FANTASY FOOTBALL
DRAFT TOOL

Phase-2 Deliverable Report

08/06/2023

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

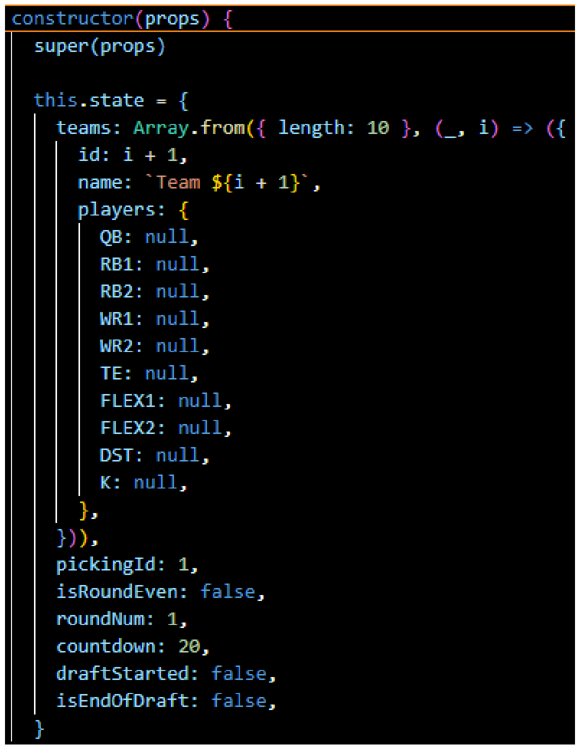
GitHub Repository:

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

Front-End:

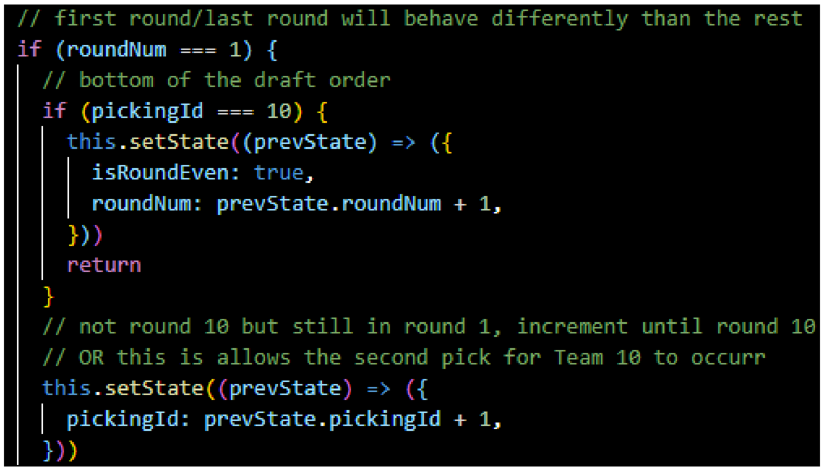
Deliverable #1 - Write React Code: The front-end react app utilizes 4 components - App.js, DraftBoard.js, AvailablePlayerList.js, and TeamManagement.js.

- **Implementation of App.js:** App.js is the main component that contains the logic and state management of our single-page website.
    - Changes made since Phase 1: App.js changed a considerable amount in Phase 2 compared to Phase 1. After spending more time with the project, it became clear that the draft state was not going to be able to be managed in DraftBoard.js. This is because many aspects of the draft state had to be checked and monitored all around the application, in multiple components. Therefore, to assure there were no issues changing state, reading state and managing the draft, we moved all of the state out of DraftBoard.js and into App.js, as well as most of the handler functions that altered the state.
    - Challenges: Initially, we ran into a lot of problems trying to get the draft logic to work like a proper snake draft. The difficult part was ensuring that Team 1 and Team 10 got 2 picks in a row as long as the draft was not on round 1 or round 10. Once the draft logic was figured out, we realized that the other two components being rendered were going to need to have access to much of the draft state that was being manipulated in DraftBoard.js. Since AvailablePlayerList.js and TeamManagement.js are not children of DraftBoard.js, it made sense to move all the state and their accompanying functions out of DraftBoard.js and into App.js (now a class component) and then pass them as props to the components that needed them.
    - App.js utilizes a constructor to maintain the state of the array of teams which includes the team id, name, and players. The component also maintains the state of the pickingId, isRoundEven, roundNum, countdown, draftStarted, and isEndOfDraft.



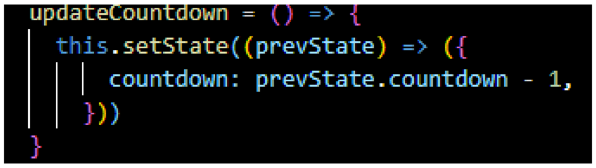
```
constructor(props) {
  super(props)

  this.state = {
    teams: Array.from({ length: 10 }, (_, i) => ({
      id: i + 1,
      name: `Team ${i + 1}`,
      players: {
        QB: null,
        RB1: null,
        RB2: null,
        WR1: null,
        WR2: null,
        TE: null,
        FLEX1: null,
        FLEX2: null,
        DST: null,
        K: null,
      },
    })),
    pickingId: 1,
    isRoundEven: false,
    roundNum: 1,
    countdown: 20,
    draftStarted: false,
    isEndOfDraft: false,
  }
```

- App.js has 10 functions that allow for the management of the fantasy football draft.
  - **updatePickingId:** Manages which team is picking and draft logic
    - Has no parameters
    - Utilizes nested conditionals to handle different scenarios in a snake draft such as the first and last rounds, or even/odd rounds.
    - Ensures that the pickingId, isRoundEven, and roundNum variables are updated so that the draft can progress in a snake draft order.
    - See snippet below to see the conditional in updatePickingId that handles the progression of the first round.



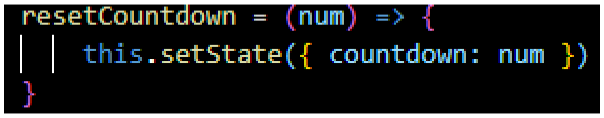
```
// first round/last round will behave differently than the rest
if (roundNum === 1) {
  // bottom of the draft order
  if (pickingId === 10) {
    this.setState((prevState) => ({
      isRoundEven: true,
      roundNum: prevState.roundNum + 1,
    }))
    return
  }
  // not round 10 but still in round 1, increment until round 10
  // OR this is allows the second pick for Team 10 to occurr
  this.setState((prevState) => ({
    pickingId: prevState.pickingId + 1,
  }))
```

- **updateCountdown:** Decrements the countdown timer.
  - Uses setState and prevState to accurately decrement the countdown at the set interval.



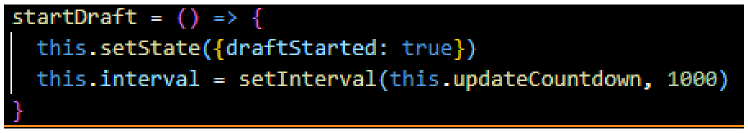
```
updateCountdown = () => {
  this.setState((prevState) => ({
      countdown: prevState.countdown - 1,
    }))
}
```

- **resetCountdown:** Resets the countdown timer
  - Has num parameter which is the number of seconds that the countdown is reset to.
  - Sets the state of countdown variable to num parameter.



