ChatGPT

Backend System Design for College Football Dynasty Tracker

Overview and Scope

This document specifies a backend system for a college football dynasty tracking application. The system will run on a local Raspberry Pi and expose a RESTful API built with **Python Flask** (for integration with a React frontend) and a **relational database** (using SQLite for simplicity, with the option to switch to PostgreSQL). The design emphasizes simplicity (manual stat entry, no user authentication) and supports 10–15 seasons of data. Key features and how the design addresses them are summarized below:

- **Season & Team Overview**: Track each season's details (team records, team prestige, overall ratings, etc.) in a relational model. Each team's season performance (wins, losses, ranking) and attributes like prestige and rating are stored per season for historical tracking.
- **Schedules & Scores**: Store a full regular season schedule and game results. Each game record will include participating teams, score, week, and flags for postseason rounds to support playoffs.
- **Team and Player Stats**: Record cumulative season stats for teams and players. Team stat rankings (offense, defense, etc.) and player stat leaders (passing yards, touchdowns, etc.) can be derived from these records.
- **Recruiting & Progression**: Manage recruiting classes and the transfer portal by modeling players and their team affiliation each season. New recruits are added as players with an initial overall rating (OVR), and their progression (OVR changes each year) is tracked season by season. Transfers are handled by updating a player's team in the next season's records.
- **Playoffs & Championships**: Support multi-round playoffs by marking playoff games and their round (e.g. quarterfinal, semifinal, final) in the schedule. This allows building a bracket view of the postseason.
- **Awards**: Track national awards and their recipients by recording award titles and the player (and team) that won each award in a given season.
- **Recruiting Class Rankings**: For user-controlled teams, store or compute the ranking of each recruiting class (e.g. via total recruit points or stars). This could be a field in the season record or computed from player data.
- **NFL Drafted Players**: Track which players get drafted each season. Each player record can carry a flag or year of being drafted, so that the number and list of drafted players per college team can be obtained for any season.
- **Promotion & Relegation**: Model conferences as entities and allow teams to change conference membership between seasons. Promotion/relegation is achieved by updating the team's conference in the new season's records (since conference is tied to the team's season entry, not the team itself).

This design uses a **RESTful API** to expose the data to the React frontend. There are no image or file uploads (all data is entered manually via forms), and no authentication or complex user management is required (only two users will use the system, and detailed tracking is primarily for one user's team). The following sections detail the data schema, API endpoints, and application structure to fulfill these requirements.

Architecture and Technology Stack

Tech Stack: The backend uses **Flask** (a lightweight Python web framework) with **Flask-SQLAlchemy** for database integration. SQLite will be used in development/local deployment, but the ORM will allow easy switching to PostgreSQL in the future (SQLAlchemy supports SQLite, PostgreSQL, etc. with just a URI change [1]). This ensures the solution is simple to set up on a Raspberry Pi and scalable if needed.

The Flask app will run a **RESTful JSON API**. The React frontend will communicate with this API using HTTP requests (e.g. using `fetch` or Axios) to read and update data. In development, the React app (running on a different port) will require CORS to be enabled on Flask (allowing cross-origin requests) [2], or a proxy can be configured. In production on the Pi, the Flask app can serve the React app's static files, avoiding CORS issues altogether.

Deployment: Since the app is local, a simple Flask development server (or Gunicorn for production) can be used. The React app can be built and its static files served by Flask. Flask has built-in support for serving static files from a `/static` directory [3], so the React build output (HTML, JS, CSS) can be placed there and served at the root or a specific route. For example, the React build can be published into Flask's `static/` folder and an `@app.route("/")` can return the `index.html`, allowing the single-page app to load and then use the API via AJAX calls.

No separate authentication or user account system is needed. If necessary, a simple configuration can mark which team is the user's primary team (to filter or highlight its data), but all data is open to both users.

