1. Layered architecture

Real world examples

The most well-known example is the three-tier architecture, which divides an application into three separate layers: the data layer, the business layer, and the presentation layer. Each layer is in charge of a distinct set of duties. An e-commerce website is an excellent illustration of the three-tier architecture. Web pages and APIs that let users examine product details and make purchases make up the presentation layer.

While the data layer stores and retrieves data from the database, the business layer manages business logic, including order totals and price calculations.

when would it be most appropriate to use layered architecture?

A three-tier strategy is used by enterprise software solutions, such as cloud-based apps, customer relationship management (CRM), and enterprise resource planning (ERP). Web apps: This architecture is used by the majority of contemporary web and mobile applications.

Budget or Time Restrictions: Layered architecture is well recognised and easy to implement. It is appropriate for projects with tight budgets and deadlines because it does not include the complexity of distributed systems.

Isolated Changes: This architecture is perfect if the majority of changes are limited to particular layers (such as UI redesigns or database migrations) since it reduces the influence of changes on other layers.

Alignment of Team Structure: works effectively with technical expertise-based teams, such as database, front-end, and back-end teams.

Conway's Law, which describes this alignment, guarantees effective cooperation.

General-Purpose Applications: When it's unclear which architecture is ideal for the application, this is a good place to start. It offers a flexible, comprehensible framework.

Why would it be more appropriate to use layered architecture of a theree-tier architecture in iterprise software solutions?

This division of responsibilities encourages modularity and makes development, maintenance, and scaling easier.

2. Repository architecture

Real world examples

Using the Repository Pattern in Enterprise Architecture

Situation: Assume we are creating an API for an enterprise's product management. We will be able to abstract database interactions using the Repository Pattern, which will facilitate future scalability and maintenance of the application.

Here, we'll use SQLAlchemy and FastAPI to create a simple CRUD application for managing products in the database.

The following pattern is incorporated into the data persistence layer of an enterprise architecture, which can be a component of a hexagonal architecture (sometimes called "ports and adapters") or a layered architecture.

[API / Business Logic] → [Repository (Data Access Interface)] → [Database]

The Code

2.1. Model of Data First, we use SQLAlchemy to create the product model that will represent our database table.

from sqlalchemy import Column, Integer, String, Float from sqlalchemy.ext.declarative import declarative_base

Base = declarative_base()

class Product(Base): **tablename** = 'products'

```
id = Column(Integer, primary_key=True, index=True)
name = Column(String, index=True)
price = Column(Float)
```

Justification:

Product: Indicates a product in the database, each of which contains a name, price, and ID. Base: We use declarative_base() to build the foundation for our data models.

2.2. Repository

Building the repository that would oversee CRUD activities for the goods is the next stage. By serving as an abstraction layer, this repository keeps the business logic separate from the database queries.

from sqlalchemy.orm import Session from models import Product

class ProductRepository: def **init**(self, db: Session): self.db = db

```
def create(self, name: str, price: float) -> Product:
    product = Product(name=name, price=price)
    self.db.add(product)
    self.db.commit()
    self.db.refresh(product)
    return product

def get_by_id(self, product_id: int) -> Product:
    return self.db.query(Product).filter(Product.id ==
product_id).first()

def update(self, product_id: int, name: str, price: float) -> Product:
    product = self.get_by_id(product_id)
    if product:
        product.name = name
        product.price = price
        self.db.commit()
        self.db.refresh(product)
    return product

def delete(self, product_id: int) -> None:
    product = self.get_by_id(product_id)
    if product:
        self.db.delete(product)
        self.db.commit()
```

Explanation: CRUD Methods:

create: Creates a new product in the database. get_by_id: Retrieves a product by its ID. update: Updates an existing product. delete: Deletes a product from the database. This repository layer becomes a reusable service that can be easily adapted for different needs in the application.

