DesignDojo

Team 2

Abstract

DesignDojo is an interactive learning platform that transforms how developers master design patterns—the essential building blocks of scalable, maintainable software architecture.

Unlike traditional learning resources that only teach theory, DesignDojo provides a complete hands-on environment where developers can learn, practice, and be evaluated on their implementation of design patterns. Our platform combines comprehensive learning materials with an innovative challenge system powered by advanced AI.

1 Requirements and Subsystems

1.1 Functional Requirements

- **User Authenitcation and Authorization**
  - Users are able to register and log in using their email and password.
  - System utilizes JSON Web Tokens (JWT) for stateless authentication.
  - Users have secure access to their data and actions (e.g. profile and submissions).

  Architectural Significance: Authentication is a major concern affecting all protected resources as it ensures that unauthorised users cannot access the system or its data. The JWT-based approach informs the stateless design of the system, which impacts scalability and server-side complexity.

- **Learning Module**
  - Users are able to access learning materials for each design pattern.
  - Each learning material includes theory of the pattern, a sample code, and few MCQs to test understanding.
  - Users can track their progress for each pattern as well as for the learning module as a whole.

  Architectural Significance: Consistency across lessons is required and content should be managed efficiently.

- **Challenges Module**
  - Users can solve design patterns related coding questions.
  - Challenges are dynamically generated using LLMs.
  - Users can write and submit their code submission in a dedicated on-site codespace.
  - Submissions are evaluated by LLMs, and the feedback (score and comments) is presented to the user.

  Architectural Significance: This is core functionality requiring integration of multiple complex subsystems (code editor, execution environment, LLM-based evaluation). The LLM integration informs API design and performance considerations.

- **Rating System**

- Users earn rating based on the challenges they solved.
- Users' past 5 ratings are displayed through a line graph allowing for visual self-comparison.

  Architectural Significance: Requires a well-designed data model to track user performance over time and maintain rating integrity, while ensuring that it doesn't break under heavy use.

- **Profile and Dashboard**
  - Users can update their username and password through the profile page.
  - The dashboard provides links to the learning and challenges pages.
  - Dashboard displays the user's most recent lessons and submissions.

  Architectural Significance: Dashboard aggregates data from multiple system components, informing API design and data access patterns, while the profile page allows user to view and edit their identity and authorisation details.

1.2 Non-Functional Requirements

- **Security**
  - User passwords are hashed securely before storing.
  - JWTs are signed and verified preventing unauthorized access.
  - Sensitive data (e.g. tokens, passwords) are transmitted over HTTPS.

  Architectural Significance: Security requirements necessitate specific middleware, authentication flows, and data handling practices throughout the application.

- **Usability**
  - The application provides an intuitive and user-friendly interface across devices.
  - Users can seamlessly navigate between different pages with links available on the respective pages as well as a consistent navbar.

  Architectural Significance: Impacts frontend architecture, state management approach, and API design to support responsive UI updates.

- **Performance**
  - API responses have low latency to ensure a smooth user experience.
  - LLM evaluations have been optimized to minimize delays in providing feedback.

  Architectural Significance: Performance requirements impact technology choices, caching strategies, and resource allocation, especially for LLM-based evaluations which can be computationally intensive.

- **Reliability**
  - Data integrity is maintained during user actions like profile updates.

  Architectural Significance: Requires fault-tolerant system and affects error-handling strategies as data needs to be carefully maintained and updated.

- **Extensibility**
  - System allows addition of new challenges or features without significant architectural changes.

  Architectural Significance: Demands modular, loosely-coupled and easy-to-modify component design as well as well-defined interfaces between subsytems.

1.3 Subsytem Overview

- **Authentication and User Management Subsystem**
  This system is responsible for managing user identity, authentication and profile information. Its functionalities include user registration and account creation, login/logout with JWT authentication, user profile management including password reset function, and session management and authorization controls.

- **Learning Content Management Subsystem**
  This subsystem serves and organises educational content on design patterns. Its functionalities include providing structured content for design patterns theory including an example scenario with code, multiple-choice questions with solutions, progress tracking for both completed and ongoing lessons, and content categorisation and intuitive navigation.

- **Challenge Generation and Management**
  It creates, stores and serves coding challenges related to design patterns. Its functionalities include LLM integration for dynamic challenge generation, challenge difficulty classification, on-site code editor, LLM-powered code evaluation pipeline, feedback generation on submitted solutions, and score calculation based on submission accuracy.

- **Rating and Analytics Subsytem**
  It tracks user performance and calculates skill rating. Its functionalities include rating calculation algorithms, performance history storage, grahical visualisation of ratings, and progress analytics and statistics.

