FANTASY FOOTBALL
DRAFT TOOL

Phase-1 Deliverable Report

07/30/2023

Project:

Fantasy Football Draft Tool

Team Members:

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

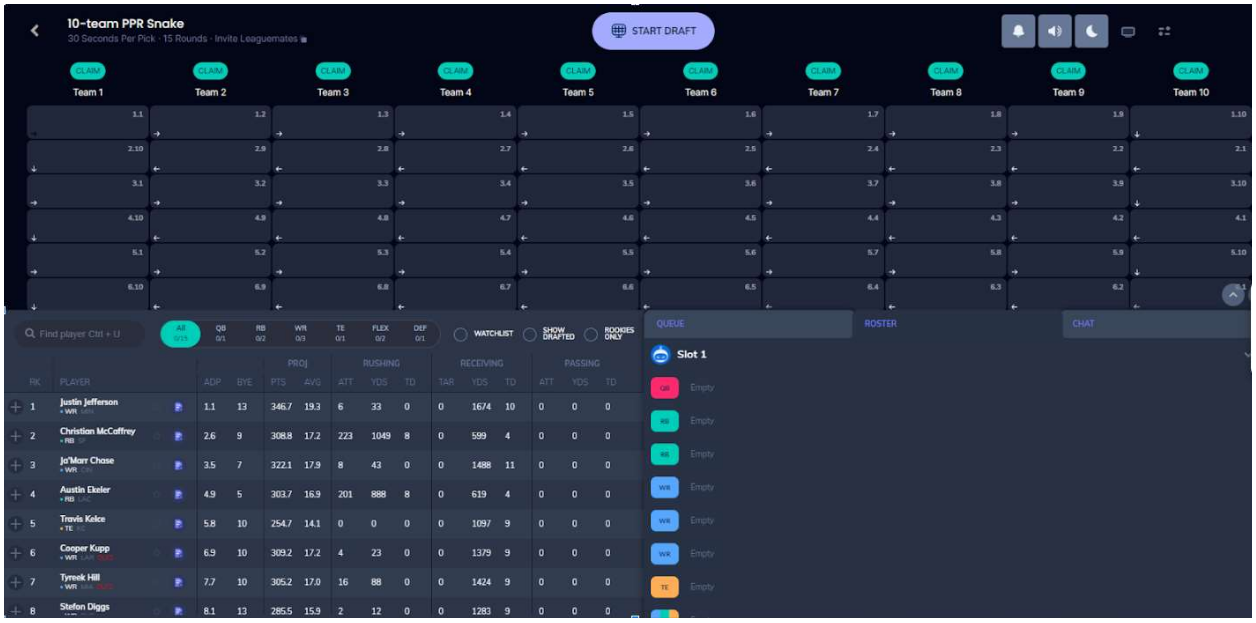
Drew Heller
Email: dh2225@nau.edu &
GitHub: https://github.com/dh2225

GitHub Repository:

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

Front-End:

Deliverable #1 - Design Front-End: The intent behind this deliverable is to begin discussing the overall look and style of the front-end. By agreeing upon a rough design of the front-end web page, we are able to better prepare for the completion of the Phase-2 deliverables. As of now, we agree to an overall design close to the below image.



				PROJ		RUSHING			RECEIVING			PASSING		
RK	PLAYER	ADP	BYE	PTS	AVG	ATT	YDS	TD	TAR	YDS	TD	ATT	YDS	TD
1	Justin Jefferson • WR	1.1	13	346.7	19.3	6	33	0	0	1674	10	0	0	0
2	Christian McCaffrey • RB	2.6	9	308.8	17.2	223	1049	8	0	599	4	0	0	0
3	Ja'Marr Chase • WR	3.5	7	322.1	17.9	8	43	0	0	1488	11	0	0	0
4	Austin Ekeler • RB	4.9	5	303.7	16.9	201	888	8	0	619	4	0	0	0
5	Travis Kelce • TE	5.8	10	254.7	14.1	0	0	0	0	1097	9	0	0	0
6	Cooper Kupp • WR	6.9	10	309.2	17.2	4	23	0	0	1379	9	0	0	0
7	Tyreek Hill • WR	7.7	10	305.2	17.0	16	88	0	0	1424	9	0	0	0
8	Stefon Diggs	8.1	13	285.5	15.9	2	12	0	0	1283	9	0	0	0

