DeepKlarity Assignment Reference Document

AI Wiki Quiz Generator

1. Objective

The primary goal of this project is to create a full-stack application that leverages AI to transform unstructured text from a Wikipedia article into a structured, engaging, and educational quiz.

Core Functionality:

1. **Input:** Accept a Wikipedia URL from a user.
2. **Process:** Scrape the article, use a Large Language Model (LLM) via LangChain/Gemini to generate a quiz (5-10 questions), and extract summary information.
3. **Storage:** Persist the original URL, scraped content, and the generated quiz data in an MYSQL or POSTGRESQL database.
4. **Output:** Display the quiz immediately after generation and allow users to view a history of all generated quizzes.

2. Technical Requirements

To successfully execute this project, you will need the following tools and libraries:

Category	Component	Purpose	Installation/Notes
Backend (Python)	**Python 3.10+**	Core programming language.	Required environment
	FastAPI	High-performance, easy-to-use Python web framework for the API.	`pip install fastapi uvicorn`

Category	Component	Purpose	Installation/Notes
	BeautifulSoup4	Library for parsing HTML and extracting data (scraping).	`pip install beautifulsoup4 requests`
	LangChain	Framework for developing applications powered by LLMs.	`pip install langchain-core langchain-community pydantic`
	Pydantic	Data validation and settings management (built into FastAPI).	Included with FastAPI
LLM Access	**Gemini Free Tier API**	The Large Language Model used for quiz generation.	Set up a virtual environment and use the LangChain integration (`langchain-google-genai`).
Frontend (Web)	**React**	JavaScript library for building the user interface.	Requires Node.js and npm/yarn for React tooling (but the backend is strictly Python).
	Tailwind CSS	Utility-first CSS framework for clean, minimal UI design.	Integrate into the React build process.
Database	**MYSQL or POSTGRESQL**	Simple, file-based database for data storage.	

NOTE: The use of **Node.js** for the backend or any core API functionality is strictly prohibited and will result in rejection. The backend **must** be implemented using Python (FastAPI/Django) as specified in the Technical Requirements.

3. Step-by-Step Instructions

Follow these steps sequentially to build your application.

Phase 1: Environment and Backend Setup

Step 1: Project Structure & Python Environment

1. **Create Project Directories:** Make a main folder (`ai-quiz-generator`) and two subfolders: `backend` and `frontend`.
2. **Create Virtual Environment:** Navigate into the `backend` folder and create a Python virtual environment:

```
python -m venv venv
source venv/bin/activate  # On Windows: venv\Scripts\activate
```

3. **Install Backend Dependencies:** Install the required Python libraries:

```
pip install fastapi uvicorn[standard] sqlalchemy beautifulsoup4 requests p
ydantic langchain-core langchain-community python-dotenv langchain-g
oogle-genai
```

4. **API Key Setup (Optional/Local):** Create a file named `.env` in the `backend` folder to store your Gemini API key:

```
GEMINI_API_KEY="YOUR_API_KEY_HERE"
```

Step 2: Database (MYSQL or POSTGRESQL) Setup

1. **Define Database Connection:** Create a file named `database.py`. Define the SQLAlchemy engine pointing to an MYSQL or POSTGRESQL file (`./quiz_history.db`).
2. **Define Base Class:** Create a declarative base class for your models.
3. **Define the `Quiz` Model:** Create a `Quiz` class that inherits from the base. It must include fields for:
    - `id` (Primary Key)
    - `url` (String)
    - `title` (String)
    - `date_generated` (DateTime)
    - `scraped_content` (Text - for Bonus)

- `full_quiz_data` (Text/JSON - **CRUCIAL:** Use a Text field to store the complex JSON structure of the quiz, key entities, and related topics after serialization using `json.dumps()`).

Step 3: Wikipedia Scraper

1. **Create Scraper File:** Create `scraper.py`.
2. **Implement `scrape_wikipedia(url)`:** Use `requests` to fetch the URL content and `BeautifulSoup` to parse it.
3. **Clean Content:** Identify the main article body (e.g., using known Wikipedia CSS classes or IDs like `#mw-content-text`). Strip out boilerplate, reference links (`sup` tags), and tables to provide a clean text input for the LLM. Return the clean text and the article title.