2.3. API - Integration with FastAPI Finally, we integrate this repository into an API built with FastAPI so we can interact with products via HTTP requests.

```
from fastapi import FastAPI, Depends from sqlalchemy import create_engine from sqlalchemy.orm import sessionmaker from repositories.product_repository import ProductRepository from models import Product, Base

DATABASE_URL = "sqlite:///./test.db" engine = create_engine(DATABASE_URL, connect_args={"check_same_thread": False}) SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)

app = FastAPI()

def get_db(): db = SessionLocal() try: yield db finally: db.close()

@app.post("/products/") def create_product(name: str, price: float, db: Session = Depends(get_db)): repo = ProductRepository(db) return repo.create(name, price)

@app.get("/products/{product_id}") def get_product(product_id: int, db: Session = Depends(get_db)): repo = ProductRepository(db) return repo.get_by_id(product_id)

@app.put("/products/{product_id}") def update_product(product_id: int, name: str, price: float, db: Session = Depends(get_db)): repo = ProductRepository(db) return repo.update(product_id, name, price)

@app.delete("/products/{product_id}") def delete_product(product_id: int, db: Session = Depends(get_db)): repo = ProductRepository(db) repo.delete(product_id) return {"message": "Product deleted successfully"}
```

Explanation:

We use FastAPI to create API routes that manage products. Each route calls the appropriate method from the repository to perform the corresponding operation.

Why would it be more appropriate to use repository architecture in Enterprise Architecture?

The Repository Pattern is essential for any enterprise application that seeks a clean, scalable, and maintainable architecture. Its ability to decouple data access from business logic improves flexibility and ease of maintenance for large systems. Implementing this pattern not only enhances the structure of the application but also enables more effective testing and reduces long-term complexity.

3. Client-server architecture

Real world examples

Web browsers like Chrome and Safari work as clients, asking web servers for pages. Mobile apps like Instagram and Amazon connect to servers to pull in user data, posts, or product listings. Online games also use the client-server setup to keep player actions, progress, and scores in sync with central servers.

when would it be most appropriate to use client-server architecture?

With a client-server architecture, you can send and receive emails, host websites, support cloud systems, and access online applications.

Why would it be more appropriate to use client-server architecture in web browsers, mobile apps and in online games?

Scalability: Easily handles multiple users.

Security: Centralized control over data. Efficiency: Reduces client processing load. Maintainability: Modular structure simplifies updates.

4.Pipe filter architecture

Real world examples

Unix/Linux Command Line: Shell pipelines in Unix and Linux environments (such as Bash) enable users to connect multiple commands in sequence using pipe operators. For instance, the command cat file.txt | grep "pattern" | sort | uniq passes a file through a chain of operations that filter content, arrange it alphabetically, and eliminate duplicate entries.

Compilers: The GNU Compiler Collection (GCC) transforms source code through multiple phases including preprocessing, syntactic analysis, optimization, and machine code generation. Each phase functions as a filter, converting the code from its current state to the next required format.

Data Processing Frameworks: Apache Flink and Apache Storm handle real-time data streams by processing them through various stages. Each processing component (such as map, filter, and reduce operations) serves as a filter within the data pipeline. Apache NiFi is a data integration platform that automates data movement between different systems, utilizing processors that act as filters to modify, direct, and control data flow.

Media Processing: GStreamer is a multimedia framework that handles audio and video streams by routing them through a sequence of processing elements (filters) that perform operations like decoding, encoding, and applying various effects.

Web Development Frameworks: In Express.js (Node.js), middleware functions serve as filters that handle HTTP requests and responses. Operations such as logging, user authentication, and request parsing are managed by distinct middleware components. Following a similar pattern to Express.js, ASP. NET Core employs middleware components that process HTTP requests through a sequential pipeline.

ETL (Extract, Transform, Load) Tools: Talend is an ETL platform that utilizes a sequence of components to retrieve data from multiple sources, modify it based on business logic, and transfer it to destination systems. Apache Hop is an open-source data integration solution that processes data through sequential transformation steps, facilitating sophisticated ETL workflows.

when would it be most appropriate to use pipe filterr architecture?

Sequential Data Transformation: This architecture is ideal when data must flow through multiple distinct processing stages in a predetermined sequence, with each stage modifying the data before passing it forward. A typical example would be an image processing workflow that sequentially resizes images, applies visual filters, adjusts brightness levels, and performs compression.

