Module 3: Accelerated Code Generation and Debugging, Lesson 1: Zero-to-Draft Code Generation

Hands-On: RICE in Action—Live JWT Service Demo

Overview

This demo proves three claims from the lecture in real time:

- Contract-first prompts produce predictable, typed output — not creative guesses
- Documentation is a byproduct of well-structured prompts, not a separate task
- The review step is non-negotiable — even a five-minute pass catches real production risks

You will build a complete JWTAuthService class using a single, layered prompt — then immediately go through the output the way a senior engineer would in a real code review. The goal is not to admire what the AI produced. The goal is to own it. By the end of this exercise you will be able to articulate exactly what came back, why it came back that way, what you would approve, and what you would push back on.

This exercise mirrors what you will do every day as an AI-native engineer: prompt with precision, evaluate with seniority, and ship with accountability.

Before You Start

- Open Cursor (or Claude.ai / GitHub Copilot Chat) with a blank file
- Have a Python 3.11+ virtual environment ready with: pip install fastapi pydantic python-jose passlib sqlalchemy
- Keep the six-part prompt template visible on screen or a second monitor
- Do NOT start a new chat between prompts — session continuity is the point
- Paste the prompt exactly as written — resist the urge to trim layers before you see the output

> **Remember**
>
> Every layer of the prompt below is load-bearing. The Role layer sets the AI's quality bar. The Schema layer eliminates ambiguity about data shapes. The Requirements layer prevents the AI from making decisions you would have made differently. Remove any layer and the output degrades in a predictable way. We will prove this in the student lab.

During Hands-On — Keep Visible

Keep the six-part prompt structure visible throughout. Map every block below to one of these layers as you write:

```
1. ROLE        —  who the AI is (seniority, specialty, quality standard)
2. TASK        —  what to build (class type, purpose, name)
3. CONSTRAINTS —  stack, versions, async/sync, library choices
4. SCHEMA      —  data shapes the output must conform to
5. METHODS     —  exact API surface (names, signatures, return types)
6. EXTRAS      —  secondary requirements (docstrings, exports, error
format)
```

Step 1 — The Layered Prompt

What to Learn

Notice how the prompt is built in layers, not written as a paragraph. Each layer narrows the solution space. By the time the AI reaches the METHODS block it cannot guess at method signatures — they are specified. By the time it reaches the EXTRAS block it cannot skip docstrings — they are required. The AI never has to decide what 'good' looks like because you have already told it.

The most common beginner mistake is writing a single-paragraph prompt that mixes task, constraints, and requirements together. Layered prompts are not just cleaner to read — they produce measurably more consistent output because each section maps to a discrete decision the model makes.

Paste this into Cursor as your opening prompt in a blank file:

```
// LAYER 1 — ROLE
You are a senior Python engineer with deep FastAPI expertise and a strong
focus on security-hardened, production-ready code.

// LAYER 2 — TASK
Create a JWTAuthService class for a FastAPI application that handles all
JWT token operations and user authentication.

// LAYER 3 — TECHNICAL CONSTRAINTS
- FastAPI with fully async/await patterns throughout
- Pydantic v2 for all data models (use model_config + ConfigDict,
  never class Config)
- python-jose for JWT encode/decode operations
- passlib with bcrypt for password hashing
- SQLAlchemy AsyncSession for database access
```

```
- All public methods must be async where I/O is involved

// LAYER 4 — USER SCHEMA (work from exactly this)
class User(BaseModel):
    model_config = ConfigDict(from_attributes=True)
    id: UUID
    email: EmailStr
    hashed_password: str
    is_active: bool = True
    is_superuser: bool = False
    created_at: datetime

// LAYER 5 — REQUIRED METHODS
- create_access_token(data: dict, expires_delta: timedelta | None) -> str
- create_refresh_token(data: dict) -> str
- verify_token(token: str, expected_type: str) -> TokenPayload
- authenticate_user(db: AsyncSession, email: str, password: str) -> User
| None
- get_current_user(db: AsyncSession, token: str) -> User
- get_current_active_user(current_user: User) -> User
- hash_password(password: str) -> str
- verify_password(plain_password: str, hashed_password: str) -> bool

// LAYER 6 — ADDITIONAL REQUIREMENTS
- A TokenPayload Pydantic v2 model: sub, exp, type fields
- A Token response model: access_token, refresh_token, token_type
- UUID validator on TokenPayload.sub
- Settings from config object (SECRET_KEY, ALGORITHM,
  ACCESS_TOKEN_EXPIRE_MINUTES, REFRESH_TOKEN_EXPIRE_DAYS)
- HTTPException with WWW-Authenticate headers on auth failures (RFC 6750)
- Type hints on every parameter and return value
- A one-line docstring on each method
- Module-level singleton: auth_service = JWTAuthService()
```

