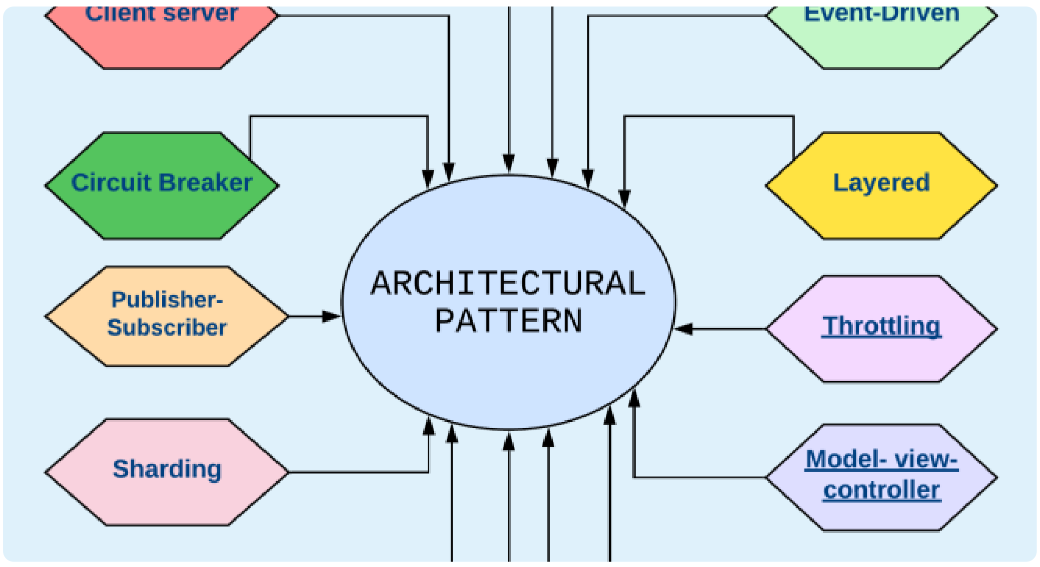


One post to refer to for Software Architectural Design Pattern

Rocky Bhatia

+ Follow

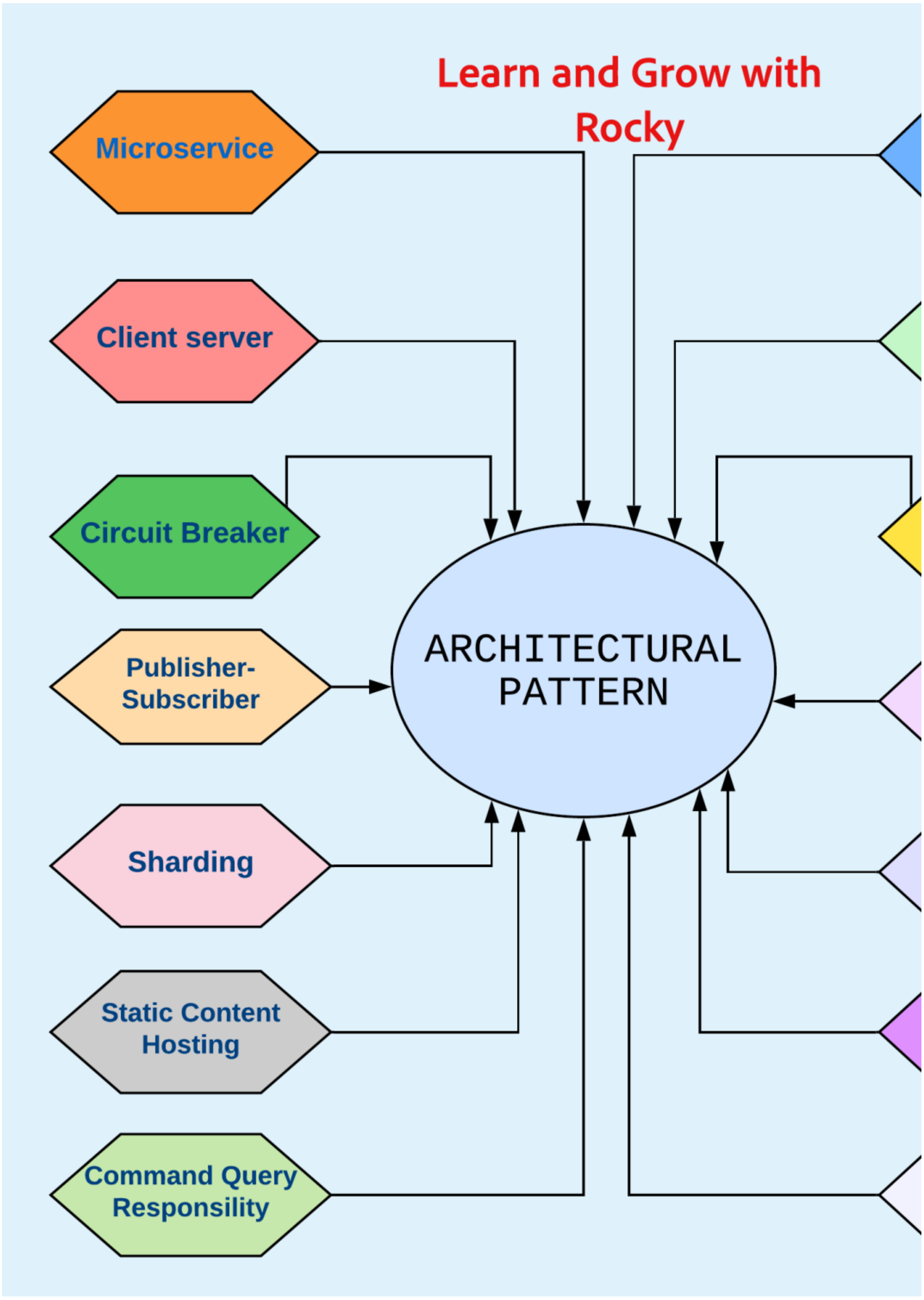
Architect@Adobe || 70K + Linkedin || International Speaker || Learner || Teacher || Content Creator || 25+ Million Impression

Published Feb 7, 2023

In **#softwareengineering**, a software design pattern is a general, reusable solution of how to solve a common problem when designing an application or system. Unlike a library or framework, which can be inserted and used right away, a design pattern is more of a template to approach the problem at hand.

#Architectural patterns provide a blueprint for the overall structure and organization of a software system, and help to ensure that the system is scalable, maintainable, and easy to understand.

There are 14 Important and widely used design Patterns ,Please find the details :


Learn and Grow with Rocky
Microservice
Client server
Circuit Breaker
Publisher-Subscriber
Sharding
Static Content Hosting
Command Query Responsility
ARCHITECTURAL PATTERN

Master-slave

Master-slave architecture is a type of client-server architecture where one node, the master, controls and coordinates the actions of one or more other nodes, the slaves. The master is responsible for distributing tasks and managing the overall flow of work, while the slaves perform the actual work. This architecture is often used in distributed systems, where the master node acts as a coordinator and the slave nodes perform the processing and storage of **#data**.

Examples of systems that use the master-slave architecture include:

Database Replication: A database replication system, where a primary database server, the master, holds the original and up-to-date data, and one or more secondary database servers, the slaves, receive and store a copy of the data from the master.
Load Balancing: A load balancing system, where a master node distributes incoming network requests to a pool of slave nodes for processing.
Parallel Processing: A system that divides a large task into smaller subtasks and assigns them to multiple slave nodes for parallel processing, with a master node coordinating the task distribution and results collection.
Distributed File Systems: A distributed file system, where the master node manages the file system metadata, such as file and directory names and their locations, while the slave nodes store the actual data.
Network Services: A network service, such as a DNS service, where the master node acts as the primary source of information and the slave nodes provide redundant backups of the data.

