Josh Brown

Humans vs Zombies website refactor

Final state of system

Most of the application was implemented but unfortunately I did not have time to implement the weeklong games functionality. I will work on it before next semester starts so that the CU HvZ club can deploy it as their new website. All of the main pages can be visited by any user, including the past weeklong games which dynamically display the statistics from those games. Users can sign up for an account and a confirmation email will be sent to them. Some pages are locked and will redirect if the user is not logged in or is already logged in. Users can also activate their account by clicking on the link sent to them then they signed up.

Not implemented:

- Weeklong game functionality
- Admin page

Patterns implemented

MVC

The MVC pattern is implemented in a majority of the frontend application. Views are kept in the components folder, they will format the data from the models to html for display. Page views are the main views that have controllers. The page controllers talk to the API and deal with making the models and then sending them to the pages so the data is displayed correctly.

Interpreter

The interpreter pattern is implemented through the FormattedText class. This class takes text that is saved in the database and converts it into html that React can render. This is what it can interpret:
BOLD[content] - Formats to tag
LINK[name][link] - Formats to <a> tag
LINK_NEW_TAB[name][link] - Formats to <a> tag that open in a new tab
IMAGE[link] - Formats to <img> tag
IMAGE[link][size %] - Formats to <img> tag and sets the size
[LINE] - Formats to <hr/> tag

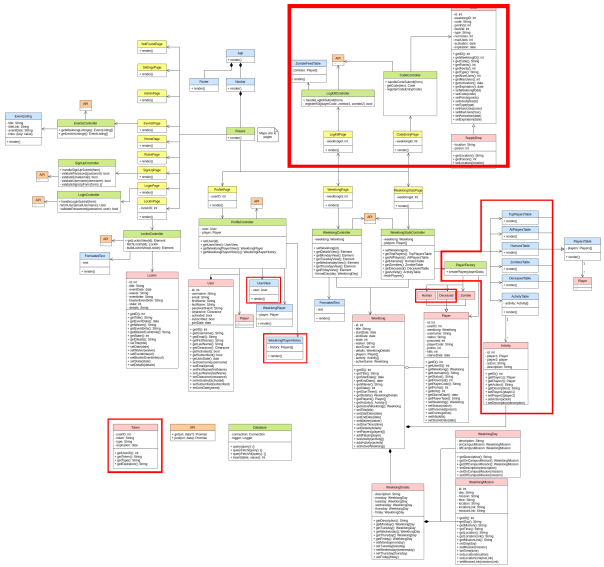
Singleton

The singleton pattern is implemented through the Database class on the backend. I used the singleton class here so that only one connection to the database would be open for each user.

Final Class Diagram and Comparison Statement

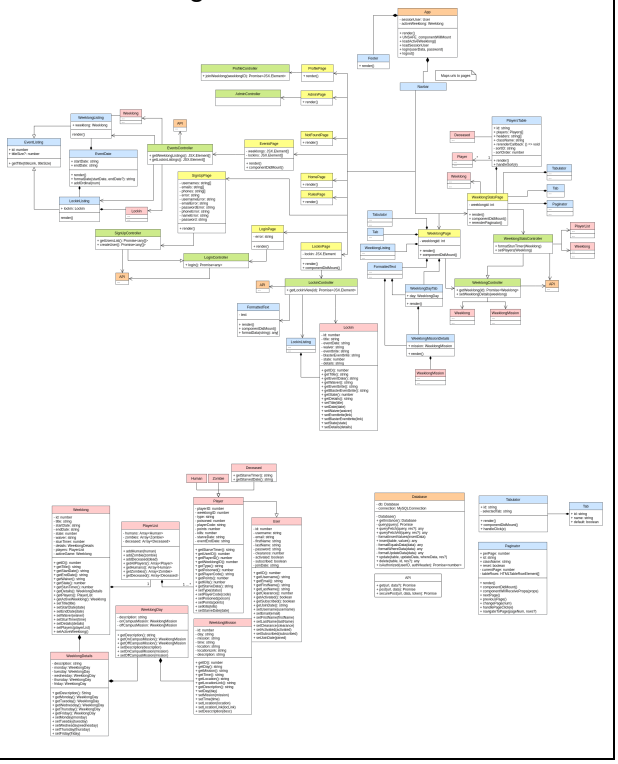
Full pictures of the class diagrams can be found on the github repository.
Color codes for UML diagram:
Green - Controller or Service class
Blue - View class
Yellow - Page view class
Pink - Model class
Orange - Utility class