Database Schema and Data Models

The data is organized into relational tables with clear relationships. We use a normalized schema to minimize redundancy and ensure data integrity [4]. The main entities include seasons, teams, conferences, players, games (matches), and various supporting tables for stats and awards [4]. Below is the schema with each table and its key fields:

Seasons

Tracks each year/season of the dynasty. - **season_id** (PK) – unique ID for the season. - **year** – Year or label of the season (e.g. 2025). Could also be an ordinal season number. - *Relationships*: One-to-many with TeamSeason records, games, award records, etc.

Conferences

Defines the conferences/leagues in which teams play (needed for promotion/relegation). - **conference_id** (PK) – unique ID for the conference. - **name** – conference name (e.g. *SEC*, *Big Ten*, *Tier-2 League*, etc.). - **tier/level** – *(Optional)* an indicator of conference level if using a tiered system (e.g. 1 for top tier, 2 for lower). - *Relationships*: One-to-many with TeamSeason (each TeamSeason entry links to a conference for that year).

Teams

Stores static information about each team (which remains across seasons). - **team_id** (PK) – unique team identifier. - **name** – team name (e.g. *Alabama Crimson Tide*). - **abbreviation** – short code (e.g. *BAMA*) if needed. - **primary_conference_id** – *(Optional)* default/home conference (if we want to note where the team started). - **is_user_controlled** – *(Optional)* boolean to mark the user's team or the friend's team, if needed for UI highlighting. - *Relationships*: Does **not** directly contain season-specific data to allow conference changes. Season-specific info is in TeamSeason.

TeamSeason (Season Team Records / Standings)

Associative table linking Teams to Seasons, capturing a team's performance and attributes in a specific season. This is essentially the "standings" or team season stats table [4]. - **team_season_id** (PK) – unique ID (or use composite of team_id+season_id as PK). - **team_id** (FK to Teams) - **season_id** (FK to Seasons) - **conference_id** (FK to Conferences) – conference the team played in **that season**. This allows reflecting promotion/relegation by changing this per season. - **wins**, **losses** *(int)* – Season record for the team. (Add **ties** if needed, though college OT usually prevents ties.) - **points_for**, **points_against** *(int, optional)* – Total points scored and allowed by the team that season. - **offense_yards**, **defense_yards** *(optional)* – Total offensive and defensive yardage (if tracking team stats). - **prestige** *(int or text)* – Team prestige level for that season (e.g. 1–6 for star prestige, or letter grade). This can change year to year as the program grows or declines. - **team_rating** *(int or text)* – Overall team rating at start of that season (e.g. an aggregate of player ratings, could be stored as an integer or letter like "93" or "A-"). - **final_rank** *(optional)* – Season-end ranking or playoff placement (if the team finished #1, etc.). - **recruiting_rank** *(optional)* – National rank of the team's recruiting class for that season. - *Relationships*: Many-to-one to Season and Team. This table is the source for generating standings, team stat rankings, etc. (For example, one can query TeamSeason to find the team with most points, best defense, etc., to identify rankings.)

By storing conference in TeamSeason, teams can be moved between conferences each year. Promotion/relegation is handled by creating the next season's TeamSeason entries with teams in their new conferences as appropriate.

Players

Represents individual players in the dynasty. Each player entry is persistent across seasons (until graduation or leaving). - **player_id** (PK) – unique player identifier. - **name** – player's name. - **position** – player position (QB, RB, WR, etc.). - **recruit_stars** *(optional)* – Recruit rating (e.g. 5-star, 3-star) when the player was recruited. - **recruit_rank_nat** *(optional)* – National ranking of the player as a recruit (if tracked). - **team_id (current)** – *Optional:* current team of the player. This can be updated as players transfer, but historical team data is in PlayerSeason. It might be convenient to store for quick reference the current team or last known team. - **current_year** – *Optional:* current year in college (e.g. FR/SO/JR/SR). - **career_stats** – *Optional:* aggregate career stats or achievements (could also be derived from PlayerSeason). - **drafted_year** *(optional)* – The year (season/year) the player was drafted to the NFL, if applicable. Null if not drafted or still in college. This allows querying drafted players per team by looking at players with drafted_year = X. - *Relationships*: One-to-many with PlayerSeason (each player has entries for each season played) and possibly AwardWinners.

