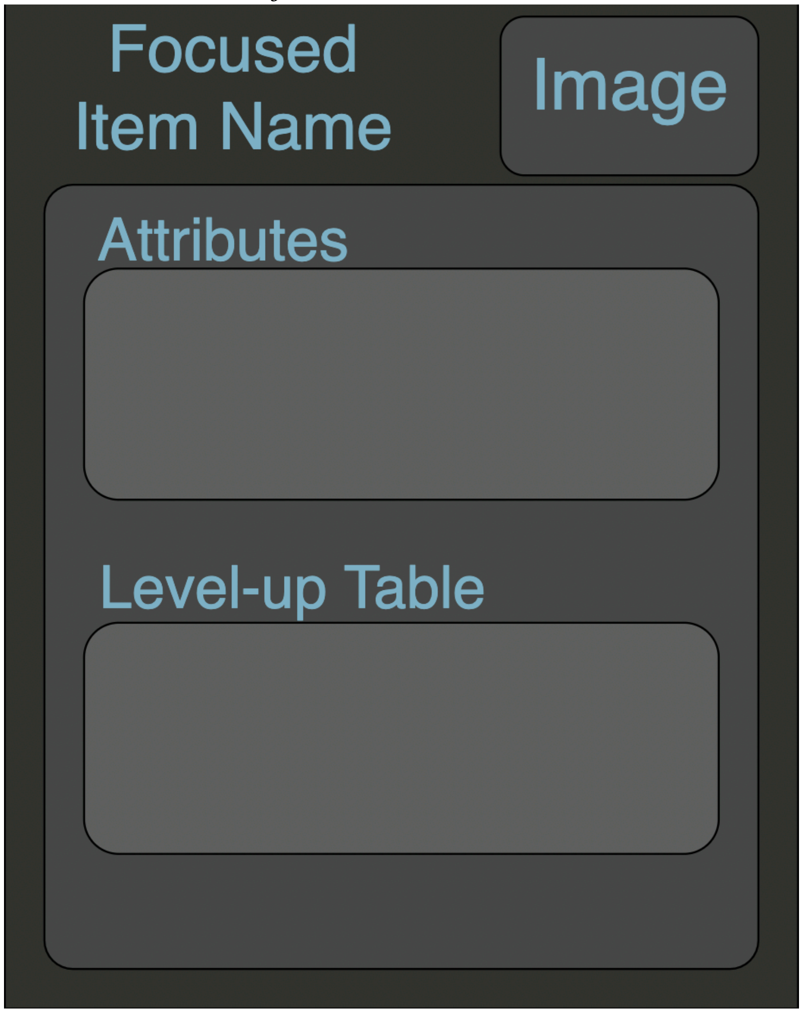
Couldn't Add Character There was a Database Error

Database error, Couldn't get Characters

Various bootstrap alerts will insert into the DOM when actions are completed. These alerts are dismissible by clicking on the ’X’ in the top right corner of each alert. Due to our website design of having most of the information be on the same page these alerts show the user what happens instead of redirecting to a success or error page.

Focused Object

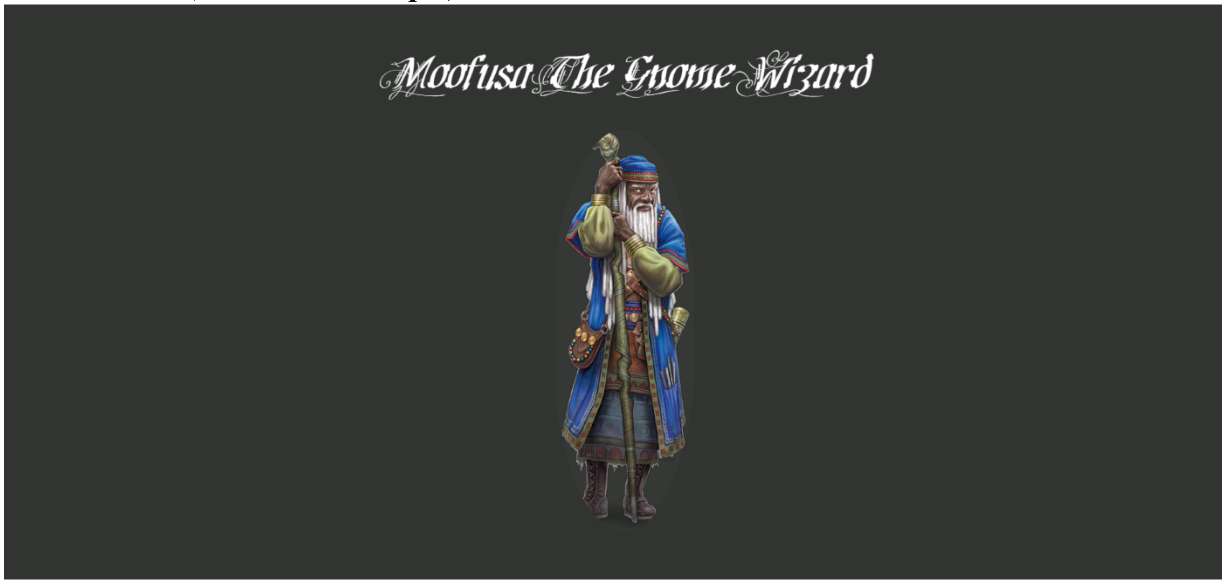
Generic Focused Object



Once focused, items such as a specific race or class will display with a simple view interface that describes the unique features of the object as well as a table that contains the level up table so that users can see how a specific character would level up over time.

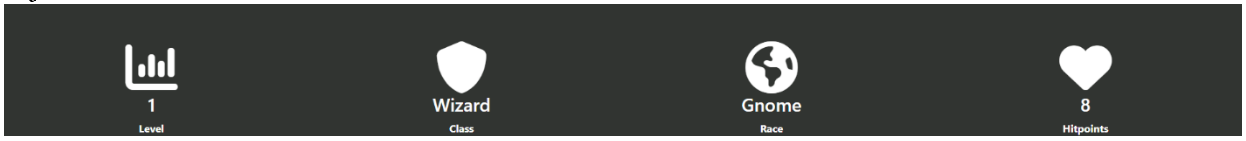
A small image will show up on the top right in order to differentiate the different objects that could be focused and give a unique flair to the page.