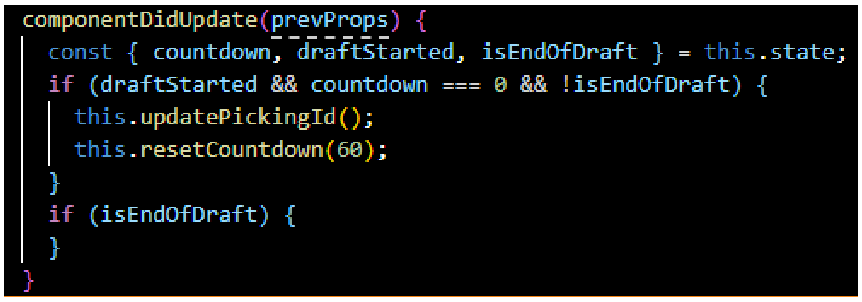
```
resetCountdown = (num) => {
    this.setState({ countdown: num })
}
```

- **componentDidMount:** Component Lifecycle Method
  - Sets initial value of this.interval that is used to manage the countdown timer once the draft is started.
- **componentWillUnmount:** Component Lifecycle Method
  - Uses clearInterval function to stop the interval that was setup for the countdown timer.
- **startDraft:** Starts the draft process
  - Updates the state of the boolean variable draftStarted to true.
  - Uses setInterval function to call updateCountdown function, setting function to be called every 1000 milliseconds.



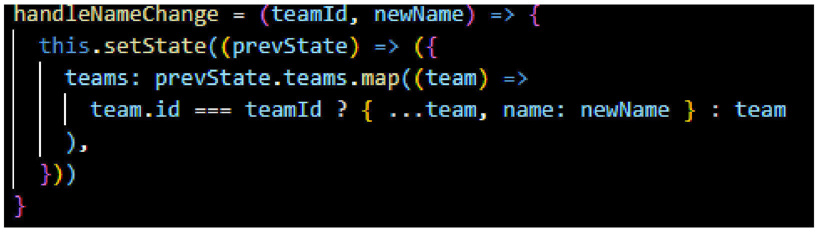
```
startDraft = () => {
  this.setState({draftStarted: true})
  this.interval = setInterval(this.updateCountdown, 1000)
}
```

- **componentDidUpdate:** Component Lifecycle Method used to manage draft flow and update draft order.
  - If the draft has started, the countdown is 0, and the draft has not ended, the picking id will be updated and the countdown will be resetted. Essentially skipping that player’s turn. The player will end the draft with an empty spot on the roster due to the skipped turn.
  - If the draft has ended, do nothing. Preventing weird interactions that occur when components are re-rendered.



```
componentDidUpdate(prevProps) {
  const { countdown, draftStarted, isEndOfDraft } = this.state;
  if (draftStarted && countdown === 0 && !isEndOfDraft) {
    this.updatePickingId();
    this.resetCountdown(60);
  }
  if (isEndOfDraft) {
  }
}
```

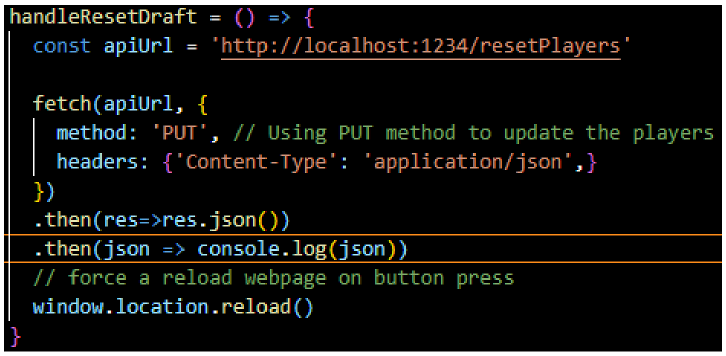
- **handleStartDraft: Starts Draft - Passed as prop to Draftboard.js**
  - Invokes startDraft function.
- **handleNameChange:** Updates name of team upon handleTeamNameClick() in Draftboard.js
  - Utilizes setState, prevState, and map function to update the name in the state teams array.



```
handleNameChange = (teamId, newName) => {
  this.setState((prevState) => ({
    teams: prevState.teams.map((team) =>
      team.id === teamId ? { ...team, name: newName } : team
    ),
  }))
}
```

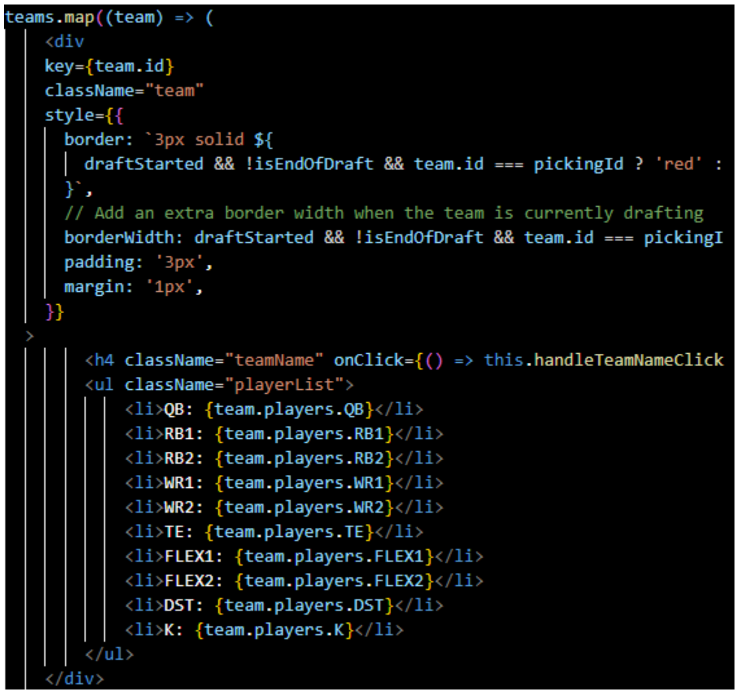
- **render:** The App.js component renders the other components - DraftBoard.js, AvailablePlayerList.js, and TeamManagement.js. Passes appropriate props and functions to each component.

- **Implementation of DraftBoard.js:** DraftBoard.js is a class component that is responsible for displaying the live draft board and functionality for resetting the draft.
    - DraftBoard.js utilizes a constructor that manages the state of the individual team colors. These are randomly generated per team id.
    - Changes made since Phase 1: DraftBoard.js went through considerable changes from Phase 1 to Phase 2. Originally, we had all the draft states being handled by DraftBoard.js, however, as we continued to code the application, it became apparent that other components were going to need access to the draft states and their methods. This lended us to refactoring DraftBoard.js and moving the state and their functions to App.js. We then passed the state and methods as props to the components as needed. After this change was made, DraftBoard became responsible for properly displaying all 10 teams and their players as well as the draft buttons and the countdown.
    - Challenges: We had a lot of trouble with the draft state and the accompanying methods. As we started working on the other components of the application, we were encountering errors trying to keep the draft countdown and the pickingId up to date. Once we moved it all into App.js and passed the state and method as props, everything began to work as expected.
    - DraftBoard.js has 2 functions allowing for the display and reset of the draft.
        - **getRandomColor:** Generates random ASCII color to be used for team borders.
        - **handleResetDraft:** Resets the draft.
            - Invokes the resetPlayers endpoint.
            - Fetches using reset endpoint url and put method updating the status and manager of all players in the database.
            - Reloads the webpage.



```
handleResetDraft = () => {
  const apiUrl = 'http://localhost:1234/resetPlayers'

  fetch(apiUrl, {
    method: 'PUT', // Using PUT method to update the players
    headers: {'Content-Type': 'application/json',}
  })
  .then(res=>res.json())
  .then(json => console.log(json))
  // force a reload webpage on button press
  window.location.reload()
}
```

- **render:** Renders the startDraftButton, resetDraftButton, end of draft message, and teams (with colored borders).
    - If draft is not started or finished, display the start draft button and reset button.
    - If draft is started but not finished, display current round, picking team, and countdown.
    - If draft is over, display draft is over message.
    - Uses map function to display each team.
        - Uses inline css styling to customize the border colors. Red for picking team and teamColor if not picking.
        - Lists all players on each team by position
        - The below snippet shows teams.map