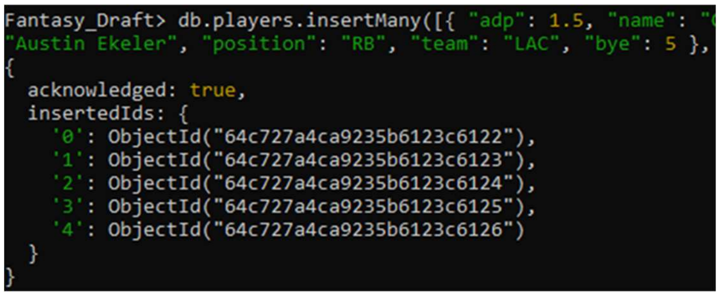
The draft board will be the centerpiece component showing which team is picking along with all of the previous picks.

Below that there will be two sides, the player board and team management console. The player board will contain any of the available players with filtering functionality. The team management console will allow the user to show each manager's team and the spots that are currently filled on their roster.

By having this look and style agreed upon, our team will be capable of making a great looking fantasy draft site in the Phase-2 deliverables.

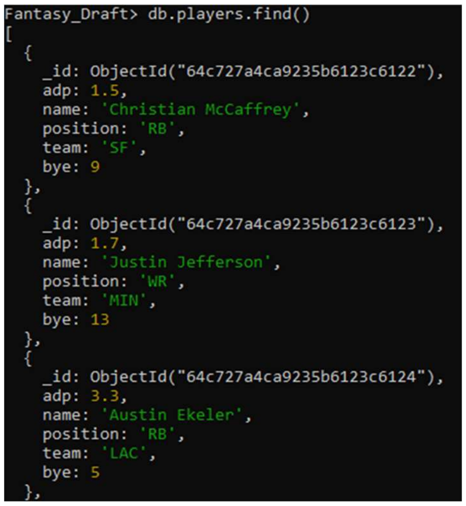
Back-End:

Deliverable #2 - Create MongoDB: The database was created using the command "use Fantasy_Draft". Following the creation of the database, the collection named "players" was created to hold the documents/players for the database. See the below screenshot to see the command that created the collection as well as the first 5 documents.



```
Fantasy_Draft> db.players.insertMany([{ "adp": 1.5, "name": "C
"Austin Ekeler", "position": "RB", "team": "LAC", "bye": 5 },
{
  acknowledged: true,
  insertedIds: {
    '0': ObjectId("64c727a4ca9235b6123c6122"),
    '1': ObjectId("64c727a4ca9235b6123c6123"),
    '2': ObjectId("64c727a4ca9235b6123c6124"),
    '3': ObjectId("64c727a4ca9235b6123c6125"),
    '4': ObjectId("64c727a4ca9235b6123c6126")
  }
}
```

Deliverable #3 - Populate MongoDB: In order to test the creation of our API endpoints and Service Function code, a populated database was needed. In this deliverable, the database was populated with the first 5 players of our intended database. The initial fields were adp, name, position, team, and bye. Later, the list would be updated to have the two missing fields that are used by our api code, status and manager. The following function within mongosh was used to add these fields, db.players.updateMany({}, { "$set": { "manager": null, "status": true } }). Below is the screenshot showing the players after the initial insert.

