

Hector Gonzalez
Sonya Cates
Data Science
04/01/2025

Project Title: Fraud Detection via API Integration Using Flask & React

Paragraph Summary

This project involved building a full-stack fraud detection web application that integrates the FraudLabs Pro API to screen and validate online transactions. The backend was built using Python with Flask, and the frontend was built using React.js. The application allows users to submit transaction data such as billing details, IP address, and card information, which is then sent to FraudLabs Pro for fraud risk analysis. This project simulates a real-world implementation of payment risk evaluation, with a focus on correct API communication and robust error handling.

Purpose and Personal Relevance

This project was personally important to me because I am in the process of creating a full-stack web application for my father's truck mechanic and tire distribution business. The business has a high potential to grow digitally as most of his clients currently schedule work over the phone. Digitizing scheduling, inventory, and service logs through a web platform will streamline operations, improve scheduling, and ultimately scale the business. I also want to understand how credit and debit card transaction systems work so I can incorporate a secure online payment portal into the app. This would allow clients to pay for services and products (like tires) directly through the platform, creating convenience for customers and more structure for the business. Learning to handle APIs, form validation, and real-time backend communication directly aligns with features I will need to build for his company's logistics, order management, and client communication.

Process and Methodology

The project began with setting up a basic Flask server to receive POST requests from the React frontend. The React form collected all necessary billing and transaction data. Once submitted, this data was sent via POST to the backend, which would then forward the information to FraudLabs Pro using an HTTP GET request. Initially, the app used the **v1/order/screen** endpoint, which checks transactions but does not save them to the merchant dashboard. I revised the code to use

`v1/order/validate`, which validates and logs transactions. I also implemented response parsing for both JSON and XML formats to prevent backend crashes. This was a frequent issue that included a lot of going back to the old code and starting there, luckily I didn't continuously post on github so my history was left in the last current application of this app that was working.

To ensure the API key was functioning properly and that the API itself was responsive to requests, I used Postman to test the GET, POST, and parameter handling processes outside of the full application. Postman allowed me to isolate issues in the request formatting, inspect raw responses, and confirm that my API key was valid before moving those tests into the React-Flask pipeline.

- **Backend Setup:** Flask server with endpoint `/check-fraud` to receive POST data.
- **Frontend:** React form with fields for billing info, IP, currency, and card BIN.
- **API Integration:** Used `requests.get()` to send payload to FraudLabs Pro.
- **Validation:** Ensured required fields like `bill_email` and `amount` were included
- **API Testing:** Used Postman to confirm the API and key worked as expected before final integration.

Key Issues Encountered

The most persistent issue was a `500 Internal Server Error` that occurred during POST operations. This was caused by multiple factors:

- Empty or missing fields in the POST body, especially `bill_email` or `amount`
- FraudLabs rejecting invalid country codes like "United States" instead of "US"
- The use of the `screen` endpoint instead of `validate`, which led to the data not showing up on the merchant dashboard
- API returning XML instead of JSON when errors occurred, which initially broke the `.json()` call

To resolve these, I:


- Added backend validation for required fields

- Defaulted `bill_country` to "US" and uppercased any input
- Switched the endpoint to `/v1/order/validate`
- Implemented conditional logic to parse either JSON or XML responses
- Used Postman to manually test different edge cases to ensure consistent API behavior

Results and Discussion

Once the backend was stabilized, the application returned accurate fraud analysis results including `fraudlabspro_score`, `fraudlabspro_status`, and `fraudlabspro_message`. These results are now visible in the merchant dashboard after each transaction. This validated that the data was correctly formatted and the API was properly integrated. A surprising discovery was how silently the API could fail if even one field was incorrect — highlighting the importance of defensive programming.

 FraudChecker-FrontEnd-Result.pdf

 TheAPIResult-Transaction-Dashboard.pdf

What I Learned

This project taught me how to integrate third-party APIs with full-stack applications, handle dynamic response types, and design for real-world reliability. More importantly, it helped me identify how critical clean API communication is when scaling a business solution — something I plan to use as I build out scheduling, ordering, inventory systems, and online payment functionality for my father's mechanic and distribution business. This assignment helped me grow not just as a developer, but as someone applying technical skills to real, tangible needs.

Citations:

- FraudLabs Pro API Documentation: <https://www.fraudlabspro.com/developer/api>
- Flask Documentation: <https://flask.palletsprojects.com/>
- React Documentation: <https://react.dev/>
- Python XML Parsing: <https://docs.python.org/3/library/xml.etree.elementtree.html>
- StackOverflow discussions on Flask error handling and `requests.get` usage ** This was a consistent URL used **