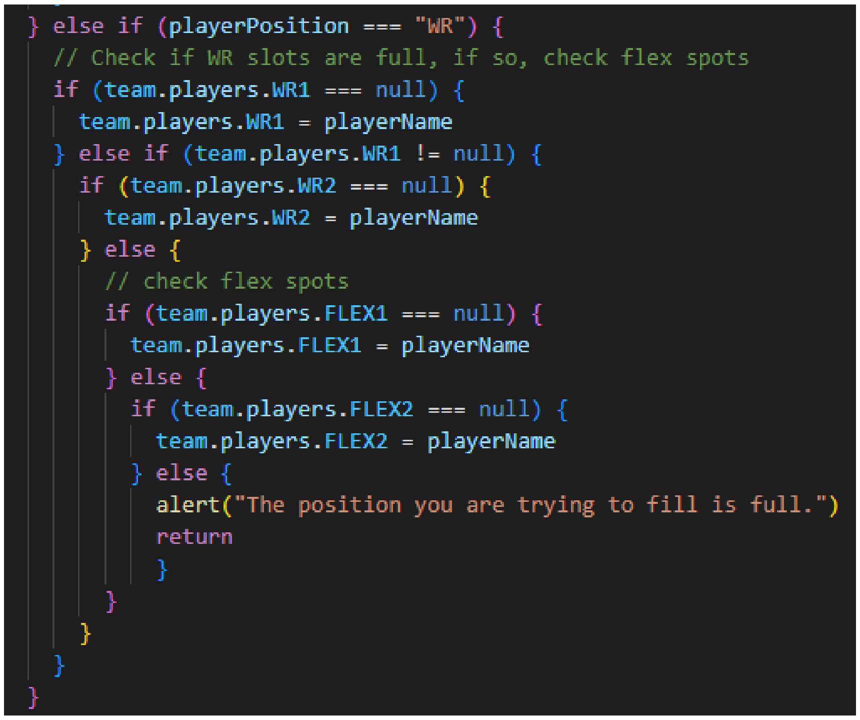


```
teams.map((team) => (
    <div
    key={team.id}
    className="team"
    style={{
      border: `3px solid ${
        draftStarted && !isEndOfDraft && team.id === pickingId ? 'red' :
      }`,
      // Add an extra border width when the team is currently drafting
      borderWidth: draftStarted && !isEndOfDraft && team.id === pickingI
      padding: '3px',
      margin: '1px',
    }}
  >
        <h4 className="teamName" onClick={() => this.handleTeamNameClick
        <ul className="playerList">
            <li>QB: {team.players.QB}</li>
            <li>RB1: {team.players.RB1}</li>
            <li>RB2: {team.players.RB2}</li>
            <li>WR1: {team.players.WR1}</li>
            <li>WR2: {team.players.WR2}</li>
            <li>TE: {team.players.TE}</li>
            <li>FLEX1: {team.players.FLEX1}</li>
            <li>FLEX2: {team.players.FLEX2}</li>
            <li>DST: {team.players.DST}</li>
            <li>K: {team.players.K}</li>
        </ul>
    </div>
```

- **Implementation of AvailablePlayerList.js:** AvailablePlayerList.js is a function component responsible for rendering and displaying all the players with a status that evaluates to true. That means that the player has not been touched by the updatePlayer endpoint or service functions and that the player is free to draft by the next team in the draft. Shows relevant draft data such as ADP, position, player name, bye, and team.
    - Changes made since Phase 1: Originally, this component rendered the available players in a table using grid to style and organize. However, since we decided that our users should have the ability to search, sort and

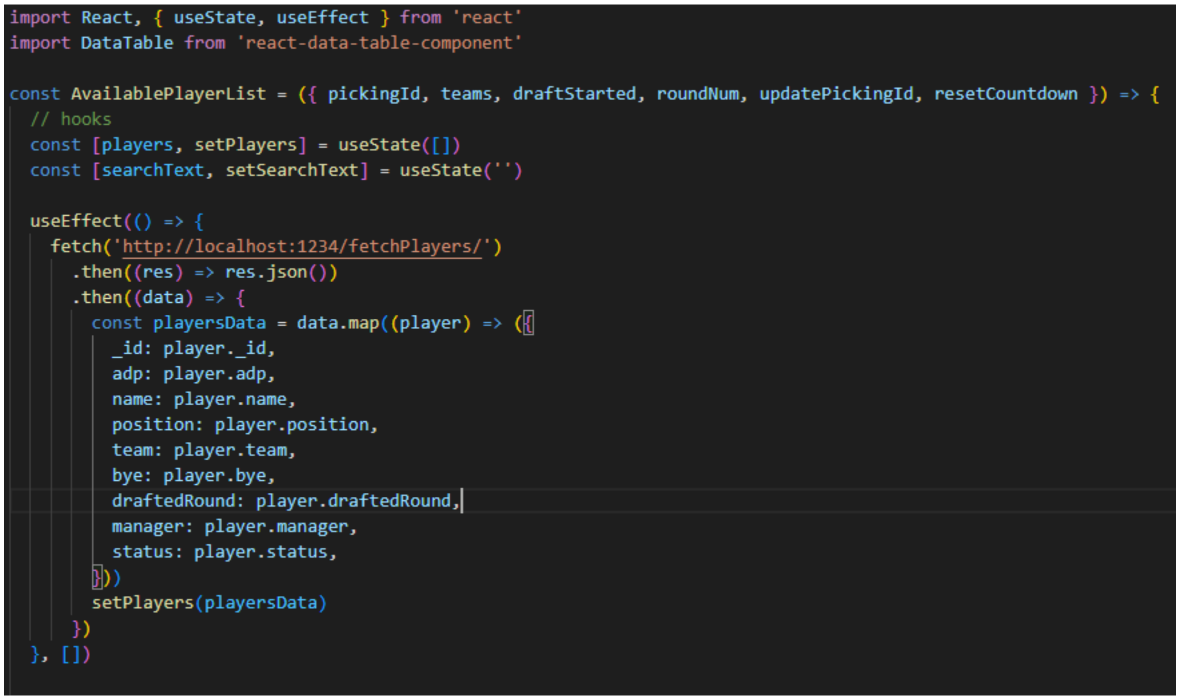
paginate their players, we implemented a package called “react-data-table-component” that has built in sorting and pagination. From there we created a custom search function that uses the built in filter() and includes() functions to search through the players useState() hook. We also added the button that allows the user to add a player to the team that is currently drafting.

- Challenges: It was not hard to implement the “react-data-table-component” package as invoking the DataTable custom html tag is straightforward. We did have some trouble getting the “Add Player” button to work properly along with its accompanying functions and methods. The hardest part was actually coding all the conditions that needed to be checked every time a player was added to a new team. Fantasy Football teams have a number of open slots per position, as well as a number of flex spots that can hold any type of player. This meant that as the user clicks the “Add Player” button, the players should be dynamically added to the team in the next available position or be placed in an open flex spot. We have included the conditions that were checked when a wide-receiver is attempted to be added to a team in order to better display what had to be checked for each player to be added properly.