What you should get

A complete auth.py file with two Pydantic models (TokenPayload and Token) and a JWTAuthService class containing all eight methods. The class should have a constructor that reads from settings, module-level CryptContext and OAuth2PasswordBearer instances, and a singleton export at the bottom. All methods should carry type hints and one-line docstrings.

If the model skips any method, adds a class Config instead of ConfigDict, or omits the singleton — that is signal that a layer was ambiguous. Note what failed and why before continuing.

You should ask:

“Why is CryptContext defined at module level instead of inside __init__?”

Answer: Creating a CryptContext is expensive. It loads and initialises the bcrypt algorithm. Doing it at module level means it pays that cost once per process, not once per request. This is the kind of performance decision a senior engineer makes automatically — and the kind you should notice, verify, and endorse or change consciously.

You should also ask:

“Why does verify_token take an expected_type parameter instead of two separate methods?”

Answer: A single verification entry point with a type parameter makes the token type check mandatory. If it were two separate methods (verify_access_token / verify_refresh_token), callers could invoke the wrong one. This is a defensive API design choice — the kind worth recognising and reinforcing.

Step 2 — The Generated Code

What to Say

Read the output top to bottom before commenting. You are looking for three things: structure, decisions, and surprises. Structure means: is it laid out the way a senior engineer would lay it out? Decisions means: did the AI make choices you would have made? Surprises means: did it add something you did not ask for, or leave out something you expected?

Resist the urge to run the code immediately. The review step comes first. An untested code review is better than no review at all.