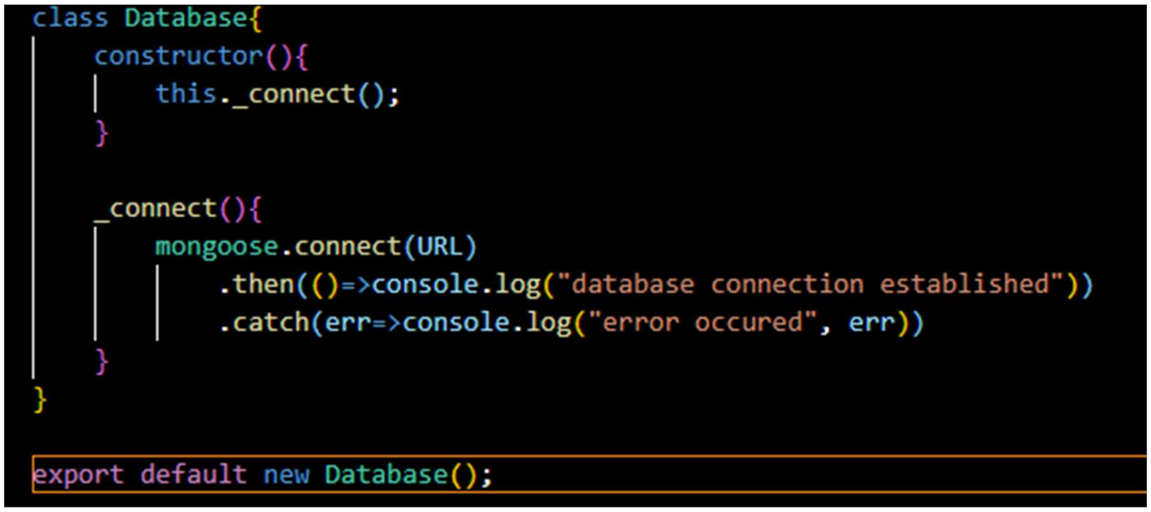


```
Fantasy_Draft> db.players.find()
[
  {
    _id: ObjectId("64c727a4ca9235b6123c6122"),
    adp: 1.5,
    name: 'Christian McCaffrey',
    position: 'RB',
    team: 'SF',
    bye: 9
  },
  {
    _id: ObjectId("64c727a4ca9235b6123c6123"),
    adp: 1.7,
    name: 'Justin Jefferson',
    position: 'WR',
    team: 'MIN',
    bye: 13
  },
  {
    _id: ObjectId("64c727a4ca9235b6123c6124"),
    adp: 3.3,
    name: 'Austin Ekeler',
    position: 'RB',
    team: 'LAC',
    bye: 5
  },
```

Deliverable #4 - Create Endpoints & Service Function Code:

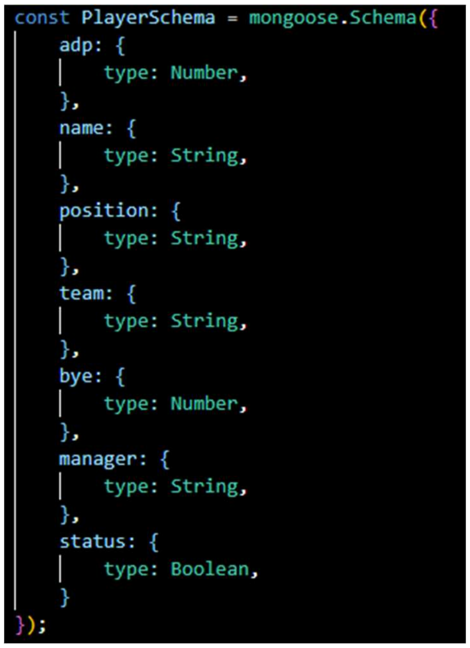
- **Creation of connect.js**
    - Connect.js was created in this deliverable so that the api and endpoints could be properly tested.
    - The connect.js code is responsible for setting up a connection to the MongoDB database using mongoose.
    - The server url was created containing our databaseName and server ip.

- Database class was created allowing an instance of the Database class to be exported and a connection to the MongoDB server to be established upon import. Below is a small snippet showing the database class.