2 Architecture Framework

2.1 Stakeholder Identification

Stakeholder	**Role/Description**	**Key Concerns**
End Users	Students and professionals learning design patterns.	• Ease of use • Intuitive UI/UX • Quality of learning material • Progress tracking
Instructors	Educators creating and managing content for design patterns.	• Content creation tools • Analytics on student performance • Customization of learning paths
Developers	Engineers building the platform (frontend and backend).	• Clear requirements • Scalable architecture • Maintainable codebase

System Administrators	Manage the deployment and maintenance of the platform.	• System reliability • Security • Monitoring and logging

2.2 Viewpoints and Views

3 View Models for DesignDojo

1. Logical View

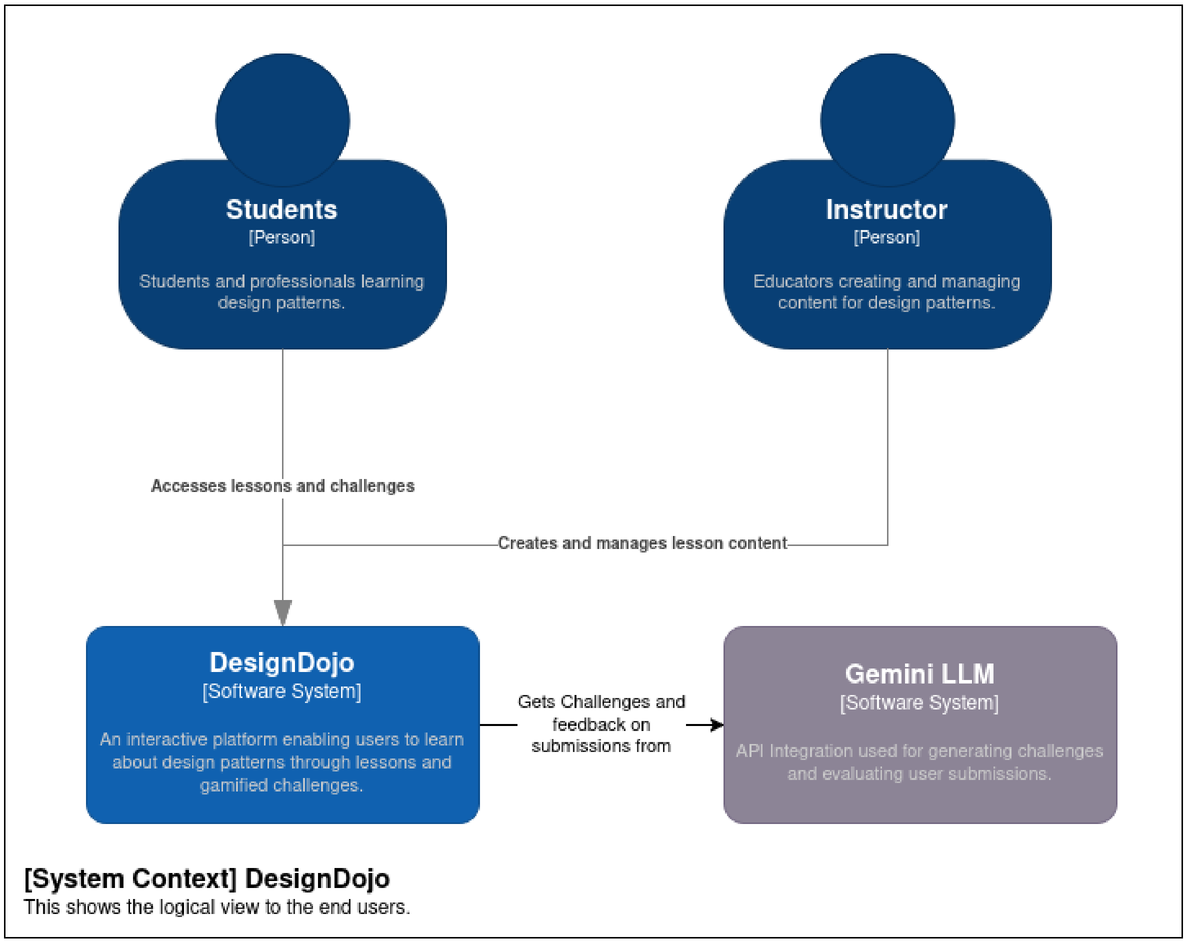
Students
[Person]
Students and professionals learning design patterns.
Instructor
[Person]
Educators creating and managing content for design patterns.
Accesses lessons and challenges
Creates and manages lesson content
DesignDojo
[Software System]
An interactive platform enabling users to learn about design patterns through lessons and gamified challenges.
Gets Challenges and feedback on submissions from
Gemini LLM
[Software System]
API Integration used for generating challenges and evaluating user submissions.
[System Context] DesignDojo
This shows the logical view to the end users.