Initial class diagram

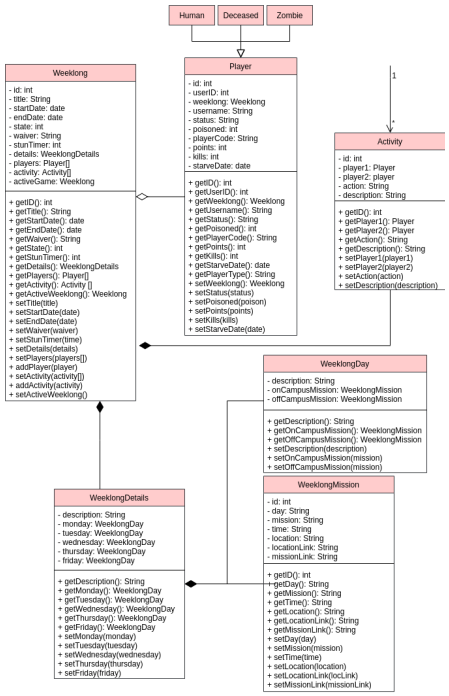
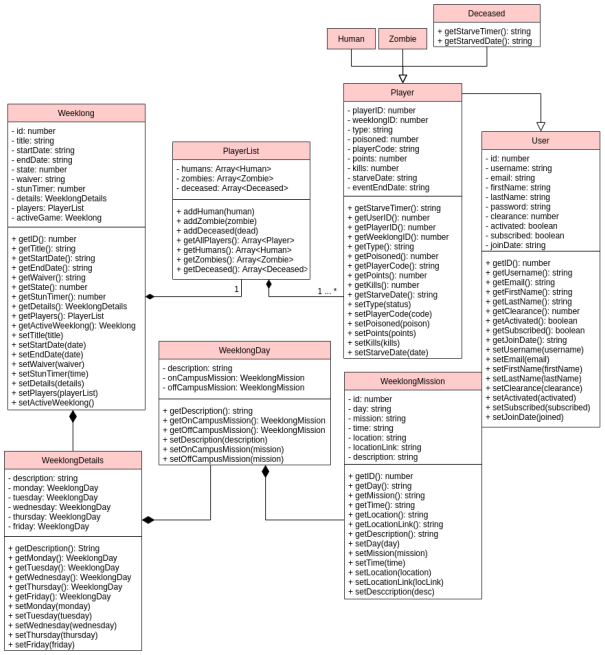


Sections in red boxes are classes that were not implemented either because of time or because they were not needed