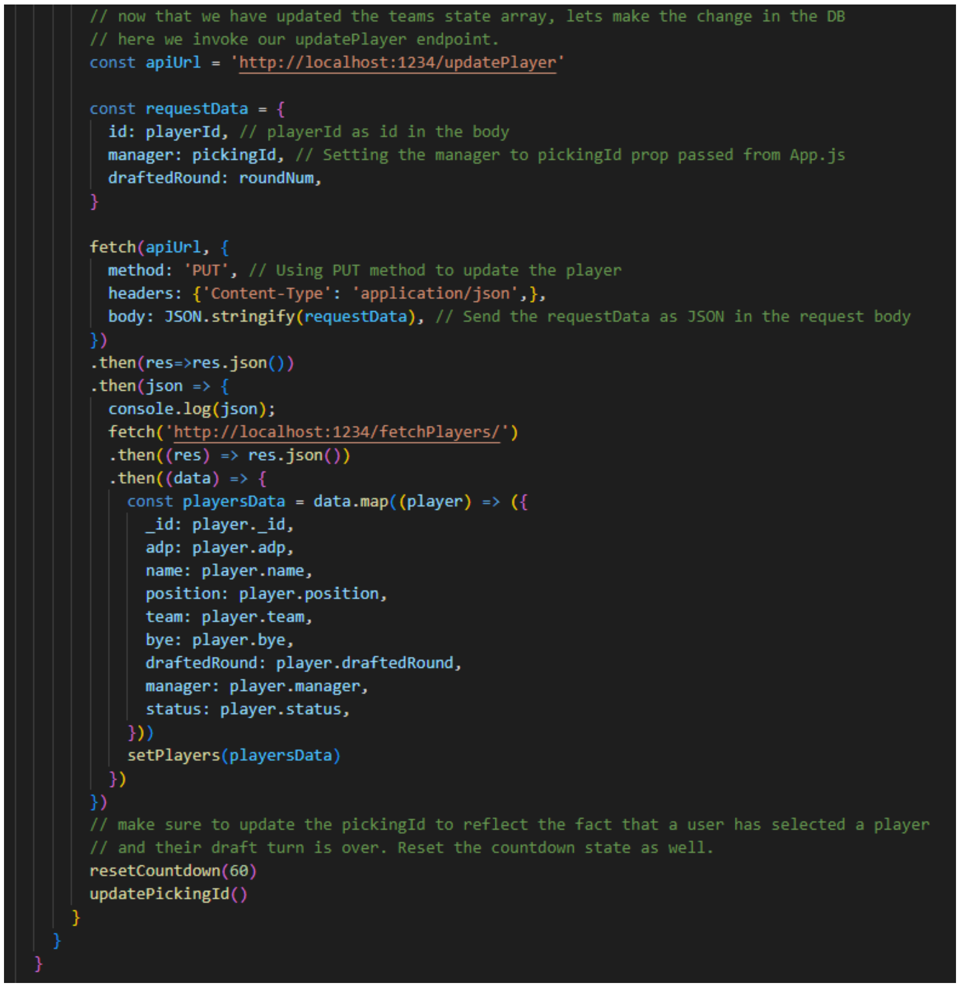
```
} else if (playerPosition === "WR") {
  // Check if WR slots are full, if so, check flex spots
  if (team.players.WR1 === null) {
    team.players.WR1 = playerName
  } else if (team.players.WR1 != null) {
    if (team.players.WR2 === null) {
      team.players.WR2 = playerName
    } else {
      // check flex spots
      if (team.players.FLEX1 === null) {
        team.players.FLEX1 = playerName
      } else {
        if (team.players.FLEX2 === null) {
          team.players.FLEX2 = playerName
        } else {
          alert("The position you are trying to fill is full.")
          return
        }
      }
    }
  }
}
```

- AvailablePlayerList.js is a function component that utilizes hooks to fetch the player data that is to be displayed.We utilized the useState() and the useEffect() hooks to accurately get and set our playersData as well as search through our playersData with our custom filteredPlayers() function.



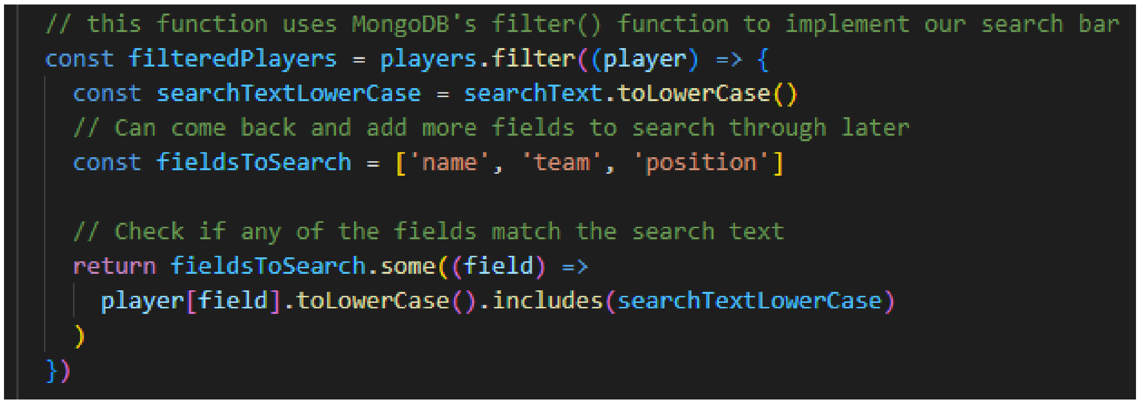
```
import React, { useState, useEffect } from 'react'
import DataTable from 'react-data-table-component'

const AvailablePlayerList = ({ pickingId, teams, draftStarted, roundNum, updatePickingId, resetCountdown }) => {
  // hooks
  const [players, setPlayers] = useState([])
  const [searchText, setSearchText] = useState('')

  useEffect(() => {
    fetch('http://localhost:1234/fetchPlayers/')
      .then((res) => res.json())
      .then((data) => {
        const playersData = data.map((player) => ({
          _id: player._id,
          adp: player.adp,
          name: player.name,
          position: player.position,
          team: player.team,
          bye: player.bye,
          draftedRound: player.draftedRound,
          manager: player.manager,
          status: player.status,
        }))
        setPlayers(playersData)
      })
  }, [])
```

- The AvailablePlayerList.js is passed a number of state variables as props as well as the updatePickingID() method and the resetCountdown() method. It consists of two functions called handleAddPlayerToTeam() and filteredPlayers().
  - handleAddPlayerToTeam(): this function is the handler function that is invoked whenever the “Add Player” button is clicked. First, it completes a large number of if statements to ensure that the player being selected actually has an open slot of the team that is attempting to draft it. If there is an open slot, we first update the teams array (passed as a prop from App.js) to include the new player. Then we make our “/updatePlayer” endpoint call to reflect those changes in the back-end. Finally, this function invokes the resetCountdown() method and the updatePickingId() method (passed as props from App.js) to move the draft along to the next team in the draft (in accordance with our snake draft logic). We included a snapshot of the function, after the if statements.