Figure 1: DesignDojo - Logical View

Purpose: Represents the system's functionality as perceived by end-users. It focuses on the object model and the relationships between components.

Key Components:

- **Frontend:** React-based user interface for students and instructors.
  - Components: Authentication, Dashboard, Learning Modules, Challenges.
- **Backend:** Python-based API (Flask) for handling business logic.

– Modules: User Management, Submissions, Learning Progress, Competitive Challenges.

- **Database:** Relational database for storing user data, submissions, and learning content.

2. Development View

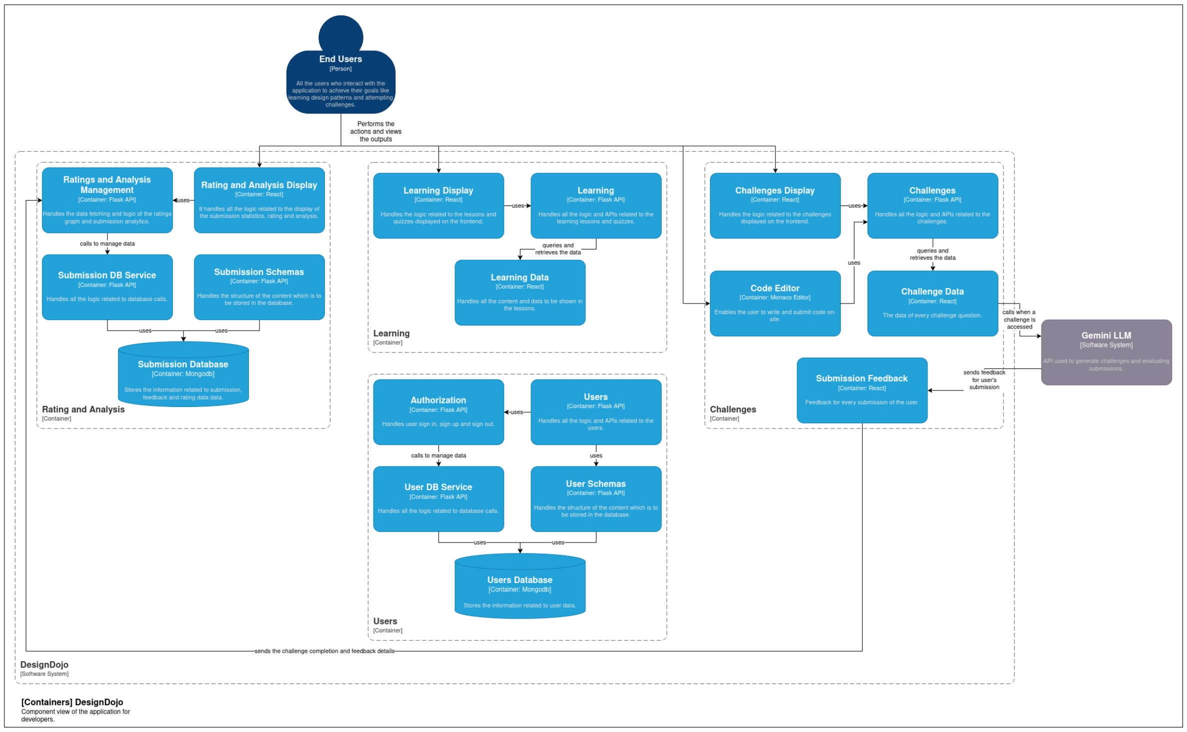
End Users
[Person]
All the users who interact with the application to achieve their goals like learning design patterns and attempting challenges.
Performs the actions and views the outputs
Ratings and Analysis Management
[Container: Flask API]
Handles the data fetching and logic of the ratings graph and submission analytics.
uses
Rating and Analysis Display
[Container: React]
It handles all the logic related to the display of the submission statistics, rating and analysis.
calls to manage data
Submission DB Service
[Container: Flask API]
Handles all the logic related to database calls.
Submission Schemas
[Container: Flask API]
Handles the structure of the content which is to be stored in the database.
uses
uses
Submission Database
[Container: Mongodb]
Stores the information related to submission, feedback and rating data data.
Rating and Analysis
[Container]
Learning Display
[Container: React]
Handles the logic related to the lessons and quizzes displayed on the frontend.
uses
Learning
[Container: Flask API]
Handles all the logic and APIs related to the learning lessons and quizzes.
queries and retrieves the data
Learning Data
[Container: React]
Handles all the content and data to be shown in the lessons.
Learning
[Container]
Authorization
[Container: Flask API]
Handles user sign in, sign up and sign out.
uses
Users
[Container: Flask API]
Handles all the logic and APIs related to the users.
calls to manage data
uses
User DB Service
[Container: Flask API]
Handles all the logic related to database calls.
User Schemas
[Container: Flask API]
Handles the structure of the content which is to be stored in the database.
uses
uses
Users Database
[Container: Mongodb]
Stores the information related to user data.
Users
[Container]
Challenges Display
[Container: React]
Handles the logic related to the challenges displayed on the frontend.
uses
Challenges
[Container: Flask API]
Handles all the logic and APIs related to the challenges.
uses
queries and retrieves the data
Code Editor
[Container: Monaco Editor]
Enables the user to write and submit code on-site.
Challenge Data
[Container: React]
The data of every challenge question.
calls when a challenge is accessed
Gemini LLM
[Software System]
API used to generate challenges and evaluating submissions.
sends feedback for user's submission
Submission Feedback
[Container: React]
Feedback for every submission of the user.
Challenges
[Container]
sends the challenge completion and feedback details
DesignDojo
[Software System]
[Containers] DesignDojo
Component view of the application for developers.

