NashBoard

CUS 720 - Design Phase

Adam Nash

Introduction

The purpose of this Design Phase document is to take the finalized NashBoard Specification and turn it into a set of concrete models that show how the system will actually be built and how it will behave when people use it. In the specification phase the focus was mainly on *what* NashBoard had to do: the features, constraints, user needs, and project scope. In this phase the focus shifts to *how* those things will be implemented using standard software-engineering tools like UML diagrams, architectural patterns, and clearly defined components.

NashBoard is meant to be a customizable sports-stats dashboard that pulls data from external sports APIs and lets users view and arrange widgets the way they like. Because there are a lot of moving parts (users, admins, APIs, layouts, widgets, caching, etc.), it is not enough to describe the system in plain text. Larger systems are easier to build and maintain when they are viewed from multiple angles. That is why this document mixes diagrams and narrative: each model shows NashBoard from a different point of view and ties back to the same set of requirements from the spec.

In this design, the context model shows where NashBoard sits in the bigger environment and which external systems and actors it talks to. The use case diagram and tables describe how different users (registered users, guests, and admins) interact with the system and what goals they can achieve. The set of activity diagrams break key flows into step-by-step behavior, such as registering or logging in, searching for players, viewing player details, customizing the dashboard layout, using admin tools, and handling predictive widgets that are planned for a later version. The sequence diagrams zoom in further and show the detailed message flow between the UI, backend services, cache layer, database, and sports data APIs for those same core scenarios.

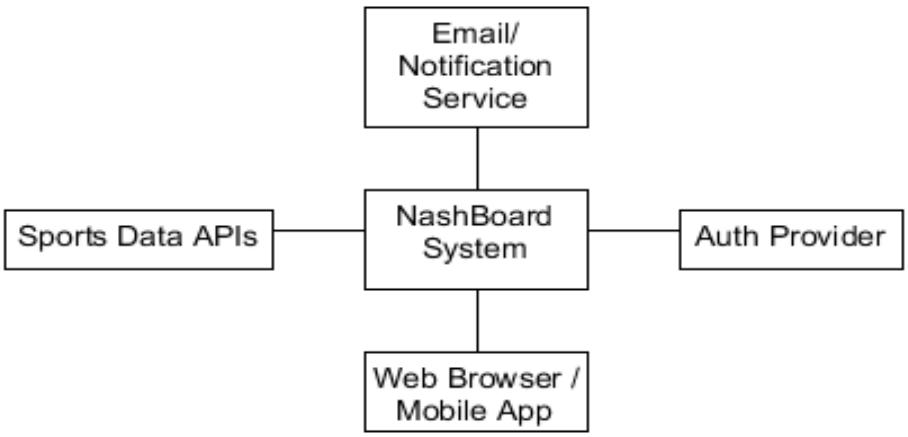
On the structural side, the domain class diagram captures the main entities NashBoard cares about (users, dashboards, widgets, players, teams, games, and cached data) and how they relate to each other. The component diagram groups the implementation into major services and shows how the frontend, backend, data layer, and external services are wired together. A state diagram models how a user account moves between states like pending verification, active, locked, and disabled, which ties into the security requirements. Finally, the architecture section explains the three-tier layered web pattern used for NashBoard and how the current Next.js / Prisma / PostgreSQL implementation fits into that structure.

This design phase is not a brand-new description of NashBoard. It sits on top of the approved specification and is meant to be a realistic blueprint that a developer could implement and extend over time. Every diagram in the following sections is traced back to the functional and non-functional requirements from the spec, and together they form

the overall picture of how NashBoard should work internally once it moves into the implementation phase.

Context Model

The context model shows NashBoard at the highest possible level—before any internal components, classes, or services come into play. The goal of this diagram is to clearly outline what sits *outside* the system and what NashBoard must communicate with in order to function. This includes people, external services, and any system that exchanges information with NashBoard during normal operation. In the center of the model, NashBoard is represented as a single unified system. Surrounding it are the major actors that interact with it. Registered Users represent the primary audience, accessing dashboards, customizing layouts, searching players, and managing watchlists. Guest Users can still browse sports dashboards and perform basic searches, but they do not have access to personalization features. The Admin actor sits apart from typical user flows and interacts with NashBoard for maintenance operations like modifying system settings or handling account-level issues. On the system side, NashBoard depends on several external services. The Sports Data APIs provide the live scores, stats, standings, injuries, and game data that populate dashboards and player pages in real time. The Notification/Email Service supports flows like account registration, email verification, and password resets. These interactions define the boundaries of NashBoard and show the environment the system operates in. Everything outside the boundary box is something NashBoard cannot directly control but must respond to reliably.Together, this context model establishes the “big picture” view of NashBoard. It sets up the foundation for the rest of the design by making the external responsibilities and data flows explicit. Each of the more detailed diagrams in later sections will zoom inward to show how the system fulfills these interactions internally.



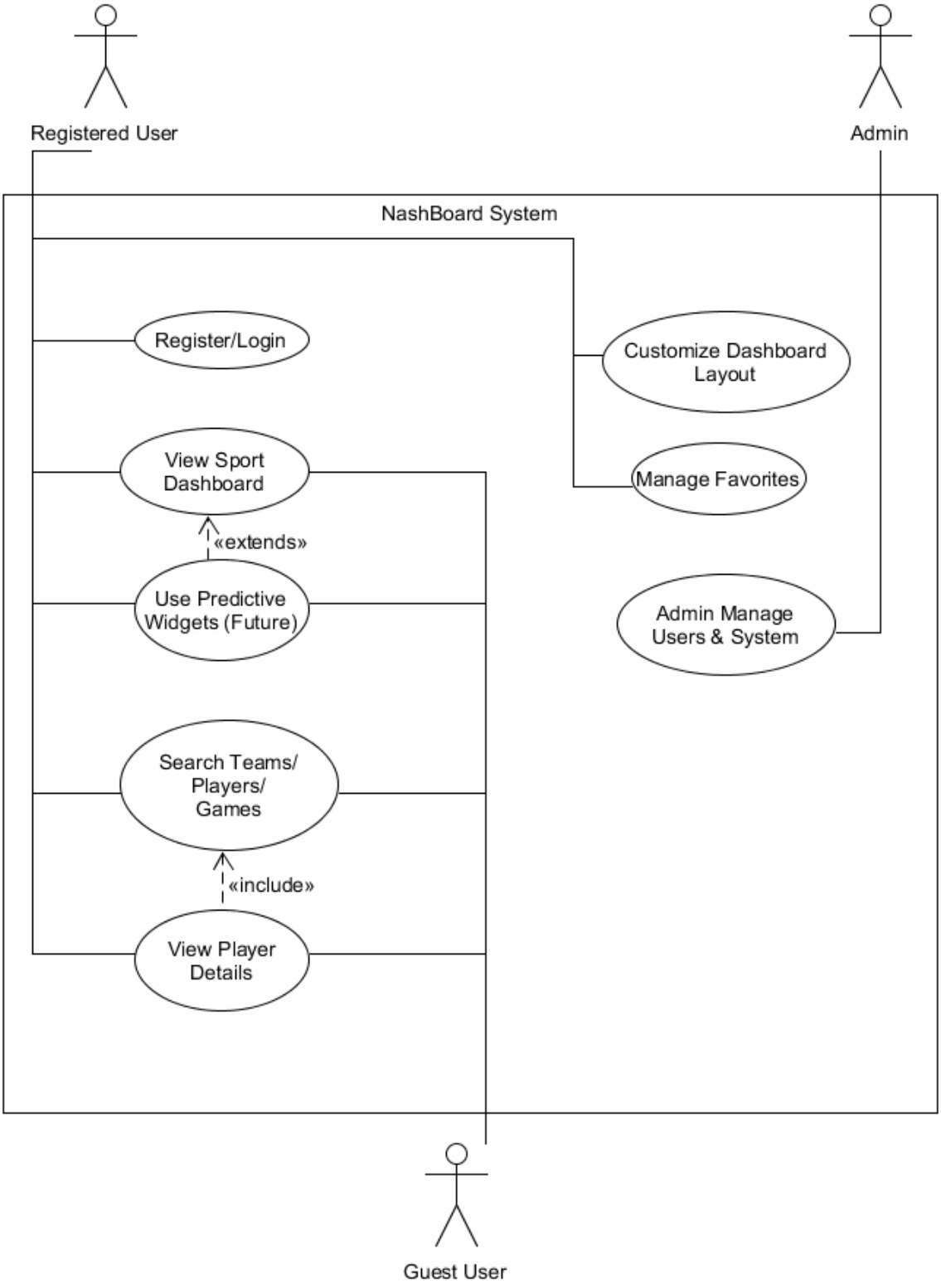
Use Case Model

The use case model provides a structured view of all the major interactions a user can have with NashBoard. While the context diagram shows the system from the outside, the use case diagram goes one step closer by mapping out *what* users can accomplish when they interact with the application. Each oval in the diagram represents a meaningful goal from the user's perspective, and each actor represents a role that initiates or participates in those goals.

In NashBoard, there are three primary actors: Registered Users, Guest Users, and Admins. Registered Users have access to the full experience, including personalized dashboards, custom widget layouts, player watchlists, and deeper analytics. Guest Users can still browse dashboards and perform searches, but they are restricted from personal features like saving layouts or managing favorites. The Admin actor interacts with the system for maintenance-oriented operations, such as updating system settings or reviewing flagged data.

The diagram clusters the main system behaviors into several key use cases. These include registering or logging in, selecting a sport and viewing the dashboard, searching for players or teams, customizing layouts, viewing detailed player pages, managing favorites, and performing administrative actions. Some relationships are expanded through UML's "include" and "extend" links—for example, viewing player details is included within the search flow, and predictive widgets are shown as a future extension of the dashboard.

