LineDriveBetting - Phase 4 Report

Team: Indigo
Github Repository: https://github.com/ArmanKhondker/EE-461L-LineDriveBetting
Team Google Drive: https://drive.google.com/drive/u/1/folders/0AF2MKdo4mjX9Uk9PVA
Project Lead: Zach Herink
Website URL: https://www.linedrivebetting.com

Name	**Email**	**Github**
Arman Khondker	armankhondker@utexas.edu	https://github.com/armankhondker
Punit Patel	punitpatel@utexas.edu	https://github.com/punitpatel9
Josh Papermaster	jpapermaster@utexas.edu	https://github.com/joshpapermaster
ThienSon Ho	thiensonho@utexas.edu	https://github.com/thienson-ho
Rohan Garg	rohanvgarg@utexas.edu	https://github.com/rohanvgarg
Zach Herink	zachary.herink@utexas.edu	https://github.com/zherink

Part IA. Information Hiding.

One of the main reasons we decided to use the React framework was for its components feature which let us split the user interface into independent reusable pieces. React allows us to think about each of these pieces in isolation. In the early phases of our design, we knew that we wanted to display data for games across the NFL, NBA, and MLB. We anticipated that we may want to add more betting data for each game such as adding new betting site sources. Although the three sports are very different, the information that we wanted to show for them were very similar.

- Each league has multiple games where one team plays another team. We modularized this into a League component.
- Every game has a set date and time that it will be played as well as point spreads and money lines. We modularized our program to have a GameBar component that displays the team logos, abbreviations, and date. The GameBar acts as a preview of the game and links to the Game component. The Game component displays the point spreads and money lines for all six of our betting site sources into tables and line charts.

- Every sports team has their own city and logo. We modularized this into assets for each league. We have a Teams.json dictionary that includes team names, locations, and abbreviations for each team.

/******* App.js

Fetch NBA data from API. (Data includes teams, datetime, money lines, point spreads for all upcoming games in the NBA)

Render League = NBA. Pass in NBA data
    For every NBA game, pass in data for that game
        Render GameBar for game preview
        Dynamically route each NBA game to a url
            Render Game page, pass in PS and ML data
                Display point spread tables and line graphs
                Display money line tables and line graphs

Repeat for MLB and NFL

*****/

Our modularization scheme allows us to reuse our components across all three leagues and the only thing that changes is the type of data we plugin. We can easily add new features by modifying one of our three main components: League, Game, or GameBar. We currently have dynamically rendered pages for over 200 games across the three leagues so adding new features would be extremely tedious if we didn't have modularization. For example, we were easily able to add the blog post feature to every page by only modifying the Game component.

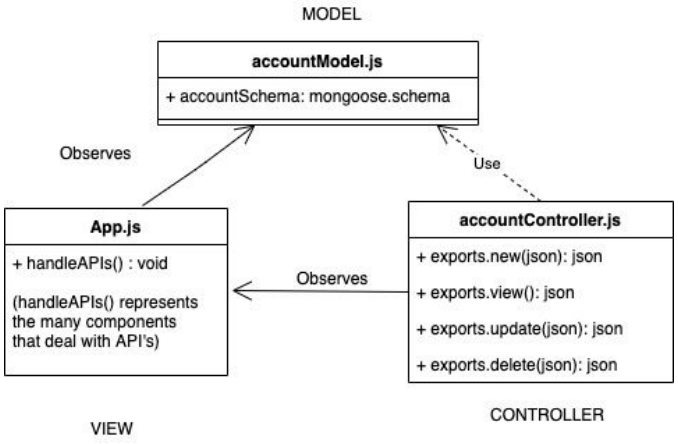
We apply information hiding because the Game component doesn't need to know about the games for the other leagues. The GameBar component doesn't need to know about all the other games there are for that league or the others. Each league's game data is passed to their child components who only use the parts of the data that they need. We could easily decide to remove any specific game or even an entire league and everything else will still work perfectly. The biggest disadvantage to our modularization is that if we add a new feature to a component, it will be shown across all sports leagues so this might be a problem if we ever need to add a feature to only one league. We could mitigate this by passing in league specific features as props so that the other leagues aren't affected.

1B. Design Patterns and Refactorings

1B.1 Design Patterns

A) Model View Controller

Each of our API's use the controller to handle api requests from the user interface. Next, the model uses the information from the controller to format and send the response data back to the user interface.



