1. Project Overview and Workflow Explanation

This project is a legal document extractor that uses a Flutter mobile application as the frontend and a FastAPI server as the backend. The core functionality is to take a legal document (specifically a land deed) as input, either by uploading a PDF or scanning it with the camera, and then extract key information from it.

Here is the complete workflow:

1. **User Interaction (Flutter App):** The user interacts with the ExtractPage in the Flutter app. They have two main options:
   - **"Pick PDF"**: The user selects a PDF file from their device's storage.
   - **"Scan & Extract"**: The user opens the device's camera, takes a picture of a physical document, and the app converts this image into a temporary PDF file.
2. **Frontend Processing (Flutter App):**
   - Once a file (either a picked PDF or a scanned PDF) is selected, the app stores a reference to it in the selectedFile variable.
   - The user then clicks the **"Extract Details"** button (or the extraction is triggered automatically after a scan).
   - The extractData() function is called, which sets the isLoading state to true to show a progress indicator.
3. **API Call (Flutter App to FastAPI Server):**
   - The ExtractService.uploadPDF() function is called.
   - This function constructs an http.MultipartRequest, which is the standard way to upload files in an HTTP request.
   - It sends the file to the backend server's /extract endpoint.
4. **Backend Processing (FastAPI Server):**
   - The FastAPI server receives the file at the /extract endpoint.
   - The main.py script saves the incoming file to a temporary uploads directory.
   - It then calls the groq_extractor.py module to handle the core extraction logic.
   - groq_extractor.extract_text() is called first. This function attempts to extract text from the PDF using PyMuPDF. If that fails (e.g., for a scanned image), it falls back to Optical Character Recognition (OCR) using pytesseract to get the text.
   - The extracted text is then passed to groq_extractor.extract_deed_info().
5. **AI Extraction (Groq API):**
   - extract_deed_info() constructs a detailed prompt for the Groq API. The prompt is a crucial part of the application, as it instructs the large language model (LLM) to perform one of two tasks:

        - If the document is a land deed, extract a specific set of fields (Deed Type, Parties, Survey Number, etc.) and return a strict JSON object.
        - If it's not a land deed, identify the document type, provide a summary, and extract any key information, also in a JSON format.
    - The function sends this prompt to the Groq API using the llama3-70b-8192 model.
    - The API's JSON response is received and validated using a regular expression to ensure a valid JSON object is returned.
6. **Backend Response & Cleanup:**
    - The validated JSON object is sent back as the response to the Flutter app.
    - The FastAPI server's finally block ensures that the temporary uploaded file is deleted, regardless of whether the extraction was successful or not.
7. **Displaying Results (Flutter App):**
    - The Flutter app's extractData() function receives the JSON response.
    - It updates the result state variable with the new data.
    - isLoading is set back to false.
    - The buildExtractedData() widget rebuilds the UI based on the result data. It intelligently checks for the presence of land deed-specific fields (Deed Type, Survey Number, etc.) to determine which type of information to display.
    - A beautifully formatted card is shown with either the land deed details and a generated summary or the general document summary and key information.

This architecture creates a powerful, scalable system where the Flutter app provides a great user experience and the FastAPI backend handles the heavy lifting of document processing and AI interaction.

---

2. Flutter and its Uses/Working Flow

What is Flutter?

Flutter is an open-source UI software development kit created by Google. It is used to build natively compiled, multi-platform applications from a single codebase. This means you can write one set of code and deploy it to iOS, Android, web, desktop (Windows, macOS, Linux), and embedded devices.

Key Features of Flutter:

- **Single Codebase:** Build applications for multiple platforms with a single codebase, drastically reducing development time and effort.
- **"Everything is a Widget":** Flutter's UI is composed of "widgets." Widgets are the basic building blocks of a Flutter app's UI. They can describe anything from a button or text field to a layout or an animation.

- **Hot Reload:** This feature allows developers to instantly see changes made to the code reflected in the running application without losing the app's state. This speeds up the development process significantly.
- **High Performance:** Flutter applications are compiled directly to native code (ARM for mobile, x64 for desktop), which eliminates the performance overhead of a bridge, as seen in some other cross-platform frameworks.
- **Rich Widget Library:** Flutter provides a rich set of pre-built, customizable widgets that follow the Material Design (Google) and Cupertino (Apple) guidelines, making it easy to create beautiful, platform-appropriate UIs.
- **Dart Programming Language:** Flutter uses the Dart language, which is an object-oriented, client-optimized language developed by Google.