The purpose of this model is not to show how these actions happen internally, but simply to map out the different goals the system must support. It acts as a bridge between the high-level requirements in the specification and the more detailed sequence diagrams that follow. Each use case shown here has a corresponding table describing the actors, triggers, success conditions, and exception handling, ensuring that the behavior of the system is fully defined before implementation begins.


Registered User
Admin
NashBoard System
Register/Login
Customize Dashboard Layout
View Sport Dashboard
Manage Favorites
«extends»
Use Predictive Widgets (Future)
Admin Manage Users & System
Search Teams/ Players/ Games
«include»
View Player Details
Guest User

Use Case Tables

Use Case 1 - Register/Login

Section	Description
Use Case Name	**Register / Login**
Primary Actor(s)	**Registered User, Admin**
Supporting Actors	**Authentication Provider, Email/Notification Service**
Goal	**Allow users to securely create an account or log into an existing one.**
Trigger	**User selects "Login" or "Create Account."**
Preconditions	**NashBoard online; Auth provider available; Email service available; Internet access**
Success Postconditions	**User authenticated; Redirected to dashboard; Session token created**
Minimal Guarantees	**No partial accounts; Invalid attempts do not modify profile**
Main Success Scenario	**1. User selects login/create accoun 2. System shows form 3. User enters credentials 4. UI validates input 5. Backend sends credentials 6. Provider verifies or creates account 7. Email verification (if new) 8. System confirms 9. Layout retrieved 10. UI loads dashboard 11. System logs event**

Alternate Flows	**A1: Incorrect credentials** **A2: Not verified** **A3: Email in use** **A4: Forgot password** **A5: Provider offline** **A6: User cancels**
Assumptions	**User owns email**
Related Requirements	**UFR-01, UFR-02, SNFR-04, UNFR-03**

Use Case 2 - Select Sport & View DashBoard

Section	**Description**
Use Case Name	**Select Sport & View Dashboard**
Primary Actor(s)	**Registered User, Guest User**
Supporting Actors	**Sports Data APIs, Backend**
Goal	**Display real-time dashboard for selected sport.**
Trigger	**User clicks a sport tile.**

Preconditions	**NashBoard running; API reachable; Layout exists**
Success Postconditions	**Dashboard rendered with live or cached data**
Minimal Guarantees	**Cached data displayed if needed**
Main Success Scenario	**1. User selects sport** **2. UI loads saved layout** **3. UI switches dashboard** **4. UI requests backend data** **5. Backend checks cache** **6. Cache freshness checked** **7. API call if needed** **8. API returns data** **9. Backend normalizes** **10. Backend sends results** **11. UI renders widgets** **12. User views dashboard**
Alternate Flows	**A1: API timeout** **A2: Guest user**

	A3: Corrupted response **A4: No internet** **A5: Change sport mid-load**
Assumptions	**Stable API data**
Related Requirements	**UFR-03, UFR-04, SNFR-02, SNFR-04**

Use Case 3 — Customize Dashboard Layout

Section	**Description**
Use Case Name	**Customize Dashboard Layout**
Primary Actor	**Registered User**
Supporting Actors	**Backend**
Goal	**Allow user to customize widget arrangement.**
Trigger	**User selects "Customize Layout Mode."**

Preconditions	**User authenticated; Layout exists; Widgets loaded**
Success Postconditions	**Updated layout saved**
Minimal Guarantees	**Old layout preserved if save fails**
Main Success Scenario	**1. User enters customize mode** **2. UI loads editable layout** **3. System loads widget library** **4. User drags widgets** **5. User resizes widgets** **6. UI previews changes** **7. User saves layout** **8. UI sends config** **9. Backend validates** **10. Backend saves** **11. Backend confirms** **12. UI renders** **13. System logs**

Alternate Flows	**A1: Invalid layout** **A2: Cancel** **A3: Save failure** **A4: Missing widget**
Assumptions	**Reliable storage**
Related Requirements	**UFR-05, UFR-22–24, SNFR-02, UNFR-03**

Use Case 4 — Search Teams / Players / Games

Section	Description
Use Case Name	**Search Teams / Players / Games**
Primary Actors	**Registered User, Guest User**
Supporting Actors	**Sports Data APIs**
Goal	**Search for teams, players, games, and stats.**
Trigger	**User types in search bar or opens Search tab.**
Preconditions	**Search bar visible; API search endpoints available**
Success Postconditions	**Results displayed**

Minimal Guarantees	**“No results found” message shown**
Main Success Scenario	**1. User opens search** **2. UI shows recent searches** **3. User types** **4. UI validates** **5. UI sends request** **6. Backend queries API** **7. API returns results** **8. Backend ranks** **9. Backend formats** **10. UI shows categorized results** **11. User selects** **12. System loads details**
Alternate Flows	**A1: No results** **A2: Rate limit** **A3: API error** **A4: Guest advanced click** **A5: Debounce input**
Assumptions	**User knows part of the name**
Related Requirements	**UFR-06, UFR-23, SNFR-02, UNFR-05**

Use Case 5 — View Player Details

Section	Description
Use Case Name	**View Player Details**
Primary Actors	**Registered User, Guest User**
Supporting Actors	**Sports Data APIs, Cache Layer**
Goal	**Display detailed player information.**

Trigger	**User clicks a player.**
Preconditions	**Player exists in API data**
Success Postconditions	**Player page displayed with up-to-date data**
Minimal Guarantees	**Cached data shown if needed**
Main Success Scenario	**1. User selects player 2. UI requests backend 3. Backend checks cache 4. Cache freshness checked 5. Return cached data OR 6. Call API 7. API returns data 8. Backend updates cache 9. Backend returns profile 10. UI renders page 11. User views details**
Alternate Flows	**A1: Player not found A2: API failure A3: Cache corrupt**
Assumptions	**Player exists in API**
Related Requirements	**UFR-07, SNFR-02**

Use Case 6 — Admin: Manage System Settings

Section	Description
Use Case Name	Admin: Manage System Settings
Primary Actor	Admin

Supporting Actors	Backend
Goal	Modify system settings and configurations.
Trigger	Admin opens Admin Panel.
Preconditions	Admin authenticated
Success Postconditions	Settings saved
Minimal Guarantees	Old settings preserved if save fails
Main Success Scenario	1. Admin opens panel 2. Backend loads settings 3. UI displays settings 4. Admin selects setting 5. Admin enters value 6. Backend validates 7. Backend saves 8. UI confirms 9. System logs
Alternate Flows	A1: Invalid setting A2: Save failure A3: Unauthorized access
Assumptions	Admin understands system controls
Related Requirements	Admin & security requirements

Use Case 7 — Predictive Widgets (Future)

Section	Description
Use Case Name	Predictive Widgets (Future)
Primary Actor	Registered User

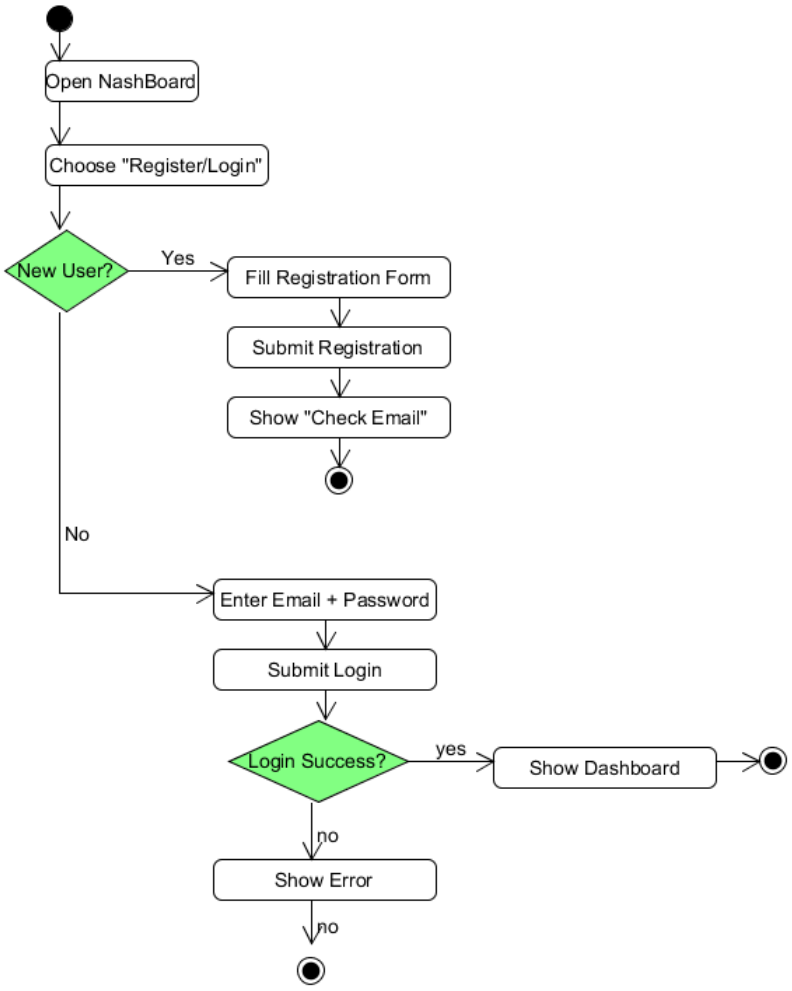
Supporting Actors	**Prediction Service (future)**
Goal	**Display placeholder predictive analytics.**
Trigger	**User clicks predictive widget.**
Preconditions	**Widget placeholders available**
Success Postconditions	**Placeholder message shown**
Minimal Guarantees	**User informed feature is not available**
Main Success Scenario	**1. User clicks widget** **2. UI sends request** **3. Service returns "coming soon"** **4. UI displays placeholder** **5. User informed**
Alternate Flows	**None**
Assumptions	**Predictive feature planned**
Related Requirements	**Future-phase requirements**

