User stories/User requirements. Describe the flows that users will go through and how they will interact with the application.

- Flow of the app
    - (eventually) user login/logout
        - All endpoints will only be accessible through authorization of the user (ie. they must be logged in)
    - Can interact with the application through the url AND also through a very simple UI (for GET calls)
        - Creating queries in the url
        - Or create queries by entering data in input field text boxes
        - The screen will display a visually reasonable format of the return data
    - Can also interact with the application through a very simple UI
        - A field and button to enter data for various endpoints
        - A section that displays a table of results and or information about the given query
- User stories
    - As a user, I want to be able to login/logout so that my info is secure
        - As a user, I want all endpoints to require authorization so that only authorized users can access the application
        - As a user, I want to be able to create an account if I do not already have one
    - As a user, I want to be able to interact with the application through the URL so that I can easily navigate and use the application
        - As a user, I want to be able to create queries in the URL so that I can retrieve specific information
        - As a user, I want to be able to add athletes to the database through the URL so that the database is up to date
    - As a user, I want to be able to interact with the application through a simple UI so that I can easily use the application

        - As a user, I want the screen to display the return data in a visually reasonable format so that I can easily read and understand it
        - As a user, I want a field and button to enter data for various endpoints so that I can input data easily
    - *Why All Sports Warehouse is better than a CRUD*: As a user, I want to be able to compare athletes within a sport

Documentation on what endpoints you will create. This should be at the same level of detail as what I provided in Assignment 1.

```
@router.get("/athletes/{id}", tags=["athletes"])
def get_athlete(id: int,
                year: int = None
                ):
    """
    This endpoint returns a single athlete by its identifier. For each athlete it returns:
    * `athlete_id`: The internal id of the athlete
    * `name`: The name of the athlete

    If the year argument is specified, the endpoint returns athlete stats for that year.
    If the year argument is not specified, for each NBA season the athlete was a part of, the endpoint returns
    the year and stats of the athlete for that season.

    Athlete stats include:
    age, team_id, team_name, games_played, minutes_played, field_goal_percentage, free_throw_percentage,
    total_rebounds, assists, steals, blocks, points
    """
```

```
@router.get("/athletes/", tags=["athletes"])
def compare_athletes(
        year: int,
        athlete_ids: List[int] = Query(None),
        stat: StatOptions = StatOptions.points,
):
    """
    This endpoint returns a comparison between the specified athletes,
    and returns the athlete id, name, and stat as specified in the input.
    It allows the user to compare athletes by a stat in `StatOptions`
    * `year`: the year to compare the athletes
    * `athlete_ids`: list of athlete names to compare (must have length >1)
    * `stat`: stat to compare athletes by (defaults to points)
    """
```

```
@router.post("/athletes/", tags=["athletes"])
def add_athlete(name: str):
    """
    This endpoint adds an athlete to the database.
    To add stats of a season in which the athlete played, use add_athlete_stats.
    The endpoint returns the id of the resulting athlete that was created
    This endpoint ensures the athlete does not already exist in the database
    """
```

```
@router.post("/athletes/season", tags=["athletes"])
def add_athlete_season(athlete: AthleteJson):
    """
    This endpoint adds the stats from an athlete's season to the database.
    The athlete is represented by the AthleteJson, which contains
    * athlete_id: the internal id of the athlete
    * age: age of the athlete
    * team_id: the team id of the athlete’s team
    * stats: a dictionary, matching StatOptions (such as "points") to values

    This endpoints ensures that athlete_id and team_id exists in the database, and that
    the athlete year pair does not already exist in the database
    The endpoint returns the name of the athlete
    """



@router.get("/teams/{team_name}/{year}", tags=["teams"])
def get_team(team_name: team_options, year: int):
    """
    This endpoint returns a single team by its identifier. For each team it returns:
        *`team_id`: The internal id of the team
        *`team_name`: The name of the team
        *`Wins`: Number of games the team won
        *`Losses`: Number of games the team lost
        *`Average Points for`: Average number of points the team scored
        *`Average Points allowed`: Average number of points team allowed
    """
```

```
@router.get("/teams/", tags=["teams"])
def compare_team(team_1: team_options,
                 team_2: team_options,
                 team_3: team_options = None,
                 team_4: team_options = None,
                 team_5: team_options = None,
                 compare_by: stat_options = stat_options.wins):
    '''
    This endpoint compares between 2 and 5 teams by a single metric
        * 'team_i': a team to be compared
        * Compare_by must be one of the following values
            * `wins`: The average wins per season
            * `points`: The average points per game
            * `rebounds`: The average rebounds per game
            * `assists`: The average assists per game
            * `steals`: The average steals per game
            * `blocks`: The average blocks per game
    '''




@router.get("/games/", tags=["games"])
def get_game(
        home_team: team_options,
        away_team: team_options
):
    """
```

```
    This endpoint returns a list of games by the teams provided ordered by date

    For each game it returns:
        * `game_id`: internal id of game
        * `home_team`: name of home team
        * `away_team`: name of away team
        * `winner_team`: name of winner team
        * `home_team_score`: score of home team
        * `away_team_score`: score of away team
        * `date`: the date the game was held
    """




@router.post("/games/add_game", tags=["games"])
def add_game(game: GameJson):
    """
    This endpoint adds a game to the database. The game is represented by:
        * `home_team_id`: the id of the home team
        * `away_team_id`: the id of the away team
        * `winner_id`: the id of the winner's team
        * Additional statistics about the game

    The endpoint returns the id of the game created
    """




@router.post("/teamratings/", tags=["ratings"])
```

```
def add_team_rating(rat: Rating):
    """
    This endpoint adds a user-generated team rating to the team_ratings table
    * `rat`: contains the team name (str) and rating (as int 1 --> 5) of the team

    The endpoint returns the id of the newly generated team rating
    """




@router.post("/athleteratings/", tags=["ratings"])
def add_athlete_rating(rat: Rating):
    """
    This endpoint adds a user-generated athlete rating to the athlete_ratings table
    * `rat`: contains the athlete name (str) and rating (as int 1 --> 5) of the team

    The endpoint returns the id of the newly generated athlete rating
    """




@router.get("/predictions/team", tags=["predictions"])
def get_team_market_price(team: team_options):
    """
    This endpoint returns the current market price of the specified team
    """
```

```
@router.get("/predictions/athlete", tags=["predictions"])
def get_athlete_market_price(athlete: str):
    """
    This endpoint returns the current market price of the specified athlete
    """
```

