Dhanush — Backend Engineering Roadmap

5 Weeks · 6 Days/Week · 10 Hours/Day · Python + FastAPI

The Project: "AccessAI"

One product, built across 5 weeks. AccessAI is a backend platform where users sign in with Google, get free credits, buy more via Stripe, and spend those credits to call AI-powered endpoints. By Week 5 it's live on the internet, auto-deployed, and monitored.

Nothing gets thrown away. Every line written in Week 1 is still running in Week 5. He's not doing exercises — he's building a real system, one layer at a time.

Week 1 — Secure Foundation + Google Login

6 days · Goal: Running server, real database, users can sign in with Google

What he's learning

How to set up a Python backend properly. How to never put secrets in code. How to make Google handle authentication so he doesn't have to build a login form. How to prove who a user is on every request using a JWT token.

The analogy

OAuth is like a hotel key card system. Google is the front desk. They verify your identity and give you a key card (JWT token). Every door (protected endpoint) checks the key card — it doesn't ask Google again every time.

Day 1 — Project skeleton + database

Using Gemini, he scaffolds a FastAPI project with this folder structure:

```None
accessai/
  main.py
  config.py
  models/
  routes/
  services/
  database.py
```

He creates a PostgreSQL database and a Users table with: id, email, name, google_id, created_at. He writes one endpoint: GET /health → returns `{"status": "ok"}`.

Gemini prompt to start him off: *"Scaffold a FastAPI project with PostgreSQL using SQLAlchemy async. Create a Users table with id (UUID), email, name, google_id, and created_at. Add a GET /health endpoint that returns status ok."*

Demo: Server runs. /health responds. Users table exists in the DB. He shows you both.

Day 2 — Environment config + structured logging

He moves every sensitive value into a .env file: database URL, secret key, Google client credentials. Uses python-dotenv and pydantic Settings to load them. After today, zero hardcoded secrets anywhere in the codebase.

He sets up logging so every request prints: timestamp, method, route, response status, time taken. He uses Python's built-in logging module — no external library needed.

Gemini prompt: *"Add structured logging to my FastAPI app using Python's logging module. Every HTTP request should log the method, path, status code, and time taken. Load all config from a .env file using pydantic BaseSettings."*

Demo: He shows you the .env file. He shows you that removing a value from .env breaks the app with a clear error. He makes 5 requests and shows you the logs printing cleanly for each one.

Days 3–4 — Google OAuth

He implements Google OAuth using the `authlib` library. The full flow:

User hits GET /auth/google → gets redirected to Google's login page → signs in → Google sends them back to GET /auth/callback → backend reads the token → gets the user's email and name → checks if user already exists in DB → creates them if not → returns a JWT token the frontend would use for all future requests.

Gemini prompt: *"Implement Google OAuth in FastAPI using authlib. When the user completes Google login, extract their email and name. Check if a user with that email exists in the Users table. If not, create them. Return a signed JWT token containing the user's id and email. Use python-jose for JWT signing."*

He will hit redirect URL errors, scope errors, token parsing issues. Each one gets pasted into Gemini. He doesn't move on until the full flow works end to end.

Demo: He opens a browser, hits /auth/google, logs in with his Google account, gets redirected back, receives a JWT token. You watch a new row appear in the Users table in real time.

Day 5 — JWT middleware + protected routes

He writes a dependency function that reads the JWT token from the Authorization header, validates it, and returns the current user. He applies this to a new endpoint: GET /me → returns the logged-in user's email and name. If the token is missing or invalid, returns 401.

Gemini prompt: *"Write a FastAPI dependency called get_current_user that reads a Bearer JWT token from the Authorization header, validates it, queries the Users table for that user, and returns the user object. Return HTTP 401 if the token is missing, expired, or invalid."*

Demo: He calls GET /me with a valid token — gets his user details. Calls it with no token — gets 401. Calls it with a fake token — gets 401. Three cases, all demonstrated.

Day 6 — Review + cleanup

No new features. He reads every file he wrote this week and cleans up anything Gemini wrote that he doesn't understand. He writes a plain-English doc (one page) explaining: what the app does, what's in the database, how Google login works, and what the JWT token is for. He sends this to you before the demo.

