

Building Resilient Backend Systems

An Advanced Guide

Environment-Aware Configurations for Personal Banking App

TARGET AUDIENCE

Python (FastAPI) Developers

SCOPE

Personal Banking Management System

The Modern Backend Challenge

"An application is not a single entity; it's a chameleon that must adapt to its surroundings."

Development

Needs fast reloads, mock data, and detailed debugging.

Staging/QA

Needs to mirror production as closely as possible, connecting to test databases and services.

Production

Needs to be secure, performant, and highly available, with robust logging and monitoring.

The Goal: A single, clean codebase that behaves correctly in any environment without if/else checks. This is the foundation of the Twelve-Factor App methodology.

Understanding Application Environments

"An application is not a single entity; it's a chameleon that must adapt to its surroundings"

A Guide to Development, Staging, and Production

The House Building Analogy

Just like building a house, software applications undergo various stages before they are ready for general use.

Key Concept: Applications need to behave differently depending on where they're running—just like a chameleon adapts to its environment.

Let's explore the three main environments where applications live...

1. Development (Dev) Environment

Your construction blueprint and workshop

What happens here:

This is where programmers write and test their code. They are actively building new features or fixing bugs.

Why it's special:

- **Fast reloads:** Developers need to see changes quickly as they code
- **Mock data:** Uses fake data to test functionality without real user information
- **Detailed debugging:** Tools to find exact problems, like a detective looking for clues

2. Staging/QA Environment

A model home for final inspection

What happens here:

After developers finish, the application moves to staging, where Quality Assurance (QA) testers thoroughly check everything.

Why it's special:

- **Mirrors production:** Made as similar as possible to the live environment
- **Test databases:** Connects to test versions of databases and services
- **Final testing:** QA testers find any remaining bugs before going live

3. Production (Prod) Environment

The finished house where real users live

What happens here:

This is the live environment where real users interact with the application.

Why it's special:

- **Secure:** Strong security to protect user data and prevent attacks
- **Performant:** Fast and responsive, even with many users
- **Highly available:** Almost always working, minimal downtime
- **Robust monitoring:** Systems to record issues and alert support teams

The Goal: A Single, Clean Codebase

One version of code that works correctly in ALL environments

Why this matters:

- Easier to manage and maintain
- Less prone to errors and inconsistencies
- No special "if this environment, then do that" checks needed
- Consistent behavior across all stages

Twelve-Factor App methodology: A set of best practices for building robust, scalable applications that adapt naturally to different environments.

Key Takeaways

Development: Build and experiment with new features

Staging/QA: Test thoroughly in a production-like environment

Production: Serve real users with security and reliability

Different environments serve different purposes, but the aim is to have a consistent and adaptable application across all of them!

Topic 1 - Environment Profiles

Profiles are the core mechanism for managing this complexity. We externalise configuration from the code.

Python/FastAPI Implementation

Mechanism

`.env` files + Environment Variables

How it Works

An environment variable (e.g., `APP_ENV=dev`) tells the app which `.env` file (`.env.dev`) to load its settings from.

Key Library

`pydantic-settings` for loading and validating configs with full type safety and validation.

💡 **Pro Tip:** Never commit secrets or environment-specific `.env` / `.properties` files to Git. Use a `.gitignore` file and provide a template (e.g., `.env.example`) for other developers.



Classroom Exercise

Short Exercise #1 - Create Your Profiles

Task: (5 minutes) Create the configuration files for our Personal Banking app for two environments: dev and prod.

Requirements:

Dev Environment

- APP_TITLE: "Personal Banking - DEV"
- DATABASE_URL: "sqlite:///./test.db"
- DEBUG_MODE: True
- REDIS_URL: "redis://localhost:6379/0"

Prod Environment

- APP_TITLE: "Personal Banking"
- DATABASE_URL: "postgresql://user:pass@prod-db:5432/banking"
- DEBUG_MODE: False
- REDIS_URL: "redis://prod-redis:6379/0"



Solution - Exercise #1

File: .env.dev

```
APP_ENV=dev
APP_TITLE="Personal Banking - DEV"
DATABASE_URL="sqlite:///./test.db"
DEBUG_MODE=True
```

File: .env.prod

```
APP_ENV=prod
APP_TITLE="Personal Banking"
DATABASE_URL="postgresql://user:pass@prod-db:5432/banking"
DEBUG_MODE=False
```

Setting Up Environment Variable Loading

```
# In your main application file
import os
from dotenv import load_dotenv

# Load the appropriate .env file based on the application environment.
# It reads the 'APP_ENV' environment variable. If it's not set, it defaults to 'dev'
env = os.getenv('APP_ENV', 'dev')

# Construct the filename (e.g., '.env.dev' or '.env.prod') and load it.
load_dotenv(f'.env.{env}')

# Now your environment variables from the specific .env file are loaded
# and ready to be used throughout the application.
```

Topic 2 - Typed Configuration Management

Plain strings are risky. Using typed configuration classes gives you auto-completion, static analysis, and runtime validation for free.

Python: pydantic-settings Implementation

Pydantic parses and validates environment variables into a clean, typed Python object with automatic validation.

```
from pydantic_settings import BaseSettings, SettingsConfigDict

class AppSettings(BaseSettings):
    # Load from .env file based on APP_ENV
    model_config = SettingsConfigDict(env_file=f".env.{os.getenv('APP_ENV', 'dev')}")

    APP_TITLE: str = "Default Title"
    DATABASE_URL: str
    DEBUG_MODE: bool = False
```

💡 **Pro Tip:** Use the `@lru_cache` decorator to ensure settings are loaded only once during the application lifecycle, improving performance.