Final class diagram

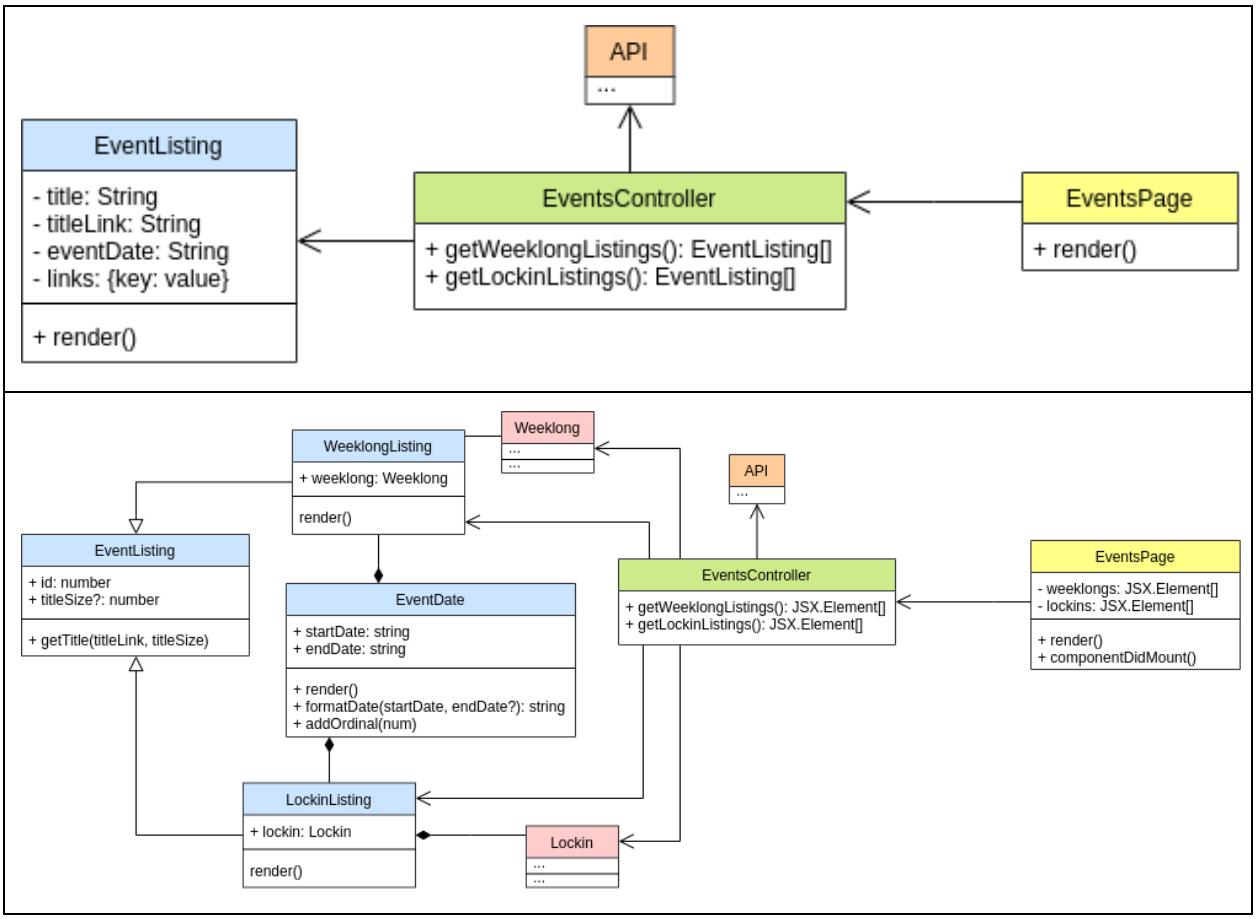


The biggest change in the weeklong classes is that I added the PlayersList class in replacement of having a list of Player objects in the weeklong class. I made this change because I needed to get separate lists of the different player types, so each list is held in the PlayerList class with getter functions for each player type. PlayerList also has a getAllPlayers function that concatenates all the lists and returns it.

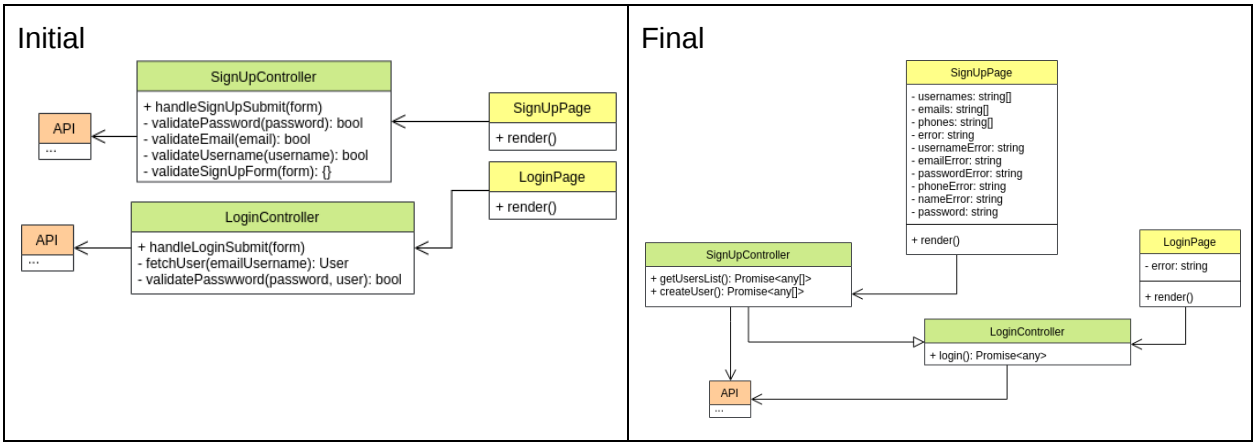
Initial weeklong class structure	Final weeklong class structure
	

Initially the class interactions for the EventsPage was pretty simple. I had to add the Lockin and Weeklong models into the diagram so the view objects could just use the data in the models. I added subclasses for the EventsListing class because some of the information needed to be displayed is different between a Lockin and a Weeklong. I also added an EventDate class that formats an event date into the proper date string.