Main Section (Character example)



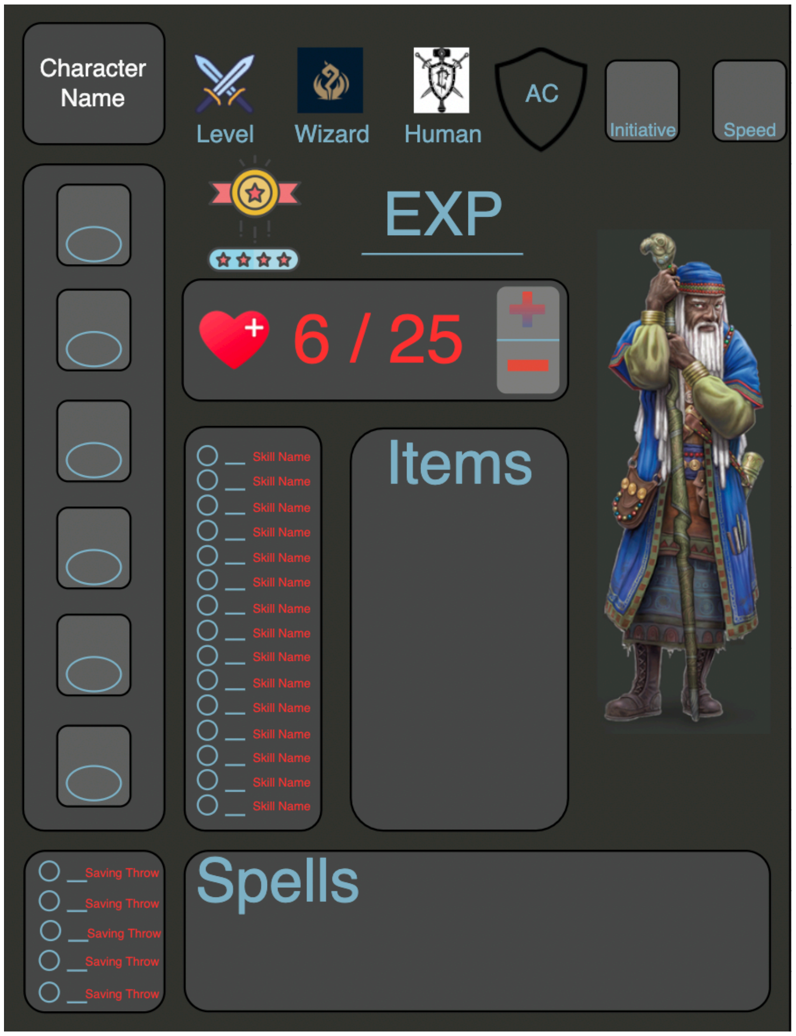
For the focused element in our page, we decided to add a title and an image showcasing a part of the focused element (in this case it is the class of the character).

Information Section



At the bottom of the main title and image there will be a section with all the stats of the specific focused element. For characters we plan on having all their attributes as well as their stats. For spells we can have a damage chart as well as their attributes, for an individual class we can have a table of the level up stats, etc.

