Frontend Integration Documentation

This document provides detailed information on integrating the DSA Practice Platform frontend with a new backend, specifically focusing on a Java and MySQL implementation. It outlines the frontend structure, API interaction patterns, and expected data formats.

1. Frontend Project Structure

The frontend is a React application built with Vite, TypeScript, and Tailwind CSS. The key directories and files are:

Plain Text

```
dsa_frontend/
├── public/                 # Static assets (e.g., favicon.ico)
├── src/
│   ├── assets/             # Images, icons, etc.
│   ├── components/         # Reusable UI components
│   │   ├── Layout/         # Header, Footer, and main Layout components
│   │   └── ui/             # Shadcn/ui components (buttons, cards, etc.)
│   ├── contexts/           # React Contexts for global state (e.g.,
AuthContext)
│   ├── hooks/              # Custom React hooks
│   ├── lib/                # Utility functions and API client
│   │   └── api.js          # Axios instance and API calls
│   │   └── utils.js        # General utility functions
│   ├── pages/              # Main application pages (Home, Login, Sheets,
SheetDetail)
│   ├── App.jsx             # Main application component, handles routing
│   ├── index.css           # Global CSS styles (Tailwind CSS)
│   └── main.jsx            # Entry point for the React application
├── index.html              # Main HTML file
├── package.json            # Project dependencies and scripts
├── pnpm-lock.yaml          # pnpm lock file
├── postcss.config.js       # PostCSS configuration
├── tailwind.config.js      # Tailwind CSS configuration
├── tsconfig.json           # TypeScript configuration
└── vite.config.js          # Vite build configuration
```

2. API Interaction and Endpoints

The frontend communicates with the backend via RESTful API calls using `axios`. All API requests are centralized in `src/lib/api.js`. The backend should expose endpoints that match the following specifications:

Base URL

The API base URL is configured in `src/lib/api.js`. You will need to update this to point to your Java backend.

JavaScript

```
// src/lib/api.js
import axios from 'axios';

const API_BASE_URL = import.meta.env.VITE_API_BASE_URL ||
'http://localhost:5000/api'; // <--- UPDATE THIS

const api = axios.create({
  baseURL: API_BASE_URL,
  withCredentials: true, // Important for sending cookies/sessions
});

export default api;
```

Authentication Endpoints

The frontend expects the following authentication endpoints:

- **`POST /api/auth/login`**
  - **Request Body:**
  - **Response (Success - HTTP 200):**
  - **Response (Failure - HTTP 401/400):**
  - **Notes:** The current frontend uses a simplified demo login. A robust Java backend should implement proper user authentication (e.g., JWT, OAuth2) and return a token or session cookie for subsequent authenticated requests. The `AuthContext` in `src/contexts/AuthContext.jsx` handles user state.
- **`GET /api/auth/me`**
  - **Description:** Retrieves information about the currently authenticated user.
  - **Response (Success - HTTP 200):**
  - **Response (Failure - HTTP 401):**

Practice Sheets Endpoints

- **`GET /api/sheets`**
  - **Description:** Retrieves a list of all available practice sheets.

    - **Response (Success - HTTP 200):**
- `GET /api/sheets/{sheet_id}`
    - **Description:** Retrieves details for a specific practice sheet, including its problems.
    - **Response (Success - HTTP 200):**

Other Endpoints (Backend Ready, Frontend Integration Pending)

The current frontend has placeholders for these features, and the Flask backend already includes the API routes. Your Java backend should implement these as well:

- **Progress Tracking:**
    - `GET /api/progress/user/{sheet_id}` : Get user progress for a sheet.
    - `POST /api/progress` : Update problem completion status.
- **Notes Management:**
    - `GET /api/notes/problem/{problem_id}` : Get user notes for a problem.
    - `POST /api/notes` : Create/update notes.
- **Admin/Moderation:**
    - `GET /api/admin/users` : List all users (admin only).
    - `POST /api/admin/sheets` : Create new sheets (admin only).

3. Running the Frontend Locally

To run the frontend independently, follow these steps:

1. **Navigate to the frontend directory:**
2. **Install dependencies (using pnpm, npm, or yarn):**
3. **Configure API Base URL:**
   Create a `.env` file in the `dsa_frontend` directory and add your Java backend API URL:
4. **Start the development server:**

4. CORS Configuration for Java Backend

When developing your Java backend, ensure proper Cross-Origin Resource Sharing (CORS) is configured to allow the React frontend to make requests. Here's a conceptual example for a Spring Boot application (adjust for your specific Java framework):

```
Java
```

```
// Example for Spring Boot (Java)
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.config.annotation.CorsRegistry;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;

@Configuration
public class WebConfig implements WebMvcConfigurer {

    @Override
    public void addCorsMappings(CorsRegistry registry) {
        registry.addMapping("/api/**") // Apply CORS to all /api endpoints
                .allowedOrigins("http://localhost:5173", "https://your-
frontend-domain.com") // Allow your frontend origin(s)
                .allowedMethods("GET", "POST", "PUT", "PATCH", "DELETE",
"OPTIONS")
                .allowedHeaders("*")
                .allowCredentials(true) // Important for session/cookie-
based authentication
                .maxAge(3600);
    }
}
```

Make sure to replace `http://localhost:5173` with the actual URL where your React frontend is running, especially in production environments.

5. Data Models (Java/MySQL)

Here are the conceptual data models that your Java backend should implement to match the frontend's expectations. You would typically map these to JPA entities or similar ORM models for MySQL.

User Model

Java

```
public class User {
    private String id; // UUID or unique identifier
    private String email;
    private String displayName;
    // Add fields for password hash, roles, etc., as per your auth system
}
```

Sheet Model

Java

```
public class Sheet {
    private String id; // UUID or unique identifier
    private String title;
    private String description;
    private int totalProblems;
    private String createdAt; // ISO 8601 format (e.g., "2025-01-
01T12:00:00Z")
    // List<Problem> problems; // For detailed sheet view
}
```

Problem Model

Java

```
public class Problem {
    private String id; // UUID or unique identifier
    private String title;
    private String difficulty; // e.g., "Easy", "Medium", "Hard"
    private List<String> topics; // e.g., ["Arrays", "Hashing"]
    private List<String> subtopics; // e.g., ["Array Manipulation"]
    private String leetcodeUrl;
    private String youtubeUrl;
    private String sheetId; // Foreign key to Sheet
}
```

6. Next Steps for Java Backend Development

1. **Choose a Java Web Framework:** Spring Boot is highly recommended for building RESTful APIs.
2. **Database Setup:** Configure MySQL and create the necessary tables based on the data models.
3. **Implement Data Models:** Create Java classes (e.g., JPA entities) for User, Sheet, and Problem.
4. **Develop Repositories/DAOs:** Create interfaces or classes for database interactions.
5. **Implement REST Controllers:** Create Java classes to handle incoming HTTP requests and expose the API endpoints as specified in Section 2.
6. **CORS Configuration:** Implement CORS as described in Section 4.
7. **Authentication:** Implement a robust authentication system (e.g., Spring Security with JWT) for user login and session management.

8. **Seed Data:** Create a mechanism to populate your MySQL database with initial data for testing.

By following this documentation, you should be able to successfully integrate the provided React frontend with your new Java and MySQL backend. Good luck!