How Flutter Works (Working Flow):

1. **Widgets:** The entire UI is built as a tree of widgets. A widget is a blueprint for a part of the UI.
2. **StatefulWidget vs. StatelessWidget:**
   - A StatelessWidget is a widget that doesn't change over time (e.g., a static text label).
   - A StatefulWidget can change its appearance in response to user interaction or data changes (e.g., a checkbox, a form input). It holds a State object that manages the dynamic data.
3. **Rendering Engine (Skia):** Flutter uses its own high-performance rendering engine called Skia. This means Flutter doesn't rely on the platform's native UI components. Instead, it draws every pixel of the UI itself, which guarantees a consistent look and feel across platforms.
4. **The setState() method:** When a StatefulWidget's data changes, the developer calls setState(). This tells the Flutter framework that the widget's state has changed and that it should be rebuilt.
5. **Rebuilding the Widget Tree:** The framework rebuilds the widget tree, compares the new tree with the old one, and efficiently updates only the parts of the UI that have changed, a process similar to a "Virtual DOM" in web frameworks like React.

In summary, Flutter's core strength lies in its "everything is a widget" philosophy, high-performance rendering, and developer-friendly features like hot reload. It is an excellent choice for building fast, beautiful, and consistent applications for a wide range of platforms from a single codebase.

---

3. Line-by-Line Code Explanation

main.py (FastAPI Backend)

Python

```
from fastapi import FastAPI, UploadFile, File, HTTPException
from fastapi.middleware.cors import CORSMiddleware
from groq_extractor import extract_deed_info, extract_text
import os
import json # Import json for cleaner error messages

# 1. Initialize the FastAPI application
app = FastAPI()

# 2. Configure CORS middleware to allow cross-origin requests from any source.
# This is essential for the Flutter app to communicate with the backend during development.
app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"],
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)

# 3. Define the directory for temporary file uploads.
UPLOAD_DIR = "uploads"
os.makedirs(UPLOAD_DIR, exist_ok=True) # Creates the directory if it doesn't exist.

# 4. Define the API endpoint to handle file uploads and extraction.
# This is a POST endpoint at the path "/extract".
# It expects a file to be uploaded.
@app.post("/extract")
async def extract(file: UploadFile = File(...)):
    print(f"Received file: {file.filename}") # Log the name of the file received.
    file_path = os.path.join(UPLOAD_DIR, file.filename) # Create the full path to save the file.

    try:
        # 5. Save the uploaded file to the server's local storage.
        with open(file_path, "wb") as f:
            f.write(await file.read())
        print(f"File saved to: {file_path}")

        # 6. Extract raw text from the saved PDF file.
        with open(file_path, "rb") as f:
            raw_text = extract_text(f) # Calls the function from groq_extractor.py.
        print(f"Raw text extracted (first 500 chars):\n{raw_text[:500]}...")

        # 7. Handle the case where no text could be extracted.
        if not raw_text.strip():
            print("Warning: Extracted text is empty or just whitespace.")
            raise HTTPException(status_code=400, detail="Could not extract any meaningful text from
the document.")

        # 8. Call the Groq-powered extraction function.
        extracted = extract_deed_info(raw_text)
        print(f"Extraction successful: {json.dumps(extracted, indent=2)}")
        return {"Details": extracted} # Return the extracted data as a JSON object.

    except Exception as e:
        print(f"An error occurred during extraction: {e}")
        # 9. If any error occurs, return a 500 Internal Server Error.
        raise HTTPException(status_code=500, detail=f"Failed to extract deed info: {e}")
    finally:
        # 10. This block always executes, ensuring the temporary file is deleted.
        if os.path.exists(file_path):
            os.remove(file_path)
            print(f"Cleaned up file: {file_path}")
```

groq_extractor.py (Backend Logic)