Process Models (Activity Diagrams)

Activity Diagram — Register/Login

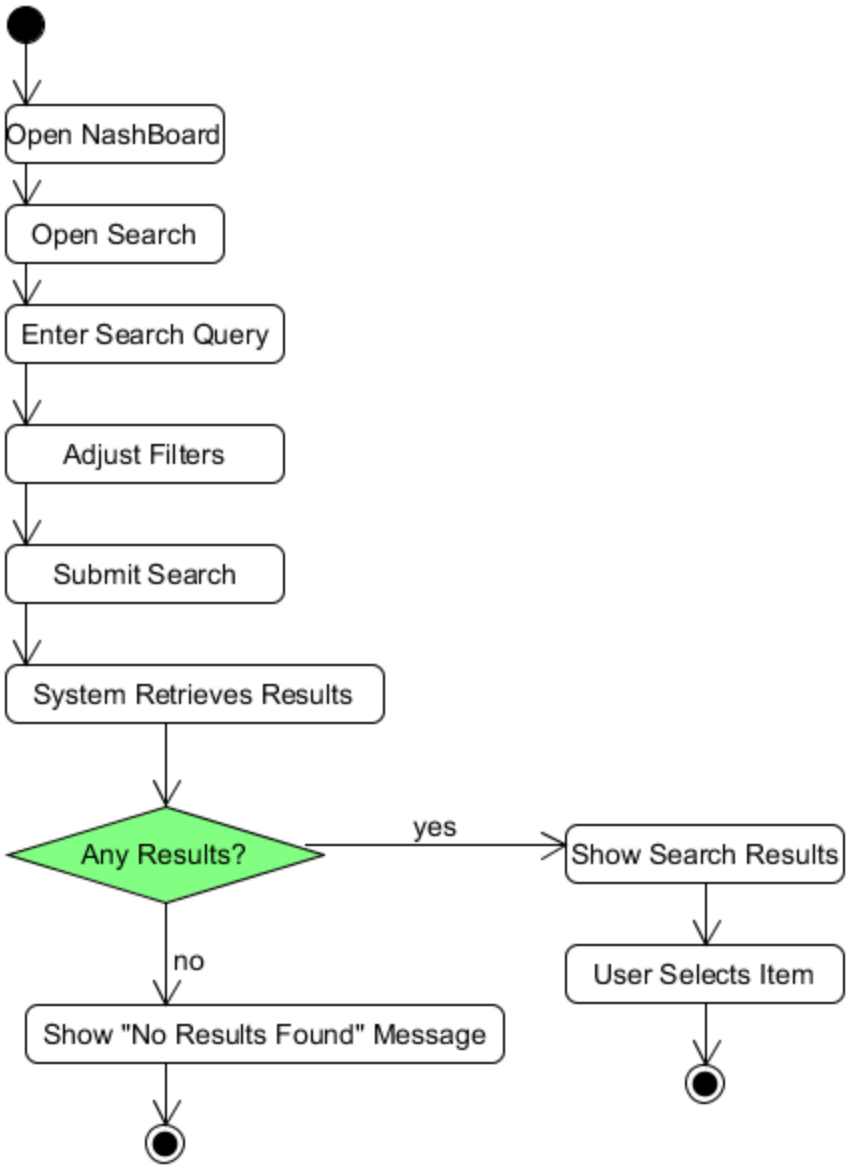
The Register/Login activity diagram shows the basic path a person follows when they first try to use NashBoard. It starts as soon as the app opens and the user picks the option to either log in or create an account. From there, the flow splits depending on whether the person is new or returning. New users go through the registration steps and email check, while returning users simply enter their credentials and wait for the system

to verify them. The diagram is really just laying out the two possible routes someone can take before they get inside the app, since everything else in NashBoard depends on the user being signed in.



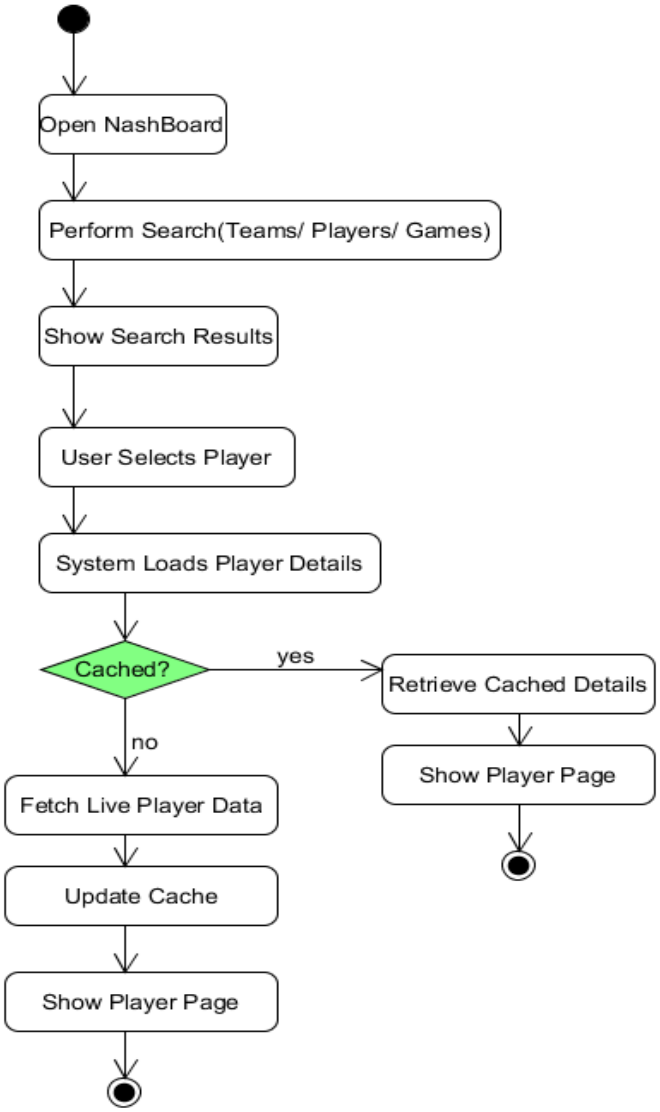
When I built this flow, I kept it simple because this is pretty much how most login systems work in real apps. Splitting it into a “new user” path and a “returning user” path made the whole thing easier to read instead of mixing everything together. It also points out the spots where things might fail, like typing the wrong password or not finishing the email check. Laying it out visually helped me see where the backend and the outside login service get involved, which wasn’t as obvious when everything was written out in the spec. This diagram also sets the stage for the later sequence diagrams, since they basically take this same process and show what’s happening behind the scenes with all the requests going back and forth.

Activity Diagram: Search Teams / Players / Games



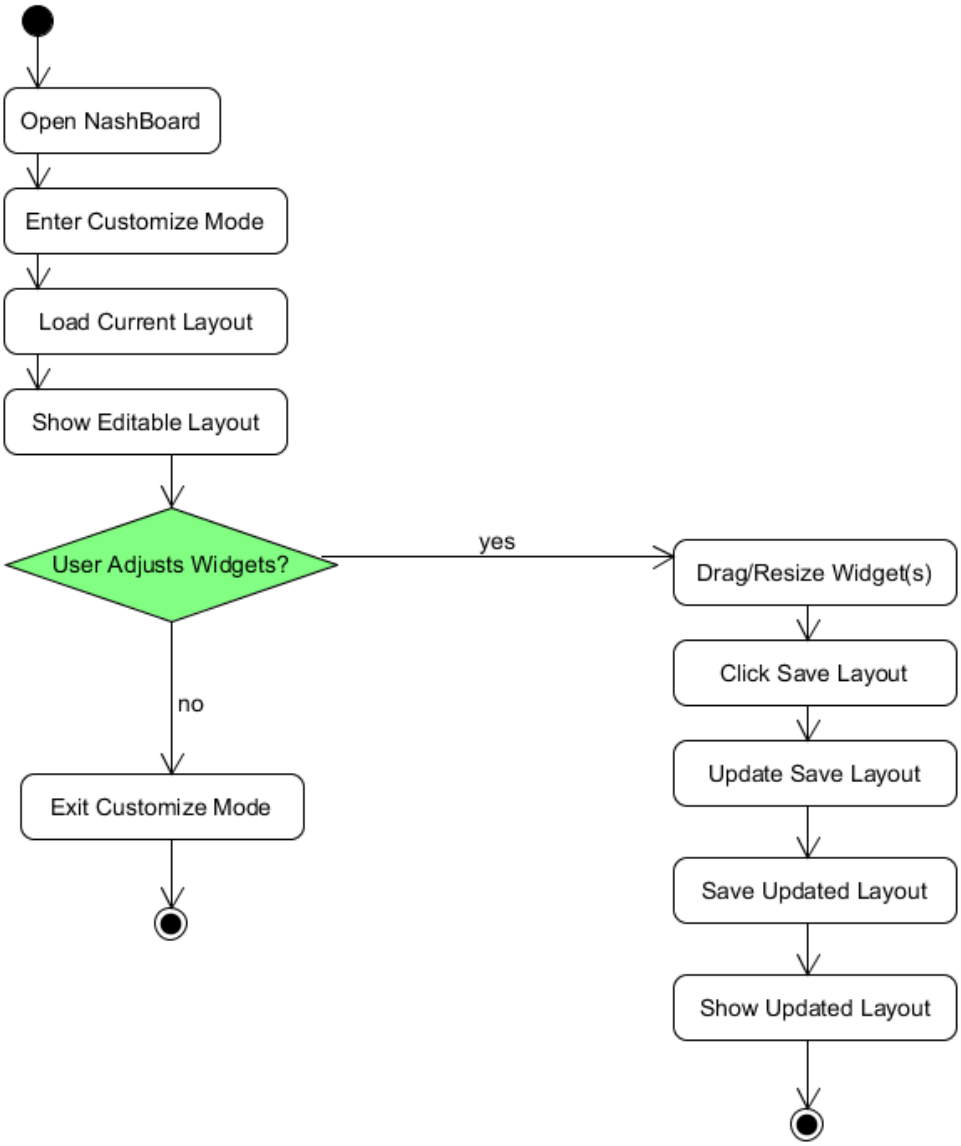
This flow matches what someone actually does when they're trying to look something up on NashBoard. Most people just open the search bar, type whatever they're looking for, maybe adjust a filter or two, and then see what comes back. Laying it out this way keeps the steps straightforward and avoids overthinking what is basically a common feature in every dashboard. The decision point about whether anything was found makes the diagram clearer, since searches don't always return something useful. It also highlights how the system reacts when there's no match or when the user picks something from the results. Overall, the diagram just walks through the normal path without adding unnecessary details, which makes it easier to connect with the sequence diagrams that follow later.