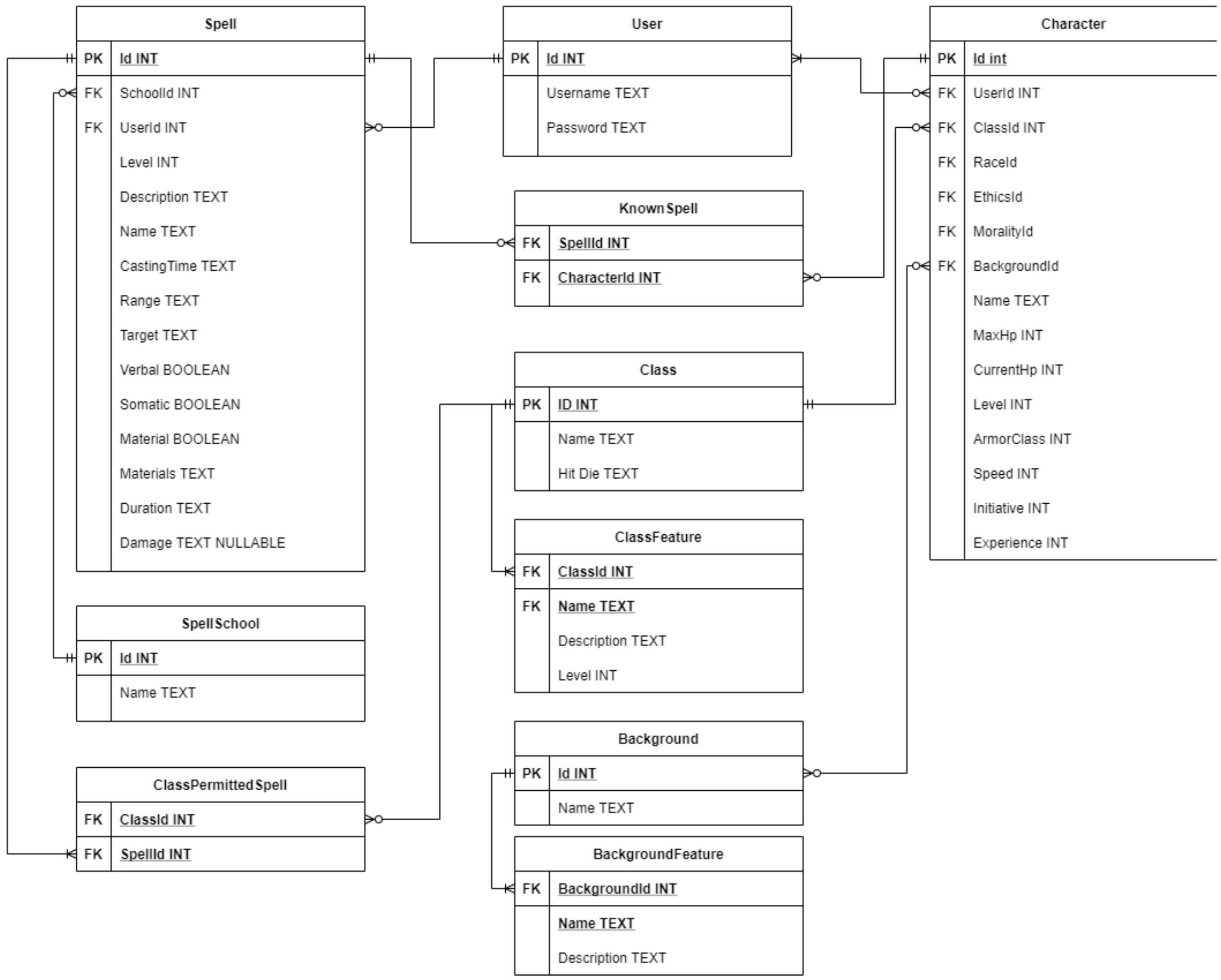
Character Sheet



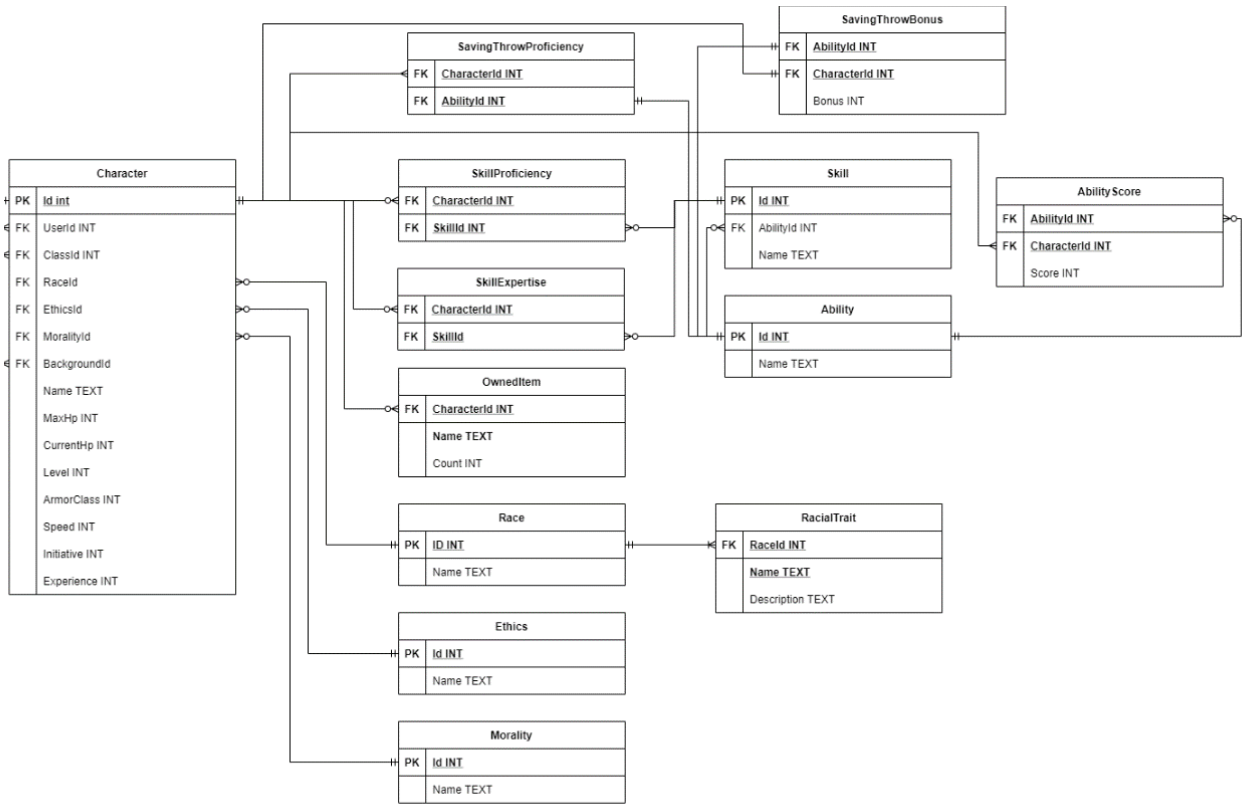
This is our rough draft for our ‘character sheet’. We want to get rid of some of the useless stuff that we won’t be using on our website and emphasize the features that will be of use on our website (image, experience points, hit points).

Styling and Coding Standards authored by Samuel

Database Design - Jeffrey



Left Side ^



Right Side ^

Database Design Explanation

Several choices were made for the database which are reflective of the requirements and features we've listed. The following section will provide insight into some choices made that may not be immediately obvious.

The ClassPermittedSpells table serves the purpose of indicating what spells can be cast by what classes. Certain spells can be cast by only specific classes. To know which spells are capable of being cast by a specific class, every time a spell is created it will be added to the ClassPermittedSpells table as well. The ClassPermittedSpells table holds the id of a spell and the id of the class which can cast it. If the spell can be cast by multiple classes, the spell id will appear multiple times with each class id that can cast it.

The ClassFeature, BackgroundFeature and RacialTrait tables will be used to hold the features of the classes, backgrounds and races. The ClassFeature table is the only one of the three which contains a level. The level represents the level at which the class gains that feature.

The Spells table contains a foreign key from the User table. This allows users to create their own homebrewed spells. The users table will always have at least one user with id 0. This will be the "Default" user. When a user gets a list of spells, they will get all the spells with User ID 0, which will be all the spells from the Player's Handbook and all the spells with their User ID. This means that they get all the default spells and any homebrew spells they created themselves.

The Morality and Ethics tables contain the values for alignment. They will contain the Good, Neutral, Evil / Lawful, Neutral, Chaotic values. These tables are there to help populate dropdowns in the views. They also help save storage space since each character only has to store two ids rather than two strings. The SpellSchool table serves the same purpose (populating dropdowns and saving storage space).