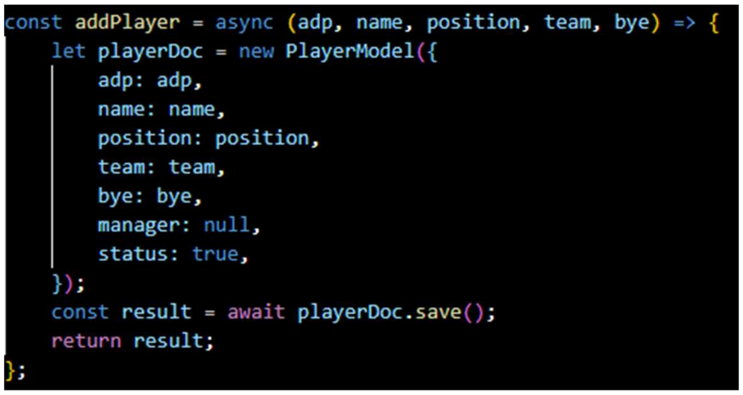
```
class Database{
    constructor(){
        this._connect();
    }

    _connect(){
        mongoose.connect(URL)
            .then(()=>console.log("database connection established"))
            .catch(err=>console.log("error occured", err))
    }
}

export default new Database();
```

- **Creation of PlayerModel.js**
    - The PlayerModel.js code is responsible for establishing a schema for the collection of players.
    - The field data types are defined and the model is exported for use in the DraftToolService.js code.
    - The below screenshot shows the schema used.



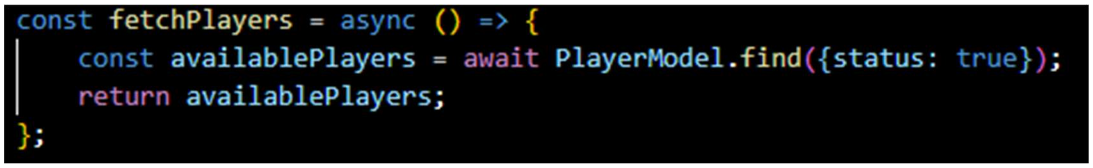
```
const PlayerSchema = mongoose.Schema({
    adp: {
        type: Number,
    },
    name: {
        type: String,
    },
    position: {
        type: String,
    },
    team: {
        type: String,
    },
    bye: {
        type: Number,
    },
    manager: {
        type: String,
    },
    status: {
        type: Boolean,
    }
});
```

- **Creation of DraftToolService.js**
    - The DraftToolService.js code contains the code for the 5 service functions that are required for the api.
        - **addPlayer**
            - Accepts the parameters adp, name, position, team and bye
            - It uses the imported PlayerModel when assigning the received values.
            - The result is then added to the database using .save() and the result of the save command is returned to the api.
            - The below screenshot shows the code for the addPlayer service function.



```
const addPlayer = async (adp, name, position, team, bye) => {
    let playerDoc = new PlayerModel({
        adp: adp,
        name: name,
        position: position,
        team: team,
        bye: bye,
        manager: null,
        status: true,
    });
    const result = await playerDoc.save();
    return result;
};
```

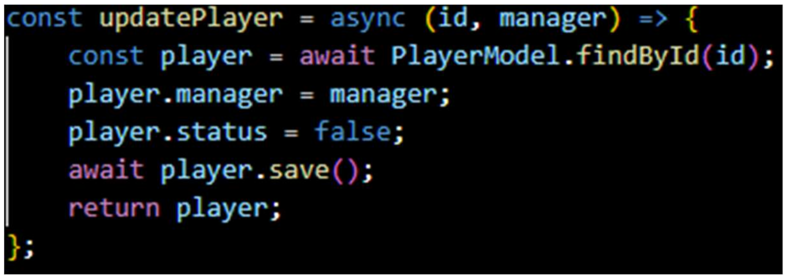
        - **fetchPlayers**
            - Uses a variable to store the results of a .find command.
            - The .find command uses a filter to only provide the players that have the status true, indicating that they are not drafted yet.
            - The response from the database is returned to the api.
            - See the below screenshot for the code.



```
const fetchPlayers = async () => {
    const availablePlayers = await PlayerModel.find({status: true});
    return availablePlayers;
};
```