Activity Diagram: View Player Details



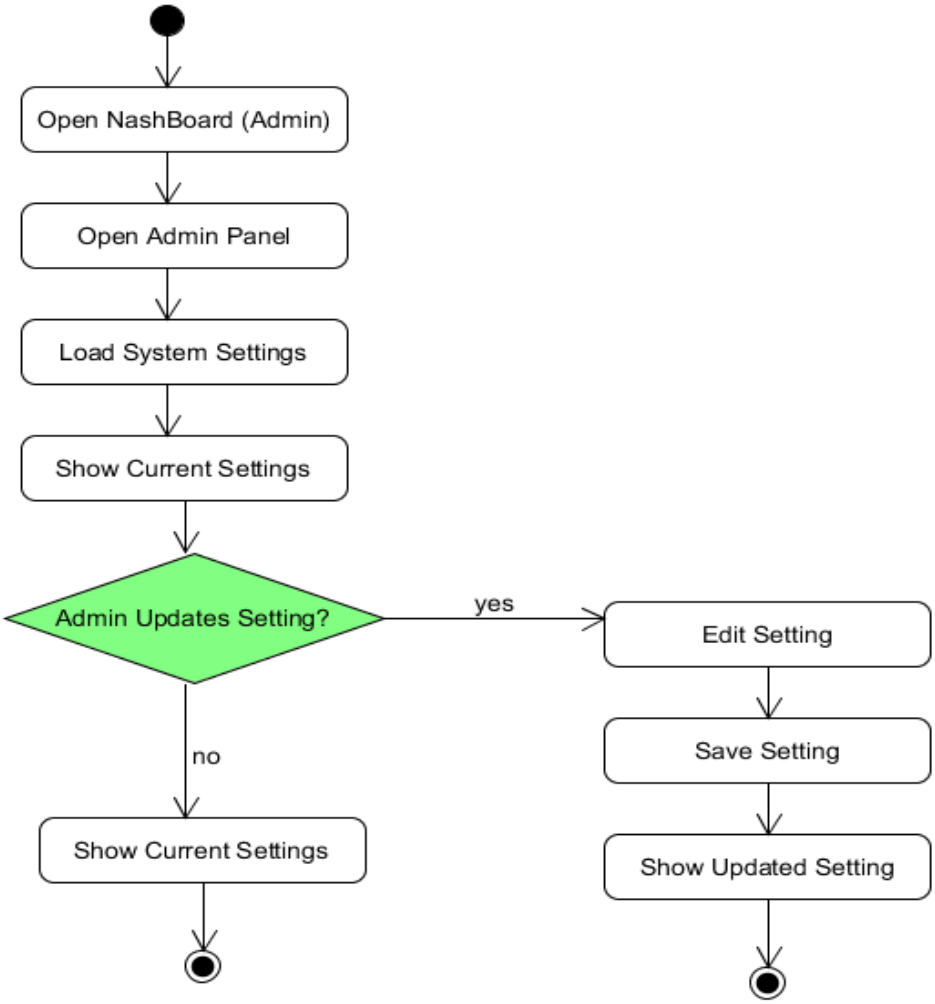
This diagram shows what happens after the user picks a player from the search results. Most of the time, if the system already has that player's information stored and it's recent, it just loads it right away. If not, it goes out to get fresh data. Breaking it into those two branches makes the whole thing easier to follow, because caching is one of those things that sounds complicated until you actually see it step-by-step. The flow ends with the player page being shown, which lines up with how people expect the app to behave. It also connects nicely with the backend diagrams, where you can see more detail about how the system decides whether to reuse cached data or fetch new stats.

Activity Diagram: Customize Dashboard Layout



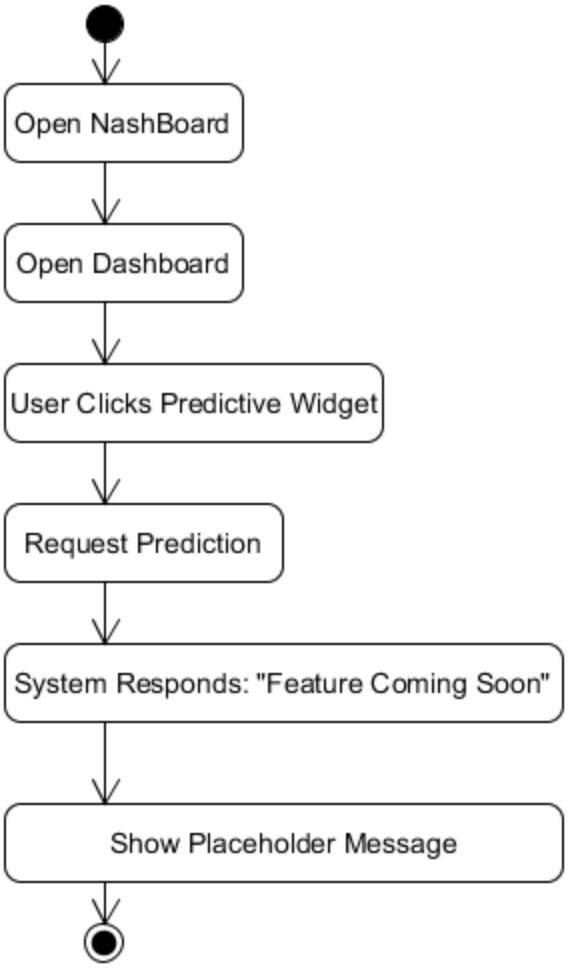
The customization flow was written in a way that reflects how people normally interact with drag-and-drop editors. You enter the mode, look at your layout, and then mess around with widgets until things look right. Splitting the diagram into the “made changes” and “did not make changes” branches keeps things from feeling cluttered. The save process is also broken out so you can see each little step instead of it all being lumped into one big action. This also makes the later sequence diagrams easier to understand, since they build directly on this same idea of loading a layout, updating it, and saving it back to the system.

Activity Diagram: Admin – Manage System



The admin flow is intentionally simple, mostly because most system-wide settings aren't things people change very often. The diagram shows the basic loop: an admin opens the panel, checks the current values, updates something if they need to, and then saves it. Keeping it short makes the diagram easier to read while still showing the branching between making a change and not making a change. It also lines up with how real admin dashboards work — you look around first, decide what needs fixing, and then move on. That same logic also ties into the sequence diagram for admin settings, which goes into more detail about the communication with the backend.

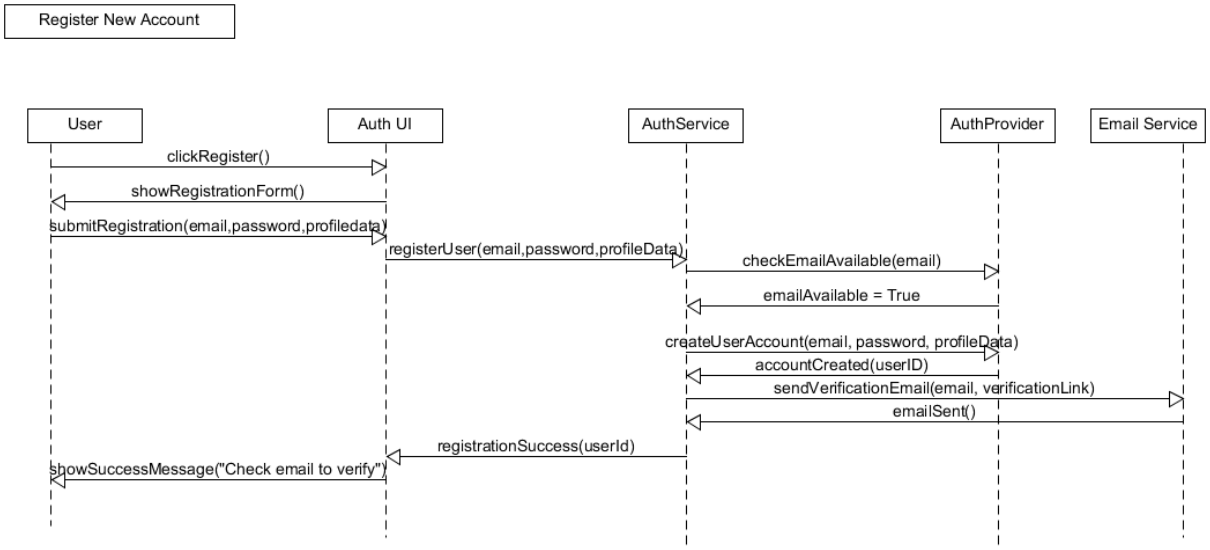
Activity Diagram: Predictive Widgets (Future Feature)