Code Performance Improvement

Optimizing Application Settings with Caching

 **Objective:** Enhance performance by implementing efficient settings management

 **Solution:** Using @lru_cache decorator for single-load configuration

 **Benefit:** Settings loaded and validated only once at application startup

Implementation Steps

Step 1: Import the Decorator

```
from functools import lru_cache
```

Step 2: Define Settings Class

```
class AppSettings(BaseSettings):  
    """  
    A class to manage application settings using Pydantic.  
  
    It automatically loads and validates settings from an environment-specific  
    .env file (e.g., .env.dev, .env.prod).  
    """  
  
    model_config = SettingsConfigDict(  
        env_file=f".env.{os.getenv('APP_ENV', 'dev')}",  
        env_file_encoding='utf-8'  
    )
```

Step 3: Create Cached Getter Function

```
@lru_cache  
def get_settings() -> AppSettings:  
    """  
    Loads the settings and returns a cached instance.  
  
    Using @lru_cache ensures that the AppSettings class is instantiated  
    only once, preventing the .env file from being read on every call.  
    This is highly efficient.  
    """  
    return AppSettings()
```



Why This Change is Important

Performance

File I/O operations are eliminated after the first load. Settings access becomes instantaneous, significantly improving application response time, especially in web applications with multiple requests.

Consistency

Guarantees the entire application uses the exact same settings object, preventing potential inconsistencies across different modules or components.

Efficiency

Subsequent calls to `get_settings()` return the cached object immediately, avoiding repeated validation and file reads.

Topic 3 - Database Connection Pooling

Opening a database connection is expensive. A connection pool is a cache of database connections that are maintained so that they can be reused.

Best Practices (Universal):

Stateless Services: Your application instances should be stateless. Don't store session state in memory.

Externalise Configs: The database URL, pool size, and timeouts must be configured via environment variables, not hardcoded.

Backing Services: Treat the database as an attached resource. Your app should connect to it via a URL. This is a core tenet of the 12-Factor App.

Implementation with SQLAlchemy + FastAPI

```
# database.py from sqlalchemy import create_engine from sqlalchemy.orm import sessionmaker from .settings import
get_settings settings = get_settings() # SQLAlchemy automatically handles connection pooling engine = create_engine(
settings.database_url, pool_size=10, # Number of connections to keep in pool max_overflow=20, # Additional connections
beyond pool_size pool_pre_ping=True, # Validate connections before use pool_recycle=3600 # Recycle connections every
hour ) SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine) # Dependency for FastAPI def
get_db(): db = SessionLocal() try: yield db finally: db.close()
```


Our application needs a way to communicate with a database. But we need to do it smartly!

Problem:

Creating a new database connection for every request is very slow and inefficient.

Solution:

We use SQLAlchemy, a powerful Python tool, to manage a "pool" of connections.

 **Analogy:** Imagine a taxi stand. Instead of calling a new taxi from home every time (slow), you go to a stand where taxis are already waiting (fast). The connection pool is our taxi stand.

The Engine - The Power Plant

The first thing we create is an **Engine**.

- It's the **central source** of database connectivity for our app.
- It's created **only ONCE** when the application starts.
- The engine **manages the connection pool**.



Analogy: The engine is the taxi company's main dispatch center. It manages the entire fleet of available taxis.

```
# The Engine is created once and manages the connection pool.  
engine = create_engine(settings.DATABASE_URL, ...)
```

Understanding the Connection Pool

We configure our engine with pooling options for maximum performance.

- **pool_size=10:** Keeps 10 connections ready and waiting at all times.
(Our taxi stand has 10 taxis waiting)
- **max_overflow=20:** Allows up to 20 extra connections if we get a sudden rush of traffic.
(The dispatch can call 20 more taxis during peak hours)
- **pool_pre_ping=True:** Before giving you a connection, it quickly checks if it's still working.
(Checks if the taxi's engine is running before you get in)
- **pool_recycle=3600:** Replaces connections every hour to prevent them from becoming stale.

Understanding the session factory

While the engine is the manager, the **Session** is what you use to actually do things.

- A Session is a **short-lived conversation** with the database.
- You use a session to **query, add, update, or delete** data.
- **Rule:** Each API request gets its own, separate session.



Analogy: A Session is a single taxi ride. You get a taxi from the stand, tell the driver where to go (run a query), and then you get out (close the session).

```
# SessionLocal is a "factory" that creates new session objects.
SessionLocal = sessionmaker(
    autocommit=False,
    autoflush=False,
    bind=engine
)
```

The get_db Dependency - The FastAPI Way

How do we give each API request its own session and make sure it's closed?
With a **FastAPI Dependency**.

The **get_db** function is a special generator that:

- **Gets a Session:** Creates a new db session from our SessionLocal factory.
- **Provides the Session:** `yield db` hands this session over to your API path function.
- **Cleans Up:** The `finally` block guarantees that `db.close()` is called after your API function is done.

✓ **This simple pattern ensures our database connections are managed perfectly and safely.**

Question 1

Why is it a bad idea to call `create_engine` inside an API endpoint function?

Answer:

The `create_engine` function is an expensive operation that sets up the entire connection pool. It should only be run **once** when the application starts. Calling it for every request would be extremely inefficient and defeat the purpose of connection pooling.

Question 2

In the `get_db` function, what would happen if we didn't have the `try...finally` block and an error occurred in our API endpoint?