Python

```
import fitz  # PyMuPDF, for PDF text/image extraction
import pytesseract # For OCR on images
```

```python
from PIL import Image # Pillow, for image manipulation
import io # To work with bytes streams
import os # For file system operations
from openai import OpenAI # Groq API client (Groq uses the OpenAI API format)
from dotenv import load_dotenv # To load environment variables (API key)
import json # For JSON parsing
import re # For regular expressions

# 1. Load environment variables from a .env file.
load_dotenv()
# 2. Initialize the Groq API client with the API key and base URL.
client = OpenAI(api_key=os.getenv("GROQ_API_KEY"), base_url="https://api.groq.com/openai/v1")

# 3. Configure the path to the Tesseract executable. This is crucial for OCR.
pytesseract.pytesseract.tesseract_cmd = r'C:\Program Files\Tesseract-OCR\tesseract.exe'

# 4. Function to extract text from a PDF file.
def extract_text(file):
    doc = fitz.open(stream=file.read(), filetype="pdf") # Open the PDF using PyMuPDF.
    text = " ".join(page.get_text() for page in doc) # Extract text from all pages.
    print(f"PyMuPDF extracted text (first 500 chars):\n{text[:500]}...")

    if text.strip():
        return text # Return the text if PyMuPDF succeeded.
    else:
        print("PyMuPDF found no text. Attempting OCR with Tesseract...")
        ocr_text = ""
        for page_num, page in enumerate(doc):
            try:
                # 5. Get a high-resolution pixmap (image) of the page.
                pix = page.get_pixmap(dpi=300)
                # 6. Convert the pixmap to a Pillow Image object.
                image = Image.open(io.BytesIO(pix.tobytes("png")))
                # 7. Use pytesseract to perform OCR and get text from the image.
                page_ocr_text = pytesseract.image_to_string(image)
                ocr_text += page_ocr_text + "\n" # Append text from each page.
                print(f"OCR extracted for page {page_num + 1} (first 200 chars):\n{page_ocr_text[:200]}...")
            except Exception as e:
                print(f"Error during OCR for page {page_num + 1}: {e}")
                continue
        if not ocr_text.strip():
            print("OCR also failed to extract any text.")
        return ocr_text # Return the OCR text if available.
```

```
# 8. Function to use the Groq API for information extraction.
def extract_deed_info(cleaned_text):
  if not cleaned_text.strip():
    print("Error: Empty text provided to extract_deed_info.")
    return {"Error": "No text available for extraction."}

  # 9. Define the prompt for the Groq LLM. This is the core instruction.
  prompt = f"""
  You are a legal assistant. Extract the following information from this Indian land deed text and give
the output as a JSON object:
  ... (rest of the detailed prompt) ...
  Text:
  {cleaned_text}
  """
  print(f"Prompt sent to Groq (first 500 chars):\n{prompt[:500]}...")

  try:
    # 10. Make the API call to Groq with the prompt.
    response = client.chat.completions.create(
      model="llama3-70b-8192",
      messages=[{"role": "user", "content": prompt}],
      temperature=0 # Low temperature ensures a deterministic, factual response.
    )

    raw_output = response.choices[0].message.content # Get the text response from the API.
    print("GROQ RAW OUTPUT:\n", raw_output)

    # 11. Use a regular expression to find and extract the JSON part of the response.
    match = re.search(r"\{[\s\S]*\}", raw_output)
    if not match:
      print("Error: No JSON found in Groq response using regex.")
      # ... (error handling) ...
      raise Exception("No JSON found in Groq response")

    cleaned_json = match.group(0)
    print(f"Cleaned JSON string:\n{cleaned_json}")

    try:
      return json.loads(cleaned_json) # 12. Parse the JSON string into a Python dictionary.
    except json.JSONDecodeError as e:
      print(f"JSON Decode Error: {e} - Raw JSON string was: {cleaned_json}")
      raise Exception(f"Failed to extract deed info: Invalid JSON format from Groq. Error: {e}")
  except Exception as e:
    print(f"Error during Groq API call or response processing: {e}")
    raise Exception(f"Groq API interaction failed: {e}")
```

extract_page.dart (Flutter Frontend - UI and Logic)