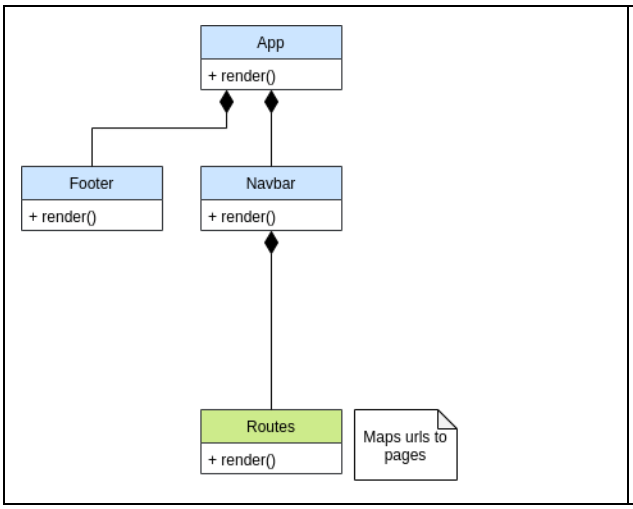
Initial
Final

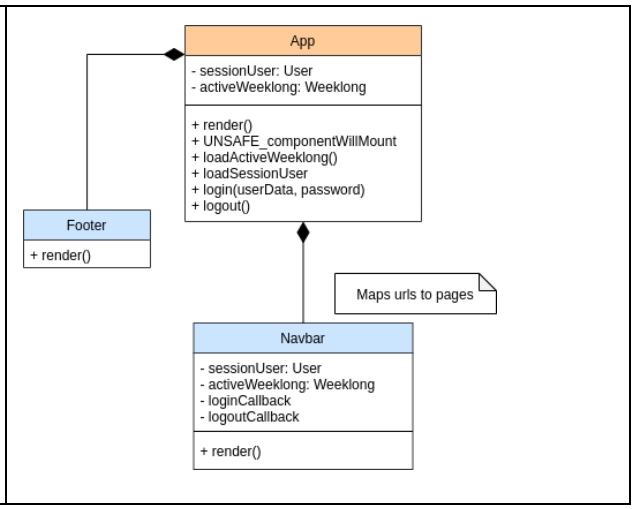


I had the SignUpController extend the LoginController because I realized that when a user signs up then they will be logged in. The SignUpPage class gained all the validating functionality and added variables to store any errors that occur when a user signs up so that it can be displayed to the user.