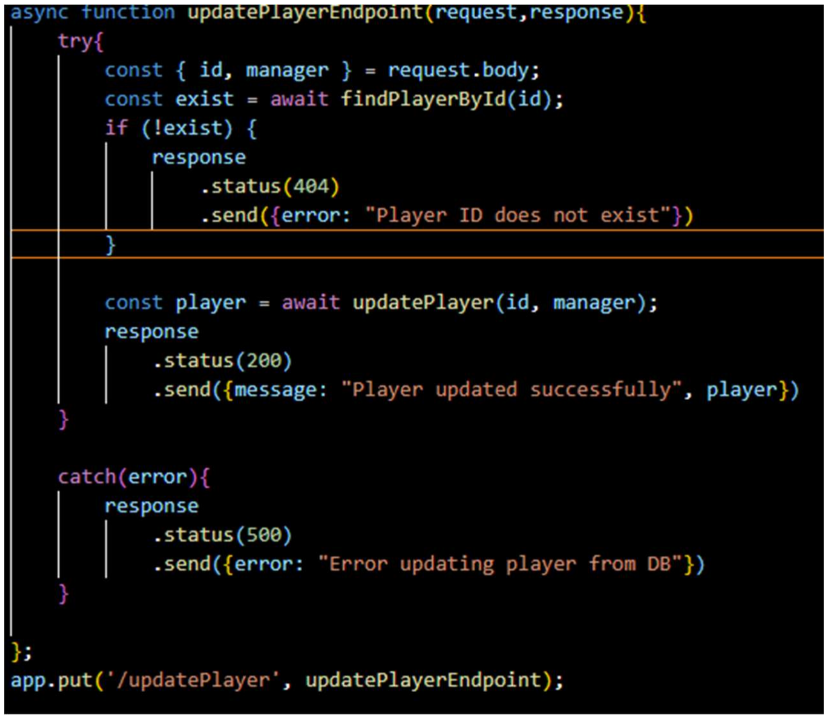
        - **findPlayerById**
            - Accepts an id as a parameter
            - Created to be used for the api to find a corresponding player by their id.
        - **updatePlayer**
            - Accepts id and manager as a parameter.

        - Finds the appropriate player by the id parameter.
        - If found, sets manager field and status field.
        - Uses .save command and returns the player.



```
const updatePlayer = async (id, manager) => {
    const player = await PlayerModel.findById(id);
    player.manager = manager;
    player.status = false;
    await player.save();
    return player;
};
```

    - **deletePlayer**
        - Accepts id as a parameter.
        - Uses findByIdAndDelete command, passing the id.
        - Returns the deleted player.
- **Creation of DraftToolAPI.js**
    - Contains the api method and endpoint functions that are required by the front end.
        - addPlayerEndpoint
        - fetchPlayersEndpoint
        - updatePlayerEndpoint
        - deletePlayerEndpoint
    - Each endpoint calls its corresponding service function.
    - Each endpoint contains error checking to make sure required fields are available, the player exists in the database, and that the service functions were performed correctly.
    - The below screenshot shows the updatePlayerEndpoint, containing its api method, try/catch, and service function call.

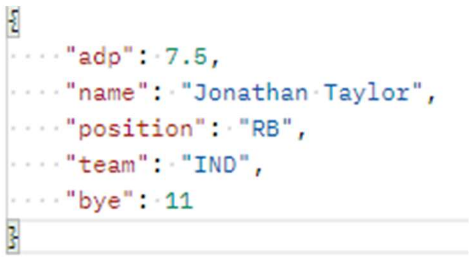


```
async function updatePlayerEndpoint(request,response){
    try{
        const { id, manager } = request.body;
        const exist = await findPlayerById(id);
        if (!exist) {
            response
                .status(404)
                .send({error: "Player ID does not exist"})
        }

        const player = await updatePlayer(id, manager);
        response
            .status(200)
            .send({message: "Player updated successfully", player})
    }

    catch(error){
        response
            .status(500)
            .send({error: "Error updating player from DB"})
    }

};
app.put('/updatePlayer', updatePlayerEndpoint);
```

Deliverable #5 - Test Endpoints in Postman: Within Postman, the four endpoints were tested to assure that the right actions were being performed within our Fantasy_Draft database. The endpoints included the following:

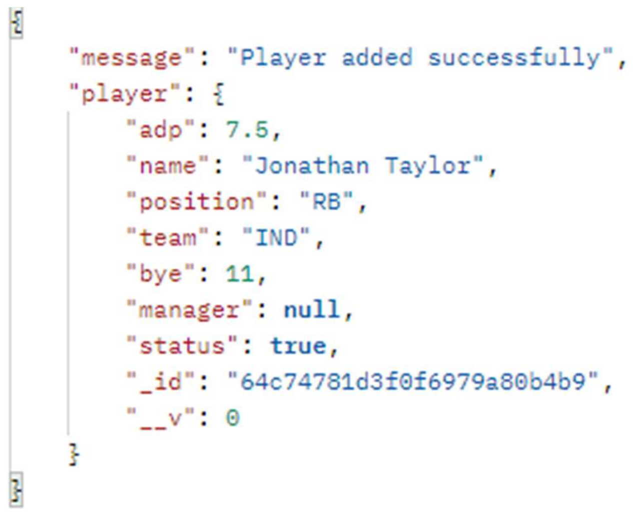
- http://localhost:1234/addPlayer/
- http://localhost:1234/fetchPlayers/
- http://localhost:1234/updatePlayer/
- http://localhost:1234/deletePlayer/

Testing Add Player Endpoint: In order to test the api functionality, the addPlayer endpoint will pass a player document in JSON format. The below screenshot shows an example of a player that would be passed from the front-end to the API.



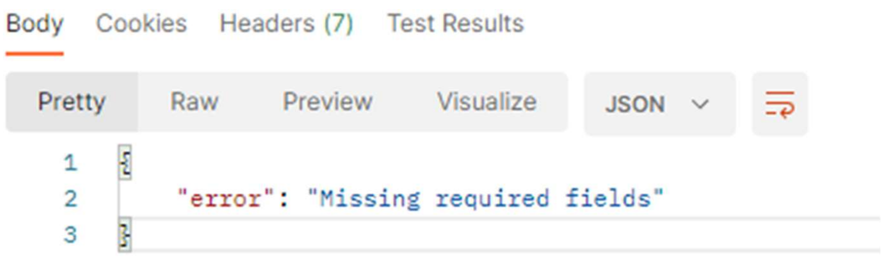
```
{
    "adp": 7.5,
    "name": "Jonathan Taylor",
    "position": "RB",
    "team": "IND",
    "bye": 11
}
```

When the body containing this player data was sent, the below screenshot shows the response.



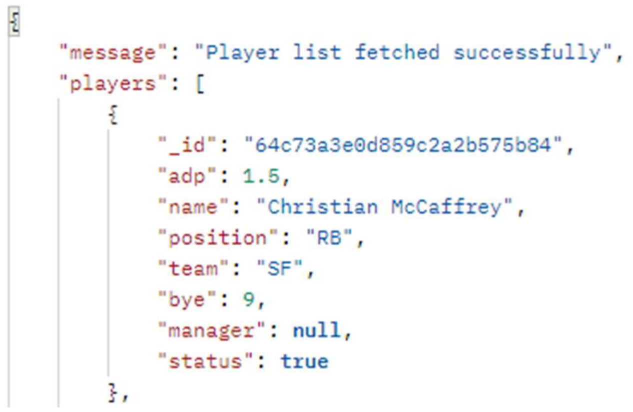
```
{
    "message": "Player added successfully",
    "player": {
        "adp": 7.5,
        "name": "Jonathan Taylor",
        "position": "RB",
        "team": "IND",
        "bye": 11,
        "manager": null,
        "status": true,
        "_id": "64c74781d3f0f6979a80b4b9",
        "__v": 0
    }
}
```

This shows that the api is able to successfully use the service functions and add players to the Fantasy_Draft database. Also, the manager and status fields are being added by the DraftToolService.js code. There are two errors that may be thrown if there is an issue with adding the player document. One is the status code 500, if there is an issue adding the player to the DB. The other will return a 400 status code if the data lacks the required fields. Both errors have been tested. See below for the status code 400 response.