Sharding

Sharding architecture is a method of horizontally partitioning large databases into smaller, more manageable parts called shards. Each shard is a self-contained subset of the database that is stored on a separate database server. The goal of sharding is to distribute the data and workload across multiple servers to improve scalability, performance, and availability.
In a sharded database system, the data is divided into shards based on a sharding key, which is a characteristic of the data used to determine which shard a particular piece of data belongs to. Queries to the database are processed by multiple shards in parallel, and the results are combined to provide a complete response to the client.
Sharding is often used in highly scalable systems, such as e-commerce websites, social media platforms, and gaming systems, where the amount of data and the number of queries per second can become very large. Sharding enables these systems to handle large amounts of data and traffic by spreading the load across multiple servers.

Publisher-subscriber

Publisher-subscriber architecture, also known as the publish-subscribe pattern, is a messaging pattern in which messages, or events, are published by a sender and delivered to multiple receivers, known as subscribers. The subscribers receive only the messages that they are interested in and are notified when a new message is available.
In this architecture, there is no direct communication between the publisher and the subscribers. Instead, a middleware component, called a message broker, acts as an intermediary between the publisher and subscribers. The message broker is responsible for receiving messages from the publisher, filtering and routing the messages to the appropriate subscribers, and delivering the messages to the subscribers.
The publish-subscribe pattern is often used in event-driven systems, where multiple components need to be notified when a certain event occurs. It provides a decoupled and scalable architecture, where components can be added or removed without affecting the others. Some examples of systems that use the publish-subscribe architecture include stock market tickers, instant messaging systems, and event-driven **#microservices**.

Circuit Breaker

Circuit breaker is a design pattern used in software architecture to handle failures in the communication between microservices or between a microservice and a remote service, such as a database or a third-party API. The circuit breaker acts as a switch that can be opened or closed to control the flow of requests.
When a service fails, it can cause a cascade of failures throughout the system, leading to a significant increase in error rates and decreased system performance. The circuit breaker pattern addresses this issue by automatically detecting failures and temporarily halting communication with the failing service. This prevents further failures and allows the system to recover and continue processing requests.
The circuit breaker pattern typically consists of three states:

Closed: In this state, requests are allowed to flow through the circuit breaker to the service.

Open: In this state, the circuit breaker is open, and requests are not allowed to flow through to the service.
Half-Open: In this state, the circuit breaker allows a limited number of requests to flow through to test the health of the service. If the service is still failing, the circuit breaker will revert to the open state. If the service is healthy, the circuit breaker will revert to the closed state.

The circuit breaker pattern helps to improve the reliability and resilience of systems by providing a mechanism for detecting and mitigating failures, and for automatically recovering from failures.

Event-Driven

Event-driven architecture (EDA) is a design pattern used in software architecture that focuses on the production, detection, and consumption of events. In EDA, components communicate with each other by generating and responding to events, rather than by direct requests or calls.
An event is a message that represents a change in state or an occurrence of interest, such as a user action, a database update, or a message received from a remote service. In EDA, events are published by event producers and consumed by event subscribers.
EDA provides a loosely coupled and scalable architecture, as components can be added or removed without affecting the others. It is particularly well-suited for systems that require real-time processing, high scalability, and complex event processing.

Examples of systems that use event-driven architecture include real-time financial trading systems, e-commerce platforms, and internet of things (IoT) systems. In these systems, events are generated and processed in real-time, and components can respond to changes in state and events quickly and efficiently.

Microservices

Microservices is an architectural pattern that structures an application as a collection of small, independent services that communicate with each other through well-defined APIs. The goal of microservices is to break down a monolithic application into smaller, manageable parts that can be developed, tested, and deployed independently.
Each microservice is responsible for a specific business capability and can be written in a different programming language, using

different technology stacks, and deployed on different servers. This allows for greater flexibility and allows for faster innovation, as teams can work on their respective services independently.