Figure 2: DesignDojo - Development View

Purpose: Focuses on the static organization of the software in the development environment. It describes the structure of the codebase.

Key Artifacts:

- **Frontend:**
  – Organized into `src/components/` for modular development.
  – Uses TypeScript for type safety and maintainability.
- **Backend:**
  – Modular structure with `routes/`, `models/`, and `middleware/`.
  – Follows RESTful API design principles.
- **Configuration:**
  – Separate configuration files for database and environment variables.

3. Physical View

Purpose: Focuses on the deployment and physical infrastructure of the system to help the system admins.

Deployment Architecture:

- **Frontend:** can be deployed on a CDN (e.g., AWS S3 or Netlify) for fast delivery.
- **Backend:** can be hosted on a cloud platform (e.g., AWS EC2 or Heroku).
- **Database:** Managed relational database service (e.g., AWS RDS or PostgreSQL). Currently using MongoDB.

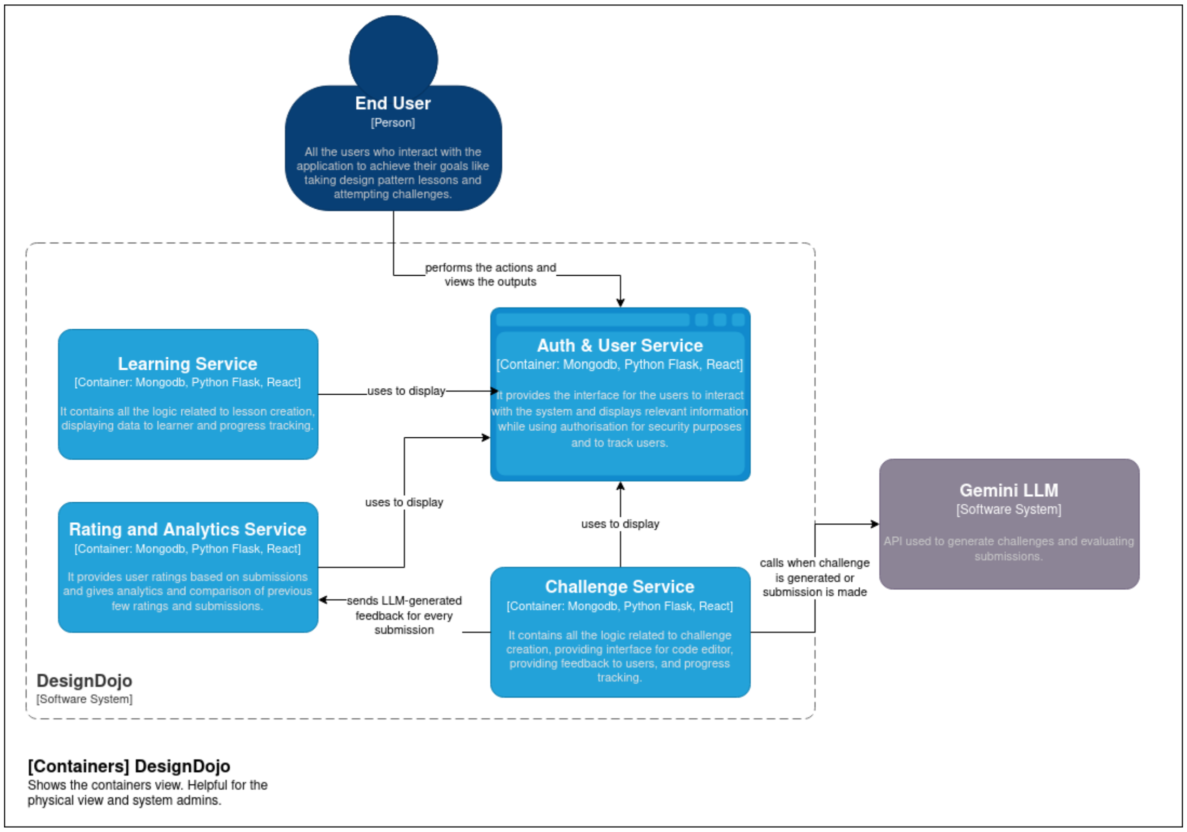
End User
[Person]
All the users who interact with the application to achieve their goals like taking design pattern lessons and attempting challenges.
performs the actions and views the outputs
Learning Service
[Container: Mongodb, Python Flask, React]
It contains all the logic related to lesson creation, displaying data to learner and progress tracking.
uses to display
Auth & User Service
[Container: Mongodb, Python Flask, React]
It provides the interface for the users to interact with the system and displays relevant information while using authorisation for security purposes and to track users.
uses to display
Rating and Analytics Service
[Container: Mongodb, Python Flask, React]
It provides user ratings based on submissions and gives analytics and comparison of previous few ratings and submissions.
uses to display
sends LLM-generated feedback for every submission
Challenge Service
[Container: Mongodb, Python Flask, React]
It contains all the logic related to challenge creation, providing interface for code editor, providing feedback to users, and progress tracking.
calls when challenge is generated or submission is made
Gemini LLM
[Software System]
API used to generate challenges and evaluating submissions.
DesignDojo
[Software System]
[Containers] DesignDojo
Shows the containers view. Helpful for the physical view and system admins.

