Group Project Proposal Final Draft

7/23/2023

Project:

Fantasy Football Draft Tool

Team Members:

Brandon Shaffer
Email: bjs397@nau.edu &
GitHub: https://github.com/brandonbjs

Drew Heller
Email: dh2225@nau.edu &
GitHub: https://github.com/dh2225

GitHub Repository:

https://github.com/dh2225/CS312-Fantasy-Draft

React Components:

DraftToolAPI.js: The DraftToolAPI file will be responsible for all of our endpoint functions. Our application will utilize the express and CORS packages to handle routing.

- fetchPlayersEndpoint() function invoked when a user is trying to fetch player data from our MongoDB
- updatePlayerEndpoint() function invoked when a user is trying to update player data in our MongoDB
- addPlayerEndpoint() function invoked when a user is trying to add player data to our MongoDB
- deletePlayerEndpoint() function invoked when a user is trying to delete player data from our MongoDB

DraftToolService.js: responsible for all of the application's service functions

- Import PlayerModel from Models folder.
- addPlayer asynchronous function used to create/add a new player to the database.
- fetchPlayers asynchronous function used to fetch all undrafted/available players.
- updatePlayers asynchronous function used to update a player based on id, setting their status to false(unavailable) and updating their manager field to the team that drafted the player.
- findPlayerByID helper function used to match a parameter ID to an player ID in the database

PlayerModel.js: responsible for mapping out how each document will look in our MongoDB

- Define PlayerSchema to structure the document (mongoose.Schema).
    - Id - required, type number, unique
    - Adp - required, type number
    - Name - required, type string
    - Position - required, type string
    - Team - required, type string
    - Bye - required, type number
    - Manager - type string, default null
    - Status - type boolean, default true
- Define PlayerModel to define a model and collection (mongoose.model).
    - Collection name players
    - Use PlayerSchema

- Export the PlayerModel

connect.js: responsible for establishing connection to mongodb

- Import mongoose.
- Set variable for serverName, databaseName, and URL
- Define database class that establishes connection to the MongoDB database.

App.js: This is the main component that contains the logic and state management of our single-page website. The App.js component will render our DraftBoard.js component, AvailablePlayerList.js component, and our TeamManagement.js component accordingly. We will also pass any required states/functions as props to these components from App.js.

DraftBoard.js: This component will be responsible for properly displaying the live draft board to the draft host by making GET requests using our “/fetchPlayers” route, which invokes our fetchPlayersEndpoint() and our fetchPlayers() service function. As players are drafted, their “status” will be updated from true to false, using our “/updatePlayer” route, updatePlayerEndpoint() endpoint function and updatePlayer() service function.

AvailablePlayerList.js: This component will be responsible for displaying all the available players remaining in the draft and their relevant statistics. This component will achieve this by making GET requests through our NodeJS API. Players who have had their “status” changed from true to false should not be visible in this section unless the user is utilizing advanced filtering to view the players that have already been drafted.

TeamManagement.js: This component will act as a way for the draft host to manually interact with individual teams. As the draft goes on, the draft host should be able to switch between teams and view their current roster, as well as execute updates on the selected team such as adding a new player.

Functions and Logic:

fetchPlayersEndpoint();

- This endpoint function will be invoked when this route is used: “/fetchPlayers”
    - try: invoke the fetchPlayers() service function which attempts to retrieve data from our MongoDB
    - 200 response upon success
    - catch: Error getting players from DB
    - 500 response upon failure

fetchPlayers();

- Asynchronous function that has no parameters.
- Set availablePlayers variable to the response of a find() function finding players with “status” field being true(available).
- Return availablePlayers.

updatePlayerEndpoint();

- This endpoint function will be invoked when this route is used: “/updatePlayer”
    - try:
        - verify id passed as parameter is valid and exists by invoking the findPlayerByID() helper function
        - invoke the updatePlayer() service function which attempts to update data in our MongoDB
    - 200 response upon success
    - catch: Error updating player in DB
    - 500 response upon failure

updatePlayer();

- Asynchronous function that has id and manager as parameters.
- Set player variable to the response of findPlayerByID().
- Set player.manager to the passed manager.
- Set player.status to false - showing player as unavailable.
- Save player to database.

addPlayerEndpoint();

- This endpoint function will be invoked when this route is used: “/addPlayer”
    - try:

        - verify that the user passed all the required fields
            - Player being added must have id, adp, name, position, team, bye, manager, status
        - invoke the addPlayer() service function which attempts to add data to our MongoDB
    - 200 response upon success
    - catch: Error adding player to DB
    - 500 response upon failure

addPlayer();

- Asynchronous function that receives id, adp, name, position, team, bye, manager, and status as parameters.
- Create new PlayerModel document - passing the received parameters to the PlayerModel constructor.
- Await and save the player to the database.

deletePlayerEndpoint();

