Top 10 Architectural Patterns

In this issue, we talk about:

- **What are the 10 most common Architectural patterns?**
- **Architecture Patterns from different points of view.**
- **What is the difference between architectural styles, patterns, and design patterns?**
- **A free e-book on the architecture of Open-source applications.**

Top 10 Architectural Patterns

Software design defines a software system's architecture, components, modules, and interfaces to meet specified requirements. During the software design phase, we create a high-level view of the system, including its structure, behavior, and interactions with other systems. This involves making decisions about the software's internal structure and organization.
The output of the software design process is a **set of design documents or diagrams** that serve as a blueprint for the software development team to follow. By following the software design plan, developers can ensure that the final product meets the stakeholders' requirements and is high quality.
Over time we saw some **common software architecture patterns** emerge. They are reusable solutions to common design problems. They are often used as a starting point for designing software systems and provide a set of best practices for solving specific architectural issues. They enable us to reduce design complexity, increase system maintainability and reduce development time. In addition, they capture best practices and proven solutions for designing reliable, scalable, maintainable, and extensible software systems.

There are many **software architecture design patterns** to know, but some of the most important ones are:

1. **Layered Architecture:** This pattern is based on dividing the application into logical layers, where each layer has a specific responsibility and interacts with the layers above and below it. A typical application could have Presentation, Business, and Data Access Layers.
2. **Microservices Architecture:** This pattern is based on decomposing the application into small, independent services that communicate with each other through well-defined APIs. Each service is self-contained and should implement a single business capability.

The services may communicate with each other via APIs and can be independently deployed.

3. **Event-Driven Architecture:** This pattern uses events to communicate between different components or services, where events trigger actions or reactions in the system. Events are processed asynchronously, meaning the system does not stop to wait for the handling of an event, increasing overall efficiency. This pattern is handy for applications that require real-time responsiveness and scalability.
4. **Space-based Architecture:** is a software design method that centers the system's structure around the idea of "spaces," independent and autonomous units. High-volume distributed systems often use it to overcome data bottlenecks, network latency, and grid failure. It divides the processing and storage between multiple servers, eliminating the need for a central server and, thus, the problem of a single point of failure.
5. **Microkernel Architecture (plug-in architecture):** this is an approach where the kernel provides minimal functionality, and services are implemented as separate modules outside the kernel (plug-ins). This separation allows developers to add, modify, or remove system components without impacting the core system functionality.
6. **Peer to Peer Architecture**: this is a decentralized model where nodes in a network can act as both clients and servers, allowing for distributed sharing of resources and information without the need for a central authority. P2P is used in blockchain technologies or file-sharing networks like BitTorrent.
7. **Cloud-native Architecture:** this is a pattern where applications are developed and deployed to run on cloud platforms, leveraging cloud services and infrastructure for scalability, reliability, and agility. It's characterized by containerizing services, allowing for maximum portability and resilience; microservices, a design that separates functionality into modular components; orchestration, managing these containers and services; and declarative APIs, which express the system’s desired state.
8. **Command-Query Responsibility Segregation Architecture (CQRS)**: This pattern separates the command and query responsibilities of an application's model, making it easier to scale and optimize the application. The command side handles creating, updating, and deleting operations, while the query side handles queries. This can significantly improve performance, simplicity, and scalability.
9. **Hexagonal Architecture (Ports and Adapters):** This pattern separates the application into an inner and outer layer, where the inner layer contains the business logic and the outer layer includes the interfaces with the outside world. It allows an application to equally be driven by users, programs, automated test or batch scripts, and to be developed and tested in isolation from its eventual run-time devices and databases.
10. **Clean Architecture:** This pattern emphasizes separating concerns and decoupling components, making it easier to maintain and change an application over time.

11.