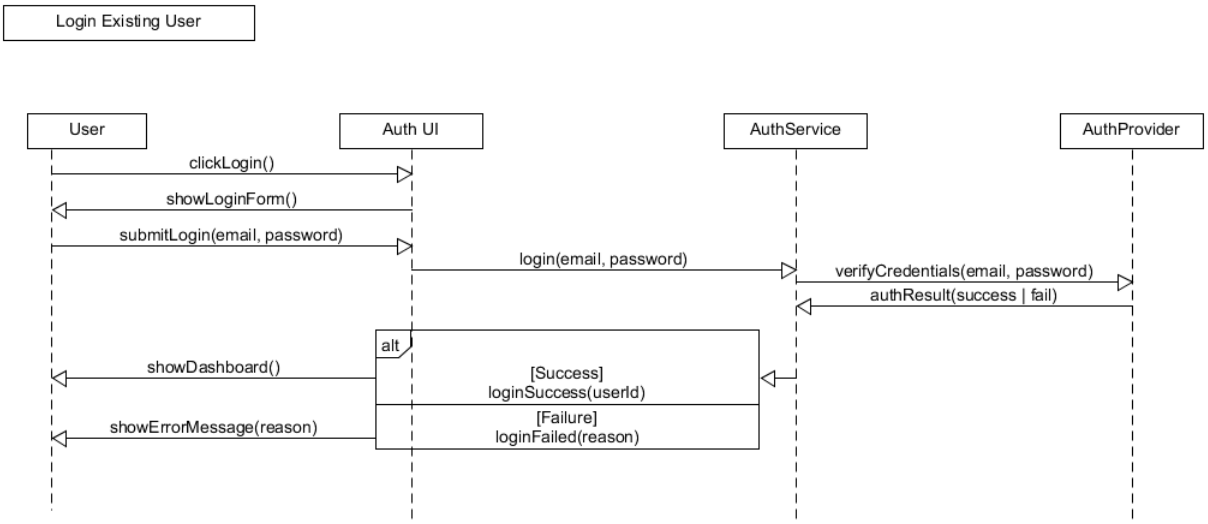


Since the predictive feature isn't actually implemented yet, this diagram just shows the placeholder behavior the app uses for now. It walks through the basic steps a user would take opening the dashboard, clicking a predictive widget, and then seeing the "coming soon" message. Even though it's simple, having it written out helps keep the design consistent with the rest of the document, and it leaves room for a more detailed flow later on once the predictive system is built. This also avoids confusion during the implementation phase because anyone reading the document can clearly see that this feature is planned but not active yet.

Sequence Diagrams

Sequence Diagrams — Register / Login

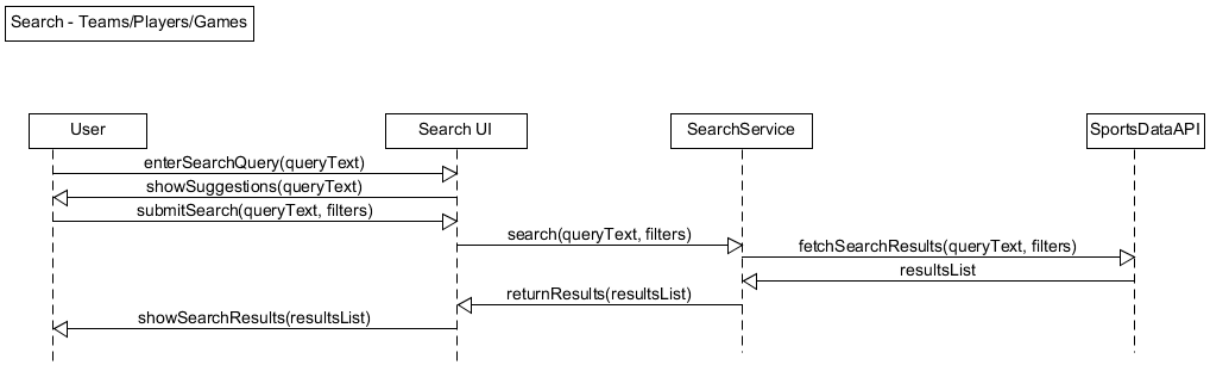




When you break down the login process into the individual steps, it becomes easier to see how many pieces are involved behind the scenes. From the user's point of view, they're just typing an email and password, but the system has to check formatting, send the credentials off to the authentication provider, wait for the response, and then either load the dashboard or show an error. Laying it out in this diagram helps show where issues might occur — like if the provider is slow or a user still needs to verify their email.

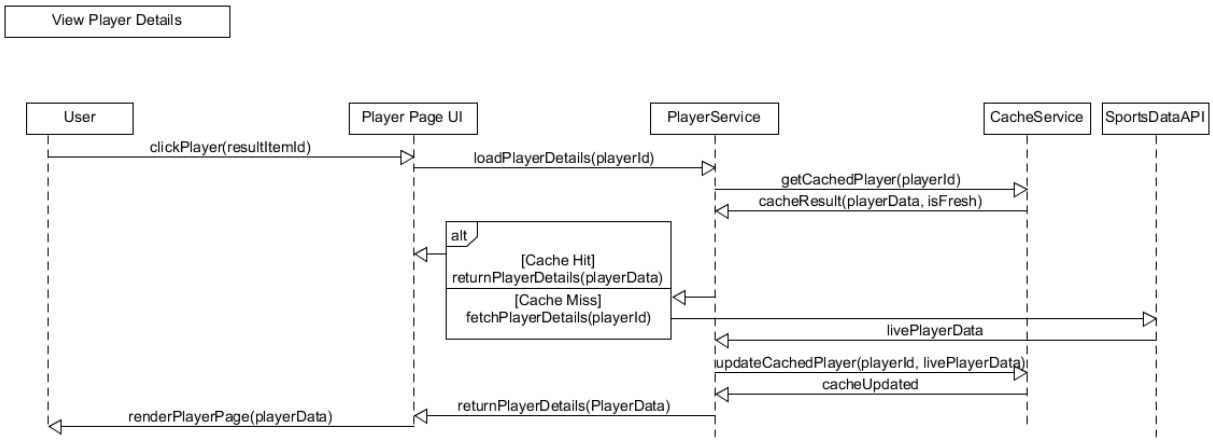
It also sets a clear picture of how the UI, backend, and external authentication service cooperate every time someone signs in.

Sequence Diagram — Search Teams / Players / Games



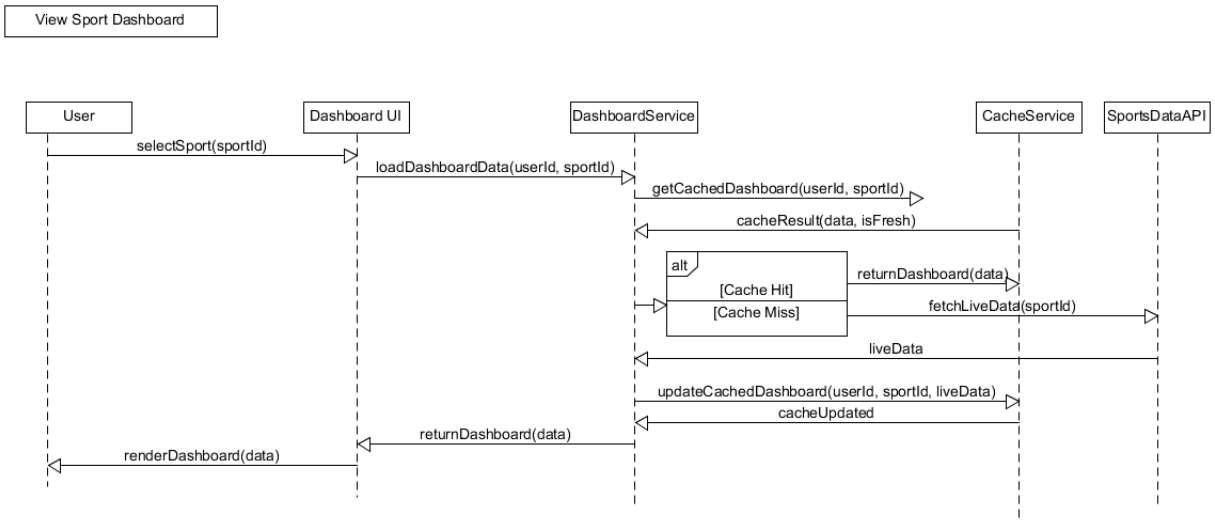
This sequence diagram follows the same path a user takes when they look something up in the search bar. The user enters a query, the UI sends it straight to the backend, and the backend reaches out to the Sports Data API to find matching results. Showing the steps this way makes it obvious that the UI doesn't actually know any sports information on its own — it's always waiting for the backend to gather and filter the data. The final handoff back to the UI reflects how search results seem instant to the user, even though a lot of small pieces are moving underneath.