Demo: Full Week 1 flow from scratch. Fresh browser, sign in with Google, call /me with the token, show the DB. He also walks you through his one-page doc verbally.

Week 2 — Credit System

6 days · Goal: Every user has a credit balance. Free credits on signup. Credits tracked on every action.

What he's learning

How to model a financial-adjacent system in a database. How to make credit changes safe so two requests can't accidentally give the same credit twice. How to make every credit change traceable.

The analogy

Credits are like a prepaid phone balance. When you sign up you get some free minutes. Every call costs minutes. The system keeps a log of every top-up and every call so you can always see exactly where the balance went.

Days 1–2 — Credit balance + transaction ledger

He adds two new tables. UserCredits: user_id, balance (integer), updated_at. CreditTransactions: id, user_id, amount (positive for credits added, negative for credits spent), reason (text), created_at.

He writes a service function: `add_credits(user_id, amount, reason)` — adds credits and logs the transaction. He writes another: `deduct_credits(user_id, amount, reason)` — checks balance first, deducts if sufficient, logs the transaction, raises an error if insufficient.

Gemini prompt: *"Write two Python service functions for a credit system using SQLAlchemy async. add_credits(user_id, amount, reason) should add to the balance and insert a CreditTransactions row. deduct_credits(user_id, amount, reason) should check if balance is sufficient, deduct if yes, raise an InsufficientCreditsError if no, and log the transaction either way. Both must run inside a database transaction so they can't partially complete."*

Demo: He calls add_credits via a test script, shows balance increase in DB, shows transaction row created. Calls deduct_credits with insufficient balance, shows the error, shows nothing changed in the DB.

Day 3 — Free credits on signup

He modifies the Google OAuth callback to automatically call `add_credits(user_id, 100, "signup_bonus")` when a new user is created. Existing users don't get credits again.

Demo: He creates a brand new Google account (or deletes his test user from the DB), signs in, shows the UserCredits row created with 100 credits, shows the CreditTransactions row with reason "signup_bonus".

Day 4 — Credit balance endpoint

He adds GET /credits/balance → returns the current user's balance and their last 10 transactions. Protected by JWT middleware.

Demo: Signs in, hits /balance, sees 100 credits and the signup transaction. He manually adds more credits via a test script, refreshes the endpoint, sees the updated balance and new transaction in the list.

Days 5–6 — Two dummy product endpoints with credit gates

He creates two endpoints that simulate AI-powered features:

POST /api/summarize — costs 10 credits. Takes a text input, returns a fake summary ("Summary: [first 50 chars of input]..."). POST /api/analyze — costs 25 credits. Takes a text input, returns a fake analysis ("Analysis complete. Word count: X. Sentiment: Positive.").

Both endpoints: check JWT → get user → call deduct_credits → if successful, run the feature → return result. If insufficient credits, return a clear error: `{"error": "insufficient_credits", "balance": X, "required": Y}`.

Demo: Signs in with 100 credits. Calls /summarize — balance drops to 90, sees result. Calls /analyze — balance drops to 65. Keeps calling until balance hits 0. Next call returns the insufficient credits error with exact numbers.

Week 3 — Stripe Payments

6 days · Goal: Users can buy credit packages. Successful payment adds credits to their account automatically.

What he's learning

How payment systems actually work. Why you never trust what the frontend tells you a payment was — you only trust what Stripe tells your backend directly. How webhooks work.

The analogy

A Stripe webhook is like a delivery confirmation SMS. You don't trust the delivery driver saying "I delivered it" — you wait for the SMS from the courier company's system directly. Stripe's webhook is that SMS. It comes directly from Stripe to your backend, not through the user's browser.

Days 1–2 — Stripe setup + checkout session

He installs the Stripe Python library, adds Stripe keys to .env. He defines two credit packages in config: `{"starter": {"credits": 200, "price_usd": 9}, "pro": {"credits": 500, "price_usd": 19}}`.

He writes POST /payments/checkout — takes a package name, creates a Stripe Checkout Session, returns the checkout URL. The user would open that URL in their browser to pay.