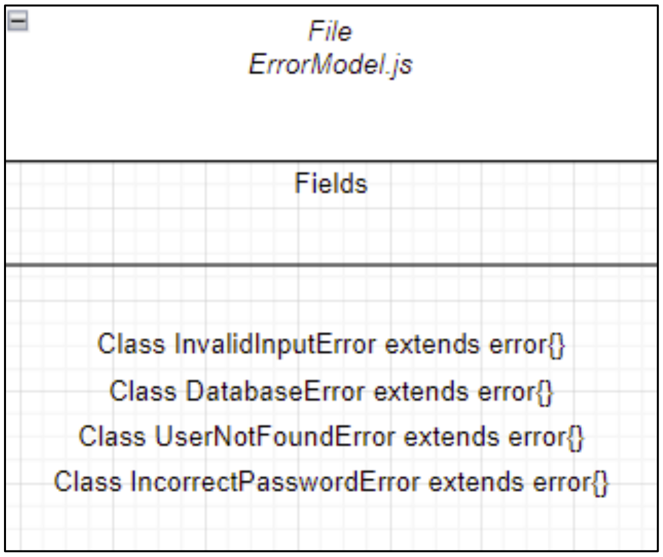
We had a hard time determining how we were going to store Character proficiencies and expertise in skills. At first, we were going to have each skill as a column in the character table which contained an integer. 0 would represent no proficiency, 1 would represent proficiency, and 2 would represent expretise. Instead, we decided on a skills table. The skills table would contain an id and a name for each skill. Then, the skillProficiency and skillExpertise table would contain the id of a character and the id of the skill they're proficient or have expertise in. A character can not have proficiency and expertise in the same skill so both tables can not have duplicate rows. The SavingThrowProficiency table serves the same purpose as the skills proficiency table but it references the ability table instead of a saving throw table. Saving throws and abilities are a one-to-one match so there's no need for a table in between.

Unlike the SkillProficiency system, saving throws can often have bonuses caused by certain gear or class features, to allow for these bonuses, the SavingThrowBonus table exists. It references a character, and an ability. This way, if a row in the table holds character id 3 and ability id 1, along with a bonus of 5, we can determine that the character with id 3 gets a bonus of +5 when performing a strength saving throw. If they are also proficient in the saving throw, their proficiency would be added to that +5 separately.

Database design section authored by Jeffrey

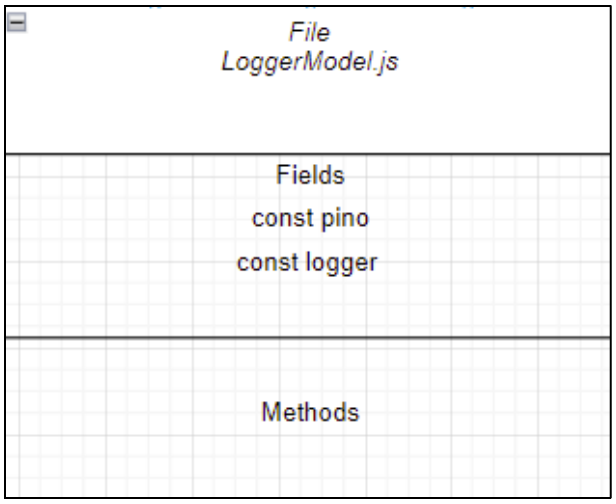
Codebase Design - Chase

An error model JavaScript file with the following details.



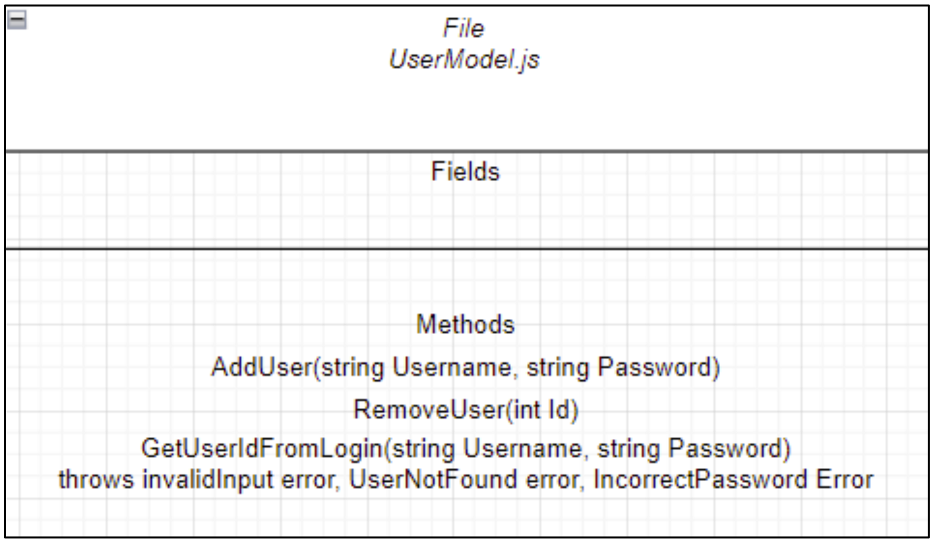
Class InvalidInputError extends error{}
Class DatabaseError extends error{}
Class UserNotFoundError extends error{}
Class IncorrectPasswordError extends error{}

A logger model JavaScript file with the following details.



const pino
const logger

A user model JavaScript file with the following details.


File
UserModel.js

Fields

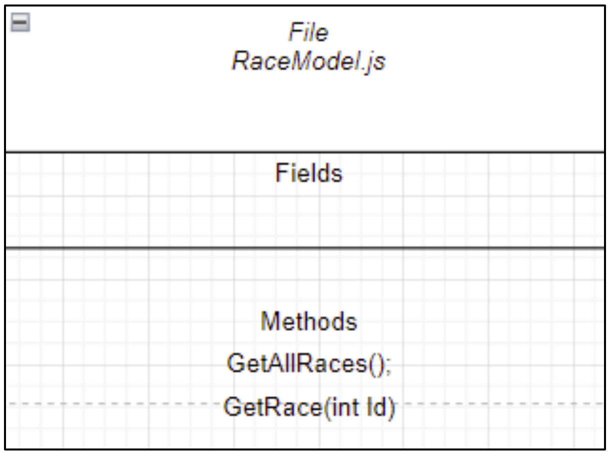
Methods

AddUser(string Username, string Password)

RemoveUser(int Id)

GetUserIdFromLogin(string Username, string Password)
throws invalidInput error, UserNotFound error, IncorrectPassword Error

A race model JavaScript file with the following details.



