**Court-Data Fetcher & Mini-Dashboard**

**Objective**

Building a small web app that lets a user choose a Case Type and Case Number for a specific Indian court, then fetches and displays the case metadata and latest orders/judgments.

**Introduction**

The Court Data Fetcher & Mini-Dashboard is a web application that allows a user to search for court case details from Delhi High Court or Faridabad District Court by entering the Court name, Case Type, Case Number and Filing Year.It scrapes live data from the selected court’s official website, extracts important metadata, and displays it in a user-friendly dashboard. Every search is recorded in the database along with raw responses for auditing.

**Purpose**

* Automate retrieval of publicly available court case information.
* Bypass session tokens/view-state variables required for valid form submission.
* Handle CAPTCHA where possible and provide fallbacks where not.
* Parse structured case information like: parties involved, filing date and next hearing date, latest Orders/Judgments PDF links.
* Store all searches & results for reuse/history.

**System Architecture**

1. **Frontend (UI Layer)**

* Files: index.html, style.css, app.js.
* Provides form inputs & renders results.
* Sends user input to backend through API calls.
* Displays data from live scrape or mock fallback.

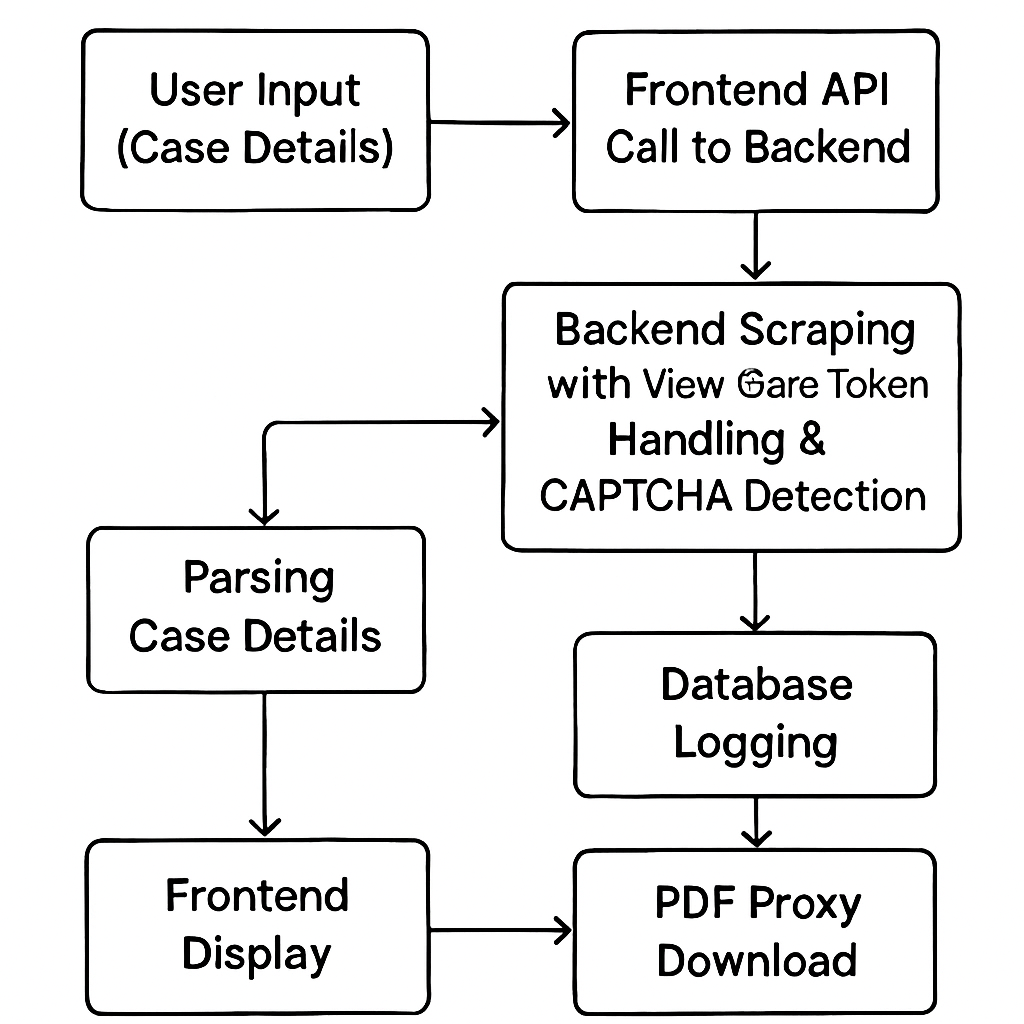
1. **Backend (Application Layer)**

* File: app.py (Flask application)
* Accepts user search parameters, initiates scraping request.
* Manages session handling, view-state tokens, CAPTCHA, and HTML parsing.
* Interfaces with database for logging queries and results.
* Provides /api/download route to proxy PDF downloads.

1. **Database (Storage Layer)**

* SQLite database court\_data.db.
* Table structure:case\_queries – logs every search: parameters, timestamps, status, raw response.

case\_data – parsed details of the case.case\_orders – links to judgment/order PDFs.



**app.py (Backend)**

**Imports**: Flask for API, requests for HTTP requests, BeautifulSoup for HTML parsing, sqlite3 for DB, json for structured data.

**Flask App:** Root (/) route serves homepage.

**/api/search route:**

* Receives search parameters from frontend as JSON.
* Logs the search to DB with status = "processing".
* Creates requests.Session() to preserve cookies and tokens.

**Scrapes court site:**

* Gets search form page.
* Extracts hidden fields (e.g., VIEWSTATE, CSRF tokens).