Phase 2: AI Integration and API Endpoints

Step 4: LLM Integration (LangChain/Pydantic)

1. **Define Pydantic Schema:** In `models.py`, define the strict JSON structure the LLM must return. (See Example Code D for snippet).
2. **Setup LLM Generator:** In `llm_quiz_generator.py`:
   - Initialize the Gemini model (e.g., `gemini-2.5-flash`) using `langchain-google-genai`.
   - Define a detailed **Prompt Template** that includes placeholders for the article text and the format instructions.
   - Use LangChain's `JsonOutputParser` to enforce the Pydantic schema, ensuring the model returns valid JSON.
   - Create a chain combining the prompt, the model, and the parser.

Step 5: FastAPI Backend Endpoints

1. **Setup `main.py`:** Initialize the FastAPI app and set up CORS middleware to allow the React frontend to communicate (See Example Code A for snippet).
2. **Endpoint 1: `/generate_quiz` (POST):**
   - Accepts a JSON body with the `url`.

- Calls `scrape_wikipedia`.
- Calls the LLM generation chain.
- Saves the data (serializing the quiz JSON to a string) into the MYSQL or POSTGRESQL database.
- Returns the full JSON data of the generated quiz.

3. **Endpoint 2: `/history` (GET)**:
   - Queries the database for a list of all saved quizzes.
   - Returns a simple list of objects containing `id`, `url`, `title`, and `date_generated`.
4. **Endpoint 3: `/quiz/{quiz_id}` (GET)**:
   - Fetches a specific quiz record by `id`.
   - **Crucial:** Deserialize the `full_quiz_data` text field back into a Python dictionary/JSON object before returning it in the response.

Phase 3: Frontend (React) Development

Step 6: React Setup and Component Structure

1. **Setup React/Tailwind:** Initialize your React app (e.g., using `vite`) and install/configure Tailwind CSS.
2. **Core Components:**
   - `App.jsx`: Manages the application state (e.g., `activeTab`).
   - `GenerateQuizTab.jsx`: Contains the URL input form, loading state, and displays the result.
   - `HistoryTab.jsx`: Fetches and displays the history table.
   - `QuizDisplay.jsx`: A reusable component to render the full structured quiz data (used in Tab 1 and the History modal).
   - `Modal.jsx`: Generic modal component.

Step 7: Frontend Integration and UI

1. **API Service:** Create a service file (e.g., `api.js`) to handle all `fetch` requests to the FastAPI backend.
2. **Tab 1 - Generate Quiz:**
    - Implement input validation for the URL.
    - Display a visually clear loading state (spinner or message) while the backend processes.
    - Use the `QuizDisplay` component to render the result in card-based layout.
3. **Tab 2 - History:**
    - Render a table listing ID, URL, and Title.
    - Implement a "Details" button for each row that:
        - Fetches the specific quiz data via the `/quiz/{quiz_id}` endpoint.
        - Opens a modal and passes the data to the reusable `QuizDisplay` component inside the modal.

4. Project Structure

```
ai-quiz-generator/
├── backend/
│   ├── venv/                  # Python Virtual Environment
│   ├── database.py              # SQLAlchemy setup and Quiz model
│   ├── models.py              # Pydantic Schemas for LLM output (QuizOutpu
t)
│   ├── scraper.py             # Functions for fetching and cleaning Wikipedia
HTML
│   ├── llm_quiz_generator.py      # LangChain setup, prompt templates, and
chain logic
│   ├── main.py                # FastAPI application and API endpoints
│   ├── requirements.txt          # List of all Python dependencies
│   └── .env                   # API keys and environment variables
│
├── frontend/
│   ├── src/
```

```
│   │   ├── components/           # Reusable UI parts (e.g., QuizCard, TabButto
n, Modal)
│   │   │   ├── QuizDisplay.jsx    # Reusable component for rendering generat
ed quiz data
│   │   │   └── HistoryTable.jsx
│   │   ├── services/
│   │   │   └── api.js            # Functions for communicating with the FastAPI b
ackend
│   │   ├── tabs/
│   │   │   ├── GenerateQuizTab.jsx
│   │   │   └── HistoryTab.jsx
│   │   ├── App.jsx               # Main React component, handles tab switching
│   │   └── index.css             # Tailwind directives and custom styles
│   ├── package.json
│
└── README.md                     # Project Setup, Endpoints, and Testing Instruc
tions
```