Patterns are essentially reusable solutions to common problems. When faced with a problem it is reasonable to consider a catalog of patterns in order to find suitable existing solutions before designing something entirely new. Then, it will be a matter of adapting these patterns to the scope of our reality.

Think of it this way: Using a design pattern is having other systems architects on your team from whom you gain knowledge without investing too much time, certainly a smart move. Although there are dozens of architectural patterns, in this post I propose to explore the most common in a language-agnostic approach.

**How are architecture patterns different from design patterns?**

The title mentions architectural patterns so it is wise to clarify how they [differ from design patterns](https://medium.com/r?url=https%3A%2F%2Fwww.oreilly.com%2Fradar%2Fcontrasting-architecture-patterns-with-design-patterns%2F) although the difference may be a gray area. Specifically, we consider here that architectural patterns differ in the type of problem they try to solve. They define a solution for a variety of quality attributes and contemplate multiple components in a software system. The scope is broader for the architectural patterns while design patterns represent a way to structure classes to build the best internal structure.

**Layers Pattern**

Let’s start with perhaps the most used (and abused) pattern which is the layered pattern. Partitioning a system into separate layers, organizing components within them by related criteria, allows developers and teams to work better together. The layers promote good practices of [low coupling and high cohesion](https://stackoverflow.com/questions/14000762/what-does-low-in-coupling-and-high-in-cohesion-mean).

Strictly speaking, in this model each module must be assigned to one and only one layer, where upper layers are allowed to use lower layers (one way only). It is a design that is justified in terms of maintainability, offering that different teams can collaborate on different modules in parallel. Here is an example of this scheme:

Diagram

Description automatically generated

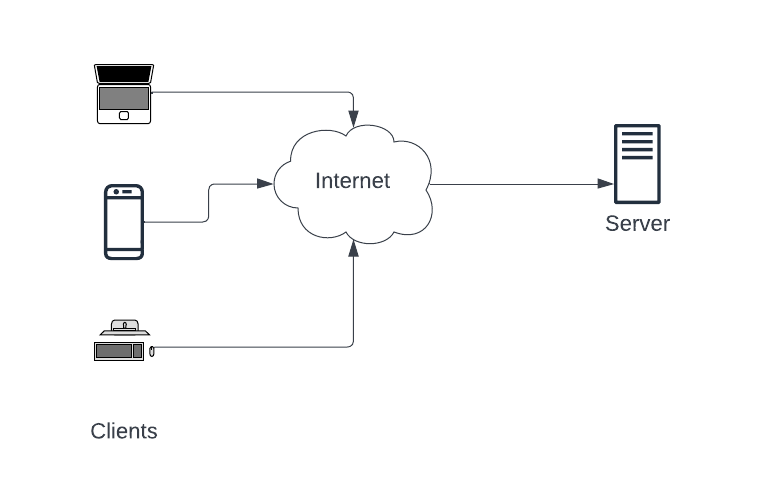
Layered Pattern Example

An API should communicate with the business logic services who at the same accesses the data layer. They all can potentially have a common set of libraries to reuse components.

**Client Server Pattern**

Another of the most common patterns in the Internet world is the Client-Server where a central authority (Server) offers one or more services that are consumed by one or more consumers (Client). Simple, but very powerful, this model offers the possibility of building distributed systems where the server centralizes the resources and workload in one place so that later multiple consumers use this data independently.

Twitter, Medium, the Email, File Sharing, mobile applications and the Web in general work with this mechanism. User applications consume the data and are responsible for providing an interface to interact with them, so the client and server often cooperate to achieve a common system. However, this pattern has a strong separation of concerns and responsibilities.



