Backend Architecture & Logic Explanation

Backend Architecture & Logic Explanation - FIR Generator

Overview

The backend is a FastAPI application that serves as the brain of the FIR Generator. It bridges the gap between the user's informal case description and the formal legal language required for a First Information Report (FIR).

Core Components

1. **API Server (`main.py`)**
   - **Framework**: Uses FastAPI for high-performance HTTP requests.
   - **Endpoints**:
     - `POST /api/generate_fir`: The main endpoint that receives case details and returns the generated FIR.
   - **LLM Integration**: Uses the `groq` client to access Llama 3.3 70B, a powerful large language model, for semantic analysis and drafting.

2. **Data Models (`models.py`)**
   - Defines the strict schema for input data using Pydantic.
   - Ensures that requests contain all necessary fields: `complainant`, `accused`, `case_description`, `fir_number`, `registration_date`, etc.

3. **Utility Logic (`utils.py`)**
   - **IPC Data Loading**: Loads the Indian Penal Code (IPC) JSON dataset into memory when the server starts.
   - **Search/Retrieval**: Implements a custom search algorithm to find relevant laws.

The Generation Workflow (How it works)

When a request hits `/api/generate_fir`, the following steps occur:

Step 1: Semantic Query Expansion

- **Problem**: Users describe crimes in simple English (e.g., "stolen bike"), but the law uses specific terms (e.g., "Theft", "Movable Property").
- **Solution**: The backend asks the LLM to analyze the informal description and identify the top

Backend Architecture & Logic Explanation

5 most relevant IPC Section Numbers and Legal Keywords.

- **Result**: "stolen bike" -> "Section 379, Theft, Dishonest Intention".

Step 2: Intelligent Legal Search (`utils.py`)

- The backend combines the user's description with the LLM's keywords to search the IPC database.
- **Scoring Logic**:
- **Keyword Matches**: Checks if keywords appear in Section Titles (High Weight) or Descriptions (Low Weight).
- **Phrase Matching**: Boosts score if exact phrases match.
- **Section Number Boost**: (Crucial) If the search query contains an exact section number (e.g., "Section 379"), that section gets a massive score boost. This ensures that if the AI identifies the correct section, the retrieval system is guaranteed to pull its full legal definition.

Step 3: Prompt Construction

- The backend assembles a massive context for the LLM, including:
- **Role**: "You are an expert Police Officer..."
- **Legal Context**: The full text of the relevant IPC sections retrieved in Step 2.
- **Rules**: Strict formatting instructions (FIR structure, 3 sections minimum).
- **Input Data**: The specific case details (Complainant, Accused, Date, FIR Number, etc.).

Step 4: FIR Drafting

- The LLM generates the final FIR text following the strict template.
- It incorporates the specific variables (e.g., FIR Number) injected into the prompt.
- It applies the legal sections provided in the context.

Step 5: Response

- The generated text is sent back to the frontend to be displayed, downloaded, or printed.

Key Features

- **Hybrid Search**: Combines keyword search (TF-IDF style logic) with Semantic Expansion (LLM) for high accuracy.
- **Context Injection**: The AI doesn't just "hallucinate" laws; it is fed the actual legal text from the database to ensure citations are correct.

- **Structured Output**: The prompt enforces a specific layout to ensure the output looks like a real official document.

Verdict Generation Logic (AI Judge)

The system uses a **First-Principles Legal Reasoning** approach to generate verdicts.

How it Works

1. **Input**: The LLM receives the **Charge Sheet** (Facts, Investigation Summary) and the **Original Complaint**.
2. **Role**: The AI is prompted to act as an "Indian District Judge" with deep knowledge of IPC and CrPC.
3. **Reasoning**: It evaluates the evidence against the cited laws to determine:
   - **Conviction Probability**: Guilty vs. Not Guilty.
   - **Punishment**: Appropriate jail term and fine based on the severity and specific section mandates.
   - **Rationale**: A brief legal explanation.

Why this is Impactful (First-Principles vs. Precedent)

We chose this "AI Judge" approach over a "Previous Verdict Search" (RAG) approach for several key reasons:

1. **Unbiased Justice**: RAG (Retrieval Augmented Generation) relies on finding "similar" past cases. If the retrieved case had a flawed or specific circumstance (e.g., a plea deal), the AI might blindly copy that outcome. Our approach evaluates the *current* facts on their own merit.
2. **Tailored Punishment**: Every case has unique details (e.g., age of accused, recovery of stolen goods). A generic search matches keywords, but a Reasoning Engine matches *context*, leading to more accurate sentencing predictions.
3. **Zero-Shot Adaptability**: The system can handle rare or complex combinations of offenses (e.g., Theft + Cheating + Assault) effectively, whereas finding a single past case with that exact combination for a RAG search would be difficult or impossible.

This transforms the tool from a "Case Search Engine" into an "Intelligent Legal Assistant" that mimics judicial thought processes.

Backend Architecture & Logic Explanation

Detailed Technical Justification: Non-RAG vs. RAG for Verdict Prediction

1. Focus on 'Legal Logic' vs. 'Pattern Matching'

- **Without RAG (Current Logic):** Forces the AI to strictly evaluate the **facts of the current case** against the **sections of law** (which it already 'knows' from pre-training). It acts like a Judge looking at the case *de novo*.
- **With RAG:** The system might retrieve 5 past cases that 'look' similar. If those past cases resulted in acquittal due to a technicality that *isn't* present in your case, the AI might blindly copy that verdict. RAG can sometimes bias the model towards 'what happened before' rather than 'what the law says about *this* specific fact pattern.'

2. Context Window Pollution

- **Without RAG:** You can feed the *entire* Charge Sheet, FIR, and Witness Statements into the context window. The model has perfect visibility of the current case details.
- **With RAG:** Valuable context window space is consumed by text from *old* cases. This leaves less room for the details of the *current* case, potentially causing the model to miss a crucial detail in the Charge Sheet.

3. Hallucination vs. Misinterpretation

- **Without RAG:** While LLMs without RAG can hallucinate *citations* (e.g., citing a non-existent case name), they are generally very good at **reasoning**.
- **With RAG:** RAG systems often suffer from **'Retriever errors'**. If the database retrieves an irrelevant case because it shared a keyword (e.g., 'knife'), the LLM might get confused and try to force-fit that irrelevant logic.

4. Zero-Shot Generalization

- Foundation models like Llama 3 or GPT-4 have been trained on millions of legal documents. They already understand the difference between *Culpable Homicide* and *Murder*. For a general 'Verdict Predictor,' their internal knowledge base is often sufficient and more consistent than a small, incomplete RAG database.

Summary

- **Current System:** Best for pure legal reasoning, applying statutory law to facts, and ease of use (no database maintenance).

Backend Architecture & Logic Explanation

- **RAG Approach:** Best for finding specific case precedents (e.g., 'Find me a case from 1995 regarding Section 302 where the weapon was a cricket bat'). Since the goal is **'Predict Verdict'** (reasoning) and not **'Find Precedent'** (search), the non-RAG approach is a valid and robust design choice.