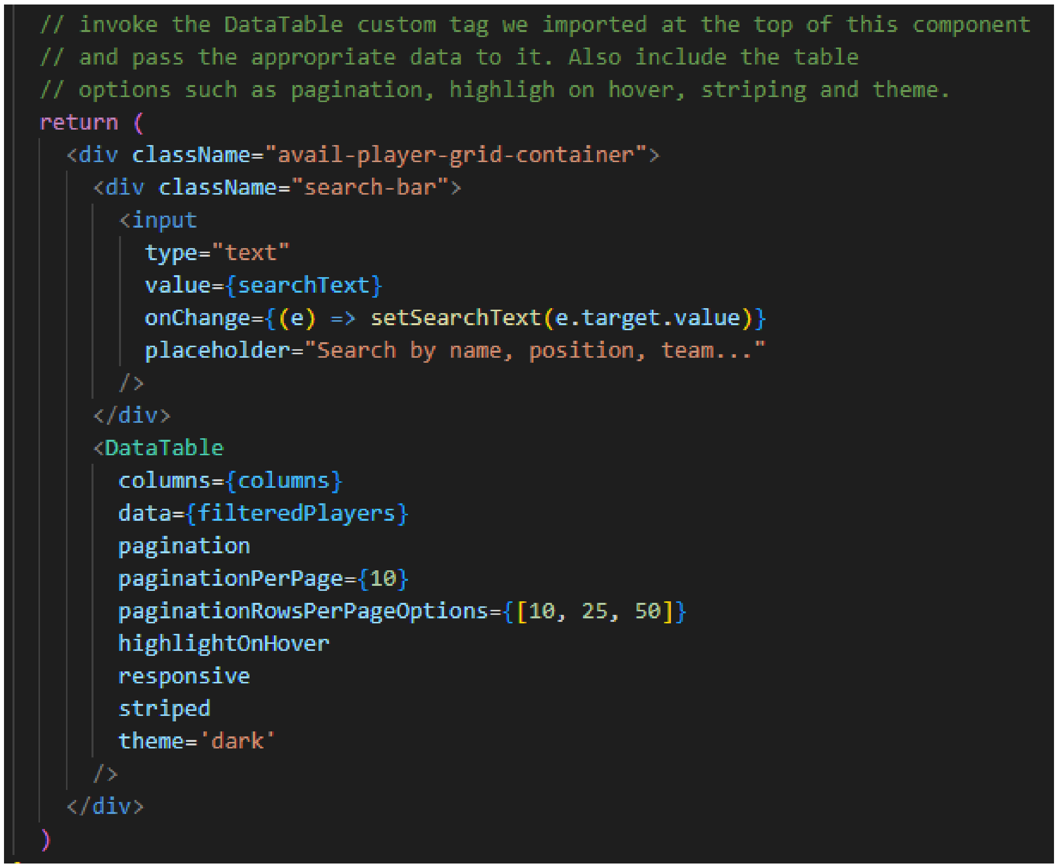
```
      // now that we have updated the teams state array, lets make the change in the DB
      // here we invoke our updatePlayer endpoint.
      const apiUrl = 'http://localhost:1234/updatePlayer'

      const requestData = {
        id: playerId, // playerId as id in the body
        manager: pickingId, // Setting the manager to pickingId prop passed from App.js
        draftedRound: roundNum,
      }

      fetch(apiUrl, {
        method: 'PUT', // Using PUT method to update the player
        headers: {'Content-Type': 'application/json',},
        body: JSON.stringify(requestData), // Send the requestData as JSON in the request body
      })
      .then(res=>res.json())
      .then(json => {
        console.log(json);
        fetch('http://localhost:1234/fetchPlayers/')
        .then((res) => res.json())
        .then((data) => {
          const playersData = data.map((player) => ({
            _id: player._id,
            adp: player.adp,
            name: player.name,
            position: player.position,
            team: player.team,
            bye: player.bye,
            draftedRound: player.draftedRound,
            manager: player.manager,
            status: player.status,
          }))
          setPlayers(playersData)
        })
      })
      // make sure to update the pickingId to reflect the fact that a user has selected a player
      // and their draft turn is over. Reset the countdown state as well.
      resetCountdown(60)
      updatePickingId()
    }
  }
}
```

- filteredPlayers(): a small helper function that utilizes the built-in functions filter() and includes() that functions as a search bar



```
// this function uses MongoDB's filter() function to implement our search bar
const filteredPlayers = players.filter((player) => {
  const searchTextLowerCase = searchText.toLowerCase()
  // Can come back and add more fields to search through later
  const fieldsToSearch = ['name', 'team', 'position']

  // Check if any of the fields match the search text
  return fieldsToSearch.some((field) =>
    player[field].toLowerCase().includes(searchTextLowerCase)
  )
})
```

- Finally, the last thing to note is that we defined an array called “columns” that outlines the DataTable that is invoked in the return statement. We pass the columns array and the filteredPlayers data to the DataTable to be displayed with sort and pagination capabilities.

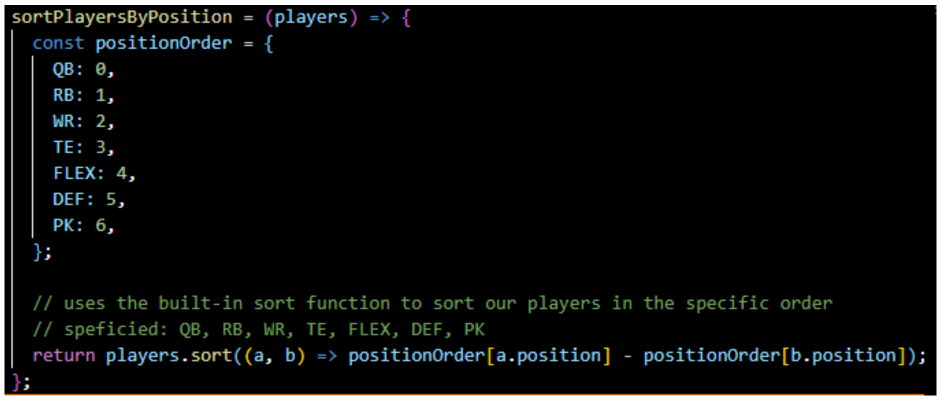


```
// invoke the DataTable custom tag we imported at the top of this component
// and pass the appropriate data to it. Also include the table
// options such as pagination, highligh on hover, striping and theme.
return (
  <div className="avail-player-grid-container">
    <div className="search-bar">
      <input
        type="text"
        value={searchText}
        onChange={(e) => setSearchText(e.target.value)}
        placeholder="Search by name, position, team..."
      />
    </div>
    <DataTable
      columns={columns}
      data={filteredPlayers}
      pagination
      paginationPerPage={10}
      paginationRowsPerPageOptions={[10, 25, 50]}
      highlightOnHover
      responsive
      striped
      theme='dark'
    />
  </div>
)
```

- **Implementation of TeamManagement.js:** TeamManagement.js is a class component that is responsible for displaying each of the teams with their players. The players will be displayed with columns showing positions, player name, team, bye week, adp, and round drafted. In the draft, this will be used by teams to see the outlook of each team allowing for strategizing throughout the draft process.
  - Changes from Phase 1: TeamManagement.js did not exist in Phase 1, we only had a placeholder div that was styled, therefore all of its code was completed in Phase 2.
  - Problems encountered with TeamManagement.js: The problems we encountered in TeamManagement.js were related to displaying the correct information in a pleasing format. We ended up using some refactored grid code that Drew wrote in Phase 1 (for AvailablePlayerList.js) to display the table in the TeamManagement window. We also created a new endpoint and service function called “/fetchTeams”, “fetchTeamsEndpoint()” and

"fetchTeams()" that accepted a manager in the request.body and used the find() method to return all players with the matching manager id. Lastly, we added a final field to our ODM Player Model called "draftedRound" that tracks the exact round the player was drafted in.

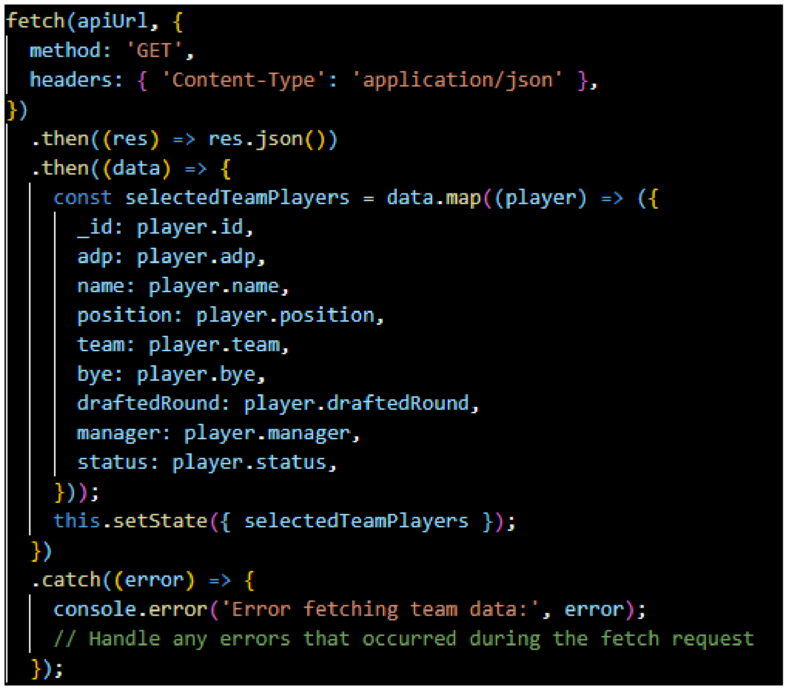
- TeamManagement.js utilizes a constructor to maintain the state of variables selectedTeamID and selectedTeamPlayers.
- TeamManagement.js has 3 functions that are used to display each team correctly.
    - **sortPlayersByPosition:** Sorts the team according to position for display purposes.
        - Accepts parameter players that are to be sorted.
        - Uses positionOrder to maintain an array of positions with order values.
        - Returns the players sorted by positionOrder.



```
sortPlayersByPosition = (players) => {
  const positionOrder = {
    QB: 0,
    RB: 1,
    WR: 2,
    TE: 3,
    FLEX: 4,
    DEF: 5,
    PK: 6,
  };

  // uses the built-in sort function to sort our players in the specific order
  // speficied: QB, RB, WR, TE, FLEX, DEF, PK
  return players.sort((a, b) => positionOrder[a.position] - positionOrder[b.position]);
};
```

