**✅ What is Software Architecture?**

**🔹 Definition:**

**Software Architecture** is the **high-level structure** of a software system. It defines how **software components** (like modules, services, databases, UI) interact with each other.

It answers **"How will the system be built?"** rather than just **"What will it do?"**

**🧱 Core Elements of Software Architecture**

| **Component** | **Description** |
| --- | --- |
| **Modules** | Logical units like UI, database access, business logic |
| **Layers** | Organize code into tiers (presentation, business, data) |
| **Interfaces (APIs)** | Define how components talk to each other |
| **Deployment Strategy** | Cloud/on-premise, microservices, monolithic |
| **Non-functional Concerns** | Performance, security, scalability, reliability |

**🧭 Why is Software Architecture Important?**

* Ensures **scalability** (can handle growing users/data)
* Improves **maintainability** (easier to fix/upgrade)
* Provides **security** and **reliability**
* Enables **team collaboration** (clear component responsibilities)
* Guides **technology selection** (database, messaging, frontend, backend)

**🎯 Types of Software Architecture (with Real-time Examples)**

**🔹 1. Layered Architecture (N-tier Architecture)**

**Structure**: Divides application into layers like Presentation, Business Logic, Data Access.

**✅ Used In:**

* Banking systems
* E-commerce websites

**💼 Example:**

**Online Banking System**

| **Layer** | **Responsibility** |
| --- | --- |
| Presentation Layer | UI to check balance and transfer money |
| Business Layer | Logic for interest calculation, transaction limits |
| Data Access Layer | Fetch data from Oracle/PostgreSQL |
| Database | Stores accounts, customer data |

**🔹 2. Client-Server Architecture**

**Structure**: A server serves multiple clients; used in desktop or web apps.

**✅ Used In:**

* ATM networks
* Online food ordering systems

**💼 Example:**

**ATM Machine System**

* **Client**: ATM Machine GUI
* **Server**: Core banking server that authenticates and processes transactions

**🔹 3. Microservices Architecture**

**Structure**: Application is split into **small, independent services**, each handling one function.

**✅ Used In:**

* E-commerce platforms (like Amazon)
* Telecom billing and customer care

**💼 Example:**

**E-commerce System**

| **Service** | **Responsibility** |
| --- | --- |
| Product Service | Manages product catalog |
| Order Service | Manages orders |
| Payment Service | Handles payments securely |
| User Service | Manages registration, login, addresses |

Each microservice communicates over REST APIs or messaging queues (like Kafka or RabbitMQ).

**🔹 4. Event-Driven Architecture**

**Structure**: System reacts to events (e.g., user clicks, data updates, notifications)

**✅ Used In:**

* Telecom recharge systems
* Fraud detection in banking
* Stock trading platforms

**💼 Example:**

**Telecom System**

* Event: User recharges a prepaid number
* Triggered services:
  + Balance gets updated
  + SMS confirmation sent
  + Transaction recorded for billing

**🔹 5. Service-Oriented Architecture (SOA)**

**Structure**: Similar to microservices but heavier; services are reusable and communicate over protocols like SOAP.

**✅ Used In:**

* Large government or banking integrations
* Insurance claims processing systems

**💼 Example:**

**Insurance Claim System**

* Authentication Service
* Policy Service
* Claim Processing Service
* Payment Settlement Service

Each department can use these reusable services through a shared service bus (ESB).

**🛠️ Non-Functional Concerns in Architecture**

| **Concern** | **Why It's Important** |
| --- | --- |
| **Scalability** | System grows with user demand |
| **Security** | Protect data (encryption, authentication) |
| **Availability** | 24/7 uptime (failover, backups) |
| **Maintainability** | Easy to change and deploy |
| **Performance** | Fast response time |

**🏁 Summary Table**

| **Architecture Style** | **Used In** | **Key Example** |
| --- | --- | --- |
| Layered (N-tier) | Banking, ERP | Internet Banking, Loan Management System |
| Client-Server | ATM, POS Systems | ATM withdraws from a central server |
| Microservices | E-commerce, Telecom | Amazon-style modular services for cart/payment |
| Event-Driven | Trading, Telecom | Kafka triggers billing or SMS on recharge |
| SOA | Govt, Insurance | SOAP services used in claim processing |

**📌 Real-World Tools That Support Software Architecture**

| **Aspect** | **Tool/Technology** |
| --- | --- |
| Microservices | Spring Boot, Docker, Kubernetes |
| Messaging/Event Bus | Kafka, RabbitMQ |
| API Layer | REST, GraphQL, Swagger |
| Database Access | JPA, Hibernate, JDBC |
| UI Layer | React, Angular, JSP |
| Security | OAuth2, JWT, SSL |