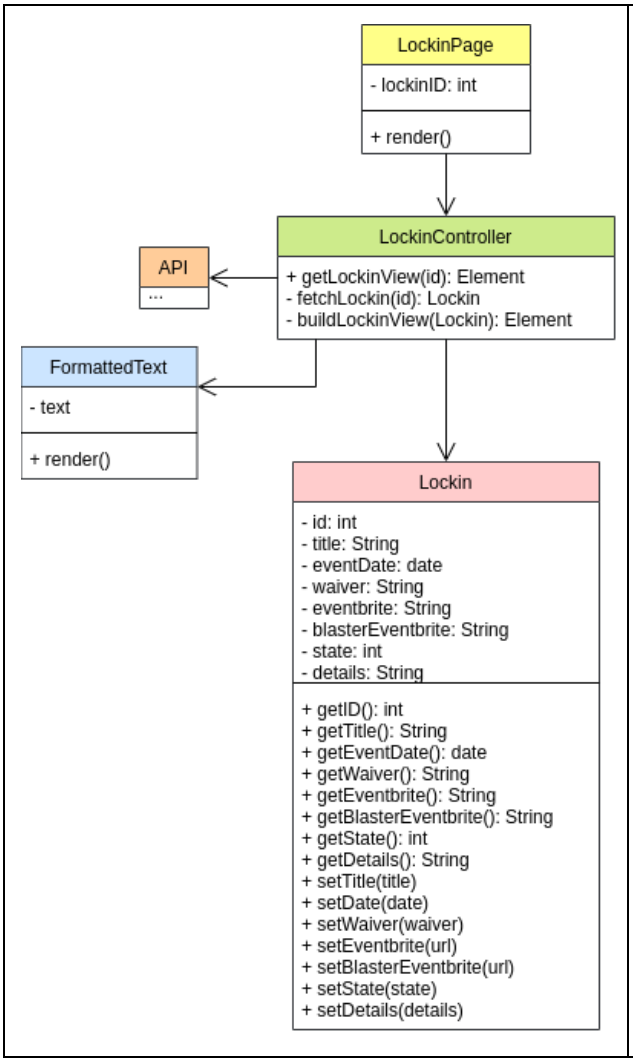


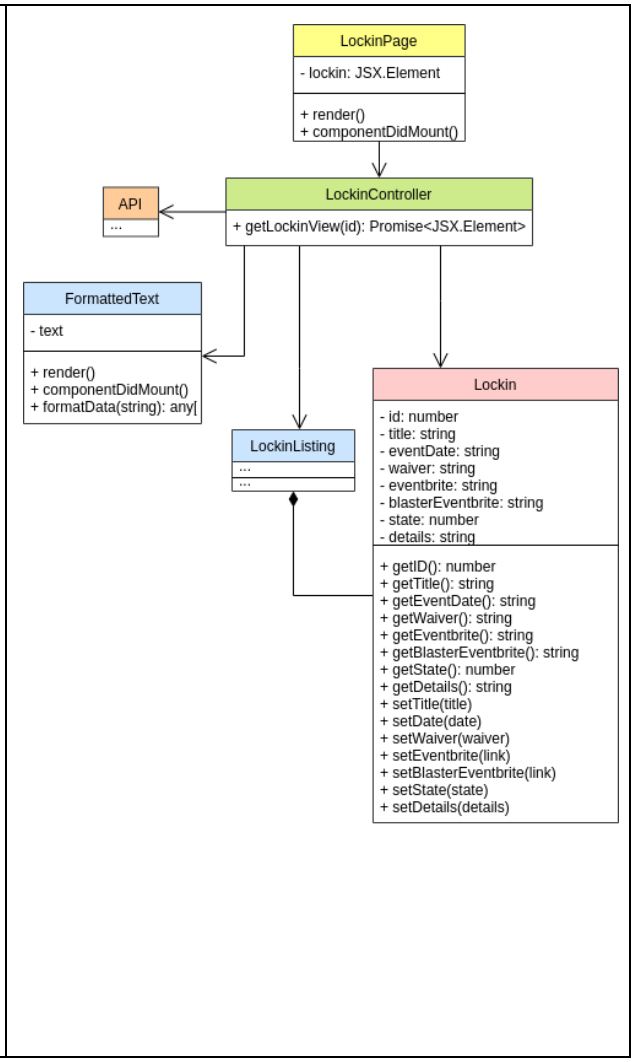
I had to drastically change my plans for the main components of the application. I removed the Routes class and changed the Navbar class to to handle the mapping of urls to page objects so that I could lock certain pages and dynamically change the navbar when a user is logged in. I had to add some functionality to the main App class to handle session information so that a user can log in and out.