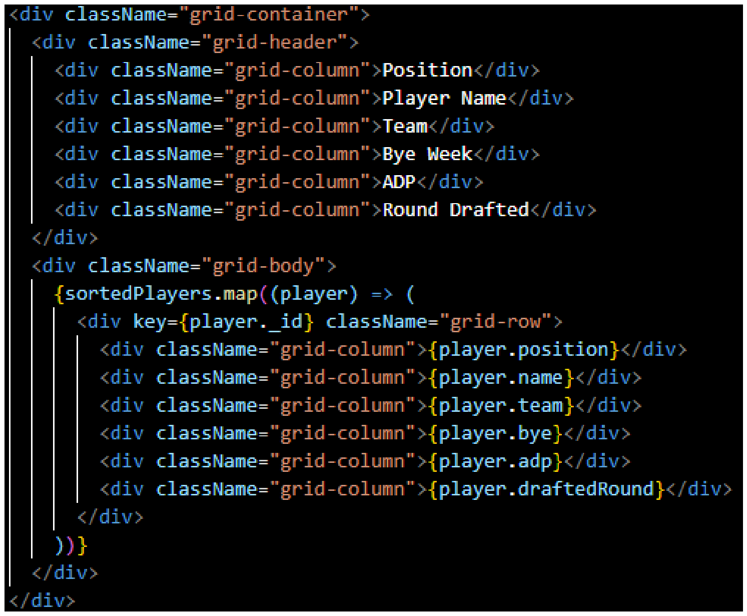
-
    - **handleTeamSelect:** Handler used to display the correct team that is selected from the drop down.
        - Sets the selectedTeamId to the event target value.
        - Fetches the fetchTeamEndpoint passing the selectedTeamId as query parameter.
        - Maps over player data.

- The below snippet shows fetch function and data handling.



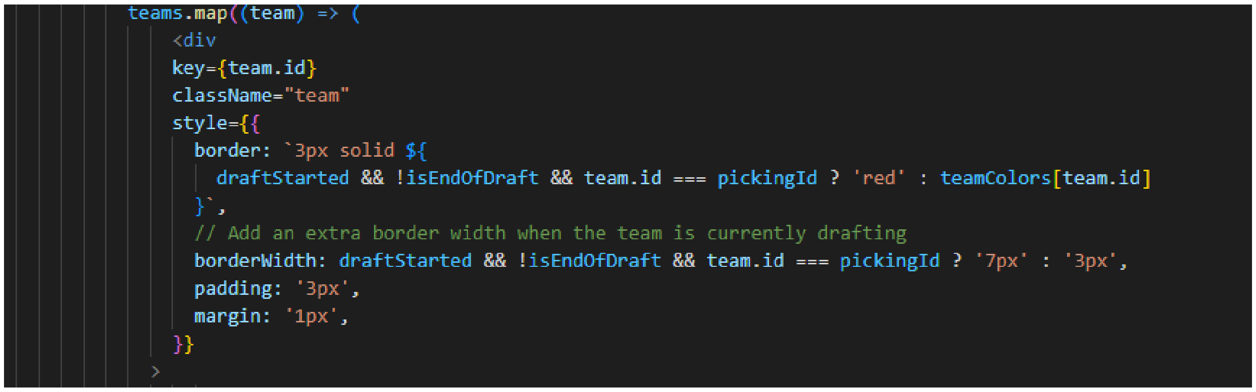
```
fetch(apiUrl, {
  method: 'GET',
  headers: { 'Content-Type': 'application/json' },
})
  .then((res) => res.json())
  .then((data) => {
    const selectedTeamPlayers = data.map((player) => ({
      _id: player.id,
      adp: player.adp,
      name: player.name,
      position: player.position,
      team: player.team,
      bye: player.bye,
      draftedRound: player.draftedRound,
      manager: player.manager,
      status: player.status,
    }));
    this.setState({ selectedTeamPlayers });
  })
  .catch((error) => {
    console.error('Error fetching team data:', error);
    // Handle any errors that occurred during the fetch request
  });
```

- **render:** Renders the drop down list and selected team.
  - Uses Drop Down List that calls handleTeamsSelect with selectedTeamId as value onChange.
  - Maps over team displaying grid headers and sortedPlayers according to position.
  - The below code snippet shows the grid container and grid body using the sortedPlayers function.



```
<div className="grid-container">
  <div className="grid-header">
    <div className="grid-column">Position</div>
    <div className="grid-column">Player Name</div>
    <div className="grid-column">Team</div>
    <div className="grid-column">Bye Week</div>
    <div className="grid-column">ADP</div>
    <div className="grid-column">Round Drafted</div>
  </div>
  <div className="grid-body">
    {sortedPlayers.map((player) => (
      <div key={player._id} className="grid-row">
        <div className="grid-column">{player.position}</div>
        <div className="grid-column">{player.name}</div>
        <div className="grid-column">{player.team}</div>
        <div className="grid-column">{player.bye}</div>
        <div className="grid-column">{player.adp}</div>
        <div className="grid-column">{player.draftedRound}</div>
      </div>
    ))}
  </div>
</div>
```

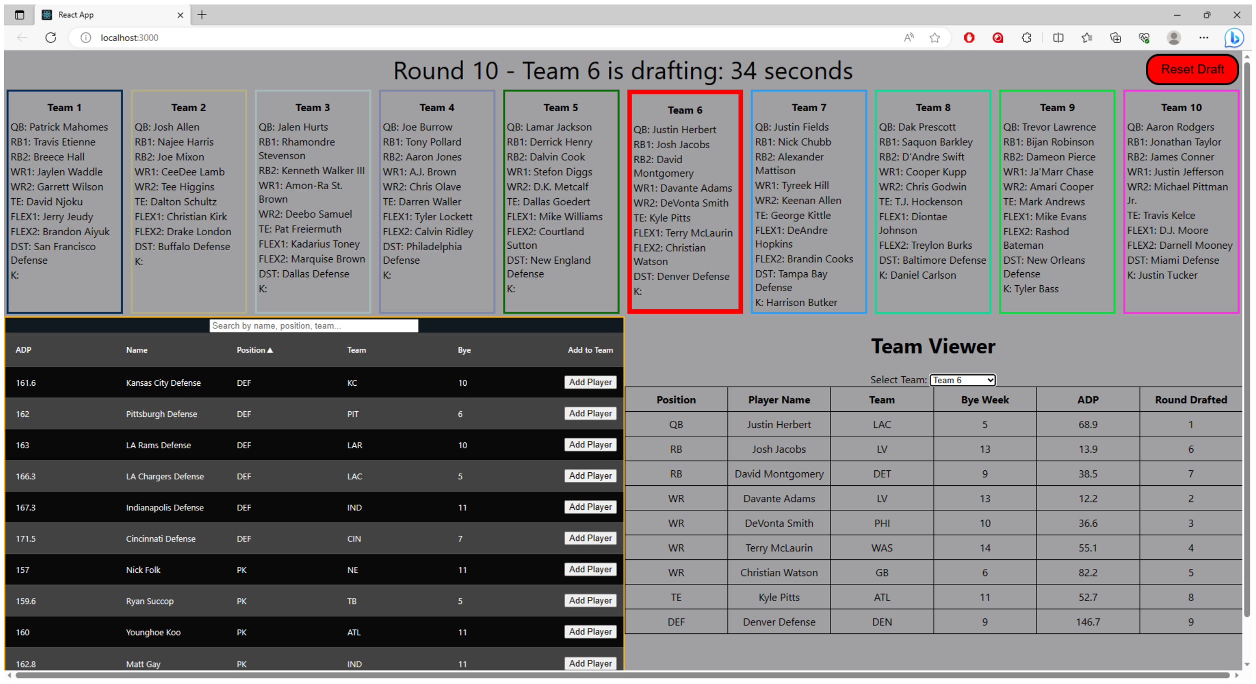
Deliverable #4 - Style Front-End: The front-end is styled exclusively using CSS. We utilized both a style sheet and in-line styling for different situations. The only time in-line styling was performed was when there was a specific condition that had to be checked that affected how the component was styled. An example of this was that we wanted the team that is currently drafting to have a thick red border around their team when they are drafting. This meant checking the teams.id with the pickingId and changing the border color accordingly. We have included a picture to illustrate one of these in-line style conditions.



```
teams.map((team) => (
  <div
  key={team.id}
  className="team"
  style={{
    border: `3px solid ${
      draftStarted && !isEndOfDraft && team.id === pickingId ? 'red' : teamColors[team.id]
    }`,
    // Add an extra border width when the team is currently drafting
    borderWidth: draftStarted && !isEndOfDraft && team.id === pickingId ? '7px' : '3px',
    padding: '3px',
    margin: '1px',
  }}
  >
```