Note: We do not tie a player permanently to one team here, because players can transfer. The historical team affiliation is handled in the PlayerSeason table. Storing only a "current team" in the Player table is mainly for convenience. As database design discussions note, relying only on a current team field would lose historical info; a separate association per season (or per tenure) is needed for full history [5] [6]. Our design uses PlayerSeason to fulfill that need.

PlayerSeason (Player Yearly Stats / Team Membership)

Associative table capturing a player's involvement each season – which team they were on, their year in school, OVR rating progression, and optionally their season statistics. This is crucial for tracking career progression and transfers. - **player_season_id** (PK) – unique ID (or use composite player_id+season_id). - **player_id** (FK to Players) - **season_id** (FK to Seasons) - **team_id** (FK to Teams) **– team the player was on during this season. -** player_class **– The player's class/year in that season (Freshman, Sophomore, etc., or redshirt status). Could also be an integer (1=FR, 2=SO, etc.). -** ovr_rating **– The player's overall rating *during that season*. This should be updated for each year to reflect progression (e.g., a recruit might start at 72 OVR as a freshman, then 78 as a sophomore, etc.). -** games_played **– number of games played (optional). -** stats **–** (Various fields or JSON)**: You can store season cumulative stats for the player. Depending on position, relevant stats differ (passing yards, rushing yards, tackles, etc.). For simplicity, one approach is to include a set of common stat fields: - e.g.** pass_yards, pass_tds, rush_yards, rush_tds, rec_yards, rec_tds, tackles, sacks, interceptions, **etc. (Any stats the user wants to track for stat leaderboards.) - Alternatively, a separate table `PlayerStats` could record individual stat categories (but for 10-15 seasons and manual entry, a flat structure in PlayerSeason might be easier). -** awards – *(Optional)* Could store flags if the player won certain awards this season (though we have an awards table for detailed info). - *Relationships*: Many-to-one to Player, Season, Team. This table enables queries like "find top performers of Season X" or "list the roster of Team Y in 2024." It's also used to log transfers: if a player changes teams, you will have two PlayerSeason records with different team_ids for consecutive seasons.

Recruiting and Transfers: A new recruit is entered as a Player plus an initial PlayerSeason entry for their first year on a team. A transfer is handled by creating a new PlayerSeason for an existing Player with a new team in the target season (the player stays the same ID). For example, if Player 123 is on Team A in 2025 and transfers to Team B in 2026, we create a PlayerSeason for 2026 with player_id=123 and team_id=B. This way, the system can answer questions like "how many years did a player play for Team A" by looking at their PlayerSeason records [5].

Games (Schedule and Results)

Stores individual games (matchups) and their results for both regular season and postseason. - **game_id** (PK) – unique ID for the game. - **season_id** (FK to Seasons) - **week** – week number or sequence of the game in that season. Regular season games might be 1-12; postseason games could be labeled differently or given high week numbers. - **home_team_id** (FK to Teams) – one team in the game (could be "home" team if applicable). - **away_team_id** (FK to Teams) – the other team. - **home_score**, **away_score** – final scores for the game. - **overtime** *(bool/int)* – optional flag if the game went to overtime (for record-keeping). - **game_type** – type of game: e.g. `"Regular"` for regular season, `"Playoff"` for playoff games, `"Championship"` or `"Bowl"` if a championship/bowl game. Could also use a boolean `is_playoff` and a separate field for round. - **playoff_round** – if game_type is playoff, specify the round: e.g. `"Quarterfinal"`, `"Semifinal"`, `"Final"`. This supports multi-round playoffs. For a 4-team playoff, you'd have Semifinal and Final; for larger playoffs, include rounds as needed. (Alternatively, round could be an integer or enum.) - **neutral_site** *(optional)* – flag if the game is at a neutral site (useful for bowls/ championships). - *Relationships*: Many-to-one to Season; many-to-one to each of two Teams. In a normalized design, you might use a separate junction for game participants, but here we assume exactly two teams per game, so two FK fields suffice.