* Detects CAPTCHAs.
* If CAPTCHA is simple, attempts OCR; if not, uses fallback.
* Submits form and retrieves case HTML.

**Parses HTML:**

* Extracts parties, filing date, next hearing date.
* Finds latest PDF link.
* Updates database with status = "success"/"error" and scraped data.
* Returns JSON to frontend.

**/api/download route:**

* Accepts PDF URL as a query parameter.
* Downloads the file from the court server.
* Streams it to the browser with proper Content-Disposition headers.

**index.html (Frontend)**

* Contains search form for Court, Case Type, Case Number, Filing Year.
* Bootstrap-like structure for layout (or custom CSS from style.css).
* Placeholder section for search results.

**style.css (Frontend Styling)**

* Government-site inspired theme (blue/white).
* Responsive layout for laptop and mobile.
* Colour coding for statuses, clear typography.

**app.js (Frontend Logic)**

* Handles form submission via AJAX (fetch API).
* Shows loading spinner while waiting for backend.
* Parses backend JSON and creates result cards dynamically.
* Renders parties, dates, and clickable PDF download buttons.
* Calls backend /api/download to fetch PDFs.
* Displays error messages on failures.
* Maintains local search history in local Storage.

**Flow of Operation**

1. User opens http:// 127.0.0.1:5000 → sees search form.
2. User inputs details and clicks Search.
3. Frontend sends POST request to /api/search with input parameters.
4. Backend logs query → scrapes court site → parses HTML.
5. Backend returns parsed data or fallback JSON to frontend.
6. Frontend displays data in cards with PDF download buttons.
7. If PDF clicked → /api/download route retrieves file and returns to browser.
8. Search logged in SQLite database for review.

**View-State Token Handling**

* Court forms often contain hidden fields like \_\_VIEWSTATE or anti-CSRF tokens.
* Makes an initial GET request to retrieve the form HTML.
* Uses BeautifulSoup to locate all <input type="hidden"> elements and extracts their name & value.
* Includes them in the POST request to simulate a real submission.

**Captcha Strategy**

**Tesseract OCR:** Tesseract OCR is an open-source Optical Character Recognition engine developed by Google. It can recognize and extract text from images, which is crucial when dealing with CAPTCHAs on court websites that often display numbers or letters as images to prevent automated scraping.