The code block below is a representative example of what a well-prompted model returns. Your actual output may differ slightly — the analysis in Step 3 applies to the patterns, not the exact lines.

```
# app/services/auth.py
from datetime import datetime, timedelta, timezone
from typing import Any
from uuid import UUID

from fastapi import Depends, HTTPException, status
from fastapi.security import OAuth2PasswordBearer
from jose import JWTError, jwt
from passlib.context import CryptContext
from pydantic import BaseModel, ConfigDict, EmailStr, field_validator
from sqlalchemy.ext.asyncio import AsyncSession

from app.core.config import settings
from app.models.user import User
```

```
# ─── Token Models ──────────────────────────────────────────────
class TokenPayload(BaseModel):
    model_config = ConfigDict(frozen=True)
    sub: str | None = None
    exp: datetime | None = None
    type: str = 'access'

    @field_validator('sub')
    @classmethod
    def sub_must_be_valid_uuid(cls, v: str | None) -> str | None:
        if v is not None:
            try:
                UUID(v)
            except ValueError:
                raise ValueError('sub must be a valid UUID string')
        return v


class Token(BaseModel):
    access_token: str
    refresh_token: str
    token_type: str = 'bearer'


# ─── Auth Service ──────────────────────────────────────────────
oauth2_scheme = OAuth2PasswordBearer(tokenUrl='/api/v1/auth/token')
pwd_context = CryptContext(schemes=['bcrypt'], deprecated='auto')


class JWTAuthService:
    """JWT-based authentication service for FastAPI applications."""

    def __init__(self) -> None:
        self._secret_key = settings.SECRET_KEY
        self._algorithm = settings.ALGORITHM
        self._access_token_expire = timedelta(
            minutes=settings.ACCESS_TOKEN_EXPIRE_MINUTES
        )
        self._refresh_token_expire = timedelta(
            days=settings.REFRESH_TOKEN_EXPIRE_DAYS
        )

    def create_access_token(
        self, data: dict[str, Any], expires_delta: timedelta | None =
None
    ) -> str:
        """Create a signed JWT access token."""
```

```
        to_encode = data.copy()
        expire = datetime.now(timezone.utc) + (expires_delta or
self._access_token_expire)
        to_encode.update({'exp': expire, 'type': 'access'})
        return jwt.encode(to_encode, self._secret_key,
algorithm=self._algorithm)

    def create_refresh_token(self, data: dict[str, Any]) -> str:
        """Create a signed JWT refresh token with a longer expiry."""
        to_encode = data.copy()
        expire = datetime.now(timezone.utc) + self._refresh_token_expire
        to_encode.update({'exp': expire, 'type': 'refresh'})
        return jwt.encode(to_encode, self._secret_key,
algorithm=self._algorithm)

    def verify_token(self, token: str, expected_type: str = 'access') ->
TokenPayload:
        """Decode and validate a JWT, raising HTTPException on any
failure."""
        credentials_exception = HTTPException(
            status_code=status.HTTP_401_UNAUTHORIZED,
            detail='Could not validate credentials',
            headers={'WWW-Authenticate': 'Bearer'},
        )
        try:
            payload = jwt.decode(token, self._secret_key,
algorithms=[self._algorithm])
            token_data = TokenPayload(**payload)
        except JWTError:
            raise credentials_exception
        if token_data.sub is None:
            raise credentials_exception
        if token_data.type != expected_type:
            raise HTTPException(
                status_code=status.HTTP_401_UNAUTHORIZED,
                detail=f"Invalid token type. Expected
'{expected_type}'.",
                headers={'WWW-Authenticate': 'Bearer'},
            )
        return token_data

    def hash_password(self, password: str) -> str:
        """Return a bcrypt hash of the given plain-text password."""
        return pwd_context.hash(password)

    def verify_password(self, plain_password: str, hashed_password: str)
-> bool:
        """Return True if the plain-text password matches the stored
hash."""
        return pwd_context.verify(plain_password, hashed_password)
```

```
    async def authenticate_user(
        self, db: AsyncSession, email: str, password: str
    ) -> User | None:
        """Look up user by email and verify password. Returns None on
failure."""
        from app.repositories.user import UserRepository
        user = await UserRepository(db).get_by_email(email)
        if not user:
            return None
        if not self.verify_password(password, user.hashed_password):
            return None
        return user

    async def get_current_user(
        self,
        db: AsyncSession,
        token: str = Depends(oauth2_scheme),
    ) -> User:
        """Resolve and return the authenticated user from a Bearer
token."""
        from app.repositories.user import UserRepository
        token_data = self.verify_token(token, expected_type='access')
        user = await UserRepository(db).get_by_id(UUID(token_data.sub))
# type: ignore
        if user is None:
            raise HTTPException(status_code=status.HTTP_404_NOT_FOUND,
                                detail='User not found')
        return user

    async def get_current_active_user(self, current_user: User) -> User:
        """Guard: raise 403 if the authenticated user account is
inactive."""
        if not current_user.is_active:
            raise HTTPException(status_code=status.HTTP_403_FORBIDDEN,
                                detail='Inactive user')
        return current_user


# ─── Singleton ──────────────────────────────────────────────
auth_service = JWTAuthService()
```

Step 3 — The Review (Non-Negotiable)

What to Learn

This is the step most students skip. Don’t. The review is where you transition from AI user to AI-native engineer. Your job is not to check whether the code compiles. Your job is to

evaluate the decisions the AI made and decide which ones you endorse, which ones you would change, and which ones represent genuine production risk.

Ask yourself: if this code arrived in a PR from a junior engineer on your team, what would you approve, what would you comment on, and what would you block? The answer is identical whether the author is a person or an LLM. The review process does not change.

What the AI Got Right

These are the decisions you should recognise, understand, and consciously carry forward into your own code.

- **Timezone-aware datetimes:** Uses datetime.now(timezone.utc) throughout. Naive datetimes in JWT logic cause silent comparison failures in production when the hosting environment runs on UTC but the application assumes local time. This is the correct call.
- **Token type field on both issue and verify:** The type claim in the payload prevents refresh tokens from being used as access tokens. This is a real and commonly missed attack vector — a refresh token is long-lived and high-value; accepting it where an access token is expected is a privilege escalation bug.
- **UUID validator on TokenPayload.sub:** The @field_validator runs before any DB call. A malformed or injected sub value is rejected at the boundary, not deep in a repository method where the error message is harder to interpret.
- **CryptContext at module level:** Creating a CryptContext is expensive. Placing it at module level means the algorithm is loaded once per process. A per-request instantiation would add measurable latency under load.
- **RFC 6750 compliance:** Every 401 response carries the WWW-Authenticate: Bearer header. This is required by the spec and expected by OAuth2 clients and API gateways for correct error handling.
- **Frozen TokenPayload:** ConfigDict(frozen=True) makes the decoded payload immutable. Token payloads are trust boundary objects — they should not be modifiable after decoding.
- **Separate access and refresh token expiry:** Storing expiry as a timedelta in __init__ means the values are read from settings once and reused, rather than re-read on every token creation. This is consistent with twelve-factor app principles.