Sequence Diagram — View Player Details



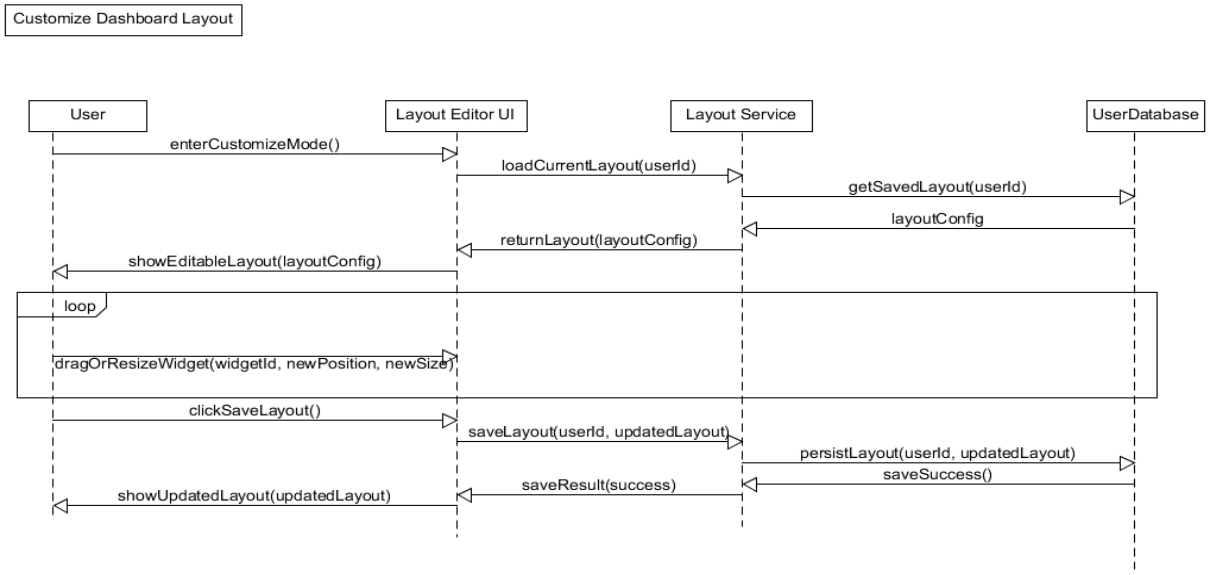
This sequence diagram follows the same path a user takes when they look something up in the search bar. The user enters a query, the UI sends it straight to the backend, and the backend reaches out to the Sports Data API to find matching results. Showing the steps this way makes it obvious that the UI doesn't actually know any sports information on its own — it's always waiting for the backend to gather and filter the data. The final handoff back to the UI reflects how search results seem instant to the user, even though a lot of small pieces are moving underneath.

Sequence Diagram — View Sport Dashboard



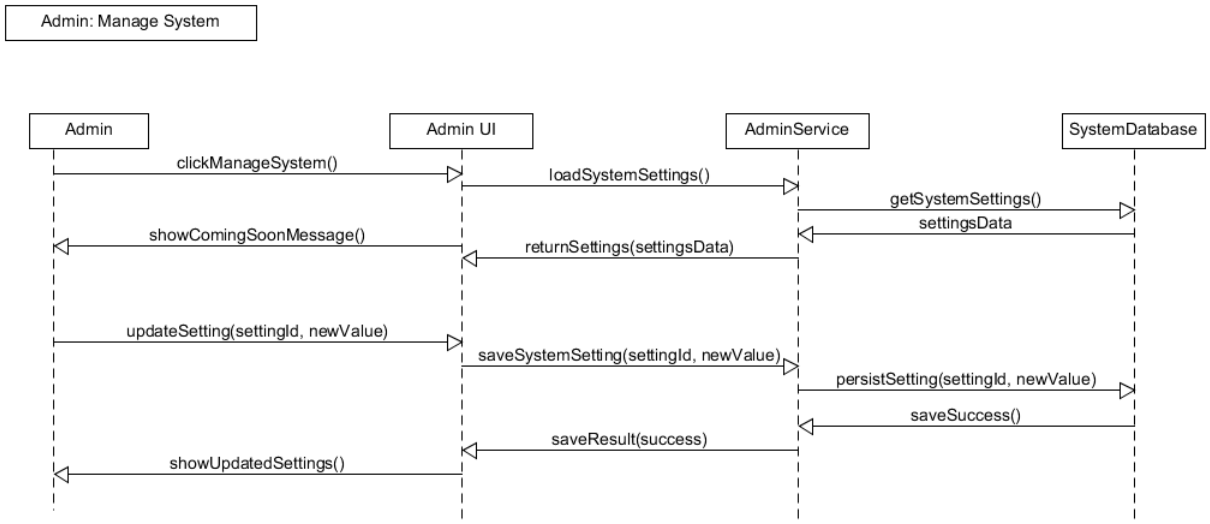
This diagram highlights the back-and-forth that happens once the user clicks on a player from the search results. The interesting part is the caching check. If the player's information is still considered "fresh," the system returns it right away. If not, it contacts the Sports Data API to pull updated stats and then refreshes the cache before sending anything back to the UI. Seeing it in sequence form helps explain why some player pages load faster than others — it depends on whether the system already has the data handy or needs to fetch it again.

Sequence Diagram — Customize Dashboard Layout



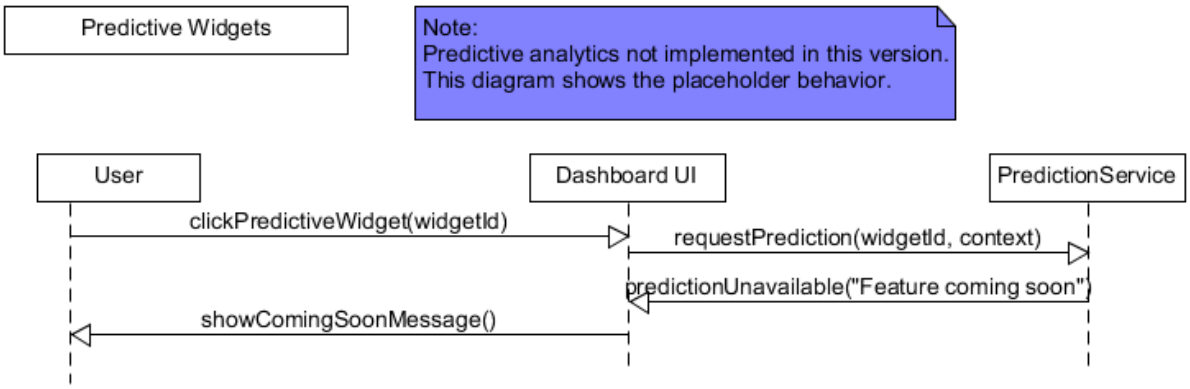
This diagram walks through the process that happens when a user saves a new dashboard layout. The UI collects all the layout changes and sends them to the backend, which checks whether the structure is valid. From there, it saves the layout in the user's profile and confirms that everything went through successfully. Showing the save workflow this way emphasizes that even though customizing the dashboard feels like a visual action, most of the real work happens in the backend when the changes get validated and stored.

Sequence Diagram — Admin: Manage System



The admin flow shows the cycle that happens when an administrator loads system settings, makes an edit, and saves the update. The diagram lays out the communication line clearly: the UI loads the existing values, the backend pulls them from storage, and then any edit gets pushed back down to the database. Seeing it in sequence form helps clarify the difference between simply viewing settings (a read operation) and actually updating them (a write operation). It also explains why admins are separated from normal users at the system level.

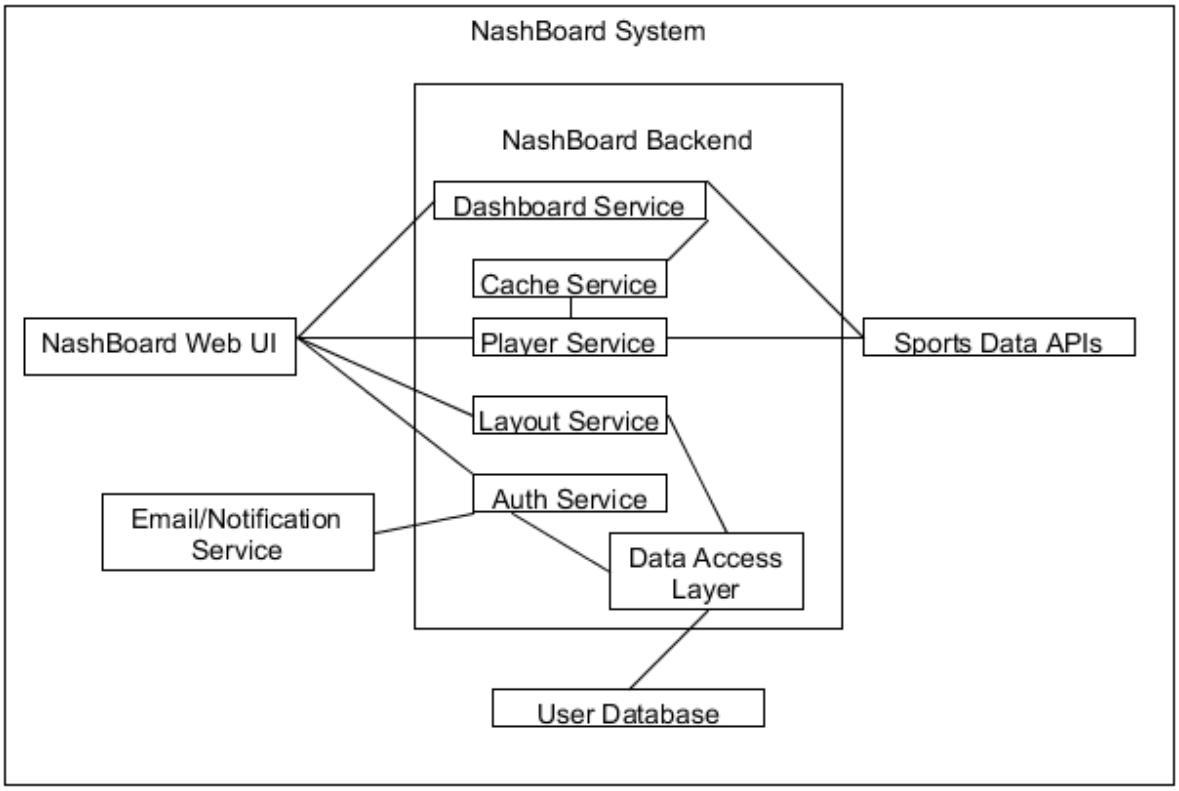
Sequence Diagram — Predictive Widgets (Future)



Since predictive widgets aren't functional yet, this diagram describes the placeholder version of the feature. The UI sends a request to the backend, but instead of performing an actual calculation, the backend returns the "Feature Coming Soon" response immediately. Even though it's simple, including this diagram makes it clear to anyone reading the document that the feature is intentionally not implemented yet, rather than missing or forgotten. It also reserves a logical spot in the system for when the predictive algorithms eventually get added.