The Games table allows retrieving the schedule and results for each season. A **regular season schedule** can be fetched by filtering game_type = "Regular" and season_id, ordered by week. Playoff bracket can be constructed by filtering game_type = "Playoff" and grouping by playoff_round – e.g., all Semifinal games of that season form the semifinal bracket, etc.

Awards

Stores the set of national awards that can be won (e.g. Heisman Trophy, Best QB award, etc.). - **award_id** (PK) – unique ID for the award type. - **name** – name of the award (e.g. *Heisman Trophy*, *Davey O'Brien Award*). - **description** – (optional) longer description or criteria. - *Relationships*: One-to-many with AwardWinners.

This table can be pre-populated with all awards of interest.

AwardWinners

Records the winner of each award for each season. - **award_winner_id** (PK) - **award_id** (FK to Awards) - **season_id** (FK to Seasons) - **player_id** (FK to Players) – the player who won the award in that season. - **team_id** (FK to Teams) – the team of that player (redundant but convenient; can be derived via PlayerSeason). - *Relationships*: Many-to-one to Award, Season, Player. This table lets us list "National Awards and recipients" per season. For example, query all AwardWinners for season 2025 to get the list of awards and who won each.

By using this schema, we maintain a separation of concerns: - Data is **normalized** (no duplicate storage of stats across tables unnecessarily). For example, team records are in TeamSeason, players in PlayerSeason, rather than duplicating data in one giant table. - The design is flexible: if the user chooses to track only their own team's detailed stats, the system still functions (other teams can have minimal data, e.g. one could omit creating PlayerSeason entries for every player on CPU teams). - It covers all requested features: multi-season history, per-season stats, dynamic conference membership, and so on, by linking the tables appropriately.

API Endpoints Design

We will expose a set of RESTful API endpoints under a base URL (e.g. `/api/`) to allow the React frontend to read and modify the data. All data is sent and received as JSON. The API follows standard REST conventions for HTTP methods (GET for retrieving data, POST for creating, PUT/PATCH for updating, DELETE for removing) [7]. In the descriptions below, `{id}` indicates a resource identifier. (Exact URL structure can be adjusted, but the following is one logical scheme.)

Season and Conference Endpoints

- **GET** `/api/seasons` – Retrieve list of seasons. Returns an array of seasons (each with `season_id` and year, plus possibly summary info like champion or user team record).
- **POST** `/api/seasons` – Create a new season. Expects JSON with the year (and perhaps initial setup info). This would typically trigger creation of TeamSeason records for each team in that season (possibly carrying over teams from last season). If automation is desired, the backend could copy all teams from the previous season's TeamSeason and set initial values (e.g., 0-0 record).
- **GET** `/api/seasons/{season_id}` – Get details of one season, including perhaps a summary of standings or champion. (Alternatively, the client might gather details via other endpoints.)
- **GET** `/api/seasons/{season_id}/teams` – Get all teams in that season with their season stats (standings). Essentially returns the list of TeamSeason entries for the season, including fields like wins, losses, prestige, etc. This can be used to display standings or conference tables.
- **PUT** `/api/seasons/{season_id}/teams/{team_id}` – Update a team's season info. For example, after a season ends, the user can update the team's prestige or record if not already updated. This can also handle conference changes (e.g. moving a team to a new conference for next season by updating the TeamSeason's conference_id).
- **GET** `/api/conferences` – List all conferences (names and IDs).
- **POST** `/api/conferences` – (Optional) Create a new conference (likely not needed often, as conferences are mostly static).
- **GET** `/api/conferences/{id}/teams` – Get all teams that *currently* belong to a conference (could specify a season to get membership for that year). Alternatively, conference membership is seen via TeamSeason in a given season.