Microservices patterns include:

API Gateway: This pattern provides a single entry point for client requests and acts as an intermediary between the client and the microservices. The API gateway is responsible for routing requests, authentication, and rate limiting.
Service Registry: This pattern provides a registry of all the available microservices, allowing clients to discover and communicate with the appropriate service.
Load Balancer: This pattern distributes incoming requests evenly across multiple instances of a microservice, improving reliability and performance.
Circuit Breaker: This pattern is used to handle failures in communication between microservices or between a microservice and a remote service.
Event-Driven Communication: This pattern uses events to trigger communication between microservices, rather than direct requests.

Microservices is a popular architectural pattern for building highly scalable, resilient, and flexible applications. It is used in many systems, including e-commerce platforms, content management systems, and cloud-based applications.

Layered

Layered architecture is a design pattern that divides an application into multiple layers, each with a specific responsibility. The layers are organized in a stack, with each layer relying on the services provided by the layer below it and providing services to the layer above it.
The purpose of layering is to improve the modularity, maintainability, and scalability of the application by separating concerns and reducing coupling between components. A well-designed layered architecture can make it easier to understand, develop, and maintain the application, as well as to test and deploy individual components.
Typically, a layered architecture will include the following layers:

Presentation layer: This layer is responsible for presenting the user interface and handling user input.

Application layer: This layer implements the application's business logic and coordinates the flow of data between the presentation layer and the data access layer.
Data access layer: This layer provides an abstraction over the data storage mechanism and handles database access and data retrieval.
Database layer: This layer stores and manages the data used by the application.

In addition to these core layers, other specialized layers can be added, such as a security layer, a caching layer, or a logging layer.

The layered architecture provides a flexible and modular structure that can be adapted to meet the specific needs of the application.

Throttling
Throttling is a technique used in **#softwarearchitecture** to control the rate of incoming requests to an application or service. The goal of throttling is to prevent an application from being overwhelmed by too many requests, which can cause performance degradation, stability issues, or even complete failure.

In a throttling architecture, incoming requests are monitored and compared against a pre-defined threshold. If the number of requests exceeds the threshold, the application can either reject or delay some of the requests to reduce the load.

Throttling can be applied at various levels, such as at the network level, at the server level, or at the application level. It can be implemented using various techniques, such as fixed rate limiting, leaky bucket algorithms, or token bucket algorithms.
Throttling is an important aspect of performance optimization and system reliability. It helps to ensure that the application can handle spikes in traffic and maintain a consistent level of performance, even under heavy load.

Model-View-Controller

Model-View-Controller (MVC) is a software design pattern that separates an application into three main components: the model, the view, and the controller.

Model: The model represents the data and the business logic of the application. It is responsible for managing the state of the application and handling the underlying data storage.
View: The view is responsible for presenting the user interface and rendering the data from the model. It receives user input and passes it to the controller.
Controller: The controller acts as an intermediary between the model and the view. It receives user input from the view and updates the model, and it updates the view when the model changes.

The MVC pattern is used to improve the separation of concerns, increase the modularity, and reduce the complexity of the application. By separating the data, the user interface, and the control logic into separate components, the MVC pattern makes it easier to develop, maintain, and test the application.

MVC is a popular pattern in web application development and is widely used in various frameworks, such as Ruby on Rails, AngularJS, and ASP.NET MVC. It is also used in desktop application development and has been adopted by several programming languages, including Java and Swift.