Figure 3: DesignDojo - Physical View

4. Scenarios (Use Cases)

Purpose: Ties the other views together by describing how the system behaves in specific use cases.
Example Use Cases:

1. **User Registration:**
   - Frontend sends user data to the backend.
   - Backend validates and stores the data in the database.
2. **Learning Module Access:**
   - User selects a module on the frontend.
   - Backend retrieves module content from the database and sends it to the frontend.
3. **Code Submission:**
   - User submits code via the frontend.
   - Backend validates the submission and stores it in the database.
   - Results are returned to the frontend for display.

3.1 Architectural Design Records

3.1.1 Frontend Framework: React with TypeScript

- **Context:** We needed to choose a language for developing the UI/frontend of our app.
- **Decision:** The frontend of the application is built using React with TypeScript.

- **Rationale:** React provides a component-based architecture that simplifies the development of reusable UI components. TypeScript adds static typing, which improves code quality, reduces runtime errors, and enhances developer productivity. This combination ensures scalability and maintainability for a growing codebase.
- **Alternatives Considered:** Angular and Vue.js were considered but rejected due to the team's familiarity with React and the need for a lightweight, flexible framework.

3.1.2 Backend Framework: Flask

- **Context:** We needed to decide on a language to build our app's backend.
- **Decision:** Flask was chosen as the backend framework for the application.
- **Rationale:** Flask is a lightweight and flexible Python web framework that allows rapid development. It provides the necessary tools for building RESTful APIs, which are essential for the application's architecture. Having a Python framework also enables smoother integration with LLM APIs being a major factor in the decision. Flask's simplicity aligns with the project's requirements and the team's expertise in Python.
- **Alternatives Considered:** Django was considered but deemed too heavy for the project's needs, as it includes features like an ORM and admin panel that were not required.

3.1.3 Code Editor Integration: Monaco Editor

- **Context:** Our challenges page allows the user to write and submit code on-site. We needed a code editor in place for that functionality.
- **Decision:** The Monaco Editor is integrated into the frontend for the code editing functionality.
- **Rationale:** Monaco Editor, the same editor used in Visual Studio Code, provides a rich set of features such as syntax highlighting, IntelliSense, and error checking. It is highly customizable and supports multiple programming languages, making it ideal for the learning environment of DesignDojo.
- **Alternatives Considered:** Ace Editor and CodeMirror were considered but lacked some advanced features and community support compared to Monaco Editor.