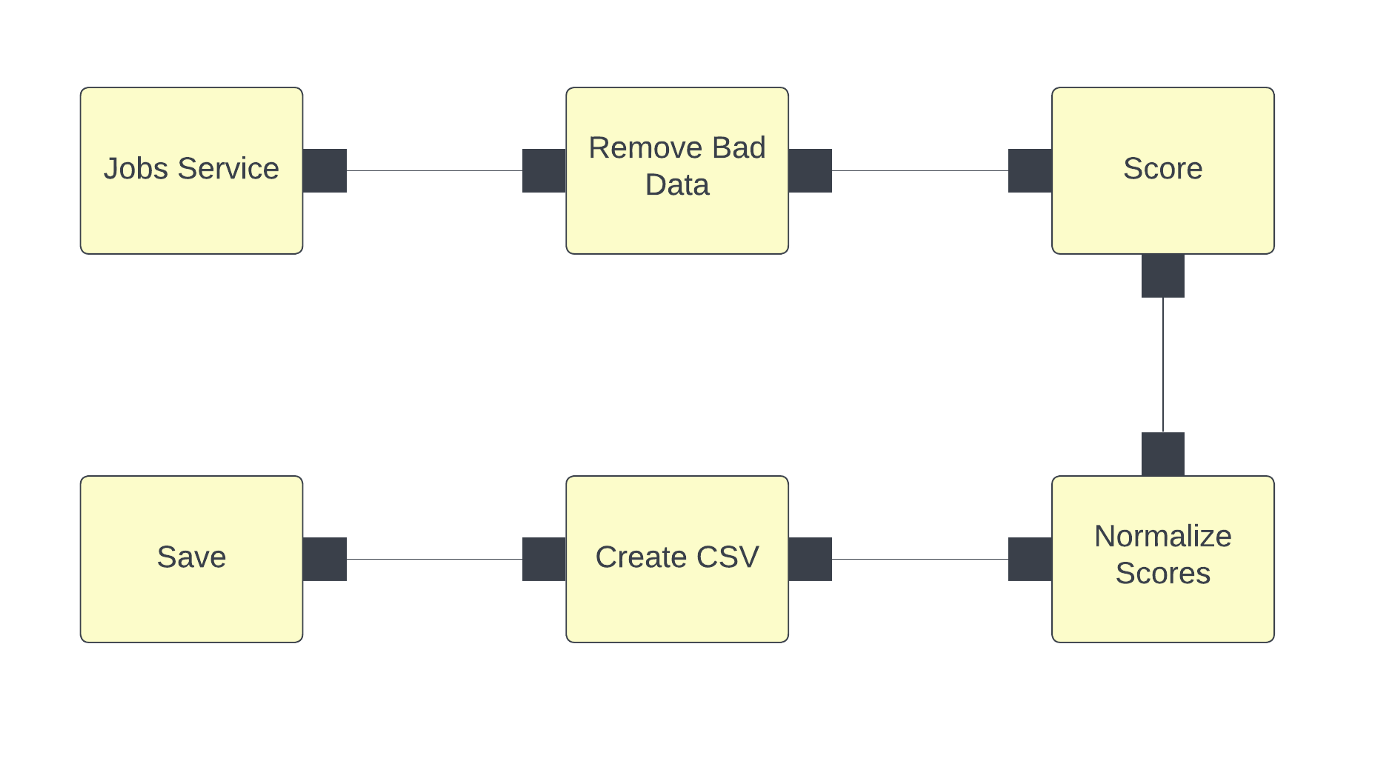
Client — Server Example

**Pipe and Filter Pattern**

With the pipe and filters pattern, each component named *filter* is responsible for a single transformation or operation on the data. Data is streamed from one filter to the next as fast as possible and data operation occurs in parallel. Decoupled filters can even be reused and swapped to create new pipelines.

It is a widely used pattern in data analysis and data transformation use cases. A more everyday example of this pattern is when we use the unix pipe function to combine commands, the essence is the same.

Here is an example diagram of this pattern



Pipes and Filter Example

Here we can define:

* *Filter*: it is the component that reads the data, transforms it and returns the transformed data
* *Pipe*: It is a connector that transports the data from one filter to the next and must ensure that the data is not modified along the way

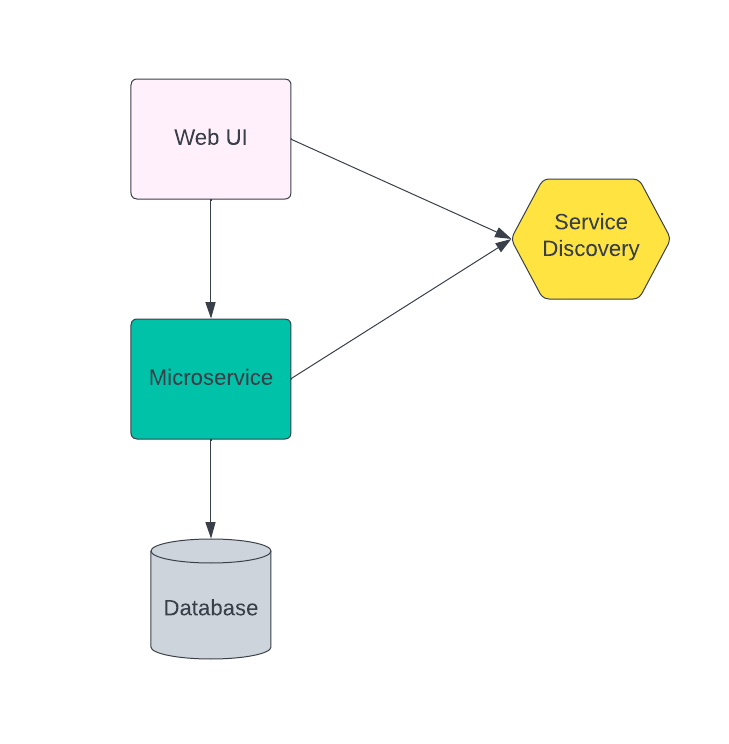
It is a pattern that can become computationally expensive due to its nature of data analysis, but it offers performance at a higher level of architecture due to the ability to post-process data, clean it, classify it, removing the workload from other more “real time” pieces.

**SOA Pattern**

In a Service-Oriented-Architecture, independent components are implemented as services, which offer specific functionality. These services are combined at run time to define the behavior of the system as a whole. For this to work, consumers of these services must be able to locate and use them without knowing the implementation details underlying these services.