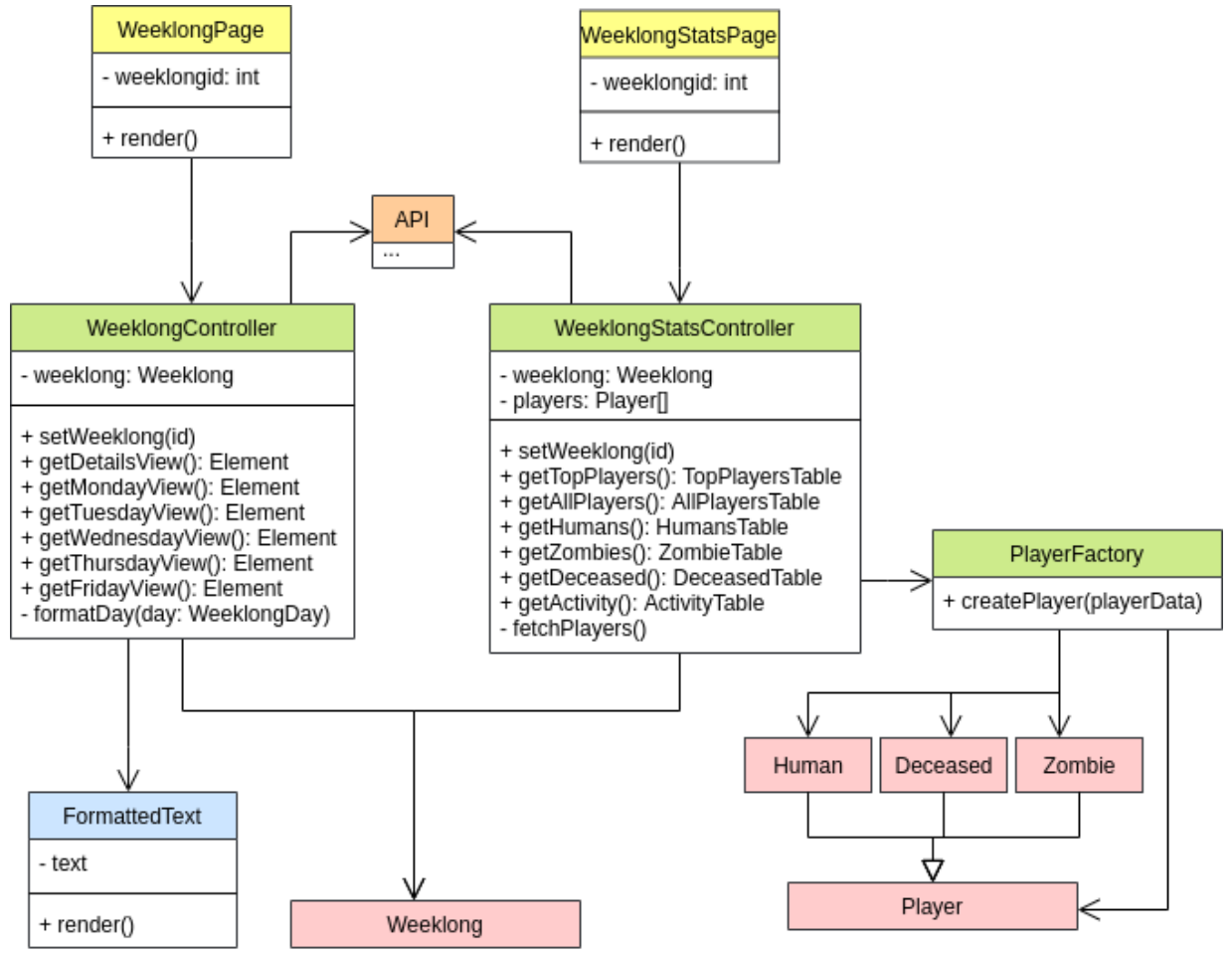
The LockinPage relations all pretty much stayed the same but I added the LockinListing object as a header since it was the same code being used.