Stream Processing: The pattern excels when dealing with continuous data flows that demand real-time or near-real-time analysis and processing. Common applications include log analysis platforms, live analytics dashboards, and systems that process sensor data streams.

Independent, Reusable Components: This approach works best when each processing step operates autonomously without dependencies on other components, can be repurposed across different processing workflows, and allows for isolated unit testing. ETL (Extract, Transform, Load) operations exemplify this, where data extraction, transformation logic, and loading mechanisms remain separate and self-contained.

Data Processing with Variable Complexity: The architecture suits scenarios where different data elements may follow distinct processing routes or require varying numbers of transformation steps. Compiler systems demonstrate this well, as they process code through stages like preprocessing, lexical analysis, syntax parsing, optimization routines, and final code generation.

Parallel Processing Opportunities: When filters can operate on data simultaneously rather than strictly sequentially, this pattern enables improved throughput and enhanced system performance. Batch processing systems leverage this capability by processing multiple data segments concurrently across different pipeline stages.

Easy Maintenance and Extension: This pattern shines when systems require the ability to introduce new processing stages without altering existing functionality, swap out or update individual filters independently, and reconfigure processing workflows with minimal effort. Media transcoding systems benefit from this flexibility, allowing new encoding formats or compression algorithms to be integrated as additional filter components without disrupting the existing pipeline.

Why would it be more appropriate to use pipe filter architecture?

Modularity and separation of concerns Flexibility in reconfiguring pipelines Easy parallel and distributed processing Simplified testing and debugging Natural support for streaming data

5.

Among the four architectures mentioned, Client-Server Architecture and Layered Architecture are the two that frequently incorporate combinations of multiple architectural patterns.

Client-Server Architecture typically integrates Layered Architecture on the server side, organizing code into distinct layers such as presentation, business logic, and data access. It also commonly employs the Repository Pattern for managing data operations and may utilize Pipe and Filter concepts for processing requests and responses through middleware sequences.

The advantages of Client-Server Architecture include centralized control over data and business logic which simplifies updates and maintenance, independent scalability of server resources, consolidated security management at the server level, efficient sharing of resources among multiple clients, flexibility in using different platforms for clients and servers, and reduced maintenance requirements for thin clients.

However, it has disadvantages including vulnerability to complete service disruption if the server fails, heavy reliance on network connectivity and performance, risk of server congestion under high client loads, expensive scaling costs for infrastructure, communication delays particularly for remote clients, and increased complexity when managing distributed server environments.

Layered Architecture also combines multiple patterns, frequently incorporating the Repository Pattern within its data access layer to abstract database interactions, implementing Client-Server concepts where the presentation layer functions as a client to the business logic layer acting as a server, and occasionally using Pipe and Filter mechanisms within layers for data validation and transformation processes.

Its advantages include clear separation of responsibilities across layers promoting organized and maintainable code, reusability of layers across different applications or ability to swap implementations, independent testing capabilities for each layer using mock objects, ease of maintenance since changes in one layer typically don't propagate to others when interfaces are properly defined, deployment flexibility allowing layers to run on separate machines, and adherence to well-established best practices and conventions.

The disadvantages encompass performance penalties from data traversing multiple layers causing processing delays and potential bottlenecks, tendency toward over-engineering for straightforward applications that don't require strict layer separation, cascading modifications when data structures change affecting multiple layers, challenges in maintaining strict layer boundaries when implementing cross-cutting functionalities, increased difficulty in debugging as issues must be traced through multiple layers, and greater upfront investment in design and implementation to establish the proper architecture.

Both Client-Server and Layered Architecture regularly combine multiple architectural patterns to capitalize on their complementary benefits, with Client-Server frequently incorporating layered design on the server side alongside repository patterns, while Layered Architecture commonly integrates repository patterns for data access and may employ client-server principles between layers, though these combinations provide enhanced flexibility and separation of concerns at the cost of increased complexity and potential performance trade-offs.