Team Endpoints

- **GET** `/api/teams` – Retrieve list of all teams (basic info). Could include current conference or overall record, etc.
- **POST** `/api/teams` – Add a new team to the league (if the user wants to introduce a new team in the dynasty).
- **GET** `/api/teams/{team_id}` – Get info for a specific team (name, etc.) plus perhaps summary of its historical performance. (We might not precompute history, but we could include links or counts of championships, etc.)
- **GET** `/api/teams/{team_id}/seasons` – Get all seasons that this team has played (list of TeamSeason records for this team). This can show the team's year-by-year record, prestige, etc. Useful for a team's history page.
- **GET** `/api/teams/{team_id}/players` – Get all current players on this team (active roster). By default, this could return the roster for the latest season. A query parameter or separate endpoint can specify the season:
- Alternatively, use **GET** `/api/seasons/{season_id}/teams/{team_id}/players` – to get that team's roster **for a specific season**. This returns players with their PlayerSeason info (OVR, class, season stats) for that year.
- **PUT** `/api/teams/{team_id}` – Update team info (e.g. rename, etc., though not likely needed). Could also be used to mark a team as user-controlled.

Player and Recruiting Endpoints

- **GET** `/api/players/{player_id}` – Get detailed info about a player (name, position, current team, career stats if any, etc.). This could also embed or link to their season-by-season stats.
- **POST** `/api/teams/{team_id}/players` – Create a new player on a given team. This endpoint would be used to add recruits or transfer players to a team. The request JSON would include player details (name, position, initial OVR, etc.) and a season. On the backend, this will create a new Player record and a PlayerSeason record for the specified season (e.g. the upcoming season as a freshman on that team). If the season isn't explicitly given, it could default to the current/offseason context.
- **POST** `/api/players/{player_id}/seasons` – Add a season entry for an existing player. This can handle player progression or transfers. For example, to record a player's new season (with increased rating and year, possibly a new team if transferred). JSON would include `season_id`, `team_id`, `ovr_rating`, etc. This creates a new PlayerSeason entry for that player. (This could also be achieved with the above endpoint by allowing the JSON to specify an existing player_id.)
- **PUT** `/api/players/{player_id}` – Update a player's information. For instance, when a season rolls over, one might update a player's `current_year` (FR->SO) or mark if they were drafted. More often, new data is added via PlayerSeason rather than altering the base Player.
- **DELETE** `/api/players/{player_id}` – Remove a player (if, say, a player quits or you input someone by mistake). This would remove their base record and cascade to their season stats.

These player endpoints allow managing rosters and progression. For example, **recruiting a new class** involves multiple calls to POST `/teams/{id}/players` for each new recruit on the user's team (with the new season's ID). **Transfers** can be handled by posting a new PlayerSeason for the player on the new team's endpoint.

Game (Schedule/Score) Endpoints

- **GET** `/api/seasons/{season_id}/games` – Retrieve all games in a season (could be huge if all teams, but for completeness). Support query params to filter, e.g. `?team={team_id}` to get games of a specific team, or `?week=X` to get a specific week's schedule.
- **GET** `/api/games/{game_id}` – Get details of a single game (including scores and teams).
- **POST** `/api/games` – Create a new game record. This can be used before the season to input the schedule, or during the season to add games as they're played. The JSON would include season, week, home team, away team, and optionally score if the game is already played.
- **PUT** `/api/games/{game_id}` – Update a game's result. This is important for manual stat entry: the user can input the score after a game is played. This could also update a game from "scheduled" to "completed" state. (Alternatively, we could create all games with no score initially, then update scores later.)
- **DELETE** `/api/games/{game_id}` – Delete a game (if needed to remove an erroneous entry or canceled game).

