SAMPLE HLD

High-Level Design (Refined): Traditional Django Online Judge

1. Introduction

This document outlines the high-level design for a monolithic, server-rendered Online Judge (OJ) platform built with the Django framework. The platform's primary goal is to provide a straightforward and effective environment for users to practice programming problems.

1.1. Core Objectives

- **User Management:** Secure user registration, login, and session management.
- **Problem Solving:** A curated list of problems for users to solve.
- **Code Submission:** An interface for users to submit code in multiple languages.
- **Automated Evaluation:** A backend system to compile, run, and judge submissions against test cases.
- **Feedback & History:** Immediate feedback on submissions and a personal history log for each user.

1.2. Scope

- **In-Scope:** All core features listed above, a basic user dashboard, problem details view, submission system, and an admin panel for content management.
- **Out-of-Scope (for this version):** Real-time contests, discussion forums, user-to-user interaction, AI-based feedback, and team-based features.

---

2. Technology Stack & Rationale

Component	Technology	Rationale
Backend Framework	**Django**	A "batteries-included" framework perfect for rapid development of monolithic apps. Its ORM, forms, and admin panel are ideal.
Frontend	**HTML, CSS, JavaScript**	Ensures simplicity and avoids frontend framework overhead. Server-side rendering is handled entirely by Django Templates.

UI Styling	**Bootstrap (Optional)**	A simple CSS framework to create a clean, responsive UI without writing extensive custom CSS from scratch.
Database	**SQLite3**	Django's default database. It's file-based, requires zero configuration, and is sufficient for development and small-scale apps.
Code Execution	**Python subprocessModule**	The simplest method for executing external code directly from the Django backend, suitable for a foundational implementation.

---

3. System Architecture

The application will be a **Django Monolith**. This single codebase will be responsible for all aspects of the application, promoting simplicity in development, testing, and deployment.

Component Breakdown:

- **manage.py**: The command-line utility for interacting with the Django project.
- **settings.py**: Contains all project configurations, including database settings, installed apps, and template directories.
- **urls.py**: The URL dispatcher. It maps incoming URL requests to the appropriate view function.
- **views.py**: Contains the business logic. Each view handles a specific request, interacts with the models, and renders a template.
- **models.py**: Defines the database schema through Django model classes. The ORM handles all database communication.
- **forms.py**: Defines form classes for data validation and rendering (e.g., RegistrationForm, SubmissionForm). This centralizes validation logic.
- **templates/**: Directory containing all HTML files. Django's template engine dynamically inserts data into these files.
- **static/**: Directory containing all static assets like CSS, JavaScript, and images.

---

4. Page and Feature Breakdown

Page	URL	Access Level	Key Features & Django Components
Home	/	All Users	- Welcome message. - Conditional links (Login/Register or Dashboard). - TemplateView
Registration	/accounts/register/	Unauthenticated	- Registration form. - Uses a UserCreationForm. - Displays validation errors. - Redirects to login on success.
Login	/accounts/login/	Unauthenticated	- Login form. - Uses Django's built-in LoginView and AuthenticationForm. - Shows errors via the **Messages Framework**.
Logout	/accounts/logout/	Authenticated	- Logs the user out. - Uses Django's built-in LogoutView.
Problem Dashboard	/problems/	Authenticated	- Lists all problems in a table (Title, Difficulty, Solved Status). - Fetches data in ListView.
Problem Detail & Submit	/problems/<int:pk>/	Authenticated	- Displays problem details and sample test cases. - A SubmissionForm with <textarea> and language <select>. - DetailView

Submission Result	/submissions/<int:pk>/	Submission Owner	- Shows the submitted code (read-only), final status, and any output/error. - DetailView with access control logic.
Submission History	/submissions/	Authenticated	- Lists the current user's submission history. - Fetches data using Submission.objects.filter(user=request.user).
Admin Panel	/admin/	Admin Users	- Built-in Django Admin. - CRUD operations for Problem and TestCasemodels.

---

5. Core Workflows

5.1. User Journey

1. A new user visits the site, clicks "Register", fills out the form, and creates an account.
2. They are redirected to "Login" and sign in.
3. Upon login, they land on the "Problem Dashboard". They see a list of problems.
4. They click on a problem, which takes them to the "Problem Detail" page.
5. After reading the problem, they write their code in the text area, select a language, and click "Submit".
6. The page reloads or redirects, showing a "Pending" status. After a few seconds, the page updates to show the final result ("Accepted", "Wrong Answer", etc.).
7. The user can view all their past attempts on their "Submission History" page.

5.2. Code Execution & Evaluation Flow

This is the most critical workflow. It must be handled carefully to ensure correctness and security.

1. **Submission Received:** The user submits the code via a POST request. The Submission view validates the form and creates a new Submission instance with status='Pending'.
2. **Trigger Execution:** The view then triggers the execution logic. For this HLD, this can be a direct function call. *(For a production system, this should be offloaded to a background worker like Celery).*
3. **Secure Sandboxing:**

    - A secure, temporary directory is created.
    - The user's code is written to a file within this directory (e.g., main.py).
4. **Compilation (if needed):**
    - If the language is C++, the subprocess module is used to run g++ main.cpp -o main.
    - If compilation fails, the stderr is captured, the submission status is updated to **"Compilation Error"**, and the flow stops.
5. **Execution and Testing:**
    - The compiled executable or the script is run in a new process using subprocess.run().
    - Crucially, the timeout parameter of subprocess.run() is set to the problem's time_limit.
    - The input parameter is used to pass the input_data from the current TestCase.
    - stdout and stderr are captured.
6. **Verdict Determination:**
    - **Time Limit Exceeded (TLE):** If subprocess.TimeoutExpired is caught.
    - **Runtime Error (RTE):** If the process returns a non-zero exit code or writes to stderr.
    - **Wrong Answer (WA):** If the captured stdout (after stripping whitespace) does not exactly match the expected_output.
    - **Accepted (AC):** If the code passes all test cases successfully.
7. **Database Update:** The Submission object is updated with the final status and the captured output.
8. **Cleanup:** The temporary directory and all its contents are deleted.
9. **Feedback to User:** The user is redirected to the submission result page, which now displays the final verdict.

---

6. Security Considerations

- **Code Execution:** Running arbitrary code with subprocess is **highly insecure**. The code must be executed by a non-root, low-privilege user. File system access should be heavily restricted. **Using Docker containers for sandboxing is the standard and recommended practice for any real-world deployment.**
- **Cross-Site Scripting (XSS):** Django's template engine auto-escapes variable content by default, providing strong protection. All user-generated content (like problem descriptions) should be carefully handled if Markdown or HTML is allowed.
- **Cross-Site Request Forgery (CSRF):** Django's built-in CSRF protection will be used for all POST forms.
- **SQL Injection:** Django's ORM uses parameterized queries, which eliminates the risk of SQL injection.