**Software Architecture Style: Layered Architecture**

**📚 Layers:**

1. **Presentation Layer** – Interacts with the user (via CLI here)
2. **Service Layer** – Contains business logic
3. **DAO Layer** – Data access (in-memory in this example)
4. **Model Layer** – Domain objects

**📂 Project Structure**

CopyEdit

BankingApp/

├── model/

│ └── Account.java

├── dao/

│ ├── AccountDao.java

│ └── AccountDaoImpl.java

├── service/

│ ├── AccountService.java

│ └── AccountServiceImpl.java

├── ui/

│ └── BankAppUI.java

**🔹 1. model/Account.java – Domain Model**

public class Account {

private int accountNumber;

private String holderName;

private double balance;

public Account(int accountNumber, String holderName, double balance) {

this.accountNumber = accountNumber;

this.holderName = holderName;

this.balance = balance;

}

public int getAccountNumber() { return accountNumber; }

public String getHolderName() { return holderName; }

public double getBalance() { return balance; }

public void setBalance(double balance) { this.balance = balance; }

@Override

public String toString() {

return "Account[" + accountNumber + ", " + holderName + ", Balance=" + balance + "]";

}

}

**🔹 2. dao/AccountDao.java – Data Access Interface**

import model.Account;

public interface AccountDao {

void save(Account account);

Account findByAccountNumber(int accNumber);

}

**🔹 3. dao/AccountDaoImpl.java – In-Memory DAO Implementation**

import model.Account;

import java.util.HashMap;

import java.util.Map;

public class AccountDaoImpl implements AccountDao {

private Map<Integer, Account> db = new HashMap<>();

@Override

public void save(Account account) {

db.put(account.getAccountNumber(), account);

}

@Override

public Account findByAccountNumber(int accNumber) {

return db.get(accNumber);

}

}

**🔹 4. service/AccountService.java – Business Logic Interface**

import model.Account;

public interface AccountService {

void createAccount(int accNum, String name, double initialBalance);

Account getAccount(int accNum);

}

**🔹 5. service/AccountServiceImpl.java – Business Logic Implementation**

import dao.AccountDao;

import dao.AccountDaoImpl;

import model.Account;

public class AccountServiceImpl implements AccountService {

private AccountDao dao = new AccountDaoImpl();

@Override

public void createAccount(int accNum, String name, double initialBalance) {

Account acc = new Account(accNum, name, initialBalance);

dao.save(acc);

}

@Override

public Account getAccount(int accNum) {

return dao.findByAccountNumber(accNum);

}

}

**🔹 6. ui/BankAppUI.java – Presentation Layer**

import service.AccountService;

import service.AccountServiceImpl;

import model.Account;

import java.util.Scanner;

public class BankAppUI {

public static void main(String[] args) {

AccountService service = new AccountServiceImpl();

Scanner scanner = new Scanner(System.in);

System.out.println("Welcome to Core Java Bank");

while (true) {

System.out.println("\n1. Create Account\n2. View Balance\n3. Exit");

int choice = scanner.nextInt();

switch (choice) {

case 1:

System.out.print("Account Number: ");

int accNum = scanner.nextInt();

scanner.nextLine(); // consume newline

System.out.print("Holder Name: ");

String name = scanner.nextLine();

System.out.print("Initial Balance: ");

double balance = scanner.nextDouble();

service.createAccount(accNum, name, balance);

System.out.println("Account Created Successfully.");

break;

case 2:

System.out.print("Enter Account Number: ");

int num = scanner.nextInt();

Account acc = service.getAccount(num);

if (acc != null)

System.out.println(acc);

else

System.out.println("Account not found.");

break;

case 3:

System.out.println("Thank you!");

return;

default:

System.out.println("Invalid choice.");

}

}

}

}

**✅ Output (Sample Run)**

markdown

CopyEdit

Welcome to Core Java Bank

1. Create Account

2. View Balance

3. Exit

> 1

Account Number: 101

Holder Name: John

Initial Balance: 5000

Account Created Successfully.

> 2

Enter Account Number: 101

Account[101, John, Balance=5000.0]

**📌 Key Architectural Benefits Demonstrated**

| **Layer** | **Responsibility** |
| --- | --- |
| ui | User input/output |
| service | Business rules and logic |
| dao | Data storage and retrieval (mock DB) |
| model | Data structure for domain (Account) |

This **Core Java layered architecture** makes the code:

* Easy to **extend** (e.g., add updateAccount)
* Easy to **test** and **maintain**
* Ready for migration to **Spring Boot**, **Hibernate**, or a **REST API**