- ## Architecture Patterns from different points of View

- **By Connections** - a way how to connect devices and applications.
    a. REST (Representational State Transfer)
    b. BFF (Backend for Frontend)
    c. RPC (Remote Procedure Call)
    d. P2P (Peer to Peer)
    e. SOA (Service-Oriented Architecture)

- **By Composition** - how we compose internal structures.
    f. Microservices
    g. Monolith
    h. Microkernel
    i. Layers
    j. Plugins
    k. Components
    l. Nano services

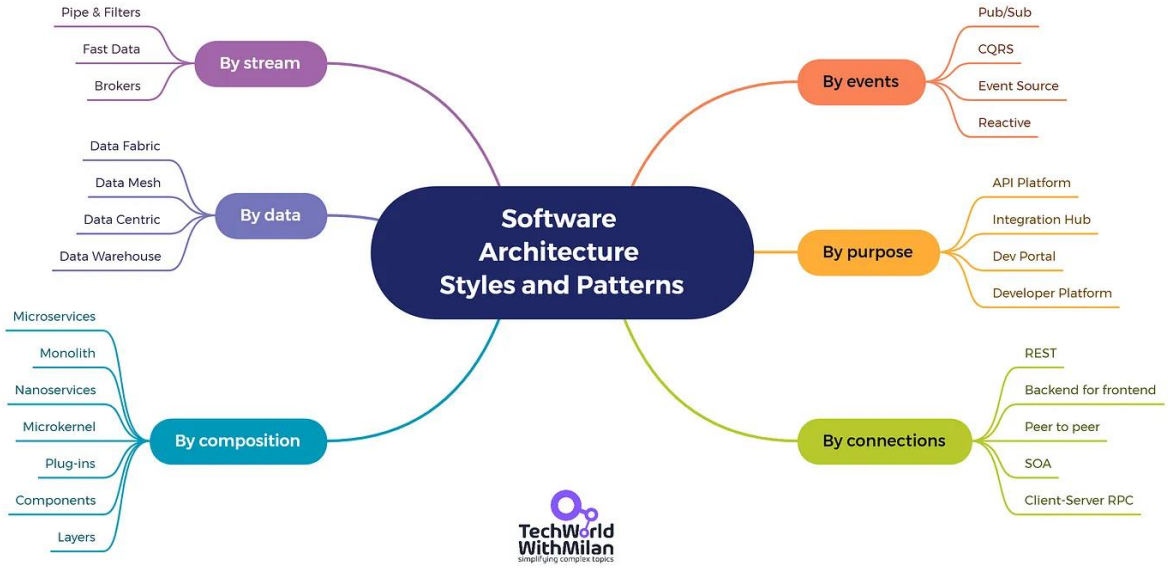
- **By Events** - use it to decouple software components in an event-driven manner.
    m. Publish/Subscribe
    n. CQRS (Command Query Responsibility Segregation)
    o. ES (Event Sourcing)
    p. Reactive

- **By Stream** - how data flow through our system.
    q. Pipe and Filters
    r. Message Brokers
    s. Fast Data

- **By Data** - for data-oriented systems.
    t. Data Mesh
    u. Data Warehouse

v. Data Fabric
w. Data-centric Business Logic
x. Data Lake

- **By Purpose**
    y. API Platform
    z. Integration Hub
    aa. Developer Portal
    bb. Developer Platform



Architecture Styles, Patterns, and Design Patterns

We can frequently see the need for clarification between architectural styles, patterns, and design patterns. These terms often need clarification, and different people give them different meanings. Let's define it a bit more precisely.

Architectural Styles

Architectural Styles and **Architectural Patterns** are both design strategies used in software architecture. They provide a means of expressing high-level design solutions, but they do so at slightly different levels of abstraction.
Architecture Styles are the highest level of abstraction where architectural designs instruct us on how to structure our code. They provide a set of constraints and guidelines that can help define the system's overall structure and high-level modules and how they relate to and interact with one another. Examples of architectural styles include:

- **Monolith**
- **Layered**
- **Event-driven**
- **Self-contained Systems**
- **Microservices**
- **Space-Based**

Architectural Patterns

Architectural Patterns represent a way to implement an architectural style so we can do this regularly. Some examples are how to separate the user interface (UI) and data, how internal modules interact, and what layers we will use. Patterns answer these types of questions. In addition, they usually impact the code base and how to structure the code inside. Examples of architectural patterns include:

