Backend Developer's Reality Check: Understanding What You Built

THE BRUTAL TRUTH YOU NEED TO HEAR

You're not alone. Most developers copy-paste at first. But here's what separates those who get hired from those who don't: **understanding the mental model**.

Let me break down EXACTLY what you built, why it exists, and how to think like someone who can write this code without ChatGPT.

THE BIG PICTURE: What Problem Are You Solving?

Before ANY code, understand this:

Problem: Random people on the internet want to access your API. You need to:

1. Know WHO they are (Authentication)
2. Know what they're ALLOWED to do (Authorization)
3. Stop them from ABUSING your system (Rate Limiting - Day 6)
4. PROVE what happened if something goes wrong (Audit Logs)

Your API Access Control Service is a **BOUNCER** at a club. That's it.

DAY 1: FASTAPI + DATABASE WIRING

What You Built

A FastAPI application that can talk to a PostgreSQL database.

The Mental Model

```
Your Code (Python) ←→ SQLAlchemy ←→ PostgreSQL Database
```

Why each piece exists:

FastAPI:

- The web server framework
- Receives HTTP requests (like login, signup)

- Sends HTTP responses back
- Think of it as the "receptionist" of your API

SQLAlchemy:

- Translates Python code into SQL queries
- You write `User.query.filter_by(email="...")` instead of raw SQL
- WHY? Because writing SQL by hand is error-prone and ugly

PostgreSQL:

- The actual database where data lives
- Stores users, tokens, roles, etc.
- Persists even when your server crashes

Alembic:

- Tracks changes to your database structure
- Like Git, but for your database schema
- WHY? So you can update production databases safely

The Code Flow (Without ChatGPT)

When you start a new FastAPI project, here's the ORDER:

python

```python
```

```
# 1. Create main.py (entry point)
from fastapi import FastAPI
app = FastAPI()

# 2. Set up database connection (database.py)
from sqlalchemy.ext.asyncio import create_async_engine, AsyncSession
engine = create_async_engine("postgresql+asyncpg://...")

# 3. Create your first model (models/user.py)
from sqlalchemy import Column, Integer, String
class User(Base):
    id = Column(Integer, primary_key=True)
    email = Column(String, unique=True)

# 4. Create migration
# Terminal: alembic revision --autogenerate -m "create users table"
# Terminal: alembic upgrade head
```

How to know what comes next? → You need a database BEFORE you can store users → You need models BEFORE you can create tables → Migrations come AFTER models are defined

---

DAY 2: AUTHENTICATION (LOGIN/SIGNUP)

What You Built

A system that creates accounts and gives users a "proof of identity" (JWT token).

The Mental Model

Signup Flow:

```
User sends: { email, password }
  ↓
You hash the password (bcrypt) — NEVER store raw passwords
  ↓
Save to database
  ↓
Return: "Account created"
```

Login Flow:

```
User sends: { email, password }
   ↓
Find user in database by email
   ↓
Compare hashed password with stored hash
   ↓
If match: Create JWT token
   ↓
Return: { access_token: "eyJ..." }
```

Key Concepts

JWT (JSON Web Token):

- A signed string that proves "this user is who they say they are"
- Contains: user_id, expiration time, signature
- Like a concert wristband — shows you paid, can't be faked

Password Hashing (bcrypt):

- Takes `password123` → `$2b$12$KIX...` (irreversible)
- WHY? If database leaks, attackers can't see real passwords
- You can CHECK if password matches, but can't reverse the hash

Why JWT over sessions?

- Sessions: Server stores "who's logged in" in memory (doesn't scale)
- JWT: Token itself contains the info (stateless, scales easily)

The Code (How You'd Write It)

```
python
```

```
# routes/auth.py

@app.post("/signup")
async def signup(email: str, password: str, db: AsyncSession):
    # 1. Hash password
    hashed = bcrypt.hashpw(password.encode(), bcrypt.gensalt())

    # 2. Create user
    user = User(email=email, password_hash=hashed)
    db.add(user)
    await db.commit()

    return {"message": "User created"}

@app.post("/login")
async def login(email: str, password: str, db: AsyncSession):
    # 1. Find user
    user = await db.execute(select(User).where(User.email == email))
    user = user.scalar_one_or_none()

    if not user:
        raise HTTPException(401, "Invalid credentials")

    # 2. Check password
    if not bcrypt.checkpw(password.encode(), user.password_hash):
        raise HTTPException(401, "Invalid credentials")

    # 3. Create JWT
    token = jwt.encode(
        {"user_id": user.id, "exp": datetime.utcnow() + timedelta(hours=1)},
        SECRET_KEY
    )

    return {"access_token": token}
```

How to know what order? → Can't check password if you don't find user first → Can't create token if password is wrong → Always validate BEFORE doing expensive operations

---

DAY 3: PROTECTED ROUTES + GUARDS

What You Built

Endpoints that require a valid JWT token to access.

