The application at a high level consists of a website, an Application Programming Interface and a Database. Going deeper into the architecture:

Database

The game will be using Mongodb to store game play data, gamer settings, scoreboard, user accounts, game store products information, player purchases data and transactions made in the Squared store. MongoDb uses Collections which acts like tables in a relational database and documents which acts like a data row in a relational database's table .This will be managed by the Flask server. The goal of this is to offer a simple solution to storing user data and being able to retrieve it at a later time.

Application Programming Interface (API)

The API will be using the Model View Controller (MVC) pattern to organize the Flask API. The Model will contain the attributes of the particular table which will be necessary to map the objects to the Mongodb's document schema which acts like a table in the traditional relational database context. The API won't have a view in the traditional sense, meaning its views will be the Flask endpoints which will be the gateway for external components like the website to communicate with the API and its functionalities. These endpoints will consist of HTTP Post, Get, Put and Delete methods. Some of these end points will be authenticated and use JSON Web Tokens (JWT) to prevent unauthorized access to the API methods. Lastly, the Controller will be where the logic that uses the model to communicate with the database to perform. Create Retrieve Update Delete (CRUD) functionalities, as well as other logic that won't be handled on

the games web interface. This approach of using the API will provide a separation of concerns and a more scalable, maintainable and modular system so that if in the future the team decides to provide a native mobile application. There won't be a need to change the API or database, instead using the existing API and database and just implementing the game board elements and functions specific to the interface. This separation of duties will also allow us to be able to upgrade and employ multiple API servers if need be, without also having to create a new view model with it.

Web/Frontend

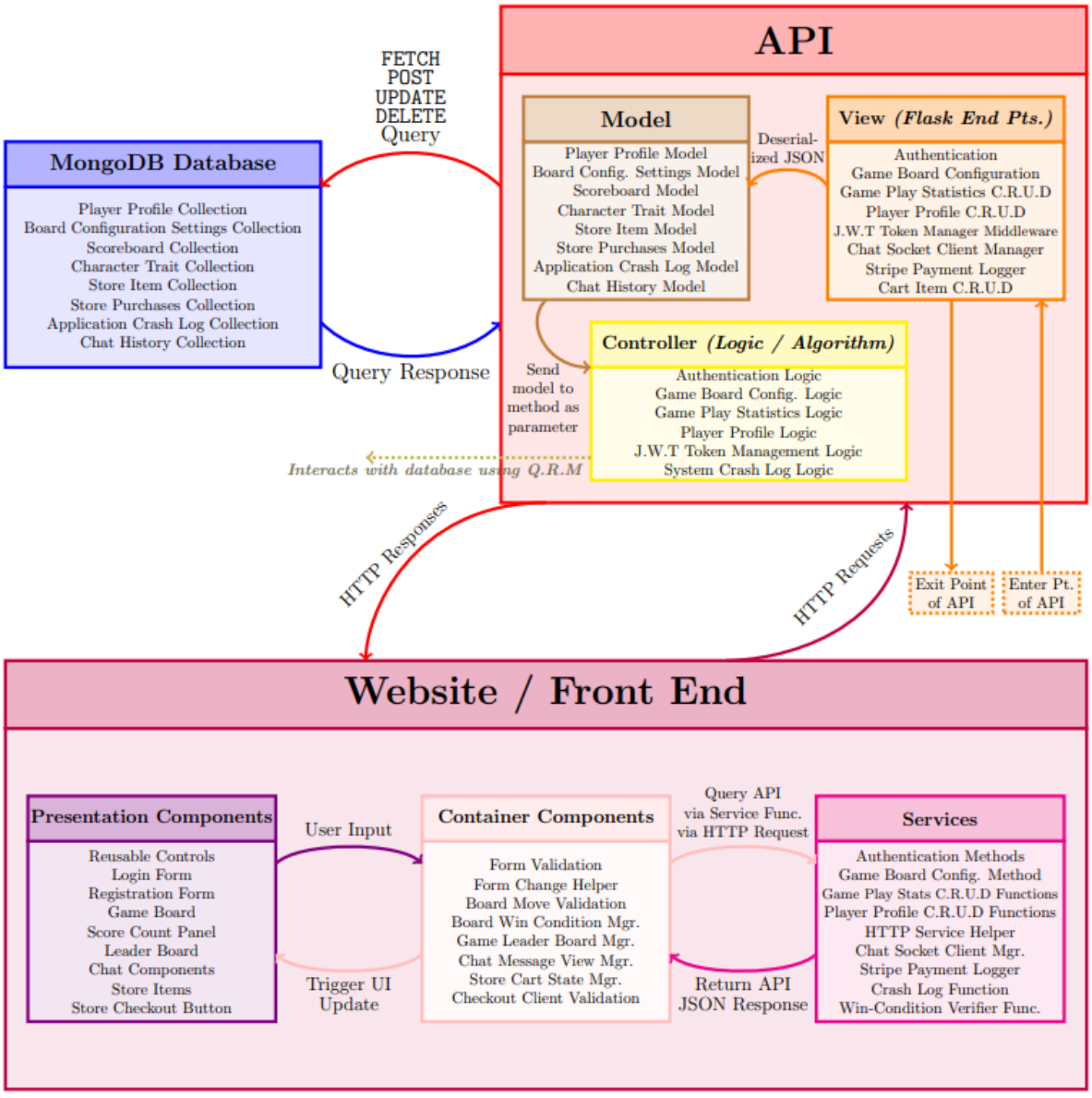
The team will be using the Presentation and Container Component Pattern for the ReactJs web interface that will contain the game board. This is not the traditional architecture discussed in Software Engineering. It is credited as a term invented by Dan Abramov. In this approach the ReactJs website is divided into Presentation Components and Container Components (Bala, 2022).

The Presentation layer can be thought up as the View In MVC. Components can be thought up as being SubViews of a view, similar to the building blocks that make up a wall. The components are responsible for forming the User Interface of the application. There is no dependency between any of the components (Bala, 2022). This means that the presentation layer responsibility is just to display data. The next part of the architecture will be the Container Component. The Container can be thought of as the Controller in the MVC context. Because, the Container Component is responsible for "how things work" (Bala, 2022). These usually consist of class components which contain lifecycle methods. Some of these lifecycle methods include

but are not limited to “componentDidMount” which runs whenever the website’s page is loaded. Another is “componentDidUpdate” which is called after the component is mounted.

The HTML components are placed inside the render method and within the return function which is responsible for returning the Html interface or elements to the browser. ReactJs elements are lightweight objects which makes it light on resources when rendering to the browser. ReactJs has a React DOM which is responsible for updating the Html “DOM to match the React elements” (“Rendering Elements”, n.d). This enables us to employ agile methods in creating our frontend. For example, we can task multiple people to create small, reusable components without their code conflicting, or having to worry about waiting for someone else to finish their code first. This also allows multiple components to easily be updated without having to edit a host of HTML documents. For example, if we have a cookie banner (to comply with GDPR), we can make it into a component and simply add it to every page. Without using a component-based model, we’d have to manually update every single HTML page if we want to consistently update the banner.

Architecture Diagram



References

Bala, S. (2022, January 11). *3 react component design patterns you should know about*. OpenReplay Blog. Retrieved March 6, 2022, from https://blog.openreplay.com/3-react-component-design-patterns-you-should-know-about

Rendering elements. React. (n.d.). Retrieved March 6, 2022, from https://reactjs.org/docs/rendering-elements.html