Answer:

If there were no **finally** block, the `db.close()` line would never be reached when an error occurs. The database connection would not be returned to the pool, leading to a "**leak**". Eventually, all connections in the pool would be used up, and the application would crash.

Question 3

What is the difference between `pool_size` and `max_overflow`?

Answer:

pool_size is the number of connections that SQLAlchemy keeps open and ready in the pool.

max_overflow is the number of additional, temporary connections it is allowed to create on top of the `pool_size` if the pool runs out during a traffic spike. These overflow connections are discarded more quickly once the demand decreases.

DB Connection Pooling – Key configuration parameters

Key Configuration Parameters

pool_size

Number of connections to maintain in the pool (default: 5)

max_overflow

Additional connections beyond pool_size when needed (default: 10)

pool_pre_ping

Test connections before use to avoid stale connections

pool_recycle

Seconds after which to recreate connections (prevents timeouts)

Tuning Our Connection Pool (The Smart Way!)



Think of it like a **high-end valet parking service** for a busy hotel.

Our goal: Get guests (API requests) a car (database connection) quickly and reliably.



pool_size

What it is:

Number of connections kept ready in the pool.

The Valet Analogy:

Valets standing at the door with keys in hand, ready to serve instantly. For normal traffic, we have **5 valets (default: 5)**.

Why it matters:

Ensures everyday requests are served immediately without waiting.



pool_pre_ping

What it is:

Tests a connection before use to ensure it's still alive.

The Valet Analogy:

Before handing keys, valet does a quick check: **Does the car start? Tires okay?** Prevents giving you a broken-down car (stale connection).

Why it matters:

Dramatically increases reliability by preventing errors from bad connections.



max_overflow

What it is:

Additional connections allowed when main pool is empty.

The Valet Analogy:

It's party night! All regular valets are busy. Manager calls **10 extra valets (default: 10)** to handle the rush. They go home after the party ends.

Why it matters:

Provides flexibility to handle traffic spikes without failing.



pool_recycle

What it is:

Time (seconds) after which a connection is automatically replaced.

The Valet Analogy:

Manager's rule: **"No car stays in service for more than 1 hour (3600 seconds)"**. After an hour, car is sent for refueling and a fresh one takes its place.

Why it matters:

Prevents connections from being dropped by database timeouts—a common error source.

Topic 4 - Caching Strategies

Caching reduces database load and improves API response times.

1. In-Memory Cache (Local)

- **Use Case:** Caching data that is frequently accessed but rarely changes within a single application instance.
- `lru_cache` is perfect for reference data that doesn't change often (account types, country codes, etc.)

```
# Using functools.lru_cache for simple caching
from functools import lru_cache
import time

@lru_cache(maxsize=128)
def get_account_type_details(account_type_id: int) -> dict:
    """
    Cache account type details - they rarely change.
    """

    print(f"Fetching account type {account_type_id} from DB...")

    # Simulate expensive DB call
    time.sleep(0.1)

    return {
        "id": account_type_id,
        "name": "Savings" if account_type_id == 1 else "Checking",
        "interest_rate": 0.02 if account_type_id == 1 else 0.01
    }
```

Questions and Answers for Understanding

Solidifying Your Caching Knowledge

? Question 1: In your own words, what is the main goal of caching?

💡 Answer: The main goal is to improve performance (speed) and reduce the load on backend resources like a database. It does this by storing copies of frequently accessed data in a location that is faster to access.

? Question 2: In the given code example, how can we prove that the cache is working after we call the function with the same input for a second time?

💡 Answer: The `print(f"Fetching account type {account_type_id} from DB...")` message will NOT appear on the second call. This proves that the function's internal code was not executed and the result was served directly from the cache.

? Question 3: What does LRU in `lru_cache` stand for, and what would happen if we set `maxsize=2` and then called the function with account IDs 1, 2, and then 3?

💡 Answer: **LRU** stands for **Least Recently Used**.

- ▶ When we call with ID **1**, it's cached. (Cache: **[1]**)
- ▶ When we call with ID **2**, it's cached. (Cache: **[1, 2]**)
- ▶ When we call with ID **3**, the cache is full (`maxsize=2`). The LRU policy kicks in and removes the "least recently used" item, which is **1**. The new cache will hold the results for IDs **2** and **3**. (Cache: **[2, 3]**)

Topic 4 - Caching Strategies

2. Distributed Cache (Redis)

- **Use Case:** Sharing cached data across multiple application instances or services. Essential for stateless, scaled applications.

```
def __init__(self, redis_url: str) -> None:
    self.redis = redis.from_url(redis_url, decode_responses=True)

async def get(self, key: str) -> Optional[dict]:
    """Retrieve and deserialize JSON data from cache."""
    value = await self.redis.get(key)
    return json.loads(value) if value else None

async def set(self, key: str, value: dict, ttl: int = 3600) -> None:
    """Serialize data to JSON and store in cache with TTL."""
    await self.redis.set(key, json.dumps(value), ex=ttl)

async def delete(self, key: str) -> None:
    """Remove key from cache."""
    await self.redis.delete(key)

# FastAPI Integration
async def get_customer_profile_cached(customer_id: int, cache: CacheService) -> dict:
    """
    Retrieve customer profile from cache if available,
    otherwise fetch from database and cache the result.
    """
    cache_key = f"customer:{customer_id}"

    # Try cache first
    if cached_data := await cache.get(cache_key):
        return cached_data

    # Cache miss - fetch from database
    profile = fetch_customer_from_db(customer_id)
    await cache.set(cache_key, profile, ttl=1800) # Cache for 30 minutes

    return profile
```

Distributed Caching with Redis

Improving Application Performance and Scalability



Learn how to optimize your applications with distributed caching

A comprehensive guide for developers

What's the Big Idea?