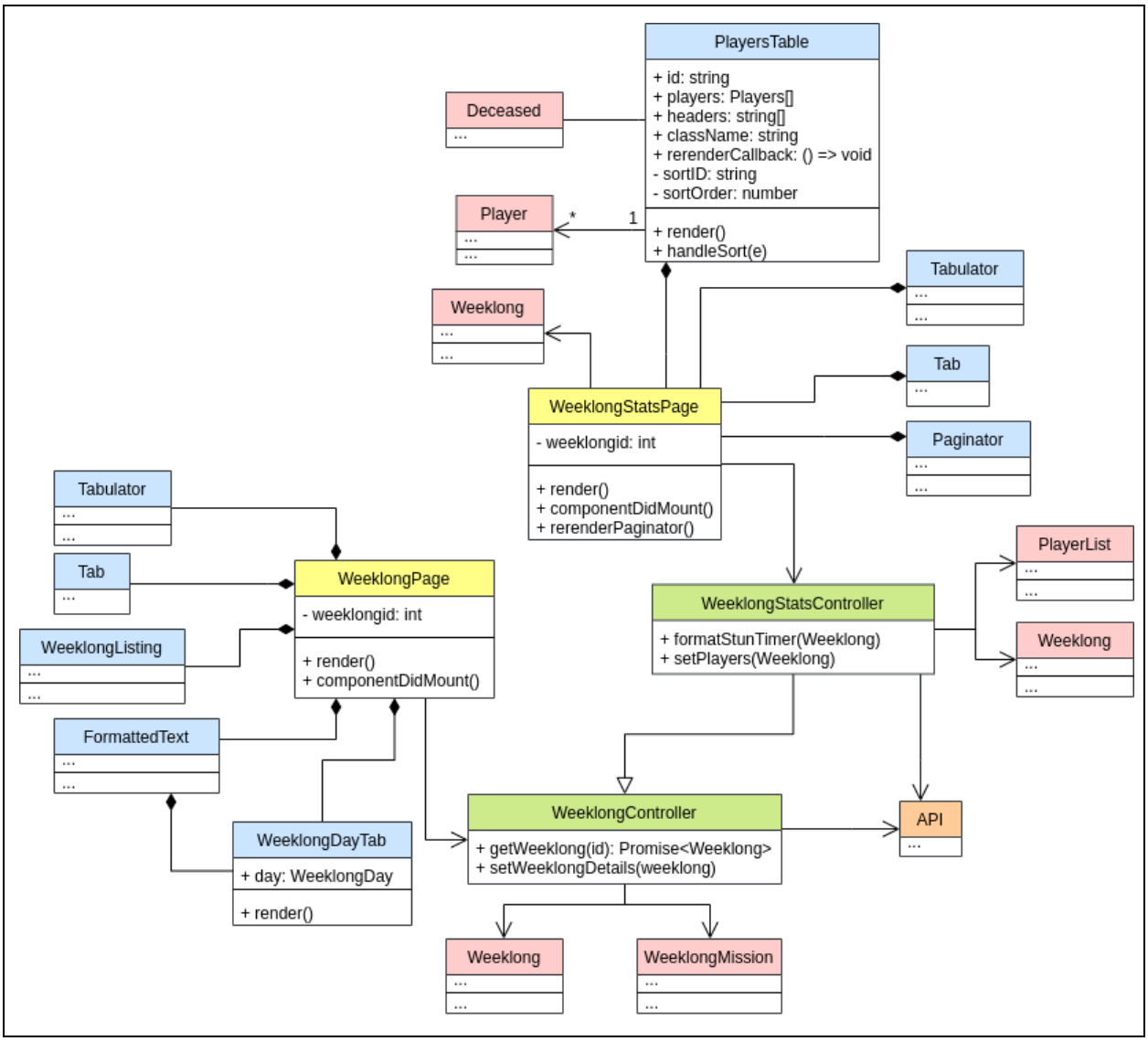


The WeeklongPage and WeeklongStatsPage diagram had some changes. The controllers became much simpler as I decided to pass around the model objects that held all the data. I also created additional view objects to display the information in the same way as the original website does.