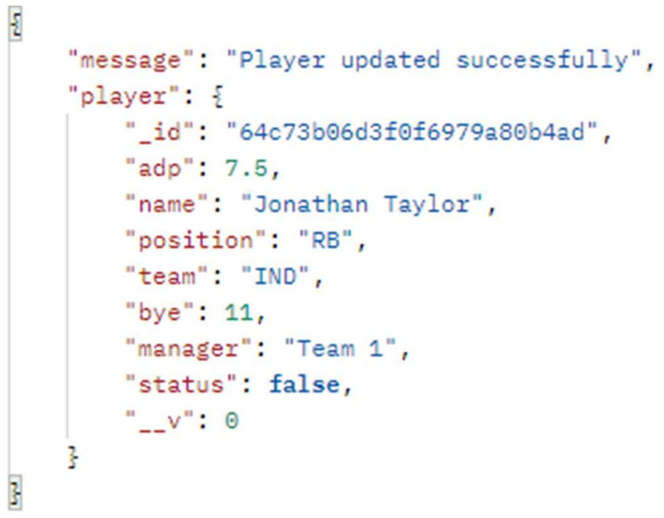
```
{
    "error": "Missing required fields"
}
```

Testing Fetch Players Endpoint: The fetch players endpoint is responsible for providing a list of players from the database that are not currently drafted by a manager/team. That is, the status of the player is true. When testing this functionality, the response should deliver a message stating that the list was fetched successfully as well as all of the players with the status of true. As seen in the below screenshot, the endpoint is capable of providing a list of players containing the status true.

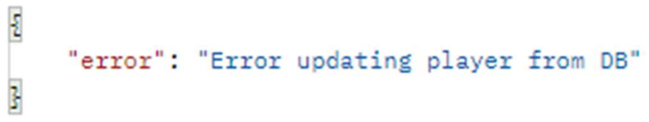

```
{
    "message": "Player list fetched successfully",
    "players": [
        {
            "_id": "64c73a3e0d859c2a2b575b84",
            "adp": 1.5,
            "name": "Christian McCaffrey",
            "position": "RB",
            "team": "SF",
            "bye": 9,
            "manager": null,
            "status": true
        },
```

If there is an error grabbing this list, the endpoint is capable of responding with the correct error message and status code.

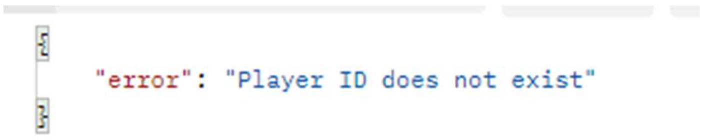
Testing Update Player Endpoint: The update player endpoint is responsible for updating a particular player's manager and status by the provided id. In the draft, this will enable the selected players to be removed from the available player list. If the update is successful, a success message will be displayed and the player data will be returned as well. See the below response showing the status changed and manager updated accordingly.


```
{
    "message": "Player updated successfully",
    "player": {
        "_id": "64c73b06d3f0f6979a80b4ad",
        "adp": 7.5,
        "name": "Jonathan Taylor",
        "position": "RB",
        "team": "IND",
        "bye": 11,
        "manager": "Team 1",
        "status": false,
        "__v": 0
    }
}
```

There are two errors that may be thrown, a 404 error stating that the player ID does not exist and the 500 error stating that there was an error updating the player in the database. The below screenshot shows the response from an incorrect id.


```
{
    "error": "Error updating player from DB"
}
```

Testing Delete Player Endpoint: The delete player endpoint is an admin only endpoint. The front-end will not be able to use this endpoint since it is a hard delete. The endpoint will accept an id of a player and either respond successfully or throw one of two errors. The first error will throw a 404 status code stating that the id doesn't exist. The other will throw a 500 status code stating that there was an error deleting the player from the database. The functionality of the delete player endpoint is working correctly. Players may be deleted and each error may successfully be thrown. The screenshot below shows that the id check of the player is working correctly.



```
{
    "error": "Player ID does not exist"
}
```