(Nice to have (not for initial 5 endpoint implementation))

list_athletes_in_team(team: str, limit: int, offset: int, sort: str): (GET)

- This endpoint will return a list of athletes in the specified team. For each athlete it will return:
    - Athlete_id
    - Athlete name
    - Team_ID
    - Age
    - Stats: stats is represented by a dictionary (see definition under get_athlete)
- It should also be able to sort by:
    - Athlete: sort athletes by name alphabetically
    - Age: sort athletes by age in either ascending or descending order
    - Stats: sort athletes by a specific stat in their sport in either ascending or descending order. This stat will be specific to `sport`.

list_all_athletes(athlete_name: str, limit: int, offset: int, sort: str) (GET)

- Returns all athletes whose name contains 'name.' If no name is given, it will return all athletes. For each athlete it returns:
    - athlete_id: The internal id of the athlete.
    - name: The name of the athlete
    - Team_id: The team id of the athlete's team
    - age : The age of the athlete
    - Stats: stats is represented by a dictionary (see definition under get_athlete)
- You can filter for athletes whose name contains a string by using the `name` query parameter.
- You can sort the results by using the `sort` query parameter:
    - athlete_id, name, sport, gender, age

- The `limit` and `offset` query parameters are used for pagination. The `limit` query parameter specifies the maximum number of results to return. The `offset` query parameter specifies the number of results to skip before returning results.

Detailed descriptions of edge cases and transaction flows. For example, if the app has a credit card checkout, describe what happens if the credit card transaction fails, what happens if the user tries to cancel mid-way through, etc.

- All user-writes will have data integrity checks
    - The data must
        - Not be a duplicate to something already in the DB
        - Match the required format and include all required fields for the given write
    - Failed writes will return an error code and error message to the screen, and prompt the user to follow the format for the given write
- All reads
    - All reads will have a limit to ensure cases where the data is large and may cause performance issues
    - Failed reads will return an error code and a message
    - Inputs will be converted to all uppercase
- User login creation
    - The username must be unique (must not already exist in the DB)
        - If it does already exist, inform the user
    - Require passwords with a minimum number of characters and special characters
    - Require the user to type in the password twice
- User login
    - Reject login after a specified number of failed logins
    - Have case sensitivity on the password