B) Singleton Design Pattern

In App.js, the main class of our front end, we use a BrowserRouter singleton. BrowserRouter is the router implementation for HTML5 browsers. Each route conditionally displays a component (webpage) based on matching a path to the url. The single instance of BrowserRouter allows user to navigate throughout our site for both predetermined and dynamically generated web pages. Predetermined pages include the home, about, and league pages. The dynamically generated pages include the hundreds of games that are made based on data fetched from our API.

```
<BrowserRouter>
    <Route path ='/' render={() => (<StickNavbar />)}/>
    <Route exact={true} path='/' render={() => (<Home />)}/>
    <Route exact={true} path='/About' render={() => (<About />)}/>
    <Route exact={true} path='/NBA' render={() => (<League games={this.state.nbaGames} league="NBA"/>)}/>
    <Route exact={true} path='/NFL' render={() => (<League games={this.state.nflGames} league="NFL"/>)}/>
    <Route exact={true} path='/MLB' render={() => (<League games={this.state.mlbGames} league="MLB" />)}/>


    {hasMounted ? (
        this.state.nflGames.map((value, index) => {
            return (
                <Route
                    key={index}
                    exact={true}
                    path={`/NFL/${value.team1.replace(' ','-')}-${value.team2.replace(' ', '-')}-${value._id}`}
                    render={() => (
                        <Game gameData={value}/>
                    )}
                />
            );
        })
    ) : (
        <div></div>// <ReactLoading type={"spin"} color={"#ffffff"} height={'20%'} width={'20%'} />
    )}
```

1B.2 Refactoring

A) Duplicated Code

Before refactoring, we had duplicated code in the NFL, NBA, and MLB page components. Each page had almost the exact same code only differing by the league. Now we have deleted the individual league components and created a generalized League component where we can specify what league it is through React props.

Before:

```
<Route exact={true} path='/Nba' render={() => (<Nba games={this.state.nbaGames} />)}/>
<Route exact={true} path='/Nfl' render={() => (<Nfl games={this.state.nflGames} />)}/>
<Route exact={true} path='/Mlb' render={() => (<Mlb games={this.state.mlbGames} />)}/>
```

After:

```
<Route exact={true} path='/NBA' render={() => (<League games={this.state.nbaGames} league="NBA"/>)}/>
<Route exact={true} path='/NFL' render={() => (<League games={this.state.nflGames} league="NFL"/>)}/>
<Route exact={true} path='/MLB' render={() => (<League games={this.state.mlbGames} league="MLB" />)}/>
```

B) Duplicated Code

In the Search component, we had three different methods to encode the team locations into their respective abbreviations. The only difference was that the leagues so we combined them into one general method with a league parameter.

Before: (One of the three)

```
encodeNflTeam(team) {
    const { nflTeams } = this.state;
    if(nflTeams === null) return 0;
    var i;
    var teams = nflTeams;
    for(i = 0; i < teams.length; i ++) {
        if(teams[i].shortName === team) {
            return teams[i];
        }
    }
}
```

After:

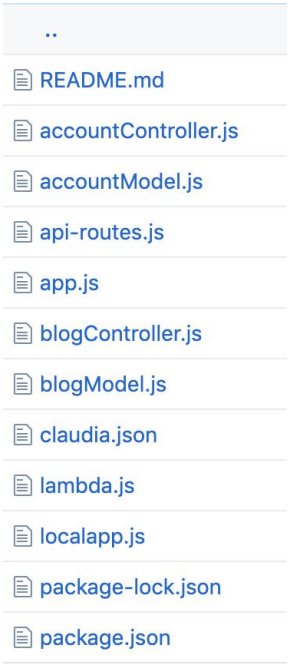
```
encodeTeam(team, league) {
    const { nbaTeams, mlbTeams, nflTeams } = this.state;
    let teams;
    league = league.toLowerCase();
    if(league === 'nba') {
        teams = nbaTeams;
    } else if(league === 'nfl') {
        teams = nflTeams;
    } else if(league === 'mlb') {
        teams = mlbTeams;
    }

    for(let i = 0; i < teams.length; i ++) {
        if (teams[i].location === team) {
            return teams[i];
        }
    }
}
```

C) Refactored File Location

For both of the API directories, all files were initially sitting inside of the root folder. The files were refactored and moved to the ‘models’ and ‘controllers’ folders. This allows for clearer understability and higher extensibility.

Before



After