- This endpoint function will be invoked when this route is used: “/deletePlayer”
    - try:
        - verify id passed as parameter is valid and exists by invoking the findPlayerByID() helper function
        - invoke the deletePlayer() service function which attempts to delete data from our MongoDB
    - 200 response upon success
    - catch: Error deleting player from DB
    - 500 response upon failure

hardDeletePlayer();

- Asynchronous function that receives an id as the sole parameter.
- This function utilizes the deleteOne() method on our PlayerModel with the id as the filter field.

findPlayerByID();

- This async function is simply a helper function that accepts an id as a parameter
- It uses the findOne() method on our PlayerModel to match the parameter id to a player id in our MongoDB.

componentDidMount();

- This lifecycle method will be used to initially populate our page with our MongoDB data.

componentDidUpdate();

- This lifecycle method will be used when the user makes updates to player status during the drafts themselves.

handleChange();

- This is a handler function that will handle any field changes that might occur in our application. An example of this would be when a team manager wants to add a player to their team using a submit button or a drop down menu selection. We must have a handler to handle these field changes.

handleSubmit();

- This is a handler function that will handle when a user clicks a button and submits a data change. The most likely scenario would be when a team manager wants to end their turn and add a player to their team. We need a handler to handle the button press the user makes when they select their player.

API Request-Response Formats:

GET - In order to fetch player documents from our MongoDB, we will need to utilize the GET request-response format in our DraftToolAPI.js file. The website will need to implement a number of GET requests during one page render in order to populate the live draft board with players as they are drafted and the available players list.

POST - We wish for our application to satisfy all CRUD capabilities, therefore we are including the ability to add players to our MongoDB playerbase for developer purposes. We will need to utilize the POST request-response format in our DraftToolAPI.js file. The user will never have the ability to add new players to and from the database as the database will already be populated with all the active NFL players. Users will only ever get player data and update player data. Therefore, this request-response format will only be used by developers during the testing phase of the application.

PUT - Other than making get requests, the user will also be making a large number of PUT requests during the duration of a draft. Every time an available player is drafted to a fantasy team, their “manager” field will be updated to the team name of the current drafting manager and their “status” field will be set to false. This indicates that the player is no longer available and allows them to be viewed in their respective team on the live draft board.

DELETE - We will never need to implement the DELETE request-response format because our application does not need to perform any hard deletions. In the event a player is no longer an active player in the NFL, we can simply utilize a PUT method to change the player's “status” from “Active” to their current status. In this way, we will implement a form of soft-deleting the players from our MongoDB rather than hard-deleting them. However, we will still include a deletion endpoint for possible developer use only.

Endpoint Routes:

```
app.use("/fetchPlayers", fetchPlayersEndpoint);
```

Our first route will be the “/fetchPlayers” route which will invoke the fetchPlayersEndpoint(). This route is responsible for making a GET request to our MongoDB. The endpoint invoked will call the fetchPlayers() service method to fetch all the players from our database or throw errors and console messages upon failure.

```
app.use("/updatePlayer", updatePlayerEndpoint);
```

Our second route will be the “/updatePlayer” route which will invoke the updatePlayerEndpoint(). This route is responsible for making a PUT request to our MongoDB. The endpoint invoked will call the updatePlayer() service method which takes an id as a parameter and updates the selected player’s “manager” and “status” or throws errors and console messages upon failure.