Dart

```dart
import 'package:flutter/material.dart';
import 'dart:io'; // For working with files
import 'package:file_picker/file_picker.dart'; // To pick files from the device
import 'package:image_picker/image_picker.dart'; // To pick images from the camera/gallery
import 'package:path_provider/path_provider.dart'; // To get temporary directories
import 'package:pdf/widgets.dart' as pw; // For creating PDFs
import 'package:http/http.dart' as http; // For making HTTP requests
import 'package:clipboard/clipboard.dart'; // For copying text to the clipboard
import 'package:google_fonts/google_fonts.dart'; // For custom fonts
import 'package:flutter_animate/flutter_animate.dart'; // For animations
import 'package:permission_handler/permission_handler.dart'; // To request permissions
import 'package:share_plus/share_plus.dart'; // To share content
import 'package:video_player/video_player.dart'; // For video background
import 'dart:convert'; // To encode/decode JSON
import '../services/extract_service.dart'; // The API service class
import 'scan_page.dart'; // The page for scanning documents

// 1. A StatefulWidget to manage the state of the extraction page.
class ExtractPage extends StatefulWidget {
  const ExtractPage({super.key});

  @override
  State<ExtractPage> createState() => _ExtractPageState();
}

class _ExtractPageState extends State<ExtractPage> {
  // 2. State variables to hold data related to the page.
  File? selectedFile;
```

```
Map<String, dynamic>? result;
bool isLoading = false;
late VideoPlayerController _videoController;

@override
void initState() {
  super.initState();
  // 3. Initialize the video player for the background video.
  _videoController = VideoPlayerController.asset('assets/videos/background.mp4')
    ..initialize().then((_) {
      _videoController.setLooping(true);
      _videoController.setVolume(0.0);
      _videoController.play();
      setState(() {});
    });
  // 4. Request necessary permissions when the page is initialized.
  requestPermissions();
}

// 5. Function to request storage and camera permissions.
Future<void> requestPermissions() async {
  await [
    Permission.storage,
    Permission.camera,
    Permission.manageExternalStorage,
  ].request();
}

@override
void dispose() {
  _videoController.dispose();
  super.dispose();
}

// 6. Function to handle picking a PDF file.
Future<void> pickPDF() async {
  FilePickerResult? picked = await FilePicker.platform.pickFiles(
    type: FileType.custom,
    allowedExtensions: ['pdf'],
  );
  if (picked != null && picked.files.single.path != null) {
    setState(() {
      selectedFile = File(picked.files.single.path!); // Update the state with the selected file.
    result = null; // Clear old results.
   });
  }
 }

 // 7. Function to send the file to the backend for extraction.
 Future<void> extractData() async {
  if (selectedFile == null) {
   ScaffoldMessenger.of(context).showSnackBar(
    const SnackBar(content: Text("Please pick a PDF file first.")),
   );
   return;
  }
  setState(() => isLoading = true); // Start loading state.
  try {
   final response = await ExtractService.uploadPDF(selectedFile!); // Make the API call.
   setState(() => result = response); // Update the state with the API response.
  } catch (e) {
   ScaffoldMessenger.of(context).showSnackBar(
    SnackBar(content: Text("Failed to extract: ${e.toString()}")),
   );
  } finally {
   setState(() => isLoading = false); // End loading state.
  }
 }

 // 8. Function to handle scanning a document with the camera.
 Future<void> scanAndExtract() async {
  // 9. Check for camera permission.
  var cameraStatus = await Permission.camera.status;
  if (!cameraStatus.isGranted) {
   cameraStatus = await Permission.camera.request();
   if (!cameraStatus.isGranted) {
    // ... (error message) ...
    return;
   }
  }

  // 10. Navigate to the ScanPage and wait for a result.
  final File? scannedPdf = await Navigator.push(
   context,
   MaterialPageRoute(builder: (context) => const ScanPage()),
  );
  if (scannedPdf != null) {
    setState(() {
      selectedFile = scannedPdf; // Use the scanned PDF as the selected file.
      result = null;
    });
    extractData(); // Automatically start extraction.
  } else {
    // ... (cancellation message) ...
  }
}

// 11. UI helper function to display status messages.
Widget buildStatusMessage() {
  // ... (logic to show loading, file name, or initial message) ...
}

// 12. UI helper function to display the extracted data.
Widget buildExtractedData() {
  final details = result?['Details'];
  if (details == null || details.isEmpty) {
    // ... (error card) ...
  }

  // 13. Check if the document is a land deed by looking for specific keys.
  bool isLandDeed = details.containsKey("Deed Type") &&
      details.containsKey("Party 1") &&
      details.containsKey("Survey Number");

  if (isLandDeed) {
    // 14. Build the UI for a land deed.
    String summaryText =
        "This ${details["Deed Type"]} dated ${details["Date of Execution"]} executed by ...";
    // ... (code for showing a summary, a table of details, and share/copy buttons) ...
  } else {
    // 15. Build the UI for a general document.
    String documentType = details["Document Type"] ?? "Unknown Document";
    // ... (code for showing document type, summary, and key information) ...
  }
  return Card(...);
}

@override
```

```
Widget build(BuildContext context) {
  // 16. Build the main UI of the page.
  return Scaffold(
    appBar: AppBar(...),
    body: Stack(
      children: [
        // 17. The video background.
        if (_videoController.value.isInitialized)
          Positioned.fill(
            child: Opacity(
              opacity: 0.24,
              child: FittedBox(...),
            ),
          ),
        SingleChildScrollView(
          child: Center(
            child: Column(
              children: [
                const SizedBox(height: 40),
                // 18. The three main buttons: Pick PDF, Extract, Scan & Extract.
                ElevatedButton.icon(
                  onPressed: pickPDF,
                  icon: const Icon(Icons.upload_file_rounded),
                  label: Text("Pick PDF", ...),
                  style: getButtonStyle(),
                ),
                const SizedBox(height: 12),
                ElevatedButton.icon(
                  onPressed: extractData,
                  icon: const Icon(Icons.auto_fix_high_rounded),
                  label: Text("Extract Details", ...),
                  style: getButtonStyle(),
                ),
                const SizedBox(height: 12),
                ElevatedButton.icon(
                  onPressed: scanAndExtract,
                  icon: const Icon(Icons.camera_alt_outlined),
                  label: Text("Scan & Extract", ...),
                  style: getButtonStyle(),
                ),
                const SizedBox(height: 30),
                Text("Scan. Extract. Understand.", ...).animate(...), // 19. Animated text.
                buildStatusMessage(), // 20. Call the status message widget.
```

```
                  const SizedBox(height: 20),
                  if (result != null && !isLoading) buildExtractedData(), // 21. Show results if available.
                  const SizedBox(height: 40),
                ],
              ),
            ),
          ),
        ],
      ),
    );
  }
}
```