3.1.4 Design Pattern Learning Module: Modular Data Files

- **Context:** The learning page has lessons for different design patterns. We required a way to store and present the data for each lesson while allowing future extensions.
- **Decision:** Design pattern lessons and quizzes are stored in modular TypeScript files under the src/data directory.
- **Rationale:** This approach allows for easy addition and modification of design pattern content without altering the core application logic. It also keeps the content decoupled from the UI, promoting separation of concerns.
- **Alternatives Considered:** Storing content in a database was considered but rejected to simplify the initial implementation and avoid unnecessary backend complexity.

3.1.5 User Interface Framework: Material-UI

- **Context:** We had to make the user interface more appealing and user-friendly.
- **Decision:** Material-UI is used as the primary UI framework for the website.
- **Rationale:** Material-UI provides pre-built, customizable components that align with modern design principles. This reduces development time while maintaining a professional and consistent look across the platform.

- **Alternatives Considered:** Bootstrap and Tailwind CSS were considered but rejected due to the team's familiarity with Material-UI and its component-based approach. Also there was no need for an advanced tool we wanted a simplistic UI.

4 Architectural Tactics and Patterns

1. **Flask JWT Authentication Middleware** (*Security*)
   Description: All incoming requests to protected endpoints (e.g. profile, submissions) pass through a custom Flask middleware that enforces stateless JWT usage. **Implementation:**

   - On `/login` and `/register`, generate a JWT signed with a rotating HMAC secret and set a short expiration (e.g. 15 min).
   - On each protected route, verify token signature, expiration, and user roles.
   - Reject any request over HTTP; only allow HTTPS transport.
   - Automatically hash incoming `password` fields (bcrypt) before persisting.

   Addresses: Ensures credentials never travel or persist in plaintext, prevents token forgery, and centralizes all auth logic in middleware.

2. **Shared React Layout & Centralized Material-UI Theme** (*Usability*)
   Description: A single `<Layout>` component wraps every page, providing a consistent navbar, sidebar, and footer. All visual styles derive from a shared Material-UI theme configuration. **Implementation:**

   - `src/components/Layout.tsx` includes responsive `AppBar`, `Drawer`, and `Container`.
   - `src/theme.ts` defines primary/secondary palette, typography, and breakpoints for mobile/desktop.
   - Common widgets (buttons, cards, forms) live in `src/components/ui/` and consume theme variables.

   Addresses: Guarantees uniform navigation and appearance; enables seamless user journeys and responsive behavior across devices.

3. **Atomic Database Transactions** (*Reliability*)
   Description: Groups related database operations into a single transaction so that either all succeed or all roll back. **Implementation:**

   - Use SQLAlchemy's `session.begin()` context manager for multi-step flows (e.g. profile update & audit log).
   - Employ `INSERT ... ON CONFLICT DO UPDATE` for idempotent upserts.
   - Capture and handle exceptions to trigger an explicit `session.rollback()`.

   Addresses: Prevents partial writes, maintains data integrity on failures, and ensures consistent state for user profiles and ratings.

4. **Plugin-Style Challenge Registry** (*Extensibility*)
   Description: Dynamically discovers and registers challenge modules at startup, each implementing a standard interface. **Implementation:**

   - In `src/challenges/index.ts`, scan the folder for `*.ts` files and `import()` each module.
   - Each module exports `generate(request):Challenge` and `evaluate(submission):Result`.
   - Populate a `ChallengeRegistry` mapping pattern names to module handlers.

   Addresses: Allows new design-pattern challenges to be added by dropping in a file—no changes required in core API or routing logic.

Implementation Patterns

We employ two primary architectural patterns in DesignDojo: Service-Oriented Architecture (SOA) for backend decomposition, and Model–View–Controller (MVC) for frontend structure. Below we describe each pattern's role and include representative diagrams.

1. **Service-Oriented Architecture (SOA)**
   Role in DesignDojo:

   - *Auth Service:*
     - Manages user registration, login/logout, and JWT issuance/validation.
     - Stateless—no server-side session storage, all tokens verified by middleware.
   - *Content Service:*
     - Serves learning materials (pattern theory, sample code, MCQs) via REST endpoints.
     - Publishes "lesson-updated" events so caches and dashboards stay in sync.
   - *Challenge Service:*
     - Generates coding challenges on-the-fly by calling the LLM adapter.
     - Hosts the Monaco Editor integration and routes submissions through the evaluation pipeline.
   - *Rating Service:*
     - Calculates and persists user ratings based on challenge results.
     - Exposes rating history for the dashboard's analytics charts.