File
RaceModel.js

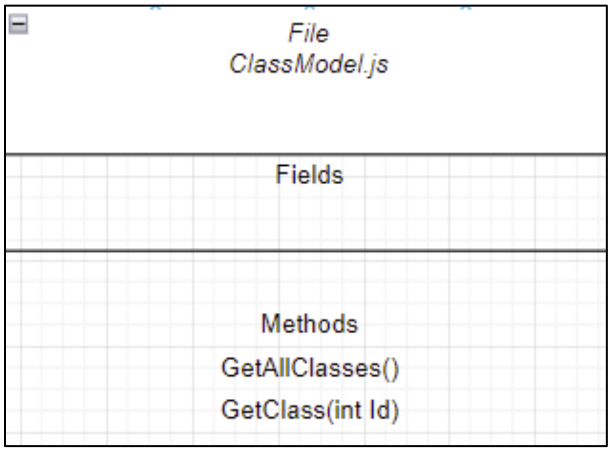
Fields

Methods

GetAllRaces();

GetRace(int Id)

A class model JavaScript file with the following details.



File
ClassModel.js

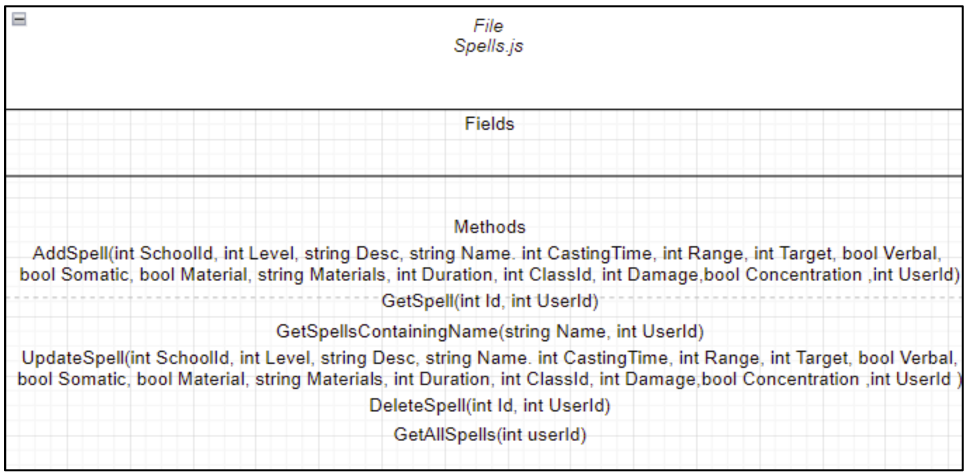
Fields

Methods

GetAllClasses()

GetClass(int Id)

A spell model JavaScript file with the following details.



File
Spells.js

Fields

Methods

AddSpell(int SchoolId, int Level, string Desc, string Name. int CastingTime, int Range, int Target, bool Verbal, bool Somatic, bool Material, string Materials, int Duration, int ClassId, int Damage,bool Concentration ,int UserId)

GetSpell(int Id, int UserId)

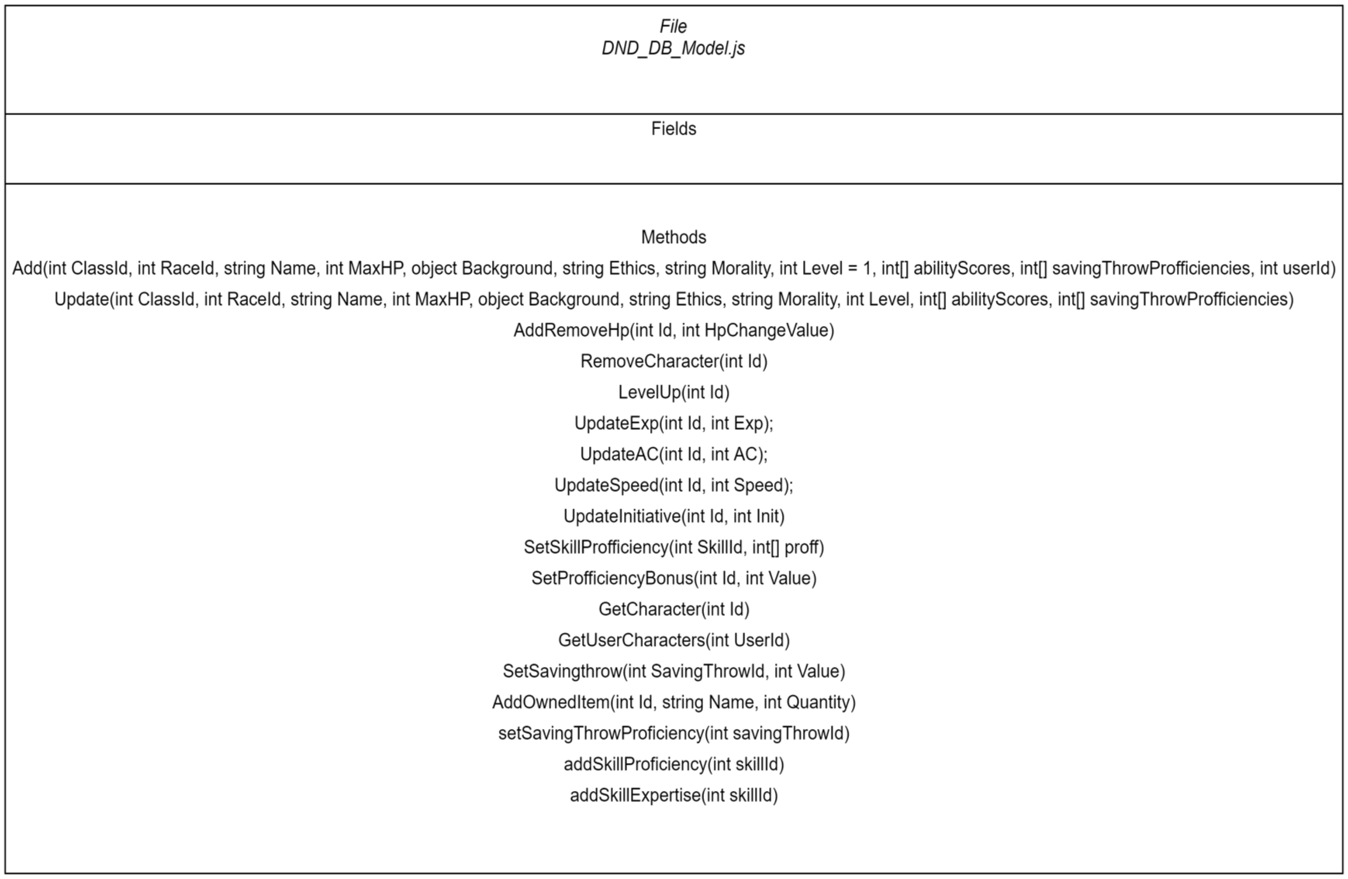
GetSpellsContainingName(string Name, int UserId)

UpdateSpell(int SchoolId, int Level, string Desc, string Name. int CastingTime, int Range, int Target, bool Verbal, bool Somatic, bool Material, string Materials, int Duration, int ClassId, int Damage,bool Concentration ,int UserId)

DeleteSpell(int Id, int UserId)

GetAllSpells(int userId)

A DND database model JavaScript file with the following details.