For playoffs specifically: - **GET** `/api/seasons/{season_id}/games?type=playoff` – Filter to playoff games of that season. The frontend can then group by `playoff_round` to display the bracket. - (If needed, a convenience **GET** `/api/seasons/{season_id}/bracket` could return games grouped by round in structured format, but this can also be done client-side.)

Stats and Leaderboard Endpoints

While much of the stats can be derived on the frontend by fetching the relevant data, we can provide endpoints to directly get rankings and leaders: - **GET** `/api/seasons/{season_id}/leaders` – Returns top players in key stat categories for that season. For example, it could return a JSON with separate lists: passing leaders, rushing leaders, etc., each list containing players with their stat values. (The server would query PlayerSeason table, sort by the stat field, and take top N.) - **GET** `/api/seasons/{season_id}/standings` – Returns the standings by conference. This could be similar to `/seasons/{id}/teams` but possibly formatted by conference (e.g. a nested structure of conference -> teams). It could also include computed metrics like win percentage or streaks. - **GET** `/api/seasons/{season_id}/awards` – Returns the list of awards and winners for that season (joining AwardWinners with Players and Teams to get names). This provides the "national awards and recipients" in one call. - **GET** `/api/teams/{team_id}/drafted` – (Optional) Returns players from a given team who were drafted (possibly filter by season, e.g. drafted in a specific year). This can be derived from players' drafted_year field combined with their team history. For example, to get "drafted players per team for season X", find all players with drafted_year = X+1 (draft after that season) and who have a PlayerSeason in season X with that team. We might not need a dedicated endpoint if the front-end can query all players drafted that year and filter by team.

These specialized endpoints are not strictly required, but they illustrate how the API can directly answer front-end needs: - The **React app** could also compute leaders by pulling all players, but providing it via an API simplifies the front-end. - Similarly, recruiting rankings for user's team could be computed (sum of recruit ratings) and even stored in TeamSeason as `recruiting_rank` or a computed field; an endpoint could expose a list of team recruiting rankings for a given year if the user wants to compare (though if only one user team is tracked in detail, this might not be heavily used).

All endpoints will return JSON. For example, a GET on `/api/seasons/2025/teams` might return:

```
[
  {
    "team_id": 1, "team_name": "Team A", "conference": "SEC",
    "wins": 12, "losses": 1, "prestige": 5, "team_rating": 95,
    "points_for": 450, "points_against": 300, "recruiting_rank": 3
  },
  { "team_id": 2, "team_name": "Team B", ... },
  ...
]
```

And a GET on `/api/seasons/2025/teams/1/players` (Team A's roster in 2025) might return:

```
[
  {
    "player_id": 101, "name": "John QB", "position": "QB", "class": "JR",
    "ovr_rating": 90, "pass_yards": 3500, "pass_tds": 30, "rush_yards": 200, ...
  },
  { "player_id": 102, "name": "Mike RB", "position": "RB", "class": "SO",
    "ovr_rating": 85, "rush_yards": 1200, "rush_tds": 15, ... },
  ...
]
```

This shows how the data might be structured in JSON for use in the frontend.

The API is stateless and uses standard HTTP status codes for success or errors. For instance, `POST /api/games` returns 201 Created with the new game JSON, `PUT` returns 200 OK, etc. We will implement validation (e.g., not allowing a game to be created with teams not in that season, etc., to maintain referential integrity).

