Atom Football App: Full Architecture Explained

Here is a complete breakdown of how your application functions, file by file, following the flow of data from the API to the user's screen.

High-Level Summary

Your app works in three main phases:

1. **Phase 1: Initialization (Once per session)** When a user first loads the app, `app.py` runs a one-time, *blocking* `full_sync()`. This populates the database with all necessary data (teams, leagues, matches) so the app is immediately usable. It then starts a background `live_poller` thread.
2. **Phase 2: UI & Interaction (User-facing)** The user interacts with a Streamlit UI in `app.py`. They see a dynamic list of day-tabs. When they click a day, the app fetches the pre-synced data from its database (using `utils.py`) and displays it using components from `widgets.py`.
3. **Phase 3: Background Sync (Continuous)** The `live_poller` thread (from `fetch.py`) runs continuously in the background. Its only jobs are to (a) check for live match score updates every 5 minutes and (b) check if it's past midnight UTC. If it is midnight, it triggers a new `full_sync()` to fetch the next day's matches and update all data, ensuring the app is always current.

File-by-File Breakdown

1. `app.py` (The Conductor & UI)

This is the main entry point that the user runs with `streamlit run app.py`.

- **Imports:** It imports all necessary modules, including key functions from your other files like `full_sync`, `live_poller`, `match_card_component`, and `load_all_match_data`.
- **Config:** It sets the page title, icon, and layout (`st.set_page_config`).
- **PWA:** It calls `inject_pwa()` (from `pwa.py`) to add the necessary HTML tags to make the app a Progressive Web App (installable on mobile).
- **CSS:** It injects all your custom CSS styles for a responsive, dark-mode UI.

Key Logic: The `if "initialized" not in st.session_state:` Block

This is the most important part of `app.py`. It only runs **once** when a user's session starts.

1. `st.spinner("..."):` Shows the user a loading message.
2. `viewport()`: Uses JavaScript to get the user's screen width and saves it to `st.session_state`.
3. `test_connection()`: Pings your Supabase DB to ensure it's online.
4. `init_db()`: If the tables don't exist, this creates them (defined in `fetch.py`).

5. `full_sync()` : This is the **first fix** we implemented. It runs the *entire* data sync process (fetching areas, teams, standings, and matches) one time, *blocking* the app. This ensures that when the UI loads, there is data to show.
6. `threading.Thread(target=live_poller, ...)` : This is the **second fix**. After the initial sync, it starts the `live_poller` function in a separate background thread. This thread will run forever, handling all future updates without blocking the UI.
7. `st.rerun()` : Forces Streamlit to reload the app from the top, skipping the `if "initialized"` block.

Main UI (After Initialization):

1. **Dynamic Tabs:** It reads the screen width from `st.session_state` and calculates how many day-tabs to show (from 3 on mobile to 15 on desktop).
2. **Data Load:** It calls `load_all_match_data()` (from `utils.py`). This function is cached (`@st.cache_data`) for 5 minutes to avoid constant database calls. It pulls *all* match data from the database.
3. **Data Sorting:** It organizes the list of matches into a dictionary (`matches_by_date`) to easily find matches for each tab.
4. **View Routing:** It checks `if st.session_state.selected_match:` .
   - **If a match IS selected:** It calls `show_match_details()` (from `widgets.py`) and stops. This is how it switches to the "Details" page.
   - **If no match is selected:** It proceeds to draw the main tabbed interface.
5. **Tab Rendering:** It loops through each date-tab. Inside a tab, it groups matches by competition (e.g., "Premier League") and puts them inside an `st.expander` .
6. **Component Rendering:** Inside each expander, it loops through the matches and calls `match_card_component(match_data)` (from `widgets.py`) to draw each individual match card.
7. **Sidebar:** It displays database statistics (e.g., "Upcoming: 50") using `count_table()` (from `utils.py`).

2. `fetch.py` (The Data Engine)

This file is responsible for all communication with the football-data.org API and the database.

- **Config:** Sets API keys, rate limits (`MAX_REQUESTS_PER_MINUTE = 8`), and date ranges. The **critical fix** here was setting `ENRICH_ONLY_FUTURE = False` to ensure H2H/Last7 data is fetched for *all* matches.
- `rate_limiter` : A class that forces the app to pause between API calls to stay within the 8-requests-per-minute free tier limit.
- `fetch_api()` : A helper function that makes the actual `requests.get()` call, handles rate-limiting, and retries on errors.