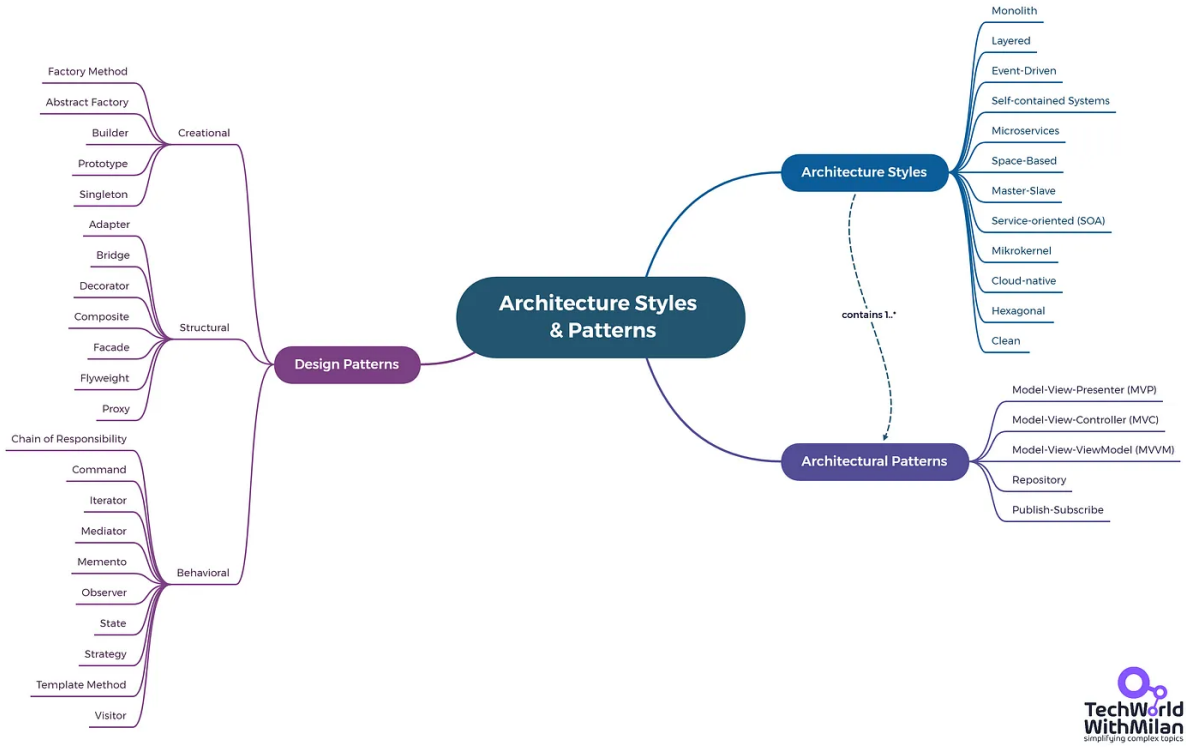
- **Model-View-Presenter (MVP)**
- **Model-View-Controller (MVC)**
- **Model–View–ViewModel (MVVM)**
- **Repository Pattern**
- **Publish-Subscribe Pattern**

Design Patterns

Design Patterns are different from architectural patterns in that they focus on a smaller area of the code base and have a minor influence (focus on a local problem). Nevertheless, they are specific solutions to common problems in software design. These might include limiting the creation of a class to only one object or notifying all dependent things when the internal state of an object is changed. These patterns are described in the book "**Design Patterns: Elements of Reusable Object-Oriented Software**" by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides from 1994.

We have three groups of patterns:

- Creational: Factory Method, Builder, Singleton, etc.
- Structural: Adapter, Bridge, Decorator, etc.
- Behavioral: Command, Iterator, State, Strategy, etc.



The Architecture Of Open Source Applications (Free E-book)

The authors of 40 open-source apps explain the structure of their software and its purpose in these two publications. What are the main elements of each program? How do they communicate? And what did their creators discover as they evolved? The authors of these works offer fascinating insights into their thought processes in their responses to these questions.
You can learn here about the architecture of:

- **Firefox Release Engineering**
- **Git**
- **Ngnix**
- **Puppet**
- **ZeroMQ**
- **CMake**
- **The Hadoop Distributed File System**
- **The NoSQL Ecosystem**
- **Selenium WebDriver**
- **High-Performance Networking in Chrome**