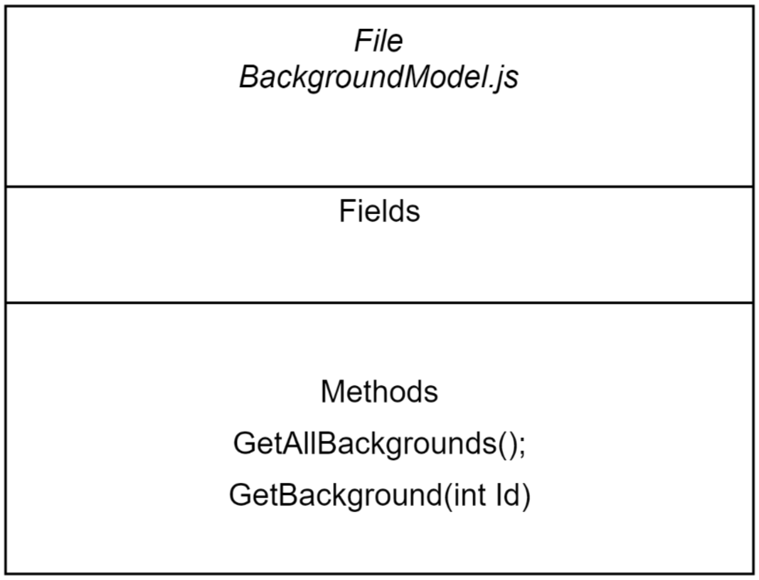
File
DND_DB_Model.js

Fields

Methods

```
Add(int ClassId, int RaceId, string Name, int MaxHP, object Background, string Ethics, string Morality, int Level = 1, int[] abilityScores, int[] savingThrowProfficiencies, int userId)
Update(int ClassId, int RaceId, string Name, int MaxHP, object Background, string Ethics, string Morality, int Level, int[] abilityScores, int[] savingThrowProfficiencies)
AddRemoveHp(int Id, int HpChangeValue)
RemoveCharacter(int Id)
LevelUp(int Id)
UpdateExp(int Id, int Exp);
UpdateAC(int Id, int AC);
UpdateSpeed(int Id, int Speed);
UpdateInitiative(int Id, int Init)
SetSkillProfficiency(int SkillId, int[] proff)
SetProfficiencyBonus(int Id, int Value)
GetCharacter(int Id)
GetUserCharacters(int UserId)
SetSavingthrow(int SavingThrowId, int Value)
AddOwnedItem(int Id, string Name, int Quantity)
setSavingThrowProficiency(int savingThrowId)
addSkillProficiency(int skillId)
addSkillExpertise(int skillId)
```

A background model Javascript file with the following details



File
BackgroundModel.js

Fields

Methods

```
GetAllBackgrounds();
GetBackground(int Id)
```

UML created by Chase

Coding Techniques / Technologies - Samuel

NPM

Node Package Manager (NPM) is what we will be using to include the packages we will require for this website.

Bootstrap

Bootstrap was heavily utilised in the creation of the general template for each web page. The header, footer, carousel, as well as the tables were all created using a bootstrap template.

Handlebars

Handlebars helper methods are used to display data such as the characters and spells. We used techniques taught in class such as *if* and *each* statements. We also used our own handlebars methods by registering a helper method, an *'equals'* helper which allows us to test equality in a handlebars if statement. This allowed us to generate the inline form for the updating methods for both the characters and the spells.

Handlebars is also a package that we will download with NPM.

JavaScript

Front-end JavaScript will be implemented in our website. We will rely on it, mostly, for event handlers / event listeners. Manipulating the DOM strictly with handlebars isn't the best because the page must be reloaded every single time due to handlebars' innate functionality. With JavaScript it is possible to add event listeners to all the buttons and dynamically change without reloading the page or submitting a form, which causes a lot of server request and means the data must be fetched from the database more often causing more IO requests than what is needed.

For example, instead of having an edit button that acts as a form which loads a whole new page in order to insert a form into the DOM that will have a submit button whose method will send a PUT request, we can have an event listener on the buttons and edit the DOM to dynamically create a form an insert it wherever we want without having to load a new page.

JavaScript can also be used on our tables to sort and filter the data without sending requests to the database.

Pino

Pino is a logger that will be installed through NPM. It allows us to have different logging levels. These logging levels can range from *Info* to *Fatal.* These logging levels allow us, as programmers, to write different messages for different reasons and if there is ever a bug that needs debugging, we can choose to filter these logs to only see some of these levels.

Another feature of Pino is its ability to log to a file. We will be using this to keep our server logs in a specific log file that we can refer to in case of errors or failure.

Validator
Validator is a package available from NPM which contains a library for string validation. We will be implementing Validator in our validation functions of the models and in our password checking function to ensure the password is strong enough.

MySQL
MySQL is an open-source relational database management system (DBMS). In our scope MySQL will be installed from NPM and will allow us to create a connection to a database that we will create in a docker container.

After our connection is created, we can query the database. This allows us to create tables, add data to the table, and query the database.

Because the DBMS is relational, we can join multiple tables and query from multiple tables at once. This allows us to get specific data based on select queries done in SQL.

Bcrypt
Bcrypt is a library that will allow us to hash passwords in order to store them securely in our database. Bcrypt is a popular hashing function which allows us to add some salt to make the passwords even more secure.

Cookies
HTTP cookies are small blocks of data sent by the server in order to keep track of certain things. These can include preferences (light vs. dark mode), logged in status, shopping cart information, etc. They will allow us to make sure a user stays signed in, as well as keep track of their shopping cart if we decide on implementing one.