Flask Application Structure and Integration with React

The Flask application will be organized to separate concerns and enable easy maintenance: - We will use **Flask blueprints** or a modular structure for the API. For example, an `api` blueprint could be created with sub-modules for different resources (teams, players, games, etc.), or we can define routes in separate files and register them. - **Models**: Using Flask-SQLAlchemy, define a model class for each table described in the schema (Season, Conference, Team, TeamSeason, Player, PlayerSeason, Game, Award, AwardWinner, etc.). SQLAlchemy will manage relationships (e.g., `PlayerSeason` model can have a relationship to `Player` and `Team` model). By using SQLAlchemy, we can switch the database by simply changing the `SQLALCHEMY_DATABASE_URI` config: e.g., `"sqlite:///dynasty.db"` for SQLite or a `"postgresql://user:pass@host/dbname"` for Postgres [8]. SQLAlchemy's ORM provides a high-level interface to query and update the data. - **Database initialization**: The app will include setup to create tables from the models (using `db.create_all()` if no migration tool is used). We might create a script to initialize with some default data (teams, conferences, awards). - **Routes**: Each endpoint corresponds to a Flask route function. These functions will interact with the database via the models: - On GET, query the database (possibly using SQLAlchemy queries) and return results as JSON (Flask can automatically JSONify dictionaries and lists [9]). - On POST/PUT, parse `request.json`, perform the appropriate create or update, commit to the DB, and return the new/updated object as JSON. - We will ensure that CORS is handled. In development, since the React dev server runs on a different port (e.g., 3000) than Flask (5000), we must enable cross-origin requests. We can use the `flask_cors` extension: `from flask_cors import CORS; CORS(app)` will allow all origins by default [2]. (Alternatively, in development one can use the React proxy setting or configure specific allowed origins.) - **Error Handling**: The API will include basic error handling – e.g., if a resource is not found, return 404, if invalid data, return 400 with an error message, etc. Flask's `abort()` or custom error handlers can be used.

Application Structure Example:

```
dynasty_app/
├── app.py (Flask app creation and blueprint registration)
├── models.py (SQLAlchemy models for all tables)
├── routes/
│   ├── seasons.py  (endpoints for seasons and conferences)
│   ├── teams.py    (endpoints for teams and team-season data)
│   ├── players.py  (endpoints for players and player-season)
│   ├── games.py    (endpoints for games)
│   └── awards.py   (endpoints for awards and award winners)
├── static/
│   └── (React build output: index.html, static JS/CSS files)
└── templates/ (if needed, e.g., could hold index.html if using render_template)
```

In `app.py`, we initialize the Flask app and database:

```
app = Flask(__name__)
app.config["SQLALCHEMY_DATABASE_URI"] = "sqlite:///dynasty.db"
# or use .env to configure
db.init_app(app)
# Register blueprints
from routes import seasons, teams, players, games, awards
app.register_blueprint(seasons.bp, url_prefix="/api")
app.register_blueprint(teams.bp, url_prefix="/api")
...
# Serve React front-end
@app.route("/", defaults={"path": ""})
@app.route("/<path:path>")
def serve_frontend(path):
    if path != "" and os.path.exists("dynasty_app/static/" + path):
        return send_from_directory("static", path)
    else:
        return send_from_directory("static", "index.html")
```

In the above example, any unknown route is served the React app's `index.html` (to support client-side routing). Flask's static file serving is leveraged for the React assets, as Flask will automatically serve files in the `static/` folder [3].

React Integration: The React app will communicate with the Flask API via HTTP requests. For instance, it might use `fetch('/api/seasons/2025/teams')` to get standings, or `axios.post('/api/games', {...})` to submit a new score. Because we are serving the React app from the same domain (when deployed on the Pi), these calls have the same origin and no extra CORS configuration is needed in production. In development, as noted, we either use Flask-CORS or set up a proxy so that API calls from React (localhost:3000) to Flask (localhost:5000) are permitted.

