Architectural Concepts 1 Cheat Sheet

by vaibhavsk via cheatography.com/205698/cs/43899/

SOLID Principles

Single Responsibility Principle (SRP): A class should have only one reason to change, meaning it should have a single responsibility or concern.

Open-Closed Principle (OCP): Software entities (classes, modules, functions) should be open for extension but closed for modification. This means that you should be able to add new functionality without modifying existing code.

Liskov Substitution Principle (LSP): Subtypes must be substitutable for their base types without altering the correctness of the program. In other words, derived classes should be able to be used in place of their base classes without causing issues.

Interface Segregation Principle (ISP): Clients should not be forced to depend on interfaces they do not use. Instead of having a single large interface, it is better to have smaller and more specific interfaces.

Dependency Inversion Principle (DIP): High-level modules should not depend on low-level modules. Both should depend on abstractions. Abstractions should not depend on details; details should depend on abstractions. This principle promotes loose coupling and allows for easier modification and testing.

Creational Patterns

Singleton: Ensures only one instance of a class is created and provides a global point of access to it.

Factory Method: Defines an interface for creating objects but allows subclasses to decide which class to instantiate.

Abstract Factory: Provides an interface for creating families of related or dependent objects without specifying their concrete classes.

Builder: Separates the construction of complex objects from their representation, allowing the same construction process to create different representations.

Creational Patterns (cont)

Prototype: Creates new objects by cloning existing ones, avoiding the need for complex initialization.

Why MICROSERVICES?

Scalability: Each microservice can be deployed and scaled individually, enabling better resource utilization and handling varying levels of load

Flexibility and Agility: using different technologies and programming languages if needed for different services. Faster development and deployment cycles

Fault Isolation and Resilience: Failures are isolated to individual services. Resiliency can be implemented by redundancy, fallback mechanisms, and graceful degradation

Continuous Delivery and DevOps: Each microservice can have its own development, testing, and deployment pipeline, allowing for faster iterations and faster time to market.

Structural Patterns

Adapter: Converts the interface of a class into another interface that clients expect, allowing classes with incompatible interfaces to work together.

Decorator: Dynamically adds new behaviors to an object by wrapping it in a decorator object that provides additional functionality.

Proxy: Provides a surrogate or placeholder object that controls access to another object, adding extra functionality or controlling the object's access permissions.

Composite: Composes objects into tree structures to represent part-whole hierarchies, allowing clients to treat individual objects and compositions uniformly.

Facade: Provides a unified interface to a set of interfaces in a subsystem, simplifying its usage for clients.

Microservices Communication

HTTP/REST: HTTP with RESTful APIs. Each microservice exposes a set of well-defined endpoints.

Messaging/Event-driven: Communicate with each other through asynchronous messaging using a message broker or event bus. One microservice publishes events or messages, and other microservices can subscribe to these events and react accordingly. Allows for loose coupling and enables better scalability and fault tolerance.

RPC (Remote Procedure Call): RPC is a communication pattern where one microservice directly calls a method or service in another microservice.

Service Mesh: A service mesh is a dedicated infrastructure layer that handles communication between microservices. Provides features like service discovery, load balancing, and security.

API Gateway: An API gateway acts as a single entry point for client applications to communicate with multiple microservices. It can handle authentication, request routing, load balancing, and protocol translation. API gateways provide a unified interface for clients and help to decouple frontend applications from the complexities of microservices communication.

Behavioral Patterns

Observer: Defines a one-to-many dependency between objects, so that when one object changes state, all its dependents are notified and updated automatically.

Strategy: Defines a family of interchangeable algorithms and encapsulates each one, allowing them to be used interchangeably based on the context.

Command: Encapsulates a request as an object, allowing clients to parameterize clients with queues, requests, and operations.

Iterator: Provides a way to access elements of an aggregate object sequentially without exposing its underlying representation.

Template Method: Defines the skeleton of an algorithm in a base class, while allowing subclasses to override certain steps of the algorithm.