UUID
UUID or Universally Unique Identifier is exactly what it sounds like, a unique number that can identify something. NPM contains a UUID package that will allow us to create a unique session ID for each of the users that request one from the server. This will allow us to keep track about a logged-in status as well as shopping cart information.

Docker
Docker is a lightweight virtualization program that allows the creation of *'images'* of specific software. In our website we will be using it for the creation of our MySQL server.

Puppeteer
We will be using puppeteer for our UI tests. In order to test that our forms function properly and send to the right view we must use a package that can input text that we want into the forms and then have the software click the submit button. Puppeteer allows us to do this.

Jest
Jest is a testing framework that allows us to create our tests and have expected values match with the ones provided by our model. Jest also works with Puppeteer which is a bonus.

Coding Technologies and Techniques authored by Samuel

Application Features – All contributed, details at the bottom of the section

Required features for a successful site.

On each page there will be the same header and footer that will do the same thing on each page. The header will have tabs for each core feature of the site. On the right there will be a sign-up and a login button that will change to your username if you're currently logged in. The footer will contain images as buttons that will redirect (in a new tab) to the requested page. There will be buttons for related websites as well as our teams GitHub page and professional pages (LinkedIn).

When accessing our website, the user will be sent to the home page. From the home page there will be a button that brings you to the login/sign up page. There will also be a nav bar with the selection of "characters" which will bring the user to the characters page and "spells" which will bring the user to the spells page. There will be an image carousel in the middle of the page which will serve as our hero image.

When visiting the characters page, the user should expect to see a form to create their own character. Under this, there will be a table with all the characters and their respective stats. Should the user be logged in, they will be able to press buttons to edit and even delete characters that they have created. The names of the characters will be clickable and doing so will bring the user to a focus page of the character. If logged in and the character is owned by the user, on this character focus page there will be a plethora of buttons and inputs to track a character's progress through a campaign. A button appearing beside the HP will either add or remove HP from the character, either in increments of 1 if no number is added, or it will add/remove based on the number entered. The text inputs will all be editable and will update the character with the new values provided. These inputs include inventory, name, attacks, etc. The user must click on the update button in order to save their character.

The spells page will appear differently depending on whether a user is signed in. If a user is not signed in, they will see a list of all the spells in the Wizards of the Coast Player's Handbook displayed in a table format. They will not be able to edit or add any spells. Using a form, they will be able to filter the spells by level, school, or whether they are containing a string. If the user is logged in, their experience will be slightly different. They will still be able to see all the spells from the Player's Handbook, but they will also be able to add spells using a form. After a spell is added, it will be immediately available in the display table. In the table, the user will be able to delete a spell by pressing a button. The user can also press an edit button next to a spell to turn that spell's row into a form. A submission through the form will validate input and update the spell, displaying the list of spells with the update done. A user can also focus on a specific spell by clicking a magnifying button on the row of the spell they want to focus on. The button will bring them to a spell focus page which displays all the information about that specific spell. At the bottom of the page, there will be a button that brings you back to the spells list page. In addition, the filter form will be updated to contain a checkbox for personal spells only. When the checkbox is checked and the spells are filtered, only the user's personally added spells will be displayed.

For the skills, races, and background pages the names of all of them will be clickable and they will bring the user to a focus page of the object they clicked. On these focus pages there won't be any interactions possible. They will be information only and it is there to help with the character creating process.

Our website will have a sign-up / login page where users can decide to create an account. The user should be able to click the sign-up button and be prompted to enter a username and a password. Doing so will create an account if the username is not already taken and the password meets the strength standard. For the login, when the user clicks on the button, they will be prompted to enter their username and password. If there is a match, then the user will be logged in and brought to the home page. If there was an error an alert will pop up informing them if they typed their password wrong or if the username does not exist.

Our website will also have a 404-error page if the user visits a URL that doesn't have an explicit endpoint. On this page the user will find a funny message as well as a button to return home.

Spell related application features authored by Jeffrey

Home page application features authored by Chase

The remainder of the application features section authored by Samuel

Features that would be nice to have. - Chase

In our application, we have decided it would be nice to have certain elements on our page. Firstly, interacting with other users' characters could bring a fun interactive feature to our website. In addition, being able to store campaigns with existing characters on our website make it easier for users to track changes to their character throughout a certain campaign. Custom images can also be a cool feature for user-created spells. A shop was deemed to be an extra feature to our site as the core focus is on the game "DND" itself. A forum chat linked to user profiles that can have names and friends lists, can also be a nice to have, users would be able to comment on specific DND topics and communicate. A dark / light mode option is also addition that is not needed but can be implemented for a better user experience. Monster glossaries can also be added to the page but are not a focus. A level-up button for the characters is also a nice feature that can be added for the user instead of having to update specific details each time. A customized attack (ex rapier button and it rolls all the dice for you) would make editing character details easier. Replacing manual bonus modifier with built-in links and calculations to other attributes will help with calculations. An items table where users can create their own items would be nice for users who want to create magic items. A long rest button is also something that our site does not need but would be nice for the user.

Bulleted list

- Interacting with other user's characters

- Linking characters to campaigns and have campaign pages
- Upload Custom Images for characters/spells/etc.
- A shop with custom miniatures, dice, etc.
- Forum/Chat
- User Profile (Picture, name, etc.)
- Dark mode / light mode toggle
- Monster Glossary
- Friends list
- Level up button in the focused character
- Animated Dice rolls
- Customized attack (ex rapier button and it rolls all the dice for you)
- Replacing manual bonus modifier with built in links and calculations to other attributes
- Even when not logged in, can see all items from player's handbook.
- Once logged in, can add a homebrewed item.
- Items
- Long / Short Rest Button

Nice to have section authored by Chase – Slightly edited by Jeffrey

Part C

Timeline – Jeffrey & Sam

Milestones denoted as **bold.** Past April 24th, tasks are indicated by their tag (Ex. B.4)