 Understanding the "Why"

The Problem: Isolated Caches

Imagine a popular food delivery app with multiple servers:

One Server (One Kitchen):

Has a small in-memory cache (fridge) for quick access to popular items

Multiple Servers (10 Kitchens):

Each has its own separate cache. They can't share data!

If Kitchen #1 prepares a popular item, Kitchen #2 doesn't know and has to make it all over again.

This is inefficient!

The Solution: Distributed Cache

⚡ Redis as a Central Warehouse

Redis: A Central, Super-Fast Warehouse

Think of Redis as a central refrigerated warehouse that all servers can access instantly.

Use Case Benefits:

- When one server prepares data, it stores it in Redis
- Any other server can access it immediately
- Perfect for **stateless applications** that need to scale
- Eliminates redundant database queries
- Dramatically improves response times

The Code: CacheService Class



Your Warehouse Manager

```
# Our "Warehouse Manager" - a service to interact with Redis
class CacheService:
    # Connects to the Redis warehouse when our app starts
    def __init__(self, redis_url: str) -> None:
        self.redis = redis.from_url(redis_url, decode_responses=True)

    # Gets an item from the warehouse
    async def get(self, key: str) -> Optional[dict]:
        """Retrieve and deserialize JSON data from cache."""
        value = await self.redis.get(key)
        # If we find the item, convert it from text back to a Python dictionary
        return json.loads(value) if value else None

    # Puts a new item into the warehouse with an expiry date
    async def set(self, key: str, value: dict, ttl: int = 3600) -> None:
        """Serialize data to JSON and store in cache with TTL."""
        # Convert the Python dictionary into a text string (JSON) to store it
        await self.redis.set(key, json.dumps(value), ex=ttl)

    # Removes an item from the warehouse
    async def delete(self, key: str) -> None:
        """Remove key from cache."""
        await self.redis.delete(key)
```

Understanding the Methods

How the CacheService Works

`__init__`: Sets up connection to Redis warehouse using a URL

`get(key)`: Retrieves an item using its unique name. Converts stored text back to Python dictionary using `json.loads`

`set(key, value, ttl)`: Stores an item in cache. Converts Python dictionary to text using `json.dumps`

TTL (Time To Live): Expiry timer in seconds. Default 3600 = 1 hour

`delete(key)`: Removes an item from the cache

Cache-Aside Pattern



FastAPI Integration

```
# This is our main application logic (e.g., in a web server)
async def get_customer_profile_cached(customer_id: int, cache: CacheService) -> dict:

    # 1. Create a unique key for this customer's data
    cache_key = f"customer:{customer_id}"

    # 2. Try to get the data from the cache first
    if cached_data := await cache.get(cache_key):
        print("☑ Cache HIT!")
        return cached_data

    # 3. If not in cache, it's a "Cache Miss". Fetch from the main database.
    print("✗ Cache MISS!")
    profile = fetch_customer_from_db(customer_id)

    # 4. Important: Store the newly fetched data in the cache for next time.
    # Set it to expire in 30 minutes (1800 seconds).
    await cache.set(cache_key, profile, ttl=1800)

    # 5. Return the profile data to the user.
    return profile
```

Request Flow Explained

Step-by-Step Journey

Step 1: Check Cache First

Application asks Redis: "Do you have profile for customer #123?"

Step 2a: Cache Hit

If Redis has the data, return it instantly. Super fast! No database query needed.

Step 2b: Cache Miss

If Redis doesn't have the data:

- Fetch from primary database (PostgreSQL, MySQL)
- Save a copy to Redis for next time
- Return data to user

Next request for customer #123 will be a Cache Hit!

Knowledge Check

Questions 1 & 2

Q1: What is the main problem that a distributed cache like Redis solves?

Answer: It solves the problem of data sharing and consistency between multiple application instances. Without a distributed cache, each server would have its own isolated cache, leading to redundant work and inefficiency.

Q2: What is a "cache miss" and what happens after it?

Answer: A cache miss occurs when data isn't found in the cache. After a miss: (1) The application fetches data from the database, (2) it stores the data back into the cache for future requests.

Knowledge Check

Questions 3 & 4

Q3: What is the purpose of TTL (Time To Live)?

Answer: TTL is an expiration timer for cached data. It ensures stale or outdated data is automatically removed. For example, if a customer updates their profile, the old cached data will eventually expire, forcing the application to fetch the new, updated profile.

Q4: Why use `json.dumps()` and `json.loads()` with Redis?

Answer: Redis is a key-value store that works with strings. `json.dumps()` serializes Python data structures into JSON strings for storage. `json.loads()` deserializes JSON strings from Redis back into Python dictionaries for use in the application.

Key Takeaways



Remember These Points

Why Redis?

