**Architecture Pattern**

An **architecture pattern** is a general, reusable solution to a commonly occurring problem in software architecture within a given context.

1. **Layered Pattern (n-tiers architecture, most common)**

* Software is divided into units called layers:

1. Presentation Layer: handling UI
2. Application Layer:
3. Business Logic Layer:
4. Data Access Layer:

* **Pros**: lower layers can be used by different higher layers.
* **Cons**: perform poorly in high performance applications since it is not efficient to going through multiple layers to fulfill a business’s request
* **Application**: usually used in building desktop application, simple web apps => a good choice for a situation which having very tight budget and time constraints

1. **Pipe-filter Pattern**

* A single event triggers a sequence off processing steps each performing a specific function. It divides a larger processing task into a sequence of smaller independent processing steps or filters that are connected by channels or pipes.
* **Application**: used in compilers where the consecutive filters perform lexical analysis parsing semantic analysis and code generation.

1. **Client-Server Pattern**

* 2 main components:

1. The client which is the service requester
2. The server which is the service provider

* Although both client and server may be located within the same system, they often communicate over a network on separate hardware.
* **Pros**: ease of modeling a set of services where clients can request them.
* **Cons**: the server can be a performance bottleneck and a single point of failure. The decisions about where to locate functionality in the client or in the server are often complex and costly to change after a system has been built.
* **Application**: online applications (e.g email, document sharing, banking)

1. **Model-View-Controller Pattern (MVC)**

* Separating application functionalities into 3 kinds of components:

1. Model: contain the core functionality and data.
2. View: displays the information to the users where more than one view may be defined.
3. Controller: handles the input from the user

* **Application**: most commonly used in the web framework such as Django or rails

1. **Event Bus Pattern**

* Software is often built as a distributed system that can service asynchronously arriving messages associated with high volume of events, has 4 major components:

1. Event source: publish messages to particular channels on an event bus
2. Event listener: subscribe to particular channels. They are notified of messages that are published to a channel to which they have subscribed before,
3. Channel:
4. Event bus:

* **Pros**: the new publishers, subscribers and connections can be added easily.
* **Cons**: problems about scalability as all messages travel through the same bus.
* **Application**: used in Android development, e-commerce apps, and notification services.

1. **Microservices Architecture**

* The modern enterprise apps are required to support a variety of browsers and native mobile clients these days. The applications usually handle client requests by executing business logic, accessing a database, exchanging messages with other systems and returning responses. Monolithic applications can become too large and complex for efficient support and deployment
* The solution is to build applications as micro services. Each service is independently deployable and scalable and has its own APIs boundary. Different services can be written different programming languages, manage their own database and developed by different teams.
* Application: involve an extensive data pipeline.

1. **Broker Pattern**

* It is used to structure distributed system with decoupled components. These components can interact with each other by remote service invocations.

1. **A broker component** is responsible for the coordination of communication among components.
2. **Servers** publish their capabilities to a broker.
3. **Clients** request a service from the broker and the broker then redirects the client to a suitable service from its registry

* **Pros:** allow dynamic change, addition, deletion, and relocation of objects. It makes distribution transparent to the developers
* **Cons:** require standardizing of service descriptions
* **Application:** used in message broker software such as Apache, Apache Kafka, Rabbit,..

1. **Peer-to-peer Pattern**

* In this pattern, individual components are known as peer. Peer may function both as a client requesting services from other peers and as a server providing services to other peers. A peer may act as a client or as a server or as both. It can change its role dynamically with time. It supports decentralized computing and is highly robust in the failure of any given node.
* **Pros**: highly scalable in term of resources and computing power
* **Cons**: no guarantee about quality of service as nodes cooperate voluntarily. Security is difficult to ensure and the system performance often depends on the number of nodes.
* **Application**: used in the file sharing networks such as Ganutella, G2, multimedia protocols, cryptocurrency-based products such as bitcoin.

1. **Blackboard Pattern**

* This pattern is useful for problems for which no deterministic solution strategies are known the blackboard pattern consists of three main components:

1. **Blackboard** which is a structured global memory containing objects from the solution space knowledge
2. **Source** which is specialized modules with their own representation
3. **Control component** which selects, configures, and executes modules

* All the components have access to the blackboard components may produce new data objects that are added to the blackboard components look for particular kinds of data on the blackboard and may find these by pattern matching with the existing knowledge source
* **Pros**: easy extension of the structure of the data
* **Cons**: modifying the structure of the data space is hard as all applications are affected
* **Application**: this pattern is often used in speech recognition, protein structure identification and summer signals interpretation pattern

1. **Master-slave Pattern**

* This pattern consists of two parties master and slaves

1. **The master component** distributes the work among identical slave components and computes a final result from the results which the slaves return
2. **The slave component** returns the results.

* **Pros:** the accuracy in which the execution of a service is delegated to different slaves with different implementations
* **Cons:** this pattern can only be applied to a problem that can be decomposed
* **Application:** usually used in database replications where the master database is regarded as the authoritative source and the slave databases are synchronized to it

<https://www.youtube.com/watch?v=BrT3AO8bVQY&list=PLIR28YzTSt_vWnvFe1RQZLLILgqIZ2xPe&index=1>