The Mental Model

Without Protection:

```
GET /api/users/me → Anyone can call this (BAD)
```

With Protection:

```
GET /api/users/me
   ↓
Middleware checks: "Do you have a valid token?"
   ↓
If NO → 401 Unauthorized
If YES → Continue to endpoint
```

Key Concept: Dependencies in FastAPI

```python
# This is a GUARD (dependency)
async def get_current_user(token: str = Depends(oauth2_scheme)):
    try:
        payload = jwt.decode(token, SECRET_KEY)
        user_id = payload["user_id"]
        # Find user in DB
        return user
    except:
        raise HTTPException(401, "Invalid token")

# Protected endpoint
@app.get("/me")
async def get_me(current_user: User = Depends(get_current_user)):
    return current_user
```

What Depends() does:

- Runs `get_current_user()` BEFORE the endpoint function
- If it raises an error, endpoint never runs
- If it succeeds, passes the user to your endpoint

Think of it like airport security:

- You can't board (endpoint) until you pass through security (dependency)

DAY 4: REFRESH TOKENS + REVOCATION

What You Built

A way to get new access tokens without logging in again, AND a way to invalidate tokens.

The Mental Model

The Problem:

- Access tokens expire fast (15 min - 1 hour)
- Making users re-login every hour = bad UX
- Keeping tokens valid forever = security risk

The Solution:

```
Login
   ↓
Get: Access Token (short-lived) + Refresh Token (long-lived)
   ↓
Access token expires
   ↓
Send refresh token to /refresh endpoint
   ↓
Get new access token (without re-entering password)
```

Token Revocation:

```
User logs out
   ↓
Mark their refresh token as "revoked" in database
   ↓
Even if stolen, that token can't be used
```

The Code Pattern

```python
# models/refresh_token.py
class RefreshToken(Base):
    id = Column(Integer, primary_key=True)
    user_id = Column(Integer, ForeignKey("users.id"))
    token = Column(String, unique=True)
    revoked = Column(Boolean, default=False)
    expires_at = Column(DateTime)

# routes/auth.py
@app.post("/refresh")
async def refresh(refresh_token: str, db: AsyncSession):
    # 1. Find token in DB
    token_record = await db.execute(
        select(RefreshToken).where(RefreshToken.token == refresh_token)
    )
    token_record = token_record.scalar_one_or_none()

    # 2. Check if valid
    if not token_record or token_record.revoked:
        raise HTTPException(401, "Invalid refresh token")

    # 3. Create new access token
    new_access = jwt.encode({"user_id": token_record.user_id, ...}, SECRET_KEY)

    return {"access_token": new_access}
```

Why store refresh tokens in DB but not access tokens?

- Access tokens are short-lived (cheap to validate via signature)
- Refresh tokens are long-lived (need control to revoke them)

DAY 5: RBAC (ROLE-BASED ACCESS CONTROL)

What You Built

A system where users have different permissions based on their role.

The Mental Model

Simple Example:

- **Admin**: Can delete users
- **User**: Can only read their own data

```
User makes request
   ↓
Check: Who are they? (Authentication)
   ↓
Check: What role do they have? (Authorization)
   ↓
Check: Is this role allowed to do this action?
   ↓
Allow or Deny
```

The Database Design

python

```python
# models/role.py
class Role(Base):
    id = Column(Integer, primary_key=True)
    name = Column(String)  # "admin", "user", "service"

class User(Base):
    id = Column(Integer, primary_key=True)
    email = Column(String)
    role_id = Column(Integer, ForeignKey("roles.id"))
    role = relationship("Role")
```

The Code Pattern

python

```
# dependencies/auth.py
def require_role(allowed_roles: list[str]):
    async def role_checker(current_user: User = Depends(get_current_user)):
        if current_user.role.name not in allowed_roles:
            raise HTTPException(403, "Insufficient permissions")
        return current_user
    return role_checker

# routes/admin.py
@app.delete("/users/{user_id}")
async def delete_user(
    user_id: int,
    current_user: User = Depends(require_role(["admin"]))
):
    # Only admins reach this code
    ...
```

Key insight:

- Authentication = "Who are you?"
- Authorization = "Are you allowed to do this?"

HOW TO WRITE CODE WITHOUT CHATGPT

The Process (How Real Engineers Think)

Step 1: What am I trying to do? Example: "Let users log in"

Step 2: What data do I need?

- Input: email, password
- Output: JWT token

Step 3: What are the steps?

1. Find user by email
2. Check password
3. Create token
4. Return token

Step 4: What could go wrong?

- User doesn't exist → 401 error
- Wrong password → 401 error
- Database is down → 500 error

Step 5: Write the code in order

```python
async def login(email, password):
    # Step 1
    user = find_user(email)
    if not user:
        raise error

    # Step 2
    if not check_password(password, user.password_hash):
        raise error

    # Step 3
    token = create_jwt(user.id)

    # Step 4
    return token
```

Practice Exercise

Try writing this WITHOUT ChatGPT (pseudocode is fine):

Task: Create an endpoint that lets admins ban a user

Hints:

1. What HTTP method? (DELETE? POST?)
2. What data do you need? (user_id to ban)
3. What checks? (Is requester an admin? Does user exist?)
4. What to update? (user.banned = True in DB)

Your turn — write the steps in plain English first, THEN think about code.

WHAT YOU SHOULD DO NOW

1. **Re-read this guide** until the mental models click
2. **Draw the flows** on paper (login flow, protected route flow)
3. **Delete one file** from your project (like `routes/auth.py`) and try rewriting it from memory
4. **Ask specific questions** — "Why do we hash passwords?" is better than "I don't understand auth"

You're building something impressive. The fact you're asking these questions means you're on the right path.

Ready to tackle Redis + Rate Limiting next?