Central shared cache for multiple application servers

Cache-Aside Pattern

Check cache first, fetch from DB on miss, store for next time

TTL is Essential

Prevents serving stale data by auto-expiring cached items

JSON Serialization

Convert between Python objects and Redis-compatible strings

Topic 5 - Secrets, Deployment & Logging

Secrets Management

The Problem: How do we provide production database passwords or API keys to our app without putting them in .env files or Git?

```
@lru_cache()
def get_secret(secret_name: str, region_name: str = "us-east-1"):
    """
    Retrieve a secret from AWS Secrets Manager.

    Args:
        secret_name (str): Name of the secret in AWS Secrets Manager.
        region_name (str): AWS region where the secret is stored.

    Returns:
        dict: Parsed JSON secret.
    """
    session = boto3.session.Session()
    client = session.client(service_name="secretsmanager",
region_name=region_name)
    response = client.get_secret_value(SecretId=secret_name)
    return json.loads(response["SecretString"])
```

```
# In your settings.py
class AppSettings(BaseSettings):
    """
    Application settings class.
    Uses local environment variables for development
    and AWS Secrets Manager for production.
    """

    app_env: str = os.getenv("APP_ENV", "dev")

    @property
    def database_password(self) -> str:
        if self.app_env == "prod":
            secrets = get_secret("banking-app-secrets")
            return secrets["db_password"]
        return os.getenv("DB_PASSWORD", "dev_password")
```

Secrets Management

Keeping Your Application Credentials Safe



**How to protect sensitive data like passwords,
API keys, and certificates**

The Problem

Question: Your app needs a database password. Where do you store it?



✗ The Dangerous Way (Hardcoding)

- ▶ Writing passwords directly in code
- ▶ Storing secrets in .env files committed to Git
- ▶ Keeping API keys in configuration files

The Problem

Why This is Risky

- ▶ Anyone with code access sees your passwords
- ▶ Secrets get leaked through version control
- ▶ Difficult to rotate or update credentials
- ▶ No audit trail of who accessed what

It's like leaving your house key under the doormat!  

The Solution

✓ Use a Dedicated Secrets Manager

Think of it as a highly secure digital vault

- ▶ Store secrets in a centralized, encrypted vault
- ▶ Give your app permission to fetch secrets at runtime
- ▶ Secrets never live in your codebase
- ▶ Easy to rotate and audit access



Example: AWS Secrets Manager, HashiCorp Vault, Azure Key Vault

Before vs After

✗ Insecure Approach

```
# config.py
DB_PASSWORD = "MyS3cr3tP@ss"

# This gets committed to Git! 🤖
# Everyone can see it!
# Hard to change without
# updating code everywhere
```

✓ Secure Approach

```
# Fetch from secure vault
secrets = get_secret(
    "banking-app-secrets"
)
db_password = secrets["db_password"]

# Secret never in code! 🎉
# Can rotate without code changes
# Audit who accessed when
```

The secret is fetched at runtime, not stored in code! 🔒

Implementation: Fetching Secrets

1 The Secret Retriever Function

```
def get_secret(secret_name, region_name="us-east-1"):
    """Fetch secret from AWS Secrets Manager"""

    # Connect to AWS
    session = boto3.session.Session()
    client = session.client(
        service_name="secretsmanager",
        region_name=region_name
    )

    # Request the secret
    response = client.get_secret_value(SecretId=secret_name)

    # Return the secret value
    return json.loads(response["SecretString"])
```

What it does: Acts as a trusted messenger that goes to the vault, asks for a specific secret by name, and brings it back to your application.


Implementation: Smart Configuration

2 Environment-Aware Settings

```
class AppSettings:
    def __init__(self):
        # Check environment: "dev" or "prod"
        self.app_env = os.getenv("APP_ENV", "dev")

    @property
    def database_password(self) -> str:
        if self.app_env == "prod":
            # Production: Fetch from secure vault
            secrets = get_secret("banking-app-secrets")
            return secrets["db_password"]
        else:
            # Development: Use local env variable
            return os.getenv("DB_PASSWORD", "dev_password")
```

- **Production (Live): Fetches from AWS Secrets Manager**
- **Development (Local): Uses a simple local password**

 **Key Concept: Different sources for different environments!**

The Big Picture

Security Benefits

- ▶ Production secrets never appear in code or Git
- ▶ Centralized management and rotation
- ▶ Access control and audit logging
- ▶ Encryption at rest and in transit


Developer Benefits

- ▶ Easy local development with simple passwords
- ▶ No risk of accidentally committing secrets
- ▶ Change secrets without code deployment
- ▶ Clear separation between environments

Security + Convenience = ✨ Best Practice!

Key Takeaways

- ▶ **Never hardcode secrets** in your code or commit them to Git
- ▶ **Use a secrets manager** for production environments
- ▶ **Fetch secrets at runtime**, not at build time
- ▶ **Separate dev and prod** configurations clearly
- ▶ **Rotate secrets regularly** using your secrets manager

Remember: Your secrets are only as secure as how you manage them! 

Topic 5 - Secrets, Deployment & Logging

Deployment with Docker + Gunicorn

Dockerfile

```
FROM python:3.11-slim

# Set working directory inside container
WORKDIR /app

# Copy dependency list
COPY requirements.txt .

# Install dependencies
RUN pip install --no-cache-dir -r requirements.txt

# Copy application code
COPY . .

# Use gunicorn with uvicorn workers for production
CMD ["gunicorn", "main:app", "-w", "4", "-k", "uvicorn.workers.UvicornWorker",
    "-b", "0.0.0.0:8000"]
```

Deploying a Python App with Docker & Gunicorn

A Comprehensive Guide for Freshers

Objective

Understand how a Dockerfile works by breaking down each instruction line by line.



From Blueprint to Running Container



What is a Dockerfile?

A Dockerfile is a text document that contains all the commands a user could call on the command line to assemble an image. It's like a **recipe or a blueprint** for creating a Docker container. 📄 ➡ 📦

Goal

To package our Python application, its dependencies, and a server into a single, isolated, and runnable unit called a **container**.

Why?

This ensures that our application runs the same way everywhere—on a developer's laptop, a testing server, or in production.