Pipe-Filter

A pipe-filter architecture is a software design pattern where data is passed from one processing stage to another in a pipeline. Each stage, called a filter, processes the data in some way and outputs the result to the next filter in the pipeline. This pattern allows for modular design and improves efficiency by processing data in parallel, with each filter working on a portion of the data. The pipeline is similar to an assembly line, where each worker performs a specific task and hands off the result to the next worker in line.
An example of a pipe-filter architecture is an image processing system, where each filter performs a specific task on an image.
For example:
The first filter reads an image file and converts it into a digital format.
The second filter performs color correction to adjust the image brightness and contrast.
The third filter applies a blur effect to reduce image noise.
The fourth filter performs edge detection to identify the boundaries of objects in the image.
The final filter outputs the processed image to a file or displays it on a screen
In this example, each filter operates independently and performs its task on a portion of the data, allowing for efficient parallel processing. The pipeline architecture allows for modular design, where each filter can be added or removed as needed, making the system scalable and flexible.
Peer-to-peer
Peer-to-peer (P2P) architecture is a decentralized network structure in which each node acts as both a client and a server. In a P2P network, nodes communicate directly with each other to share resources and distribute workloads, rather than relying on a centralized server to manage all data transfers. This eliminates the need for a single point of control or failure, making the system more resilient and scalable.

Examples of P2P systems include file sharing networks such as BitTorrent, instant messaging platforms like WhatsApp, and decentralized digital currencies like Bitcoin. In these systems,

nodes can join or leave the network dynamically, and the network can continue to function even if some nodes are offline or unavailable.
P2P architecture has the potential to provide more efficient and robust solutions compared to traditional client-server architectures, but it also introduces new security and privacy challenges, as nodes must trust each other to share information correctly.

Command Query Responsibility Segregation

Command Query Responsibility Segregation (CQRS) is an architectural pattern that separates the responsibilities of handling commands (write operations) and queries (read operations) in a system.
In CQRS, write operations are handled by command handlers, which validate the command and update the state of the system. Read operations are handled by query handlers, which retrieve information from the system and present it to the client. This separation of concerns allows for optimization of each aspect of the system, with write operations optimized for consistency and write performance, and read operations optimized for read performance and scalability.
CQRS can be used in various types of systems, such as web applications, microservices, and event-driven systems. By using CQRS, the system can improve scalability, reliability, and maintainability by separating the write and read paths and allowing for separate data stores optimized for each type of operation.
CQRS is often used in combination with Event Sourcing, a technique where the state of the system is represented as a sequence of events, rather than a single, current state.

Static content

Static content hosting architecture refers to a system designed to serve static resources such as HTML, CSS, JavaScript, and images over the web.
In this architecture, static content is stored on a server and served to clients upon request. The server does not require any dynamic

processing for each request, as the content is static and does not change. This allows for high performance and scalability, as the server can handle many requests concurrently without significant computational overhead.
Static content hosting can be achieved using various types of servers, such as web servers like Apache or Nginx, or content delivery networks (CDN) like Amazon CloudFront or Akamai. The server can be configured to serve content directly or with the use of a load balancer to distribute requests across multiple servers for better performance and availability.
Static content hosting is often used in conjunction with dynamic content hosting, where dynamic resources such as APIs, databases, and application servers provide the dynamic functionality of a web application. The static content is used to present the user interface and interact with the dynamic content.

Client-server

Client-server architecture is a distributed system architecture where the tasks are divided between clients and servers. In this architecture, clients request services or resources from servers, which provide those services or resources. The client-server relationship is defined by a protocol that specifies the request and response format. The clients are typically responsible for rendering the user interface, while the servers are responsible for processing the data and providing the requested services. This architecture is widely used in networked computer systems, such as web applications and database systems.

Examples of systems that use the client-server architecture include:

Web Applications: A web browser acts as a client, sending requests to web servers for HTML, CSS, and JavaScript files.
Database Systems: A client program, such as a database management system, sends requests to a database server to retrieve or update data.
File Sharing Systems: A client program, such as a file explorer, sends requests to a file server to access and download files.
Email Systems: An email client sends requests to an email server to send and receive messages.

By using architectural patterns, software developers can create systems that are scalable, maintainable, and easy to understand. Additionally, by reusing proven patterns, developers can reduce the risk of introducing errors or design flaws into their systems.

Please let me know which architecture pattern have you used ?

#softwareengineering #software #technology #learning #career #personaldevelopment #architecture #designpatte #softwareengineering #learningdesign #learningeveryday