Example integration snippet (React):

```
// Using fetch to get the user team's roster for 2025
useEffect(() => {
  fetch('/api/seasons/2025/teams/1/players')
    .then(res => res.json())
    .then(data => setRoster(data));
}, []);
```

This would call our Flask endpoint and retrieve JSON, which the React component can then use to render the roster. The medium tutorial confirms this pattern of fetching data from Flask and setting React state [10]. Similarly, when the user submits a form (e.g. to add a recruit or update a score), the React code will send a POST/PUT request to the appropriate endpoint and handle the response.

Security & Auth: Since this application is for personal use with two known users and runs locally, we skip authentication entirely. All API endpoints are unprotected. If in the future this were exposed beyond the Pi, one might add token auth or simple login, but it's not in scope now.

Performance Considerations: The dataset (10–15 seasons, two user-controlled teams in detail, possibly ~100 teams total if a full league) is relatively small. SQLite can handle this with ease on a Raspberry Pi. Queries for leaders or standings are simple aggregations or sorts on at most a few thousand records, which is fine. If we scale up (more teams or many seasons), PostgreSQL could be used for better concurrency and robustness – the code would not drastically change thanks to the ORM abstraction (just the connection string changes to point to Postgres [1]). We also ensure to index key fields (primary keys are indexed by default; we might add indexes on foreign keys like season_id in the PlayerSeason table to speed up season queries).

Manual Data Entry: The frontend will provide forms and tables for the user to input stats. For example, after a game, the user can enter the score and some key player stats. These will call the appropriate Flask endpoints (e.g., update game score, update player stats). The design aims to make this as straightforward as possible: e.g., one could have a single form to input a game's stats and the backend could accept a complex JSON with game score and a few player stat highlights to update multiple tables in one go. However, for clarity and simplicity, we keep endpoints discrete (each responsible for one part of the data), and the React app can orchestrate multiple calls if needed (e.g., one to update game score, another to update a player's stat line).

In summary, this technical design provides a clear separation of data models, a robust relational schema for tracking a college football dynasty across seasons, and a set of RESTful API endpoints for all CRUD operations. The Flask backend, coupled with a React frontend, will allow the two users to easily input and view detailed information about each season – from schedules and scores to player stats and awards – with all data persisted in a local database. By following REST principles and using JSON throughout, the system remains intuitive to use and easy to integrate [7]. The use of Flask and SQLAlchemy ensures that the solution is lightweight enough for a Raspberry Pi yet flexible enough to evolve (e.g., switching to PostgreSQL or adding new features) without a complete rewrite.

---

[1] How to Use Flask-SQLAlchemy to Interact with Databases in a Flask Application | DigitalOcean
https://www.digitalocean.com/community/tutorials/how-to-use-flask-sqlalchemy-to-interact-with-databases-in-a-flask-application

[2] [10] Connecting your React App to your Flask API (A step by step guide) | by Nuburooj Khan | Medium
https://medium.com/@nuburoojkhattak/connecting-your-react-app-to-your-flask-api-a-step-by-step-guide-3daa8ce9d3f2

[3] Hot reloading with React and Flask | by Andrew Hyndman | Medium
https://ajhyndman.medium.com/hot-reloading-with-react-and-flask-b5dae60d9898

[4] GitHub - kaimg/Sports-League-Management-System: A comprehensive sports league management system that streamlines the organization of sports competitions, handling team registrations, scheduling matches, tracking scores, and managing player statistics in real-time.
https://github.com/kaimg/Sports-League-Management-System

[5] [6] Sports database - help with schema : r/SQL
https://www.reddit.com/r/SQL/comments/sbtns7/sports_database_help_with_schema/

[7] Designing a RESTful API with Python and Flask - miguelgrinberg.com
https://blog.miguelgrinberg.com/post/designing-a-restful-api-with-python-and-flask

[8] Quick Start — Flask-SQLAlchemy Documentation (3.1.x)
https://flask-sqlalchemy.readthedocs.io/en/stable/quickstart/

[9] How To Create a React + Flask Project - miguelgrinberg.com
https://blog.miguelgrinberg.com/post/how-to-create-a-react--flask-project