"It works on my machine" is no longer a problem! ✅

The Blueprint - Line by Line (Part 1)



Step 1: FROM - The Foundation

```
FROM python:3.11-slim
```

What it does: This is the foundation of our image.

Explanation: Every Docker image starts from a base image. Here, we're using an official image from Docker Hub that already has Python 3.11 installed. The `-slim` version is a lightweight, minimal version, which is great for keeping our final image size small.



Step 2: WORKDIR - Set the Working Directory

```
WORKDIR /app
```

What it does: Sets the working directory inside the container.

Explanation: This command is like running `mkdir /app` and then `cd /app`. All subsequent commands (COPY, RUN, CMD) will be executed from within this `/app` directory inside our container.

The Blueprint - Line by Line (Part 2)

Step 3: Installing Dependencies

```
# Copy dependency list
COPY requirements.txt .

# Install dependencies
RUN pip install --no-cache-dir -r requirements.txt
```

COPY requirements.txt .:

This copies the `requirements.txt` file (which lists all the Python libraries your app needs) from your computer into the `/app` directory inside the container. We copy this file first for a smart reason called

layer caching, which makes future builds faster! ⚡

The Blueprint - Line by Line (Part 2)

RUN pip install ...:

- This command is executed during the build process
- It reads the `requirements.txt` file and installs all necessary libraries (like FastAPI, requests, etc.)
- `--no-cache-dir`: Prevents pip from storing a cache, helping keep the final image size smaller

The Blueprint - Line by Line (Part 3)

Step 4: Copy Application Code

```
# Copy application code  
COPY . .
```

What it does: Copies your application files.

Explanation:

- The first `.` means "everything in the current directory on my local machine" (where the Dockerfile is)
- The second `.` means "the current working directory inside the container" (which we set to `/app`)
- This command copies all your Python files (e.g., `main.py`, `models.py`, etc.) into the container

The Blueprint - The Final Command!

Step 5: CMD - Bringing Your App to Life

```
CMD ["gunicorn", "main:app", "-w", "4", "-k", "uvicorn.workers.UvicornWorker", "-b",  
"0.0.0.0:8000"]
```

What it does: This specifies the default command to run when the container starts.

Breaking it down:

- **gunicorn:** A production-ready web server. The simple server you use for local development isn't safe or powerful enough for real users
- **main:app:** Tells Gunicorn to look in the `main.py` file for a Python object named `app`
- **-w 4:** Sets the number of worker processes to 4. Like opening four checkout counters instead of one!
- **-k uvicorn.workers.UvicornWorker:** Uses Uvicorn workers designed for high performance with async code
- **-b 0.0.0.0:8000:** Binds the server to listen on all network interfaces on port 8000

Summary - From Code to Container

What did we just do?



FROM: Started with a clean Python environment



WORKDIR: Created a home for our app inside the container



COPY/RUN: Installed our app's dependencies

Summary - From Code to Container



COPY: Added our own Python source code



CMD: Defined the command to launch our app with a production server

The result is a self-contained, portable image ready for deployment!



Knowledge Check!

Time to test your understanding:

Question 1: What is the main purpose of a Dockerfile?

Answer 1:

A Dockerfile is a blueprint or a set of instructions used to automatically build a Docker image. It packages an application with all its dependencies and configurations into a standardized, portable unit.



Knowledge Check!

Time to test your understanding:

Question 2: What is the difference between the RUN and CMD instructions?

Answer 2:

- **RUN** executes a command during the build process to create the image (e.g., `pip install`). The results are saved as a layer in the final image.
- **CMD** provides the default command that is executed when a container is started from the image (e.g., starting the web server).



Knowledge Check!

Time to test your understanding:

Question 3: In the CMD line, why do we use a server like Gunicorn instead of the development server that comes with Flask or FastAPI?

Answer 3:

Development servers are designed for convenience during coding, not for handling real-world traffic. Gunicorn is a production-grade server that is more secure, stable, and can handle many concurrent connections efficiently, which is essential for a live application.



Knowledge Check!

Time to test your understanding:

Question 4: What does `-w 4` in the Gunicorn command signify, and why is it important for production?

Answer 4:

`-w 4` tells Gunicorn to start 4 worker processes. This is crucial for production because it allows the application to handle multiple user requests simultaneously, leading to better performance and responsiveness. A single worker could only process one request at a time.

Topic 5 - Secrets, Deployment & Logging

Deployment with Docker + Gunicorn

Structured Logging

```
import structlog
import sys

# Configure structured logging
structlog.configure(
    processors=[
        structlog.processors.TimeStamper(fmt="ISO"),
        structlog.dev.ConsoleRenderer()
    ],
    wrapper_class=structlog.make_filtering_bound_logger(20), # INFO level
    logger_factory=structlog.WriteLoggerFactory(file=sys.stdout),
    cache_logger_on_first_use=True,
)

# Create logger instance
logger = structlog.get_logger()

# Usage example
logger.info("Customer login", customer_id=123, ip="192.168.1.1")
```

What is Structured Logging? 🤔

Imagine you're writing a diary.

Unstructured Logging

Like scribbling a quick, messy note:

```
"A customer logged in. It was user  
123. Their IP was 192.168.1.1."
```

It's okay for you to read, but a computer would have a hard time understanding the different pieces of information.

Structured Logging

Like filling out a neat form in your diary:

- **Event:** Customer login
- **Customer ID:** 123
- **IP Address:** 192.168.1.1

This "form-filling" approach is what structured logging does for our applications. It logs events as **key-value pairs**, making them easy for both humans and machines to read and analyze. This is incredibly powerful for searching, filtering, and debugging! 🔍

Let's Break Down the Code

1. Configuration `structlog.configure`

This is the one-time setup where we tell `structlog` **how** we want our logs to look and behave.