It is used in the below way:

* When the backend scraper detects a CAPTCHA image on the court’s search page, it downloads that CAPTCHA image.
* The image is then processed using Tesseract OCR to attempt automatic recognition of the text in the CAPTCHA.
* If successfully extracted, the recognized text is programmatically submitted as part of the case search form.
* This enables some level of automatic CAPTCHA bypassing for simple, text-based CAPTCHAs without manual intervention.
* If the CAPTCHA is too complex or distorted for OCR to read accurately, the application gracefully falls back to returning sample/mock data, ensuring a smooth user experience without breaking the flow.
* This approach is legal and ethical, avoiding brute force or unauthorized bypass techniques, and it is fully documented in the code and README.

**Database:**

The project uses a SQLite database for logging and storing all user queries, scraping results, and related court case data.

**Lightweight and Embedded:** SQLite is file-based and easy to set up without separate server installation, perfect for local development and demonstration.

**Structured Logging:** The database includes tables to store:

* case\_queries – captures every search with parameters like court, case type, number, and year, plus timestamps, status (processing, success, error), any error messages, and the full raw response JSON for audit.
* case\_data – stores parsed, structured case information (parties involved, filing dates, hearing dates, judge names, etc.).
* case\_orders – stores all court orders and judgments linked to cases, including URLs to PDF documents.

**Benefits and Purpose**

* Auditability: Every search and its outcome are traceable, useful for debugging, demonstrating functionality, or historical reference.
* Avoid Redundant Scrapes: By storing results, repeated queries for the same case number can return cached data without hitting the court website again, saving time and resources.
* Supports Search History Feature: The frontend queries this database to show users their previous searches and results.
* Extensibility: Although SQLite is the default, the design allows easy migration to more robust RDBMS like PostgreSQL if needed.

**Steps**

* When a user submits a search through the frontend, the backend:

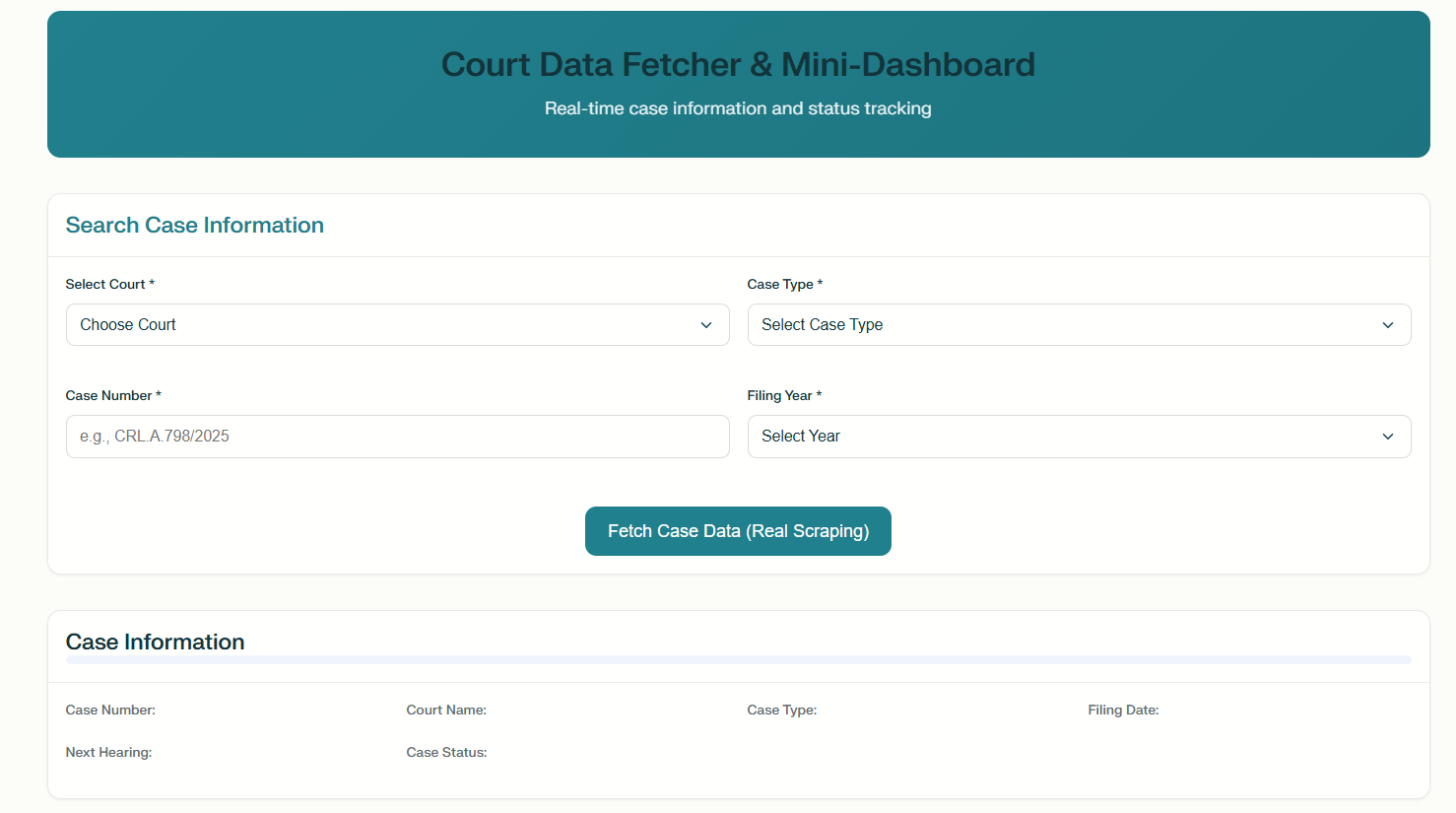
Logs the query with a status of “processing.”