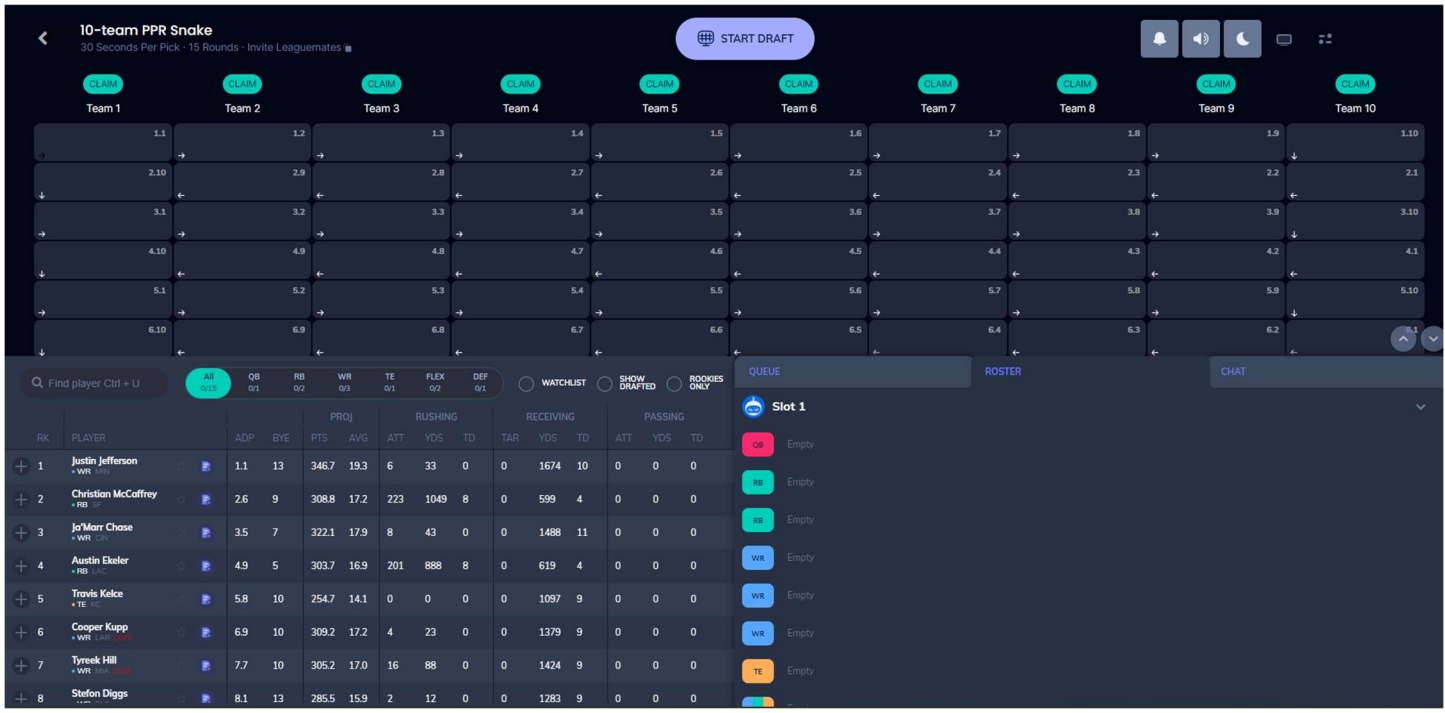
```
app.use("/addPlayer", addPlayerEndpoint);
```

Our third route will strictly be used by developers during the development phase. We are populating our own MongoDB with player data, therefore to ensure we have a smooth experience handling our database, we are implementing an “/addPlayer” endpoint. The developer must explicitly include all the fields in the JSON body (other than ID, which MongoDB generates for us by default) when using this endpoint. This endpoint will invoke the updatePlayer() service method.

```
app.use("/deletePlayer", deletePlayerEndpoint);
```

Our last route will also strictly be used by developers during the development phase. We are populating our own MongoDB with player data, therefore to ensure we have a smooth experience handling our database, we are implementing a “/deletePlayer” endpoint. This endpoint will invoke the hardDeletePlayer() service method which takes a single id parameter and uses the deleteOne() method to hard delete a player from the database. To reiterate, this hard deletion method will only be used during the development phase and will not be a public endpoint.

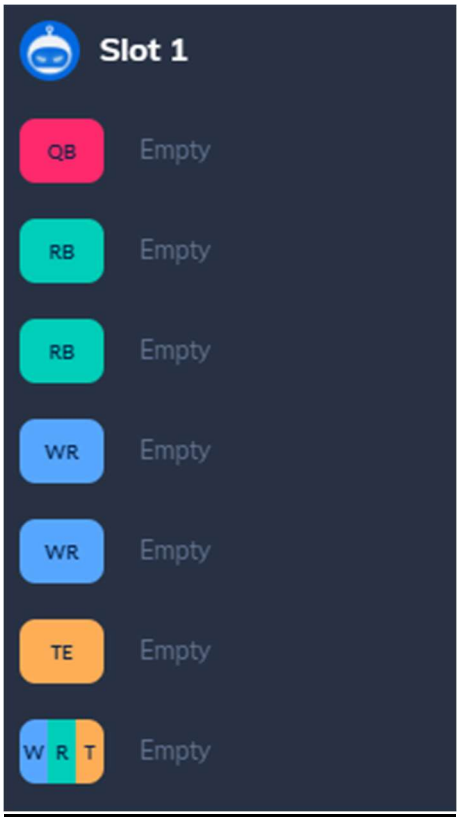
Design References:



Draft Tool Inspiration

Draft Board - The draft board will be in grid form showing each of the team names at the top of their respective columns. Underneath the team names will be the team's selections. On the draft board will be an indicator that shows which team is currently picking as well as a timer. At the expiration of the timer, if the team manager has not made a selection, the top adp player will be selected.

Player Board - The player board will also be in grid format. The default view will show all players in order according to their average draft position or ADP. On the far left of each player row will be a button that allows the current manager to make a player selection. In the header section of the player board, there will be filtering functionality that will allow the user to filter the list according to player position.



Team Management Board - The team management board will allow the user to select any team and see what positions have been filled on their roster. The design will have tabs or possibly a drop down list that will allow the desired team to be displayed.