Structural Model (Classes/Components)

Component Diagram



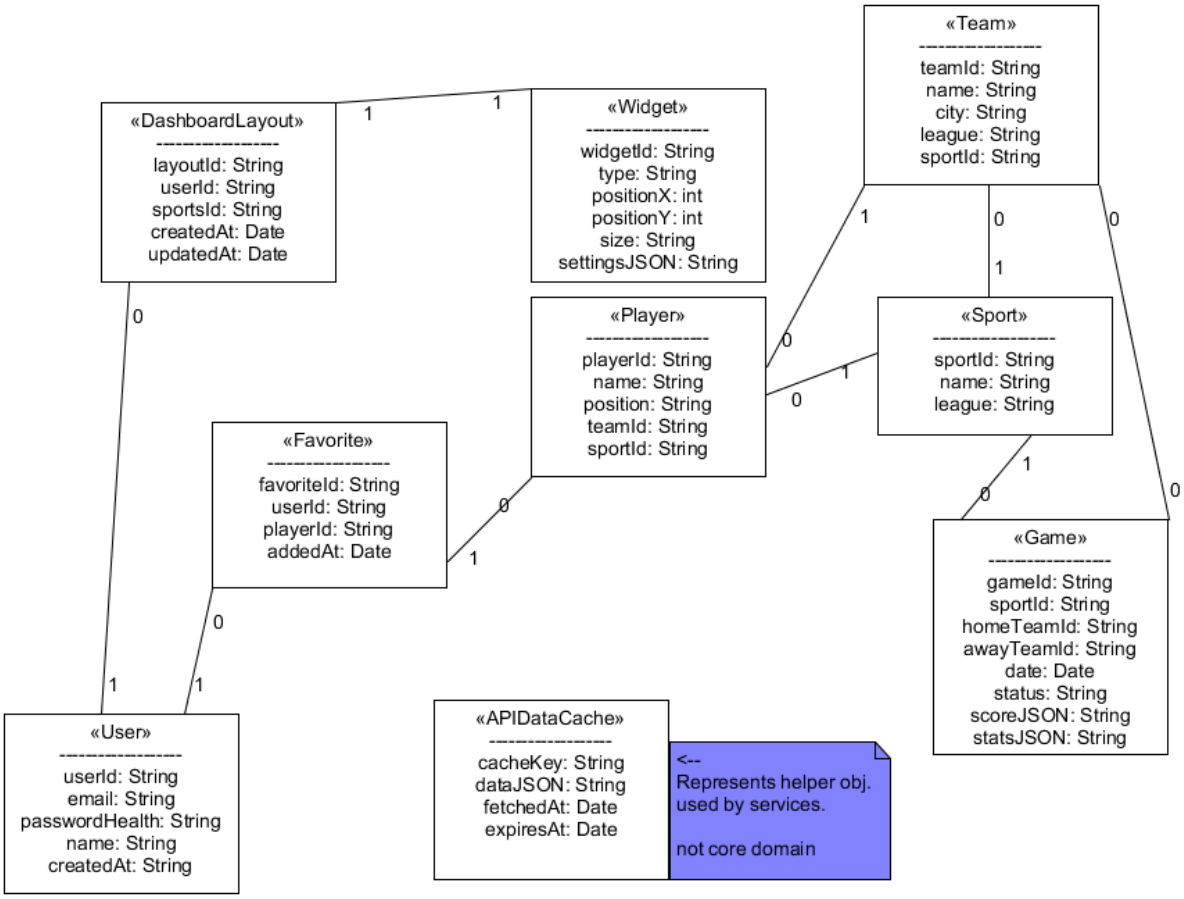
The component diagram is meant to show the big picture of how NashBoard is organized behind the scenes. Instead of focusing on individual screens or tiny details, this diagram breaks the entire system into the major parts that interact with each other during normal use. At the top of the structure is the web interface, which is the only part the user actually sees. This layer handles things like clicking buttons, choosing sports, searching for players, and arranging widgets on the dashboard.

Behind the UI sits the backend, which is split into separate services that each focus on one area of the app. For example, the Auth Service handles logins and account creation, the Dashboard Service retrieves and formats the sports data shown on the main screen, and the Layout Service takes care of saving personalized dashboard setups. The Player Service is focused on player-specific data, such as stats, injuries, and recent games. Finally, there is a Cache Service that stores data temporarily to avoid constantly calling the Sports Data API.

At the bottom is the data storage layer, which includes the user database and any long-term saved information such as dashboard layouts and favorites. Putting the components into this layered structure helps show how the system stays organized and secure. It also reflects how the actual implementation works today, since the real NashBoard project already uses separate API routes, Prisma for data access, and

React components on the front end. The diagram essentially acts as a map of how all those pieces fit together as the system grows.

Domain Class Diagram – main entities

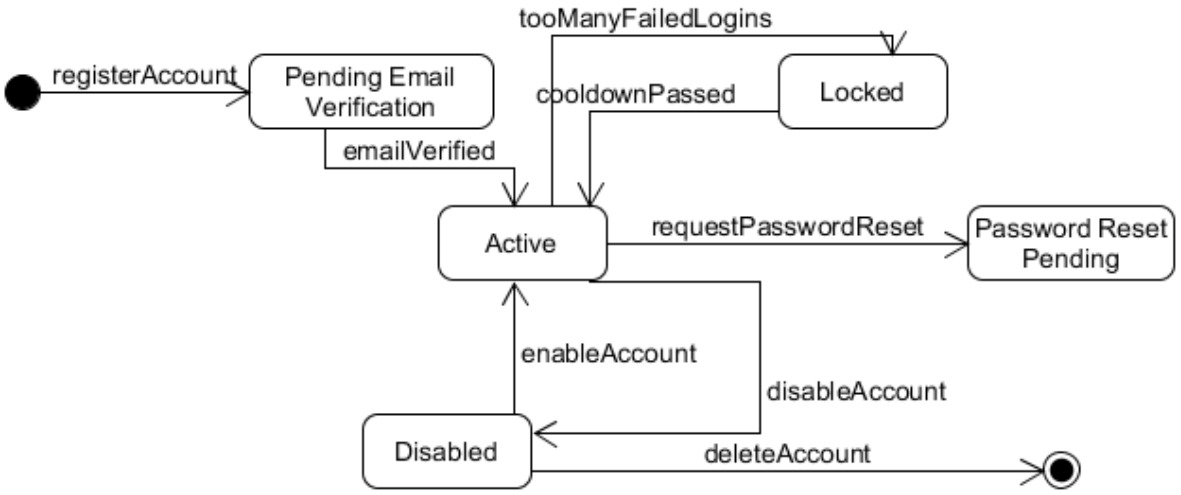


The domain class diagram for NashBoard shows the main pieces of data the system The domain class diagram focuses on the core information NashBoard needs to work. It doesn't deal with UI screens or backend functions — instead, it tries to define the actual data objects that make up the system. In the center is the User class, which represents anyone with an account. A user can have a saved dashboard layout and a list of favorite players they want to follow more closely. Each dashboard layout is made up of widgets. A widget has a type (such as "scores," "injury report," or "standings"), a position on the screen, and its own settings. This lines up with the customization features described earlier in the document, where users rearrange these widgets to match what they care about. Since each user has their own preferences, layouts are tied directly to the user profile.

The diagram also includes the sports world: players, teams, sports, and games. A player belongs to a specific team and sport, and games connect teams, scores, and dates. These classes support the search system and the detailed player pages. The APIDataCache class is included to represent cached responses from external sports APIs — this is how NashBoard avoids making repeated calls for the same data. The important thing to understand is that the diagram is slightly ahead of the current database schema on purpose. Only part of this structure exists in Prisma right now, but the full diagram reflects where the project is going. It gives a clear target for how future models will fit into the system as more features are added.

State Model

State Diagram



The state diagram is basically a simple way to show what can happen to a user account over its lifetime in NashBoard. Instead of walking through screens or UI steps, it focuses on the “status” the system assigns to an account and how that status changes depending on different events. When someone signs up, the account doesn’t jump straight into full access. It starts off in a waiting state until the person clicks the verification link that gets emailed to them. Nothing else really moves forward until that part is done. Once the email is confirmed, the account becomes active, which is the normal state for day-to-day use. From there, things can branch off depending on what the user does (or doesn’t do). If they start a password-reset request, the account shifts into a temporary state until the reset is finished. If there are too many wrong password attempts in a row, the system may lock the account to prevent anything suspicious. And

the admin has the ability to disable an account entirely, which is different from locking it disabled means the person can't get in at all until an admin turns it back on.