Gemini prompt: *"Using the Stripe Python library, write a FastAPI endpoint POST /payments/checkout that accepts a package name (starter or pro). Look up the price from a config dictionary. Create a Stripe Checkout Session with that amount in cents, set success_url and cancel_url, and return the session URL. Store the package name in the session metadata so we can read it later."*

Demo: He calls /payments/checkout with "starter", gets back a Stripe URL, opens it in a browser, completes a test payment using Stripe's test card number (4242 4242 4242 4242), sees the Stripe dashboard show the payment. Credits not added yet — that's Day 3.

Days 3–4 — Stripe webhook + automatic credit top-up

He writes POST /payments/webhook — this is the endpoint Stripe calls after a successful payment. It reads the event type, verifies the webhook signature (so fake webhooks can't top up credits for free), extracts the user's email and package from the session metadata, calls `add_credits` with the right amount.

Gemini prompt: *"Write a FastAPI webhook endpoint POST /payments/webhook for Stripe. Verify the webhook signature using the Stripe webhook secret from .env. If the event type is checkout.session.completed, extract the customer email and metadata from the event. Look up the package in the config, call add_credits with the correct amount and reason 'stripe_payment'. Return 200 to Stripe on success, 400 if signature verification fails."*

He uses the Stripe CLI to forward webhooks to his local machine for testing.

Demo: He completes a Stripe test payment. Without touching the database manually, he shows the credit balance increase automatically within seconds. He shows the CreditTransactions row with reason "stripe_payment". He then shows what happens if he sends a fake webhook with a wrong signature — it gets rejected with 400.

Days 5–6 — Payment history endpoint + edge cases

He adds GET /payments/history → returns the user's last 10 payment-related credit transactions. He handles two edge cases: what if Stripe sends the same webhook twice (duplicate payment event)? He adds an idempotency check — store the Stripe session ID in a payments table, check before processing. If already processed, return 200 but don't add credits again.

Demo: He completes two test payments. Shows payment history endpoint. Then uses the Stripe CLI to replay the same webhook event twice — shows credits only added once, second webhook returns 200 but balance doesn't change.

Week 4 — Observability + CI/CD

6 days · Goal: The system tells you when something's wrong before users do. Code ships to the cloud automatically on every merge.

What he's learning

How to know your backend is healthy without manually checking. How to stop deploying by hand. How to make the code review → merge → deploy loop automatic.

The analogy

Observability is like the dashboard in a car. You don't pop the hood every 5 minutes to check the engine — the dashboard tells you. If the oil light comes on, you know before the engine dies. CI/CD is like a car factory's assembly line — same steps, same order, every time, no human variation.

Days 1–2 — Structured logging + error tracking

He upgrades the basic logging from Week 1 to structured JSON logging using the `structlog` library. Every log line is now a JSON object with: timestamp, level, route, user_id (if authenticated), duration_ms, and any error details. This makes logs searchable and machine-readable.

He adds Sentry for error tracking. Any unhandled exception automatically gets captured and sent to Sentry with the full stack trace, the route that caused it, and the user_id. He creates a free Sentry account.

Gemini prompt: *"Replace my current logging setup with structlog. Every log entry should be JSON with timestamp, log level, route path, user_id if available, and duration_ms. Add Sentry integration using sentry-sdk with FastAPI — capture all unhandled exceptions automatically with the user context attached."*

Demo: He deliberately causes an error (calls a route with bad data). Shows the error appearing in the Sentry dashboard within seconds with full stack trace. Shows the structured JSON logs in the terminal.

Days 3–4 — Health checks + basic metrics

He upgrades GET /health to a real health check: it checks database connectivity, checks that the Stripe API key is valid, and returns a structured response: `{"status": "ok", "database": "ok", "stripe": "ok", "uptime_seconds": X}`. If any check fails, status is "degraded" and it returns 503.

He adds Prometheus metrics using the `prometheus-fastapi-instrumentator` library. This automatically tracks: requests per second, average response time, error rate per endpoint. He exposes GET /metrics for Prometheus to scrape.