Initial

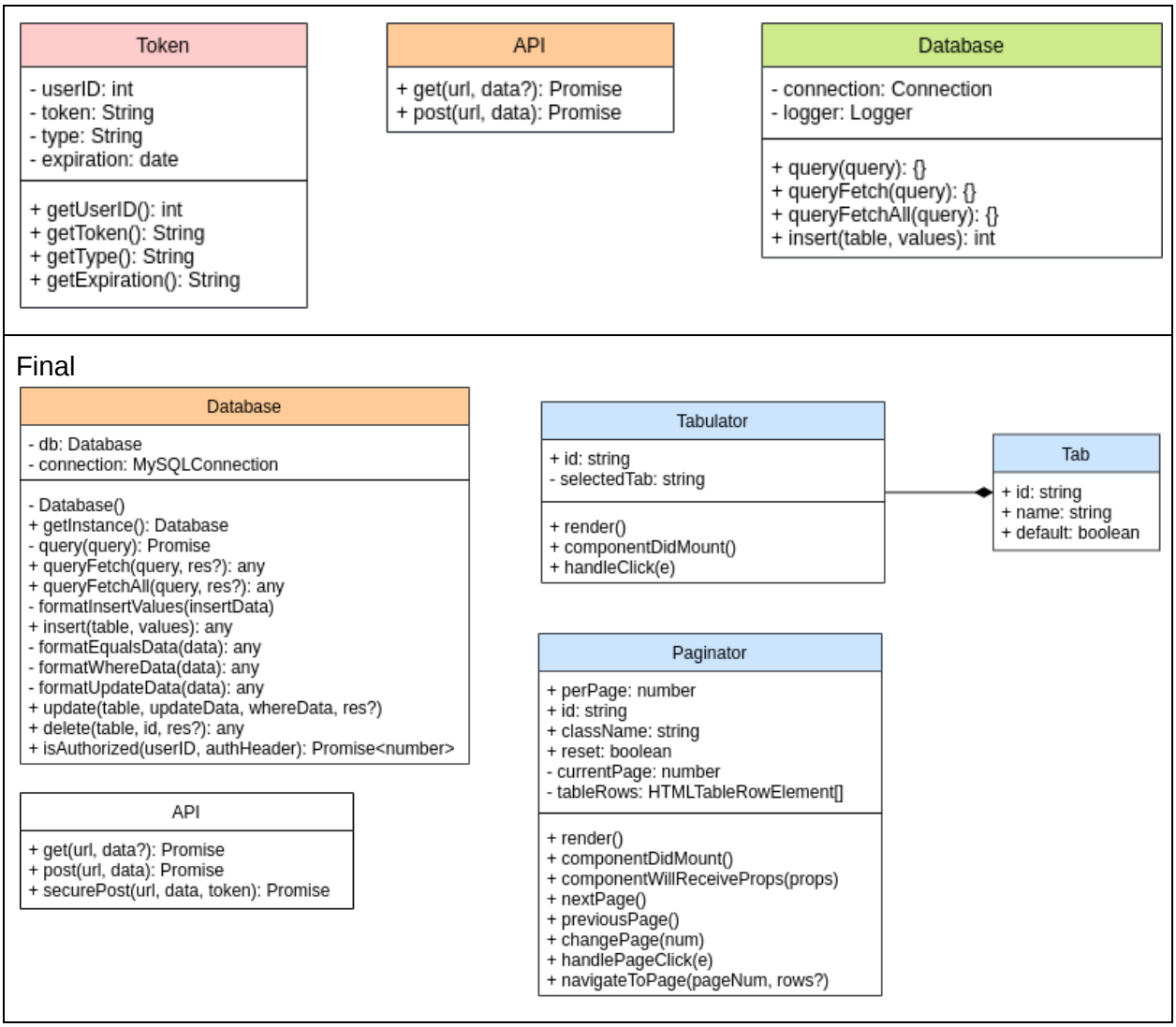


Final



These classes are some of the additional classes that have many interactions with other classes. The Token class was dropped entirely because I never actually needed it. The Database class got a lot more complicated as I had duplicated code that I could pull into separate methods. The Tabulator and Paginator classes are special view classes that help format and display data. The Tabulator class creates a div container with a navbar and tab pages that can be dynamically viewed. The paginator takes a table as a child element and “paginates” the rows of the table.

Initial



Third-Party code vs. Original code Statement

The password encryption class on the backend uses a modified version of Thomas Alrek's node-php-password package that I adapted to work with typescript as it was written for javascript.

Statement on the OOAD process for your overall Semester Project

Well when I first started the coding process I began in plain javascript because I thought trying to work with typescript and Ionic would be more difficult but I soon came to find out that it was not easier to implement the system that I had designed. I had to restart building my

frontend about halfway through project 5 but in typescript and Ionic. I wish I had stuck with the initial design in the beginning and stuck with typescript.

When initially designing the API endpoints I failed to think of many endpoints I would need. I needed to modify the endpoints in order to return data object that could be interpreted by the frontend to build the models.

Coming up with a more object oriented design was difficult as I continued to build the application because the patterns I chose didn't fit how I thought they would. Specifically the Factory pattern. Originally I was going to make a factory that would create the users and the Weeklong object would keep an array of the Player objects created by the factory. Instead, the only time I needed to create a list of Players was in one method of a controller. It wasn't necessary to create a factory because I wasn't creating generic objects, I was loading them. I'm glad I had implemented the Interpreter pattern by accident when creating the FormattedText class.