   Benefits:

   - *Scalability:* Each service can be scaled independently (e.g. more Challenge workers during peak usage).
   - *Fault Isolation:* A failure in the Rating Service does not take down Learning or Auth.
   - *Extensibility:* New services (e.g. Forum, Reporting) plug in behind the same gateway.

   C4 Container Diagram:

2. **Model–View–Controller (MVC)**

   - *Models:*
     - `AuthModel`: Manages JWT login/logout, token storage, and subscription to auth state changes.
     - `LessonModel`: Loads pattern theory, sample code, and MCQs; notifies subscribers on data arrival.
     - `ChallengeModel`: Fetches/generates LLM challenges, submits code solutions, and publishes evaluation results.
     - `RatingModel`: Retrieves and stores the user's rating history and notifies the dashboard.
   - *Views:*
     - `LoginView`, `LessonListView`, `ChallengeEditorView`, `DashboardView` render UI based on model state and capture user events (e.g. button clicks, form submissions).
   - *Controllers:*
     - `AuthController`, `LessonController`, `ChallengeController`, `DashboardController` listen for view events, orchestrate REST API calls, invoke model methods, and handle navigation.

   UML Class Diagram:

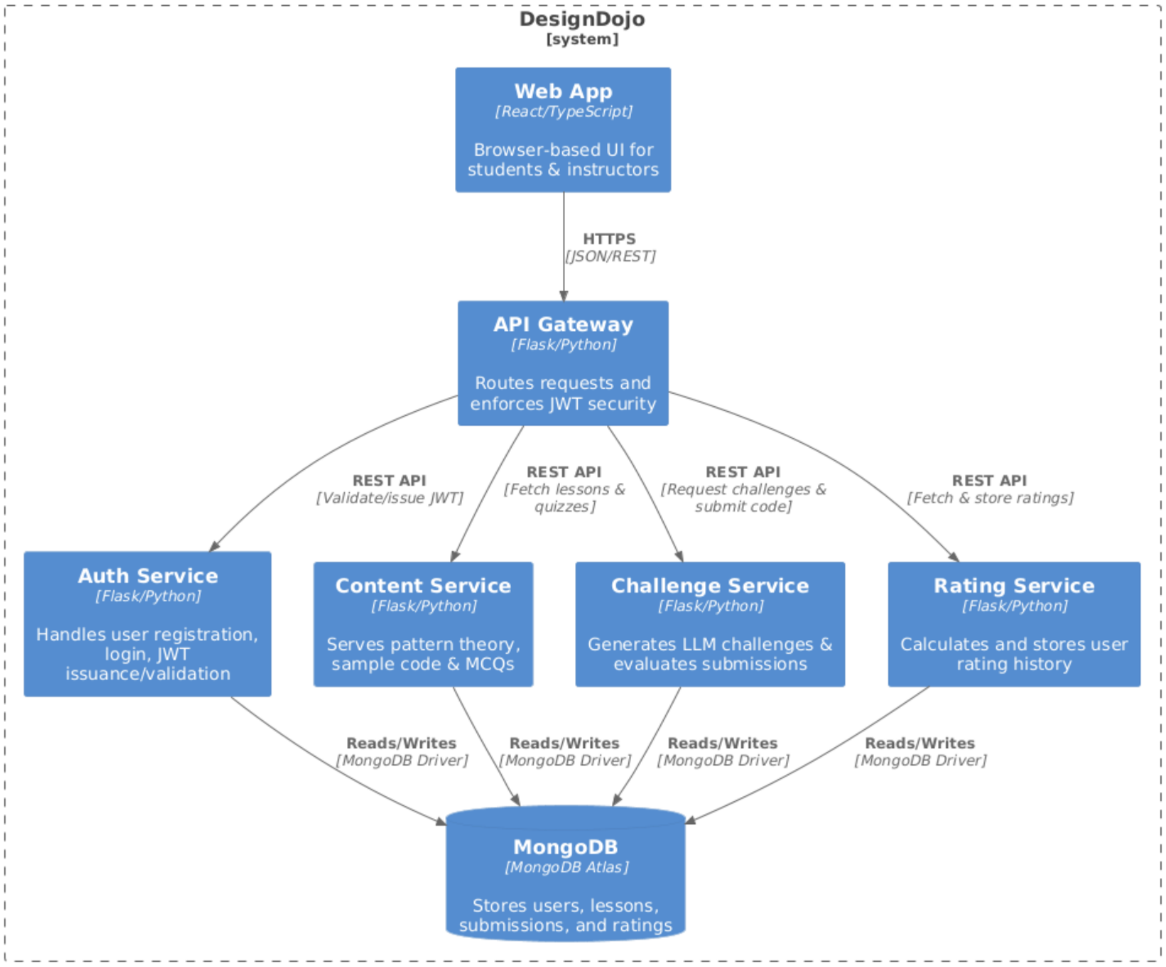

Figure 4: C4 Container diagram showing DesignDojo's SOA decomposition

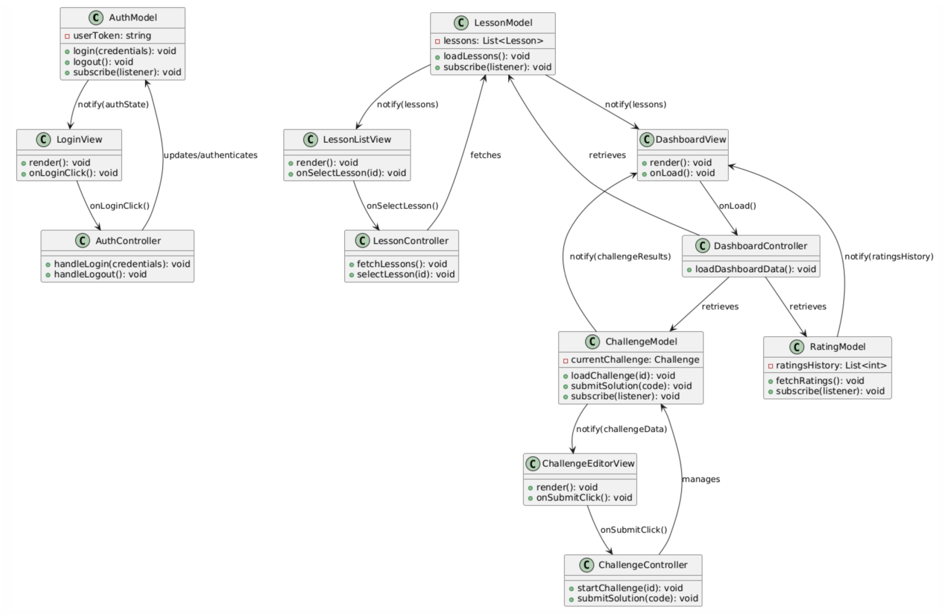

Figure 5: MVC Class Diagram for DesignDojo Frontend

5 Prototype Implementation and Analysis

Implemented Architecture: Service-Oriented Architecture (SOA)
Chosen Pattern for Comparison: Microservice Architecture

5.1 Quantification of Non-Functional Requirements

- **Development and Deployment Complexity**
  - **SOA:** Data consistency is easier to maintain since all components share the same database and application context. $\implies$ **Complexity Level:** High
  - **Microservices:** Each services has its own codebase, dependencies and deployment pipelines. This increases the complexity of development and deployment. $\implies$ **Complexity Level:** High
- **Data Consistency**
  - **SOA:** Easier to develop and deploy as all features are part of a single codebase. Deployment involves managing a single codebase. $\implies$ **Complexity Level:** Low