April 23	ERD & Database description Interface Design All UML besides Spell and Character Model Key Required Operations	Jeffrey Samuel Chase Chase
April 24	All UML besides Spell and Character Model Data needs and data dependencies UML Spell and Character Model	Samuel Jeffrey Chase
April 25	Write the introduction Write the coordination plan Write the tasking plan	Jeffrey Samuel Chase
April 26	**Planning Document Due**	
April 27		
April 28	GitHub Repository set up.	Jeffrey
April 29	1.2 - Logger complete 1.3 - Error model complete	Samuel Chase
April 30		
May 1		
May 2		
May 3	Task 4.1 & 4.2 – Users done 2.2 & 2.3 – Sign up / Log in	Jeffrey / Chase Samuel

	7.1.2 - Class model done and tested	Chase
May 4	**1.1 - Test Database made and verified for foreign key constraints** Task 8.1.2 - Background	Jeffrey Chase
May 5	Task 6.1.2 - Race	Jeffrey
May 6	Task 3.1-3.4 - Characters (Except character sheet)	Samuel
May 7	**All Models Done and Tested**	
May 8	2.1- Home hbs 5.1- spell list hbs 3.4.1 - character list hbs, 3.5.1- character focus hbs done (not all logic has to be implemented.	Samuel, Jeffrey Jeffrey Samuel Samuel
May 9	2.1.1- Home controller done and tested, 5.1 - Spell controller done and tested for list page, 3.6- Character Controller, not all data needs to be sent	 Jeffrey Samuel
May 10	**Presentation on project status / preliminary demo**	
May 11		
May 12	Task 8 - Background Done Task 6 - Race Done Task 7 – Class Done	Chase Jeffrey Chase
May 13		
May 14	Task 5 – Spells done	Jeffrey
May 15	Task 3 - Character – Character Sheet fully editable Task 4 – User Fully operational with cookies	Samuel Jeffrey/Chase
May 16	**Controllers done and tested** **Views done and tested.**	
May 17	Final touches for aesthetics, Anything that hasn't gotten done / was pushed back.	Team Team
May 18	**Finish The Project**	
May 19		
May 20	**Project Due**	

Timeline made and discussed as a team but authored by Jeffrey

Tasking Plan – Chase, then edited altogether in class

Complete the planning document

Write the introduction, containing inspiration for the project and what problem is being solved.

Complete the timeline. The timeline should contain every task from this section of the document.

Write the coordination plan, describing when and how we will meet, and explain the use of GitHub. It should explain that we will use individual branches for different tasks, not per user.

1- Configuration Tasks

1. Build Test Database

        1. Create the database tables using MySQL in a docker image, all the tables to be made are indicated in the entity relationship diagram in the database design section.
            a. Create the group 1 tables.
            b. Create the group 2 tables.
            c. Create the group 3 tables.
            d. Create the group 4 tables.
        2. Populate the database tables using MySQL in a docker image.
            a. Populate the small tables that will be done manually
                a. Populate the ability table.
                b. Populate the skill table.
                c. Populate the ethics table.
                d. Populate the morality table.
            b. Populate the larger tables using the JSON files gotten from the public GitHub repositories.
                a. Populate the background and background features table.
                b. Populate the class and class features table.
                c. Populate the race and racial traits table.
    2. Build Logger capability
        1. Create the logger file
    3. Build Error Handling Capability

2- Maintain a Home Page

1. Ability to see the home screen
    1. Create the controller for the home page
    2. Create design and view for home page
2. Ability to log in / sign up from the home page
    1. Have a popup form on the home page that shows when the user clicks on a button
    2. Make user automatically sign in if a valid session is present as a cookie
3. Ability to navigate to other pages from the home page
    1. Buttons link to another part of the website

3- Character Maintenance Capability

1. Ability to Add a Character
    1. Have a form with options to select from
    2. Validate the Character being created
    3. Query the Database
2. Ability to Delete a Character
    1. Button to send the proper request
    2. Make sure user who owns the character I logged in
3. Ability to Edit a Character

        1. Have an edit button to edit the form and send the proper request
        2. Ability to Update only the hit points
        3. Ability to edit everything about a character at once
    4. Ability to Retrieve/List Characters
        1. View the list
        2. Get a single character
        3. Get all the characters
        4. Get the characters from a specific user
    5. Ability to see a functional character sheet
        1. Sheet view
        2. Have all the inputs functional
    6. Character Controller

4- User Maintenance Capability

1. Ability to Sign up
2. Ability to log in
    1. Buttons, validation
3. Ability to have a session

5- Spell Maintenance Capability

1. Ability to list all the spells in the database that you have access to.
    1. Create the spells page
    2. Ability to list your personal custom spells
    3. Ability to list the spells from the player's handbook
2. Ability to add your own spell
    1. Ability to add a spell to the database under your user
3. Ability to delete one of your own spells.
    1. A delete button is present in the spells list if the spell is a user-created spell.
    2. Deleting a user deletes all their spells from the database.
4. Ability to view the details of a spell
    1. Clicking on a spell brings you to a page with details about it

6- Race Maintenance Capability

1. Ability to list all races in the database
    1. Create the race list page
    2. Model can get a list of races from the database.
2. Ability to view a specific race in the database
    1. Ability to view details of specific race

7- Class Maintenance Capability

1. Ability to list all classes in the database
    1. Create the class list page which lists all the classes
    2. Model can get a list of classes in the database

2. Ability to view a specific class in the database
    1. Ability to view the details of a specific class

8- Background Maintenance Capability

1. Ability to list all backgrounds in the database
    a. Create the backgrounds list page which lists all the backgrounds.
    b. Model can get a list of classes in the database.
2. Ability to view a specific background from the database.
    a. Create the view to focus on a specific background.

Tasking plan authored by Chase/Samuel/Jeffrey

Coordination plan - Sam

Meetings

Meetings are planned on Mondays after school at 4pm to discuss the deadlines coming up as well as to catch up on how people are managing their tasks.

If there is ever any task to be done in person, we are expected to be available after our Web class on Friday until 4pm.

Responses

Everyone should be reasonably available to answer questions via WhatsApp. A response should be expected within 1-2 hours Monday through Thursday after school. On the weekends a response within the same day is to be expected from at least one teammate.

GitHub

Branches will be based on a feature or task that needs to be completed for example ‘Spells Page’ or ‘Error Model’.

I, Samuel Anderson, hereby agree to this plan.

I, Jeffrey Bringolf, hereby agree to this plan

I, Chase Lipari, hereby agree to this plan

Coordination plan authored by Samuel