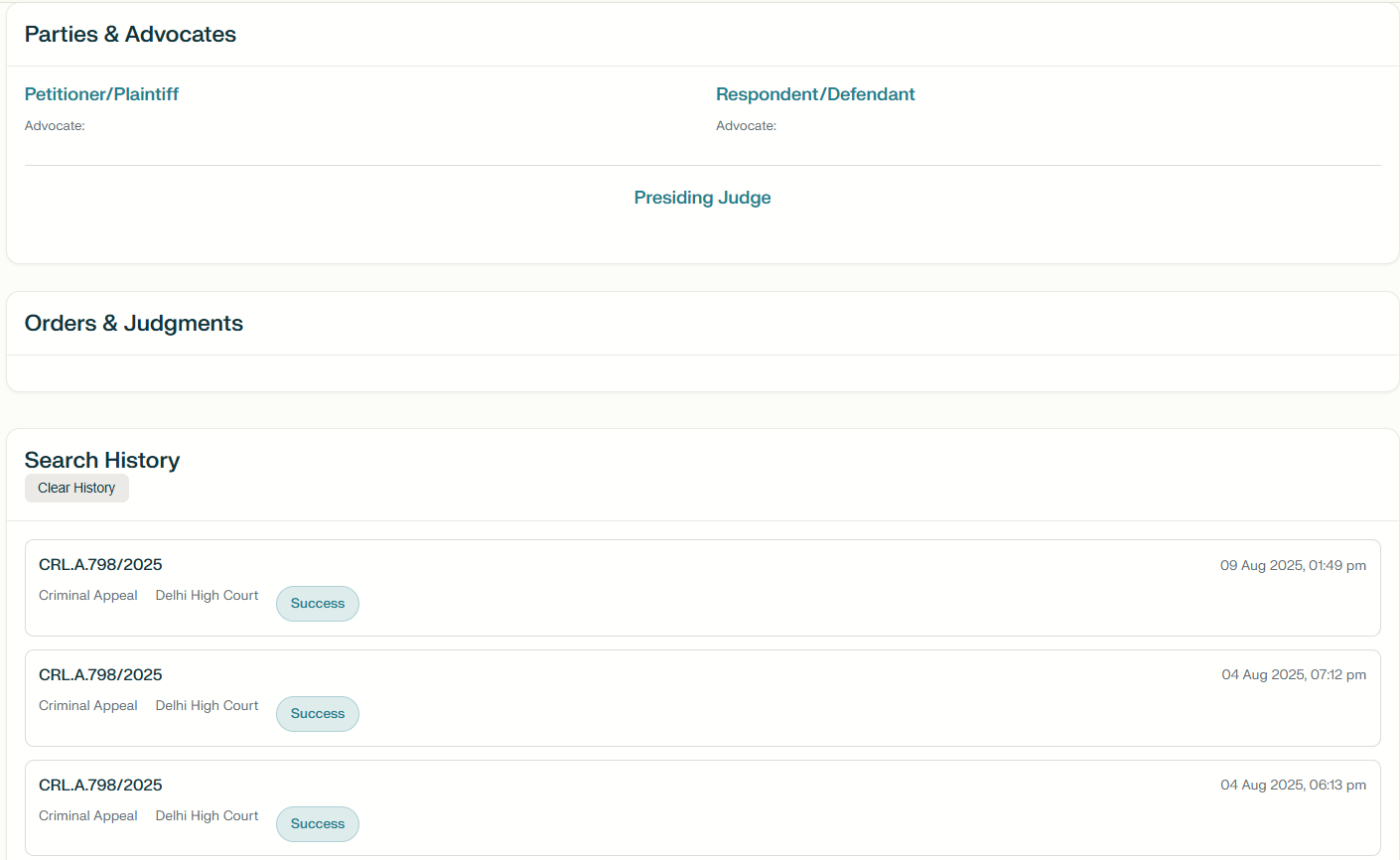
Scrapes or retrieves data.