Processors:

These are a series of steps that each log message goes through before it's printed.

- `Timestamp("ISO")` : **Adds the current time** to every log message. Super useful to know *when* something happened.
- `ConsoleRenderer()` : **Makes the log pretty** and colorful for us to read in the terminal.

Other Configuration:

- `wrapper_class=...make_filtering_bound_logger(20)` : This **sets the minimum log level**. `20` corresponds to `INFO`. This means it will show INFO, WARNING, ERROR messages, but ignore more detailed DEBUG messages.
- `logger_factory=...WriteLoggerFactory(file=sys.stdout)` : This tells the logger **where to write the logs**. `sys.stdout` means it will print directly to your console/terminal.

Let's Break Down the Code

2. Creating the Logger

```
logger = structlog.get_logger()
```

This line is simple: it just grabs the logger we just configured so we can start using it.

3. Using the Logger

This is where the magic happens! We're creating our first structured log entry.

```
logger.info("Customer login", customer_id=123, ip="192.168.1.1")
```

- `"Customer login"` : This is the main, human-readable message about the event.
- `customer_id=123` , `ip="192.168.1.1"` : These are our **structured key-value pairs** . They provide context to the event in a clean, machine-readable format.

Output Example:

```
2025-10-02T22:35:12.123Z [info ] Customer login customer_id=123 ip=192.168.1.1
```

See? It's clean, timestamped, and has our key-value pairs right there!

Check Your Understanding: Q&A

Questions

Q1: What is the main advantage of structured logging over just using a `print()` statement?

Machine-Readability: Structured logs use a consistent key-value format. This makes them easy to automatically parse, search, and analyze with monitoring tools, which is very difficult to do with simple print statements or unstructured text.

Q2: In the code example, what part determines that the logs will be printed to the console?

The `logger_factory=structlog.WriteLoggerFactory(file=sys.stdout)` line tells the logger to write to `sys.stdout`, which is the standard output stream (your console).

Check Your Understanding: Q&A

Questions

Q3: What would you change if you wanted to log an error message instead of an informational one?

You would change the method name from `.info()` to `.error()`. For example: `logger.error("Failed login attempt", user="admin")`

Q4: Why is the Timestamp processor useful?

The `Timestamp` processor automatically adds the exact time an event occurred to each log entry. This is critical for debugging and understanding the sequence of events in your application.

Best Practices & Pro-Level Tips

FastAPI Dependency Injection

```
# settings_dependency.py
from functools import lru_cache
from fastapi import Depends
from .settings import AppSettings

@lru_cache()
def get_settings() -> AppSettings:
    return AppSettings()

# In your route handlers
from fastapi import FastAPI

@app.get("/api/info")
async def get_app_info(settings: AppSettings = Depends(get_settings)):
    return {
        "title": settings.app_title,
        "environment": settings.app_env,
        "debug": settings.debug_mode
    }
```

FastAPI's Magic Wand: Dependency Injection ✨

Understanding the Power of Clean, Reusable Code

What You'll Learn:

- How Dependency Injection simplifies your code
- Real-world analogies to understand the concept
- Practical implementation examples
- Testing benefits and best practices

The Problem & Solution

The Problem: The Repetitive Chef 🍳

Imagine you're a chef (our API endpoint). For every dish you cook, you first have to run to the store to get ingredients (like database connections or settings). This is slow and repetitive!

The Solution: Your Personal Helper 🎯

What if you had a helper (a dependency) who brings you the exact ingredients you need, right when you need them? You can just focus on cooking!

That's Dependency Injection (DI): A system where FastAPI "injects" or provides your functions with the data and objects they need to work.

The Consumer & Injection

2. The Consumer (Our "Chef")

```
@app.get("/api/info")
async def get_app_info(...):
    return {
        "title": settings.app_title,
        "environment": settings.app_env,
        "debug": settings.debug_mode
    }
```

3. The Injection (The "Hand-off")

```
... (settings: AppSettings = Depends(get_settings)):
```

Depends(): Special instruction telling FastAPI to run something first

Depends(get_settings): Specifies which function to run

settings: AppSettings = ...: FastAPI passes the result to our endpoint

Let's Check Your Understanding! 🤔

Q1: In our chef analogy, what does `@lru_cache()` represent?

A: It's like the helper having a perfect memory. After going to the store once for the ingredients (creating the settings object), they keep them ready on the counter and don't need to go to the store again for the rest of the day.

Q2: What would happen if we removed `@lru_cache()`?

A: A new `AppSettings` object would be created for every single request to `/api/info`. This is very inefficient, as it might involve re-reading files or environment variables unnecessarily, slowing down our app.

Let's Check Your Understanding! 🤔

Q3: What is the main job of Depends?

A: To tell FastAPI that our endpoint function has a dependency. FastAPI then runs the dependency function and "injects" the result as a parameter into our endpoint function.

Q4: How does this pattern make testing easier?

A: FastAPI allows you to easily "override" a dependency. You can tell your test to use a fake, temporary AppSettings object instead of the real one. This means you can test your endpoint's logic with different configurations without changing any code.

Q5: Could we use the same Depends(get_settings) in another endpoint?

A: Absolutely! Any endpoint that needs the app settings can simply ask for it in the same way. The get_settings logic is defined in one place and reused everywhere. This is the DRY principle: Don't Repeat Yourself.