  - **Microservices:** Each service usually has its own database, so keeping data in sync across services is harder. You either need to use complex distributed transactions or accept delays in consistency. $\implies$ **Complexity Level:** Medium
- **Scalability**
  - **SOA:** Scaling the application requires replicating the entire monolithic backend, which can lead to inefficient resource utilization. $\implies$ **Complexity Level:** Medium
  - **Microservices:** Individual services can be scaled independently based on their specific load, making the architecture more resource-efficient. $\implies$ **Complexity Level:** High
- SOA will have a faster response time than Microservices as it avoids inter-service communication leading to lower latency.
- Microservices will have a higher throughput than SOA as it allows scaling of indivdual components while to scale SOA, the entire system needs to be replicated.

5.2 Trade-offs

- **Scalability**
  - **SOA:** Easier to develop and deploy initially but harder to scale as the system grows. A single point of failure exists in the monolithic backend.
  - **Microservices:** Highly scalable, as individual services can be scaled independently. However it introduces complexity in deployment and service orchestration.
- **Maintainability**
  - **SOA:** Easier to maintain initially due to a single codebase. However, as the application grows, it can become harder to manage.
  - **Microservices:** Each service is smaller and easier to maintain. However, managing multiple services together requires robust practices and tools.
- **Performance**
  - **SOA:** Lower latency due to fewer network calls. However, performance may degrade under heavy load due to the monolithic structure.
  - **Microservices:** Higher latency due to inter-service communication but better performance under heavy load due to independent scaling.
- **Complexity**

- **SOA:** Simpler to develop and deploy, as all features are part of a single application.
- **Microservices:** More complex to develop due to the need for inter-service communication, service discovery, and data consistency.