Data Fetching Functions (`fetch_areas` , `fetch_competitions` , `fetch_teams` , `fetch_standings`):

- These functions all follow the same pattern (this was our **third major fix**):
  1. Fetch data from the API (e.g., `fetch_api("areas")`).
  2. Open a database session (`with Session() as session:`).
  3. Loop through the items from the API.
  4. Use `session.get(Model, item_id)` to see if the item (e.g., a Team) already exists.
  5. If it exists, update its properties. If not, create a new object.
  6. Use `session.merge()` to safely "upsert" (update or insert) the record in a thread-safe way, preventing the `psycopg` errors.
  7. `session.commit()` saves all changes at once.

Match Fetching (The Core):

- `process_match_smart()`: This is the heart of the enrichment. For a *single* match from the API:
  1. It calls `safe_team_last7()` and `safe_head2head()` to get extra data.
  2. `safe_team_last7()`: Fetches a team's last 7 *finished* matches (e.g., `teams/57/matches?status=FINISHED&limit=7`).
  3. `safe_head2head()`: Fetches H2H data (e.g., `matches?h2h=57x64`).
  4. It formats this data and saves *everything* (the match data, H2H, and Last7) into a `Match` object in the database using `session.merge()`.
- `fetch_matches_smart()`: A wrapper that calls `process_match_smart()` in a `ThreadPoolExecutor` (multi-threaded) to speed up fetching dozens of matches at once.

Sync & Poller Functions:

- `full_sync()`: The main task. It runs all the `fetch_...` functions in order: areas, competitions, teams, standings, future matches, and finished matches. Finally, it calls `predict_for_matches()`.
- `live_poller()`: The background task. It's an infinite loop:
  1. `check_day_rollover()`: Checks if the UTC time is just after midnight (00:00 - 00:05). If yes, it runs the `full_sync()`.
  2. `fetch_live_matches()`: Does a quick check for any matches with status "LIVE" to update scores.
  3. `time.sleep(300)`: Sleeps for 5 minutes and repeats.

3. `predict.py` (The "Brain")

This file analyzes the data *after* `fetch.py` has saved it.

- `predict_for_matches()`: The main function, called at the end of `full_sync()`.
  1. It queries the database for all matches that *do not* have a prediction yet (`Match.prediction.is_(None)`).
  2. It loops through each of these matches.

3. It loads the `Standing` table to get the rank of the home and away teams.
4. It loads the `match.home_last7` and `match.h2h` JSON data (which `fetch.py` saved).
5. `analyze_last7()` / `analyze_h2h()`: Helper functions that parse this JSON and the `result` string (e.g., "Liverpool 2-0 Arsenal") into structured data (like `goals_for: 2`, `outcome: "W"`).
6. `assign_tags_for_team()`: This is your expert logic. It takes the analyzed data and applies your rules (e.g., `if gf and any(g >= 2 for g in gf[-3:]): tags.append("Score At least 2 goals")`).
7. It saves the resulting list of tags (e.g., `['Win', 'Score At least 2 goals']`) into the `match.prediction` column in the database and commits the change.

4. `widgets.py` (The "Frontend" Components)

This file contains functions that *draw* UI elements in `app.py`.

- `match_card_component()`: Draws the small card for the main match list. It uses `st.container` and `st.columns` to lay out the team crests, names, and the "Details" button.
- `open_match_details()`: This is the *callback* for the "Details" button. Its only job is to `st.session_state.selected_match = match`. This simple change is what tells `app.py` to switch to the details view on its next rerun.
- `show_match_details()`: Draws the full match details page.
  - It adds a "Back to Matches" button (which works by setting `st.session_state.selected_match = None`).
  - It calls `display_last7_match_list` and `display_h2h_match_list` to render the statistics.
- `display_last7_match_list()` / `display_h2h_match_list()`: These functions use `get_structured_match_info` to parse match results and then inject custom HTML/CSS (like the green "W" box) using `st.markdown(..., unsafe_allow_html=True)`.

5. `models.py`, `utils.py`, `pwa.py` (The Toolbox)

- `models.py`: Defines the database "schema" using SQLAlchemy. It tells the app what columns exist for `Area`, `Competition`, `Team`, `Standing`, and `Match`.
- `utils.py`: A collection of shared helper functions used by both `app.py` and `widgets.py`, such as `load_all_match_data` (cached DB query) and `get_structured_match_info` (result string parser).
- `pwa.py`: Contains all the logic for registering the web app's service worker and manifest file, allowing it to be "installed" on a user's device.