- **Style Implementation:** First, we created and styled three main wrapper divs that carved out the correct space for our three respective components. This has the draftBoard wrapper extending horizontally across the web page with a width of 100%. The availablePlayerList wrapper and the TeamManagement wrapper are positioned, side-by-side, below the draftBoard wrapper. With these wrapper divs in place, it was much easier to add in our buttons and tables without fear of messing up any styling. From there, all the CSS in the style sheet is organized into groupings by comments. This makes it so that, at a glance, it is easy to see what CSS is styling what part of the page.
  - Changes from Phase 1:: We implemented a large number of style changes that were not and could not be present for Phase 1. Things like the DraftBoard and the TeamManagement table were not included or styled in Phase 1. We also styled buttons to start and reset the draft and we created conditional divs that only display when the draft is over. We themed the Fantasy Draft App to have a “dark” theme feel. Lastly, the TeamManagement table was styled using grid, rather than using the “react-data-table-component”. We felt that the package was too much for the data we were trying to display in the TeamManagement.js component, therefore we implemented grid.

- Challenges: Implementing the conditional in-line styling was the hardest part about styling this application. As well as getting the “Start Draft” button and the countdown to disappear once the draft is complete.
- Below we have included a graphic of the Fantasy Draft App in action to show off our styling choices:
  - **Note:** Our pagination feature has been cut off from this graphic because the team player names are pushing it down.



Round 10 - Team 6 is drafting: 34 seconds

Team 1
QB: Patrick Mahomes
RB1: Travis Etienne
RB2: Breece Hall
WR1: Jaylen Waddle
WR2: Garrett Wilson
TE: David Njoku
FLEX1: Jerry Jeudy
FLEX2: Brandon Aiyuk
DST: San Francisco Defense
K:

Team 2
QB: Josh Allen
RB1: Najee Harris
RB2: Joe Mixon
WR1: CeeDee Lamb
WR2: Tee Higgins
TE: Dalton Schultz
FLEX1: Christian Kirk
FLEX2: Drake London
DST: Buffalo Defense
K:

Team 3
QB: Jalen Hurts
RB1: Rhamondre Stevenson
RB2: Kenneth Walker III
WR1: Amon-Ra St. Brown
WR2: Deebo Samuel
TE: Pat Freiermuth
FLEX1: Kadarius Toney
FLEX2: Marquise Brown
DST: Dallas Defense
K:

Team 4
QB: Joe Burrow
RB1: Tony Pollard
RB2: Aaron Jones
WR1: A.J. Brown
WR2: Chris Olave
TE: Darren Waller
FLEX1: Tyler Lockett
FLEX2: Calvin Ridley
DST: Philadelphia Defense
K:

Team 5
QB: Lamar Jackson
RB1: Derrick Henry
RB2: Dalvin Cook
WR1: Stefon Diggs
WR2: D.K. Metcalf
TE: Dallas Goedert
FLEX1: Mike Williams
FLEX2: Courtland Sutton
DST: New England Defense
K:

Team 6
QB: Justin Herbert
RB1: Josh Jacobs
RB2: David Montgomery
WR1: Davante Adams
WR2: DeVonta Smith
TE: Kyle Pitts
FLEX1: Terry McLaurin
FLEX2: Christian Watson
DST: Denver Defense
K:

Team 7
QB: Justin Fields
RB1: Nick Chubb
RB2: Alexander Mattison
WR1: Tyreek Hill
WR2: Keenan Allen
TE: George Kittle
FLEX1: DeAndre Hopkins
FLEX2: Brandin Cooks
DST: Tampa Bay Defense
K: Harrison Butker

Team 8
QB: Dak Prescott
RB1: Saquon Barkley
RB2: D'Andre Swift
WR1: Cooper Kupp
WR2: Chris Godwin
TE: T.J. Hockenson
FLEX1: Diontae Johnson
FLEX2: Treylon Burks
DST: Baltimore Defense
K: Daniel Carlson

Team 9
QB: Trevor Lawrence
RB1: Bijan Robinson
RB2: Dameon Pierce
WR1: Ja'Marr Chase
WR2: Amari Cooper
TE: Mark Andrews
FLEX1: Mike Evans
FLEX2: Rashod Bateman
DST: New Orleans Defense
K: Tyler Bass

Team 10
QB: Aaron Rodgers
RB1: Jonathan Taylor
RB2: James Conner
WR1: Justin Jefferson
WR2: Michael Pittman Jr.
TE: Travis Kelce
FLEX1: D.J. Moore
FLEX2: Darnell Mooney
DST: Miami Defense
K: Justin Tucker

ADP	Name	Position ▲	Team	Bye	Add to Team
161.6	Kansas City Defense	DEF	KC	10	Add Player
162	Pittsburgh Defense	DEF	PIT	6	Add Player
163	LA Rams Defense	DEF	LAR	10	Add Player
166.3	LA Chargers Defense	DEF	LAC	5	Add Player
167.3	Indianapolis Defense	DEF	IND	11	Add Player
171.5	Cincinnati Defense	DEF	CIN	7	Add Player
157	Nick Folk	PK	NE	11	Add Player
159.6	Ryan Succop	PK	TB	5	Add Player
160	Younghoe Koo	PK	ATL	11	Add Player
162.8	Matt Gay	PK	IND	11	Add Player

Team Viewer

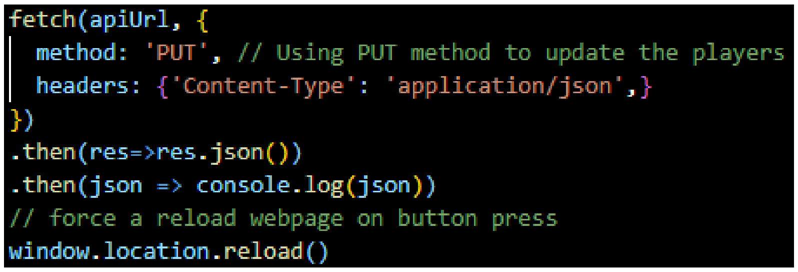
Position	Player Name	Team	Bye Week	ADP	Round Drafted
QB	Justin Herbert	LAC	5	68.9	1
RB	Josh Jacobs	LV	13	13.9	6
RB	David Montgomery	DET	9	38.5	7
WR	Davante Adams	LV	13	12.2	2
WR	DeVonta Smith	PHI	10	36.6	3
WR	Terry McLaurin	WAS	14	55.1	4
WR	Christian Watson	GB	6	82.2	5
TE	Kyle Pitts	ATL	11	52.7	8
DEF	Denver Defense	DEN	9	146.7	9