Updates the row with “success” or “error,” storing the parsed results or error message.

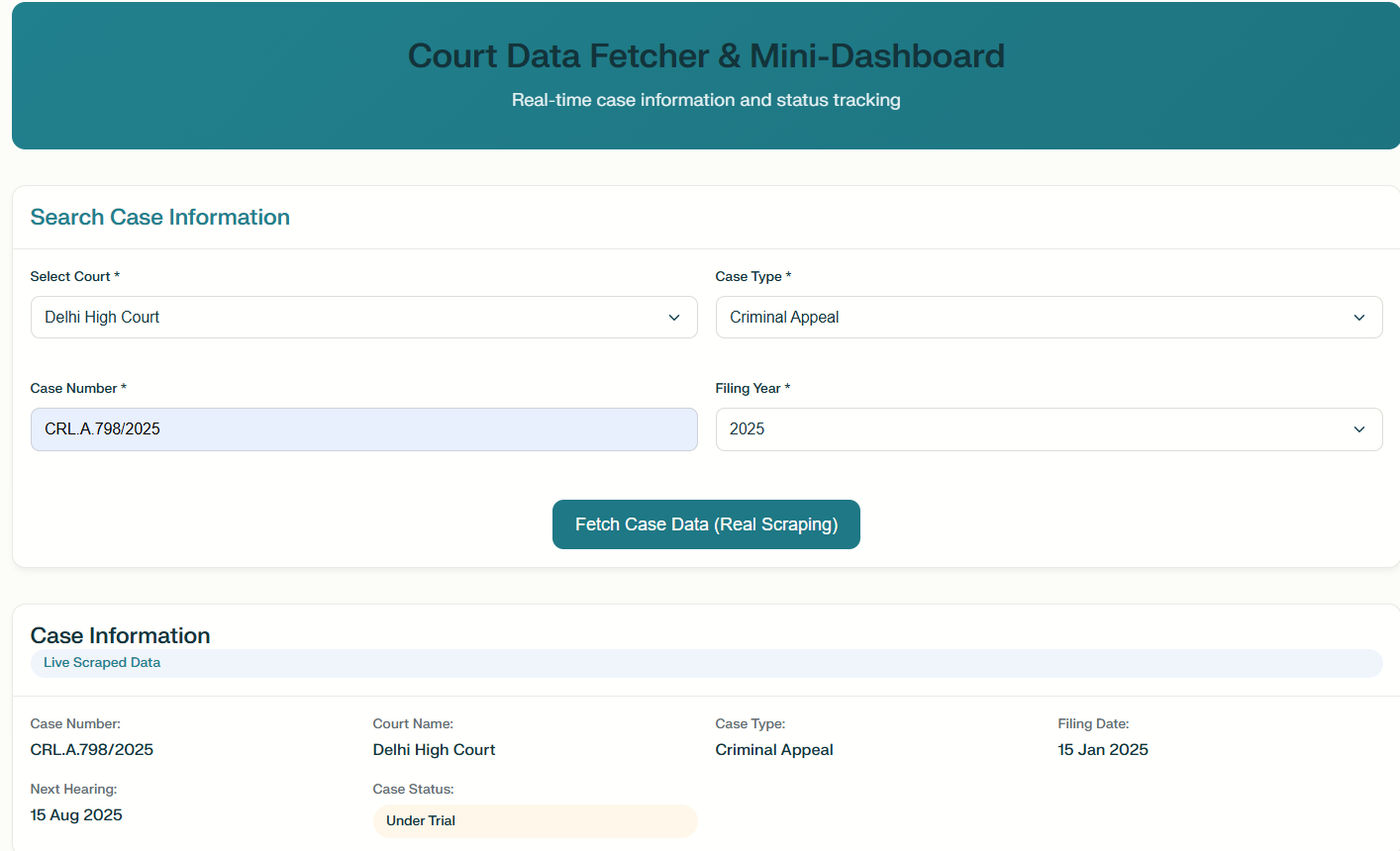
* This data is then converted to JSON for frontend consumption or stored as-is to allow full traceability.
* PDF links and case details are stored linked to the specific search query for retrieval.