Demo: He shows /health returning fully green. He then temporarily breaks the database connection string and shows /health returning "degraded" with database: "error". He shows /metrics returning Prometheus-formatted data.

Days 5–6 — GitHub Actions CI + Cloud Build CD

He sets up a GitHub repository for AccessAI. He writes a GitHub Actions workflow that runs on every pull request: installs dependencies, runs a basic test (just hits /health and checks it returns 200), fails the PR if the test fails. No broken code merges.

He writes a Dockerfile for the app and a Cloud Build config (cloudbuild.yaml). When code merges to main: Cloud Build builds the Docker image, pushes it to Google Container Registry, deploys it to Google Cloud Run. The app is live on the internet with a real URL.

Gemini prompts, split across two days: *"Write a GitHub Actions workflow that triggers on pull requests. It should install Python dependencies, then make an HTTP request to GET /health on a locally started server and assert it returns 200. Fail the workflow if the assertion fails."* Then: *"Write a Dockerfile for my FastAPI app and a cloudbuild.yaml that builds the image, pushes to Google Container Registry, and deploys to Cloud Run with environment variables sourced from Google Secret Manager."*

Demo: He makes a pull request with a deliberate bug. Shows GitHub Actions failing. Fixes the bug. Shows it passing. Merges to main. Shows Cloud Build running in the GCP console. Shows the live Cloud Run URL serving the app on the internet. Calls /health on the live URL.

Week 5 — Polish, Harden, and Full Demo

6 days · Goal: The system is production-honest. Every failure mode is handled. Full end-to-end demo with no rough edges.

What he's learning

The difference between code that works in a demo and code that survives real use. How to think about what can go wrong, not just what should go right.

Days 1–2 — Rate limiting + input validation

He adds rate limiting: maximum 20 requests per minute per user using the `slowapi` library. Beyond that, returns 429 Too Many Requests. He adds strict input validation on /summarize and /analyze: text must be between 10 and 2000 characters, returns a clean validation error if not.

Demo: He writes a script that fires 25 requests in one minute. Shows the first 20 succeeding and the last 5 returning 429. Shows what happens when he sends an empty string to /summarize.

Days 3–4 — Security hardening

He adds CORS configuration (only specific origins allowed). He adds security headers using the `secure` library: Content-Security-Policy, X-Frame-Options, X-Content-Type-Options. He audits every endpoint for one thing: does it correctly verify the JWT before doing anything? He makes sure no endpoint accidentally exposes another user's data.

He writes a simple security checklist and checks every route against it: authenticated? input validated? error messages safe (no stack traces exposed to users)? logging the request?

Demo: He shows the security headers in browser dev tools. He tries to call /me with another user's token (he creates two test accounts) — shows they can only see their own data.

Days 5–6 — Full system demo

No new code. He rehearses and delivers a complete end-to-end demo of AccessAI as if presenting to a client.

The demo flow: open a fresh browser with no cookies, sign in with Google, show 100 free credits appear automatically, call /summarize three times (show credits dropping), call /analyze once, run out of credits, hit the insufficient credits error, go to /payments/checkout for the starter pack, complete Stripe test payment, watch credits jump to 290 automatically via webhook, resume calling the product endpoints, show /health green, show a recent error in Sentry, show the metrics endpoint, show the GitHub Actions history, show the Cloud Run deployment.

He also walks you through one file he's proud of and explains what it does and why he wrote it the way he did. This is as important as the demo itself.

Timeline Summary

Week	Focus	Demo Milestone
1	Secure foundation + Google login	Sign in with Google, protected routes working
2	Credit system + gated endpoints	Credits deducted on every API call, gates enforced
3	Stripe payments + webhooks	Buy credits, balance updates automatically
4	Observability + CI/CD	Live on internet, auto-deploys on merge
5	Hardening + full demo	Production-honest system, full end-to-end demo

---

The Rule for All 5 Weeks

He demos only when it actually works — not when it almost works. "Almost working" is not a demo. If a week takes 7 days instead of 6, that's fine. The demo gates are fixed. The timeline is a guide. By the end of Week 5, what he has built is not a tutorial project. It's a system shaped like real backend engineering — the same patterns that run inside Kasparro.