What I Would Change

These are not bugs. They are design decisions the AI made that a senior engineer might make differently. Push back means a PR comment, not a rejection.

- **Lazy imports are a code smell:** Both authenticate_user and get_current_user do from app.repositories.user import UserRepository inline. This pattern is used to break circular imports, which means the underlying architecture has a circular dependency. The fix is to restructure the module graph, not to hide the symptom with a deferred import.

- **Settings baked into __init__:** Reading from settings in the constructor means the singleton reflects the settings values at import time. In tests that override settings between runs, this produces confusing state. Prefer reading settings directly in each method, or inject a Settings instance explicitly via dependency injection.
- **Depends() inside a class method:** get_current_user has Depends(oauth2_scheme) in its signature. FastAPI's DI system does not resolve Depends on class method parameters without explicit wiring. The idiomatic pattern is a module-level dependency function that calls auth_service.get_current_user(). This is worth refactoring before wiring up the routes.
- **# type:** ignore on the UUID cast: The UUID(token_data.sub) cast carries a type: ignore comment because Pyright sees sub as str | None. The guard above it proves sub is not None at that point, but the type system doesn't know that. The cleaner fix is to define a narrowed model or use assert token_data.sub is not None to communicate intent explicitly.
- **get_current_active_user is async but contains no I/O:** This method is a pure guard — it reads a boolean and raises if False. Marking it async adds nothing and slightly misleads callers who will await it expecting I/O. Drop the async unless the prompt explicitly required it, or document the reason.

What's Missing (Production Gaps)

These are real gaps that could cause incidents in a production system. Raise them in the demo and explain why each matters.

- **No token revocation or blocklist:** There is no mechanism to invalidate a token on logout or after a password change. The standard approach is a Redis-backed blocklist keyed on a jti (JWT ID) claim. The model also did not add a jti claim to the token payload — without it, individual token revocation is impossible.
- **No rate limiting on authenticate_user:** The login endpoint is an unguarded brute-force surface. In production, failed attempts should be counted in Redis with a sliding window and throttled before bcrypt even runs. bcrypt is intentionally slow — an attacker can use that to amplify the cost to your server while probing credentials.
- **verify_password blocks the event loop:** bcrypt is CPU-bound, not I/O-bound. Calling pwd_context.verify() synchronously inside an async handler blocks the event loop for every other concurrent request during the hash computation. Wrap it: await asyncio.get_event_loop().run_in_executor(None, pwd_context.verify, plain, hashed).
- **No jti claim for audit trails or replay prevention:** A jti (JWT ID) claim — a UUID per token — enables token-level revocation, security auditing, and replay attack detection. It should be standard in any production JWT implementation and takes three lines to add.
- **get_current_user returns 404 for a deleted user:** Returning 404 — Not Found on a resolved token leaks information about whether a user ID exists. For public-facing auth flows, this should collapse to the same generic 401 — Could not validate credentials response.
- **No rotation strategy for SECRET_KEY:** The service reads SECRET_KEY once at startup. There is no mechanism for key rotation without a rolling restart. In a mature system you would support multiple verification keys (current and previous) to allow zero-downtime rotation.

Decisions Worth Discussing

These are design choices where the AI made a reasonable call but a different reasonable call exists. Use these as discussion prompts.

- **Singleton vs factory:** The module-level auth_service = JWTAuthService() is clean for production use but complicates testing. A factory function (def get_auth_service() -> JWTAuthService) that FastAPI resolves as a dependency makes the service mockable without monkeypatching.
- **Repository instantiated inline, not injected:** UserRepository(db) is constructed inside the service methods. This tightly couples the service to one concrete repository class. For testability, define a UserRepositoryProtocol and inject it — then tests can pass a fake that does not touch the database.
- **Single signing key for all token types:** Using the same SECRET_KEY for both access and refresh tokens means that a key compromise exposes both. Some architectures use different keys for access vs refresh tokens to limit the blast radius of a leak.
- **No scope or role claims:** The Token models carry no claims about what the user is allowed to do. A real RBAC or ABAC system would embed scopes or roles in the access token payload so that route handlers can enforce permissions without a database call on every request.