All of these different paths tie back to the security and account-handling rules from the specification. Putting them in one diagram keeps everything in one place instead of spreading those rules across a bunch of different sections. It basically shows the "life story" of an account, from the moment someone signs up to whatever happens later, including rare cases like lockouts or admin actions.

State Description Table

State	Description	Typical Entry Events	Typical Exit Events	Related Use Cases / Requirements
Pending Email Verification	An account has been created, however, the user has not confirmed their email address yet. Login is not allowed fully	registerAccount	emailVerified, verification Expired	UC1 Security requirements
Active	Normal state where the user can log in and use NashBoard according to their role and permissions	enableAccount	requestPassword Reset, tooManyFailed Logins, disableAccount, deleteAccount	UC1 UC2
Password Reset Pending	The user has requested a password reset and must finish the reset process before the account returns to normal use	requestPassword Reset	resetCompleted, deleteAccount	UC1 alternative flow "Forgot Password"
Locked	The account is temporarily locked because there were too many failed login attempts in a short period of time.	tooMany FailedLogins	unlockByAdmin, cooldownPassed , deleteAccount	Security requirements

Disabled	The account has been disabled by an administrator or automated rule. The user cannot log in until an admin explicitly re-enabled it or it is closed.	disableAccount	enableAccount, deleteAccount	Admin management features, Policy and compliance requirements

This table basically serves as the “plain English version” of the diagram above it. Instead of tracing arrows or worrying about shapes, you can read across the row and understand what causes the transition and where the account ends up. It connects back to the security requirements and login-related use cases from the spec, so all the pieces stay consistent.

Architectural Pattern

Selected Pattern (Layered 3-tier web architecture)

For NashBoard, the whole system basically falls into three parts, which honestly is how most web apps end up being built anyway. The part people see is just the website itself, the screens where you click around, pick your sport, customize your layout, all of that. None of the real “work” happens there. Behind that, there’s the backend, which is the part that deals with all the messy stuff the user never has to think about. That includes logging people in, grabbing stats from the sports API, checking the cache, saving dashboards, and doing all the rules the system needs to follow. Then there’s the bottom layer, which is where we store everything permanently — user accounts, saved dashboards, cached data so the app doesn’t have to keep re-calling the API, and so on. The layers basically talk to each other in order, so the UI isn’t directly hitting the database or an API. It keeps things simple and pretty organized.

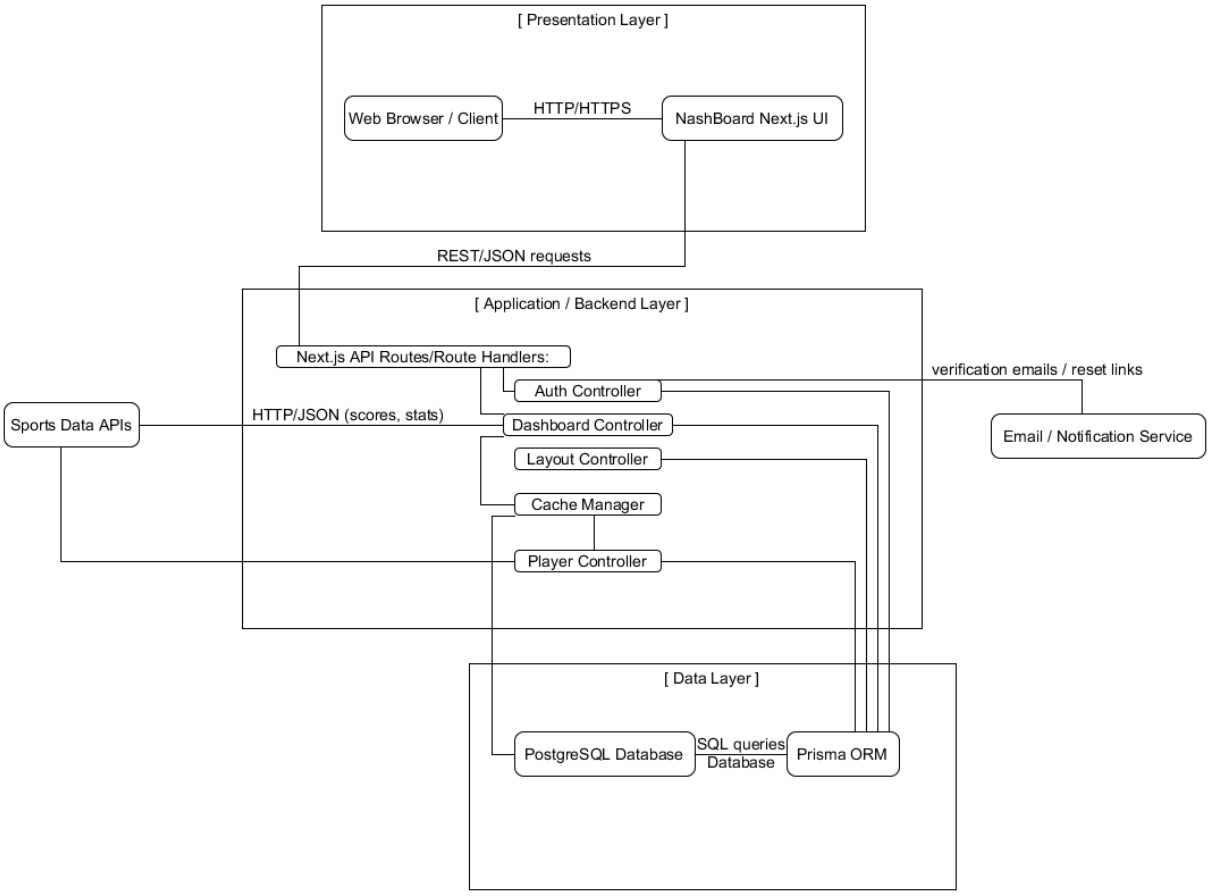
Pattern Table

Item	Content
Pattern name	Three-tier layered web architecture (UI → backend services → data layer)

Category	Web application architecture pattern
Basic structure	The system is split into three parts. The UI layer is the browser interface that users interact with. The backend layer is where all the real logic lives — login checks, fetching sports data, saving layouts, handling favorites, and so on. The data layer stores long-term information (like user accounts and dashboard layouts) and also keeps cached API results so the system doesn't constantly re-fetch the same stats. The backend also talks to external services like sports data APIs and the email/notification service.
Typical usage	This pattern is used by most interactive sites and dashboards — anywhere the visual interface is separate from the logic that processes requests or stores data. It matches the class examples where a web interface sits on top of a server that communicates with a database.
Advantages	Each layer can be changed or improved without rewriting the whole system. For example, the UI can be redesigned while the backend stays the same. It also makes security easier, since only the backend touches the database or external APIs. The structure also scales well: if the system gets more traffic, the backend layer can be expanded on its own.
Disadvantages	Sometimes this structure can feel a bit heavy because requests pass through multiple layers. If the app is not organized well, it can also make debugging harder since issues might come from any layer.
Why NashBoard fits	NashBoard already works like a layered system. The browser handles all the visual interactions, the backend gathers and cleans data from sports APIs, and the database stores user info and saved

	dashboards. This setup fits naturally with the course material and also with how you’

NashBoard Architecture Diagram



Justification

NashBoard works like a lot of dashboards and sports-tracking sites, so the layered setup fits naturally. The frontend only deals with user actions like picking a sport or arranging widgets, and the backend handles all the heavier tasks in the background. It talks to the sports API, checks login info, saves dashboard layouts, and so on. Splitting things this way makes it easier to manage the project, because each part can be worked on without breaking the others. This setup also helps with performance, since caching sports data in the data layer prevents constant API calls. It supports security

too, because only the backend touches the database or any external services. Overall, the layered structure simply matches the way NashBoard already works and lines up well with how we were taught to build these systems in class. The code that exists today already follows this layered structure, even though the feature set is still small. The Presentation Layer is implemented with Next.js and React in the /app directory and the components under /components. The main page (app/page.tsx) composes pieces like Sidebar, TopBar, TonightSlate, PlayerWatchlist, and BuildSteps to render the current NashBoard screen. The Application / Backend Layer shows up in the /app/api/watchlist/route.ts file, where a small REST-style endpoint handles watchlist operations using Prisma. On the Data Layer side, Prisma is configured in lib/prisma.ts and the schema is defined in prisma/schema.prisma, which connects to a PostgreSQL database. Even though live Sports Data APIs are not wired in yet and TonightSlate uses mock game data, those APIs already have a place in the architecture diagram as external services that will feed into the backend in a future iteration.

Traceability Note

When I put these diagrams together, I tried to make sure they actually connect back to the requirements from the NashBoard spec, instead of being random drawings I made just to fill the section. Each one lines up with something the system is supposed to do. For example, the activities and use cases are basically the same steps a normal user would take in the app — picking a sport, looking at their dashboard, searching players, messing with their layout and all that. The sequence diagrams and the state model dig more into what's happening in the background when those things occur. It's basically the "behind the curtain" version of the system, like what happens during login or when data has to be loaded or refreshed. Then there are the class and component diagrams, which show where the different kinds of information live and how the parts depend on each other. That all lines up with the data-related parts of the spec. The architecture diagram is more of a big-picture view, showing how the UI, the backend, the database, and the external sports APIs fit together. So everything in this design actually comes from the requirements; it's just presented in different ways depending on what angle I needed to show.