Deliverable #5a - Refactor Front-End Code: We refactored a large amount of code in Phase 2 in comparison to Phase 1. The main thing that was refactored was that we removed all of the draft state variables out of DraftBoard.js and moved them into App.js, as well as their accompanying methods. This made it so that our components could reference the draft state much easier and ensured that no complications would be met with state during the draft process. We then passed the relevant state and methods through to each component as props to be further utilized. We also added two more endpoints than we thought we would need. We created a new endpoint called “/fetchTeams” that accepts a manager parameter as a request.body and returns all the players with the matching manager id. We also created another endpoint called “/resetPlayers”. This endpoint is invoked whenever the big red “Reset Draft” button is pressed and it simply changes the manager and status fields to null and true, respectively. The last refactoring we did for the project involved some quality of life and style changes, such as distinguishing which team is currently drafting, adding a reset draft button, and giving each team a random colored border.

Deliverable #6a - Cleanup Front-End Code: For our clean-up process we went through the code and made sure all functions and states had comments that accurately describe what is being done. We removed any unnecessary code that may have been overlooked and we also removed all console.log() invocations. Some other small clean-up includes following ES6 and AirBnB standards, which includes removing all semicolons.

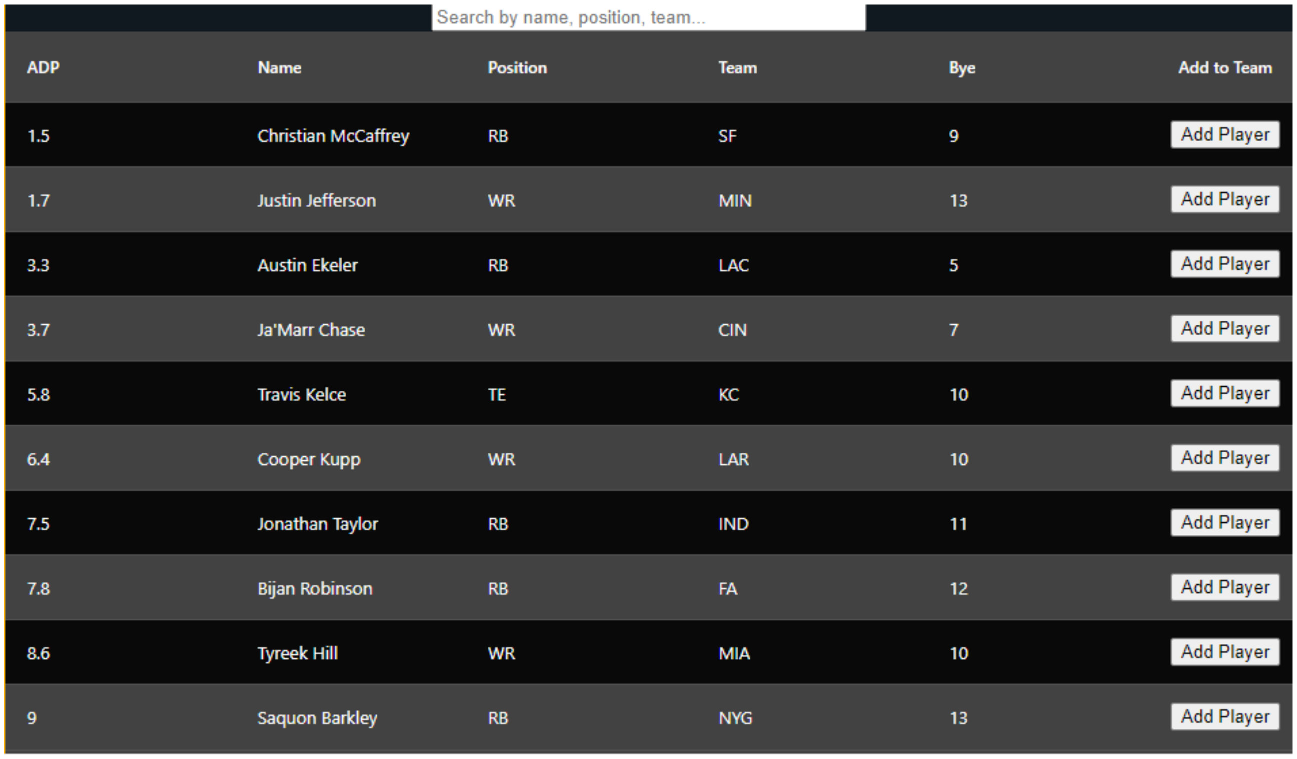
Back-End:

Deliverable #2 - Integrate React Code with NodeJS: The bulk of this deliverable was completed in Phase-1 with the creation and testing of our API and database. However, throughout Phase-2, our front-end code relied on the successful integration of React code with our API. Our DraftBoard.js, AvailablePlayerList, and TeamManagment.js components each make api requests to fetch player data from the MongoDB database. See the below snippet showing the use of the resetPlayers endpoint in DraftBoard that resets the players in our database, making them available for another draft.



```
fetch(apiUrl, {
  method: 'PUT', // Using PUT method to update the players
  headers: {'Content-Type': 'application/json',}
})
.then(res=>res.json())
.then(json => console.log(json))
// force a reload webpage on button press
window.location.reload()
```

Deliverable #3 - Ensure Data Synchronization: This deliverable is used to ensure that the data being manipulated, stored, and retrieved from the database is accurate in both our application as well as our MongoDB database. Thorough testing in Phase-2 showed that the fetch requests, api endpoints, service functions, and database are functioning correctly. See the below screenshot showing our dynamic available player list that is shown through the use of a fetch request, api endpoint, service function, and populated database.



Search by name, position, team...

ADP	Name	Position	Team	Bye	Add to Team
1.5	Christian McCaffrey	RB	SF	9	Add Player
1.7	Justin Jefferson	WR	MIN	13	Add Player
3.3	Austin Ekeler	RB	LAC	5	Add Player
3.7	Ja'Marr Chase	WR	CIN	7	Add Player
5.8	Travis Kelce	TE	KC	10	Add Player
6.4	Cooper Kupp	WR	LAR	10	Add Player
7.5	Jonathan Taylor	RB	IND	11	Add Player
7.8	Bijan Robinson	RB	FA	12	Add Player
8.6	Tyreek Hill	WR	MIA	10	Add Player
9	Saquon Barkley	RB	NYG	13	Add Player

Deliverable #5b - Refactor Back-End Code: Added two new endpoints to the back-end code that were both utilized in our finished product:

- "/fetchTeam" - the endpoint being fetched (this is a GET method that passes a query parameter to the fetchTeam() service function)
  - fetchTeamEndpoint() - invoked when the endpoint is called, which calls our fetchTeam() service function with the manager id passed as a query.
  - fetchTeam() - service function that uses the built-in find() method to sort by manager id and return all the players that match.
- "/resetPlayers" - the endpoint being fetched (this is a PUT method that takes no parameters)
  - resetPlayersEndpoint() - invoked when the endpoint is called, which calls our resetPlayers() service function.
  - resetPlayers() - service function that uses the built-in mongosh method updateMany() to change the manager and status field of every player in the DB.

Deliverable #6b - Cleanup Back-End Code: For our clean-up process we went through the code and made sure all functions and states had comments that accurately describe what is being done. We also removed any unnecessary code that may have been overlooked. We removed all console.log() calls that may have been left over as well.