CardioCare AI

Track A — Web & Software Development

Reading pack. Nothing to submit. Come with one question.

Python FastAPI · React TypeScript · PostgreSQL · Stripe

See you next week. — Dr Mahdi Torabi | Torabi Signals Ltd

What is CardioCare AI?

It's a web platform that analyses heart recordings. Someone uploads an ECG file, the backend processes it, and a report comes back saying what was found. Think of it like sending a document to a service that reads it and sends a summary back.

There are three parts to the system. The Python backend does all the analysis. The user website is what patients and doctors see. The admin panel is for managing the platform internally. Your work as a Track A student will touch all three of these.

The backend

The backend is built with FastAPI, a Python framework for building APIs. An API is just a set of URL endpoints that accept requests and return responses, usually in JSON format.

Here's a real endpoint from the codebase. When the frontend sends a POST request to /analyses, this runs:

```
@router.post('/analyses')
async def create_analysis(request: AnalysisRequest, user =
Depends(get_current_user)):
    analysis_id = await orchestrator.start_analysis(request.recording_id)
    return {'analysis_id': analysis_id, 'status': 'processing'}
```

Three things worth noticing. The async keyword means the server doesn't freeze while it waits for the analysis to finish. The Depends(get_current_user) part automatically checks the user's login token before the function runs. And the function just returns a plain Python dictionary, which FastAPI converts to JSON automatically.

Authentication

Users log in with email and password, or with Google. After login, the server gives them a JWT token. This is a long string that proves who they are. Every API request after that includes this token in the headers, and the server checks it before doing anything.

If the token is missing, the server returns a 401 error. If the token is expired, same thing. The frontend has to handle this gracefully, usually by redirecting the user back to the login page.

Real-time progress with WebSockets

Normal HTTP works like this: you ask, server answers, connection closes. But the 9-stage analysis takes several seconds. We don't want the user just staring at a blank screen.

So instead, the frontend opens a WebSocket connection after upload. This is a live, persistent connection. The backend pushes a message every time a stage completes, and the browser updates the progress bar instantly.

```
ws.onmessage = (event) => {
  const msg = JSON.parse(event.data);
  if (msg.type === 'progress') setStage(msg.stage);
  if (msg.type === 'completed') fetchResults(msg.analysis_id);
}
```

If the connection drops (which happens), the frontend retries automatically, waiting a bit longer each time: 1 second, then 2, then 4, up to 30 seconds maximum.

Subscriptions and the Database

The platform has three plans. Free users get 3 analyses total. Pro users get unlimited. Enterprise is a custom arrangement. Stripe handles the actual payments.

When someone pays, Stripe sends a webhook to the backend. A webhook is just an automatic HTTP POST from Stripe to our server, telling us something happened. The backend receives it, checks it's genuine, and updates the user's record in the database.

> There's a live bug here worth knowing about.
>
> Some parts of the code still use an old field called subscription_tier.
> The correct field is plan. When these mix, users can end up on the wrong access level without any error message. Fixing this is one of your first tasks.

The database

All data lives in PostgreSQL, hosted on Supabase. The two tables that matter most for your work are:

- analyses — stores the result of every ECG analysis, including detected arrhythmias, heart rate, and the report file path
- users / subscriptions — stores account info, plan status, and Stripe customer IDs

The backend uses SQLAlchemy to talk to the database. You write Python objects, and SQLAlchemy handles the SQL queries for you. You won't need to write raw SQL, but it helps to understand what's happening underneath.

What You'll Actually Be Working On

You won't be building from scratch. The codebase already exists. Most tasks are fixes, small features, or improvements to things that are already there.

- The React component that handles file upload and shows the progress bar
- The API client in src/services/api.ts that talks to the backend
- The subscription bug — replacing subscription_tier with plan across several files
- Possibly the Stripe webhook handler, which needs to handle edge cases gracefully

The tool for checking your work is simple. Run diagnose_af.py, it processes a test recording, and prints whether everything worked. If something is broken, it tells you where.

Before Next Week

Have a look at these. 5 to 10 minutes each is plenty:

- What FastAPI is — the homepage has a good 30-second explanation
- What a JWT token is — jwt.io has a short visual explainer
- What WebSockets do differently from normal HTTP

> One question to think about:
>
> If a user's token expires halfway through a 9-stage analysis,
> what should the frontend do? No right answer. Just have a rough thought ready.

See you next week. — Dr Mahdi Torabi | Torabi Signals Ltd