scan_page.dart (Flutter Frontend - Scan Logic)

Dart

```dart
import 'dart:io';
import 'package:flutter/material.dart';
import 'package:image_picker/image_picker.dart'; // To access the camera
import 'package:pdf/widgets.dart' as pw; // For creating PDFs
import 'package:path_provider/path_provider.dart'; // To access app directories
import 'package:permission_handler/permission_handler.dart'; // To request permissions
import 'package:open_filex/open_filex.dart'; // To open a file after saving

class ScanPage extends StatefulWidget {
  const ScanPage({super.key});

  @override
  State<ScanPage> createState() => _ScanPageState();
}

class _ScanPageState extends State<ScanPage> {
  bool _isLoading = false;
  String _statusMessage = "Opening camera...";
```

```dart
@override
void initState() {
  super.initState();
  // 1. Start the scan process as soon as the page is built.
  WidgetsBinding.instance.addPostFrameCallback((_) {
    _startScanProcess();
  });
}

Future<void> _startScanProcess() async {
  File? pdfFile;
  try {
    // 2. Pick an image from the camera.
    final pickedImage = await ImagePicker().pickImage(source: ImageSource.camera);

    if (pickedImage == null) {
      // 3. Handle user cancelling the camera operation.
      print("Camera operation cancelled by user.");
      if (mounted) Navigator.pop(context, null); // Return null to the previous page.
      return;
    }

    if (!mounted) return;
    setState(() {
      _isLoading = true;
      _statusMessage = "Converting image to PDF...";
    });

    final imageFile = File(pickedImage.path);
    // 4. Convert the captured image to a PDF.
    pdfFile = await convertImageToPdf(imageFile);

    if (!mounted) return;
    setState(() {
      _isLoading = false;
      _statusMessage = "PDF conversion complete.";
    });

    // 5. Show a snackbar with an option to save and open the PDF.
    if (pdfFile != null) {
      if (mounted) {
        ScaffoldMessenger.of(context).showSnackBar(
          SnackBar(
          content: Text("PDF created: ${pdfFile.path.split('/').last}"),
          action: SnackBarAction(
            label: 'SAVE & OPEN',
            onPressed: () async {
              await saveAndOpenPdf(pdfFile!);
            },
          ),
          duration: const Duration(seconds: 5),
        ),
      );
    }
  }

  } catch (e) {
    // 6. Handle any errors during the process.
    if (mounted) {
      setState(() {
        _isLoading = false;
        _statusMessage = "Error during scan or conversion: ${e.toString()}";
      });
    }
    print("Error in _startScanProcess: $e");
  } finally {
    if (mounted) {
      await Future.delayed(const Duration(milliseconds: 500));
      Navigator.pop(context, pdfFile); // 7. Return the temporary PDF file or null.
    }
  }
}

Future<File?> convertImageToPdf(File imageFile) async {
  try {
    final pdf = pw.Document();
    final imageBytes = await imageFile.readAsBytes();
    final image = pw.MemoryImage(imageBytes);

    // 8. Add a page to the PDF document with the image.
    pdf.addPage(
      pw.Page(
        build: (pw.Context context) => pw.Center(child: pw.Image(image)),
      ),
    );
```

```
      final dir = await getTemporaryDirectory();
      final pdfPath = '${dir.path}/scanned_doc_${DateTime.now().millisecondsSinceEpoch}.pdf';
      final pdfFile = File(pdfPath);
      // 9. Save the PDF to a temporary file.
      await pdfFile.writeAsBytes(await pdf.save());
      print("PDF saved temporarily to: ${pdfFile.path}");
      return pdfFile;
    } catch (e) {
      print("Error converting image to PDF: $e");
      return null;
    }
  }

  // 10. Function to save the PDF to a public directory and open it.