Senior Engineer Verdict

This is a genuinely solid first draft — production-closer than most code written from scratch under time pressure. The security fundamentals are right: timezone-aware tokens, token type separation, UUID validation, RFC-compliant headers. What's missing is the operational layer — revocation, rate limiting, audit trails — and a handful of architectural calls that only surface when you have been paged at 2am for a breach.

That is the gap AI fills quickly but does not yet bridge reliably. Your job as the engineer is exactly this review. The model saved you an hour of boilerplate. You spent five minutes finding three real production risks and six design decisions worth reconsidering. That is the deal.

Category	Score	Notes
Security fundamentals	8 / 10	Timezone-aware, type-separated tokens, RFC 6750 headers
Production readiness	5 / 10	Missing revocation, rate limiting, jti, event loop fix
Code quality & structure	7 / 10	Minor DI mismatch and lazy import smell, otherwise clean
Type safety	7 / 10	One type: ignore, sub narrowing could be cleaner
Testability	5 / 10	Singleton and inline repository hurt isolation

Time saved vs scratch	≈60 min	Full class with 8 typed, documented methods

Discussion Questions

Use these during or after the demo to prompt deeper engagement:

1. If this code arrived in a PR from a junior engineer, what would you approve, comment on, and block? Is your answer different because the author was an AI?
2. Which of the missing production features would you prioritise first and why? Does the answer change based on the deployment environment (startup MVP vs enterprise SaaS)?
3. The prompt specified eight method signatures. What would have happened if you had left the signatures out and just asked for a JWTAuthService? Run the experiment.
4. Which layer of the prompt was most load-bearing? What breaks first if you remove it? Try removing Layer 4 (the User schema) and compare the output.
5. The model added ConfigDict(frozen=True) to TokenPayload without being asked. Is that a good decision? What does it prevent? When would it be wrong?

Student Lab Extension

What to do

Continue in the same chat session. The AI already has the full JWTAuthService in context. Use follow-up prompts — without re-explaining the schema or method signatures — to extend the service. This proves that session continuity eliminates context repetition.

In the same session, send these follow-up prompts one at a time. Observe how each builds on the established context:

```
// Follow-up Prompt A — Add jti claim and Redis blocklist
Extend the JWTAuthService above to add a jti (UUID) claim to both
create_access_token and create_refresh_token. Add two new methods:
  - revoke_token(jti: str) -> None  (stores in Redis with TTL = token
expiry)
  - is_token_revoked(jti: str) -> bool
Inject a redis.asyncio.Redis instance via __init__. Update verify_token
to
call is_token_revoked and raise 401 if True.

// Follow-up Prompt B — Fix the event loop blocking
In the existing JWTAuthService, verify_password and hash_password call
```

```
passlib synchronously in an async context. Rewrite both methods to use
asyncio.get_event_loop().run_in_executor so bcrypt runs in a thread pool
without blocking the event loop. Update the method signatures
accordingly.

// Follow-up Prompt C — Add scope-based authorisation
Add a scopes: list[str] = [] field to TokenPayload. Add a new method:
  - require_scope(required: str) -> Callable
that returns a FastAPI dependency which raises 403 if the required scope
is not present in the token's scope list. Write two example usages as
comments showing how a route handler would use this dependency.
```

Remember

Three follow-up prompts. One session. You added token revocation, fixed the event loop blocker, and introduced scope-based authorisation — without re-explaining the User schema, the method signatures, or the library choices. The AI carried the context forward. That is what session continuity buys you.

At least one generated method will need a small fix. Read the error. The fix is usually one line. A method the AI wrote with one flaw is still better than a method you hadn't written yet.

Wrap-Up — Key Takeaways

Key takeaways:

- Layer your prompts: Role, Task, Constraints, Schema, Methods, Extras. Each layer removes a decision the AI would otherwise make for you.
- The review step is non-negotiable. What the AI gets right is usable. What it misses is your professional responsibility to catch.
- Session continuity eliminates repetition. Keep every related prompt in one window. The AI carries your context forward.
- Production readiness is not the AI's default. Revocation, rate limiting, audit trails, key rotation — these require explicit prompting or deliberate human addition.
- The code review does not change. Approve what is right, comment on what is debatable, block what is dangerous. The author being an LLM is irrelevant to that process.
- Use follow-up prompts in the same session to extend, not to start over. Contract-first thinking applied iteratively is how AI-native engineers build.

The pattern repeats every time:

Precise prompt → evaluate output → own the decisions → extend in session. That is the AI-native engineering loop.