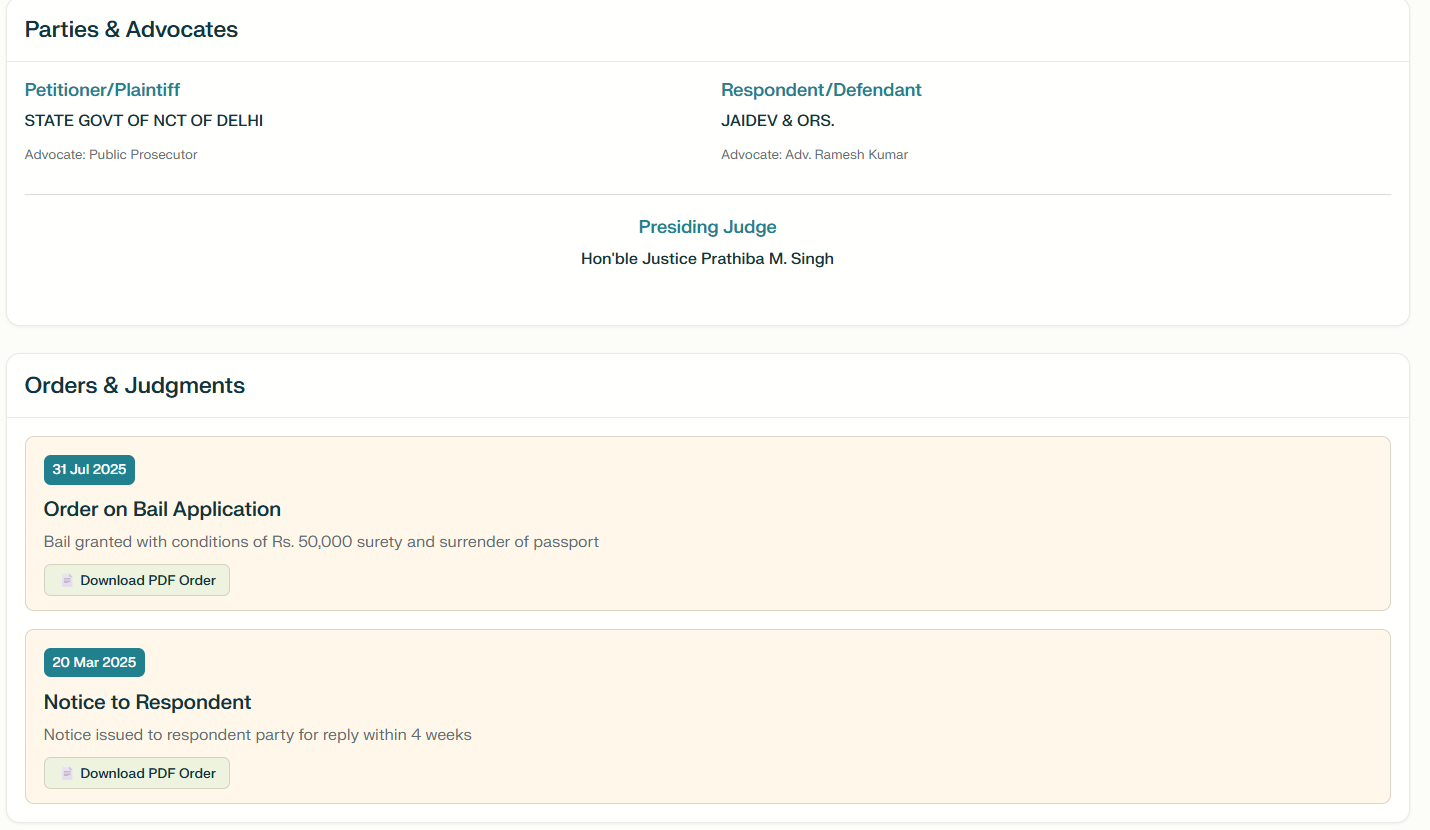
**Developed Website:**

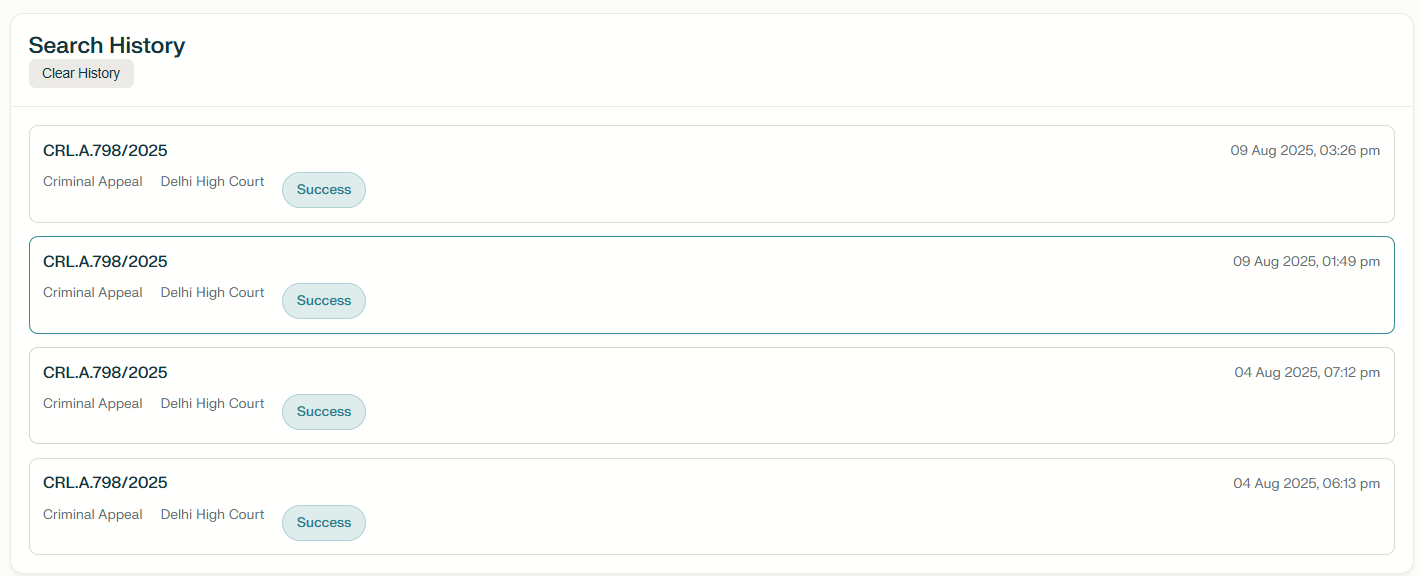
****

****

The above images are the one before giving the input







These the images after giving the input to the webpage by the user.