Key Takeaways

Key Takeaways

- ✓ Clean, testable, and reusable code
- ✓ Improved performance with caching
- ✓ Separation of concerns
- ✓ Easy to maintain and scale

Best Practices & Pro-Level Tips

Environment-Specific Services

```
# services.py - Factory pattern for different environments
```

```
# Service factory mapping
```

```
SERVICE_MAP = {  
    "prod": lambda s: AWSEmailService(s.aws_access_key),  
    "staging": lambda s: SendGridEmailService(s.sendgrid_api_key)  
}
```

```
def create_email_service(settings: AppSettings):  
    """Create email service based on environment"""  
    service_creator = SERVICE_MAP.get(settings.app_env, lambda _: MockEmailService())  
    return service_creator(settings)
```

```
# FastAPI dependency
```

```
def get_email_service(settings: AppSettings = Depends(get_settings)):  
    return create_email_service(settings)
```

Smartly Managing Services for Different Environments

Goal: Learn how to write flexible code that behaves differently in development, staging, and production

Key Pattern: The Factory Pattern



Building Maintainable and Scalable Applications

The Problem We're Solving 🤔

Scenario: Our application needs to send emails

- **Production:** Use a real, powerful email service like Amazon AWS SES
- **Staging:** Use a different service like SendGrid (cheaper or for testing)
- **Development:** Use a fake/mock service that just prints to screen (no real emails!)

Challenge: How do we manage these different services without messy code?

The Bad Way vs The Smart Way

✗ The Messy Way

```
if settings.app_env == "prod": service = AWSEmailService() elif  
settings.app_env == "staging": service = SendGridEmailService() else:  
service = MockEmailService()
```

This gets repeated everywhere you need to send an email. Hard to maintain!

✓ The Smart Way

Create a central "factory" that builds the correct service automatically!

The Code: Part 1 - The "Menu"

```
# Service factory mapping SERVICE_MAP = { "prod": lambda s:  
    AWSEmailService(s.aws_access_key), "staging": lambda s:  
    SendGridEmailService(s.sendgrid_api_key) }
```

Think of SERVICE_MAP as a menu of services:

- **Keys:** Environment names ("prod", "staging")
- **Values:** Lambda functions - small "recipes" to create services
- **lambda s:** A function that takes settings as input
- **Returns:** A newly created email service instance

The Code: Part 2 - The "Chef"

```
def create_email_service(settings: AppSettings): """Create email service based
on environment""" service_creator = SERVICE_MAP.get( settings.app_env, lambda _:
MockEmailService() ) return service_creator(settings)
```

Think of this function as the Chef:

- **settings.app_env:** The "order" (which environment?)
- **SERVICE_MAP.get(...):** Look up the order in the menu
- **Default value:** If environment not found, use MockEmailService (safety net!)
- **Returns:** The ready-to-use service object

The Code: Part 3 - The "Waiter"

```
# FastAPI dependency def get_email_service( settings: AppSettings =  
Depends(get_settings) ): return create_email_service(settings)
```

This is specific to FastAPI web framework:

- Acts as a bridge between FastAPI and our factory
- **Depends:** FastAPI's dependency injection system
- Automatically provides the email service wherever needed in the app
- Calls our "Chef" to get the correct service and "serves" it

Why Is This So Good?

- ✓ **Clean & Centralized:** All service selection logic in one place. No scattered if/else blocks
- ✓ **Flexible:** Add new environments by adding just one line to SERVICE_MAP
- ✓ **Safe for Developers:** MockEmailService as default prevents accidental real emails during development
- ✓ **Testable:** Easy to provide mock services for automated testing
- ✓ **Maintainable:** Changes in one place, not scattered throughout codebase

This is clean, scalable, and maintainable code!

Key Questions to Test Understanding

Q1: What is the main purpose of the SERVICE_MAP dictionary?

It maps environment names to functions that create the appropriate email service

Q2: What happens if settings.app_env contains "development"?

MockEmailService is created (default behavior)

Q3: Why use .get() instead of direct dictionary access?

Prevents KeyError crashes by providing a default value

Adding a New Environment

Question: How would you add support for a "user_testing" environment using Mailgun?

```
SERVICE_MAP = { "prod": lambda s: AWSEmailService(s.aws_access_key), "staging":  
lambda s: SendGridEmailService(s.sendgrid_api_key), "user_testing": lambda s:  
MailgunEmailService(s.mailgun_api_key) # ↑ Just add this one line! }
```

That's it! One line = New environment support

No other code changes needed. This is the power of the Factory Pattern!

Best Practices & Pro-Level Tips

Health Check Endpoint

```
@app.get("/health")
async def health_check(db: Session = Depends(get_db)):
    try:
        db.execute("SELECT 1") # Database health check
        await redis.from_url(settings.redis_url).ping() # Redis health check

        return {
            "status": "healthy",
            "timestamp": datetime.utcnow().isoformat(),
            "services": {"database": "up", "redis": "up"}
        }
    except Exception as e:
        return {"status": "unhealthy", "error": str(e)}
```


Conclusion & Key Takeaways

Externalize Everything

Configuration, including URLs, pool sizes, and feature flags, must live outside your code.

Embrace Typed Configs

Prevent bugs and improve developer experience with classes like `pydantic-settings` or `@ConfigurationProperties`.

Pool Connections

Never manage database connections manually. Rely on battle-tested pooling libraries.

Cache Intelligently

Use a combination of local and distributed caching to drastically improve performance.

Deploy with Containers

Use Docker + Gunicorn with Uvicorn workers for production-ready Python applications.

Log to stdout

Use structured logging with libraries like `structlog` for machine-readable logs.