  Future<void> saveAndOpenPdf(File pdfFile) async {
    // 11. Request storage permissions.
    var status = await Permission.storage.request();
    if (!status.isGranted) {
      status = await Permission.manageExternalStorage.request();
      if (!status.isGranted) {
        // ... (error message) ...
        return;
      }
    }
    // 12. Copy the temporary PDF to a public Downloads directory.
    try {
      final String? externalDir = (await getExternalStorageDirectory())?.path;
      String outputPath =
'${externalDir}/Download/scanned_land_deed_${DateTime.now().millisecondsSinceEpoch}.pdf';
      final directory = Directory(File(outputPath).parent.path);
      if (!await directory.exists()) {
        await directory.create(recursive: true);
      }
      final File savedFile = await pdfFile.copy(outputPath);
      print("PDF saved to public directory: ${savedFile.path}");
      // 13. Open the file using the OpenFilex package.
      OpenFilex.open(savedFile.path);
    } catch (e) {
      print("Error saving and opening PDF: $e");
    }
  }

  @override
```

```
 Widget build(BuildContext context) {
   // 14. The UI for the scan page, showing a progress indicator and status message.
   return Scaffold(
     appBar: AppBar(...),
     body: Center(
       child: Column(
         mainAxisAlignment: MainAxisAlignment.center,
         children: [
           _isLoading
               ? const CircularProgressIndicator()
               : const Icon(Icons.camera_alt, ...),
           const SizedBox(height: 20),
           Padding(
             padding: const EdgeInsets.symmetric(horizontal: 20.0),
             child: Text(
               _statusMessage,
               textAlign: TextAlign.center,
               style: TextStyle(fontSize: 16, color: Colors.grey.shade700),
             ),
           ),
         ],
       ),
     ),
   );
 }
}
```

extract_service.dart (Flutter Frontend - API Service)

Dart

```dart
import 'dart:convert';
import 'dart:io';
import 'package:http/http.dart' as http; // The HTTP client library

class ExtractService {
 // 1. The base URL of the FastAPI backend.
 static const String _baseUrl = "http://192.168.0.105:8000";

 // 2. The main function to upload a PDF file.
 static Future<Map<String, dynamic>> uploadPDF(File file) async {
   var uri = Uri.parse("$_baseUrl/extract"); // Construct the full URL.
   // 3. Create a multipart request, which is suitable for file uploads.
   var request = http.MultipartRequest('POST', uri);
   // 4. Add the file to the request with the field name 'file'.
   request.files.add(await http.MultipartFile.fromPath('file', file.path));

   var response = await request.send(); // Send the request.
   final body = await response.stream.bytesToString(); // Read the response body.

   if (response.statusCode == 200) {
     final decoded = jsonDecode(body); // 5. Decode the JSON response.
     return decoded;
   } else {
     print("Error ${response.statusCode}: $body");
     // 6. Throw an exception if the server returns a non-200 status code.
     throw Exception("Failed to extract deed info. Server responded with ${response.statusCode}");
   }
 }

 // 7. A helper function to parse the JSON response.
 static Map<String, dynamic> parseExtractedDetails(String responseBody) {
   try {
     return jsonDecode(responseBody);
   } catch (e) {
     print("JSON decode error: $e");
     return {
       "Details": {
         "Error": "Invalid response format or empty response."
       }
     };
   }
 }
}
```

This comprehensive breakdown should provide you with all the information you need to understand and explain your project, from the high-level architecture to the specific function of each line of code.