This type of architecture can be implemented in various ways. Traditional SOA systems rely heavily on the SOAP protocol, which works by exchanging XML messages, while more “modern” SOA applications encourage the use of microservices connected by lightweight messages on a protocol like HTTP.

Here is an over-simplified example showing a single view of a SOA system. In practice, SOA architectures are complicated and involve many architectural components. This diagram shows two services attached to the service registry. The services should then check the registry to look up connection information for other services they want to call.



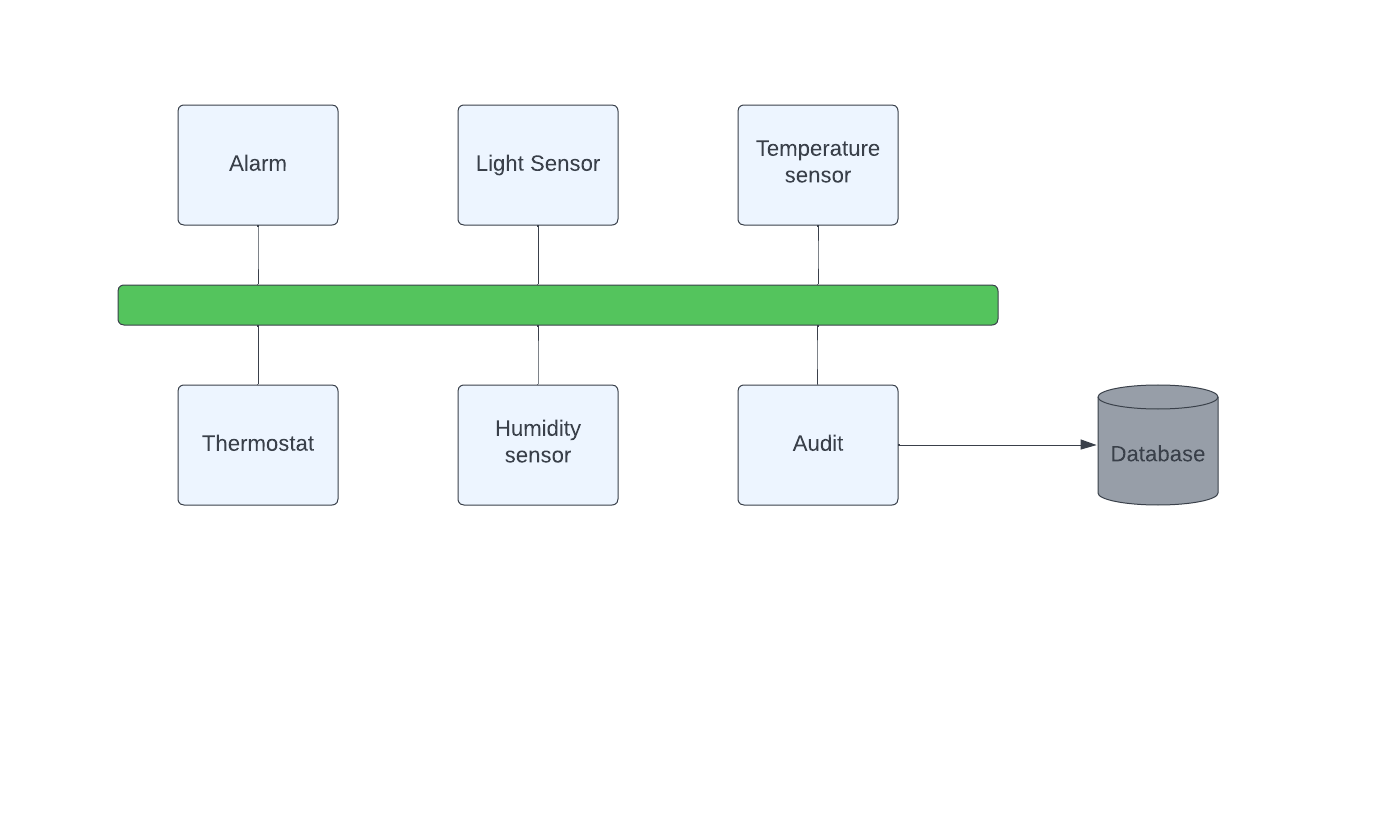
SOA Generic Example

This architecture promotes the interoperability and scalability of a system, but it also entails the complexity of distributed systems for their definition and integration, since it is often difficult to control the modifications in the messages, which can affect the consumers of the different services.

**Pub Sub Pattern**

In the Publish-Subscribe pattern, data producers and consumers exist independently and are anonymous to each other. In general, multiple consumers subscribe to events published by multiple producers. Both actors communicate indirectly through an event bus which is responsible for connecting content publishers with interested subscribers.

All communications in this pattern take place on the event bus so all components must be connected to it. The choice of technology for this bus is critical to its successful operation. Here is an example of a pub-sub system that connects different types of devices to an event bus.

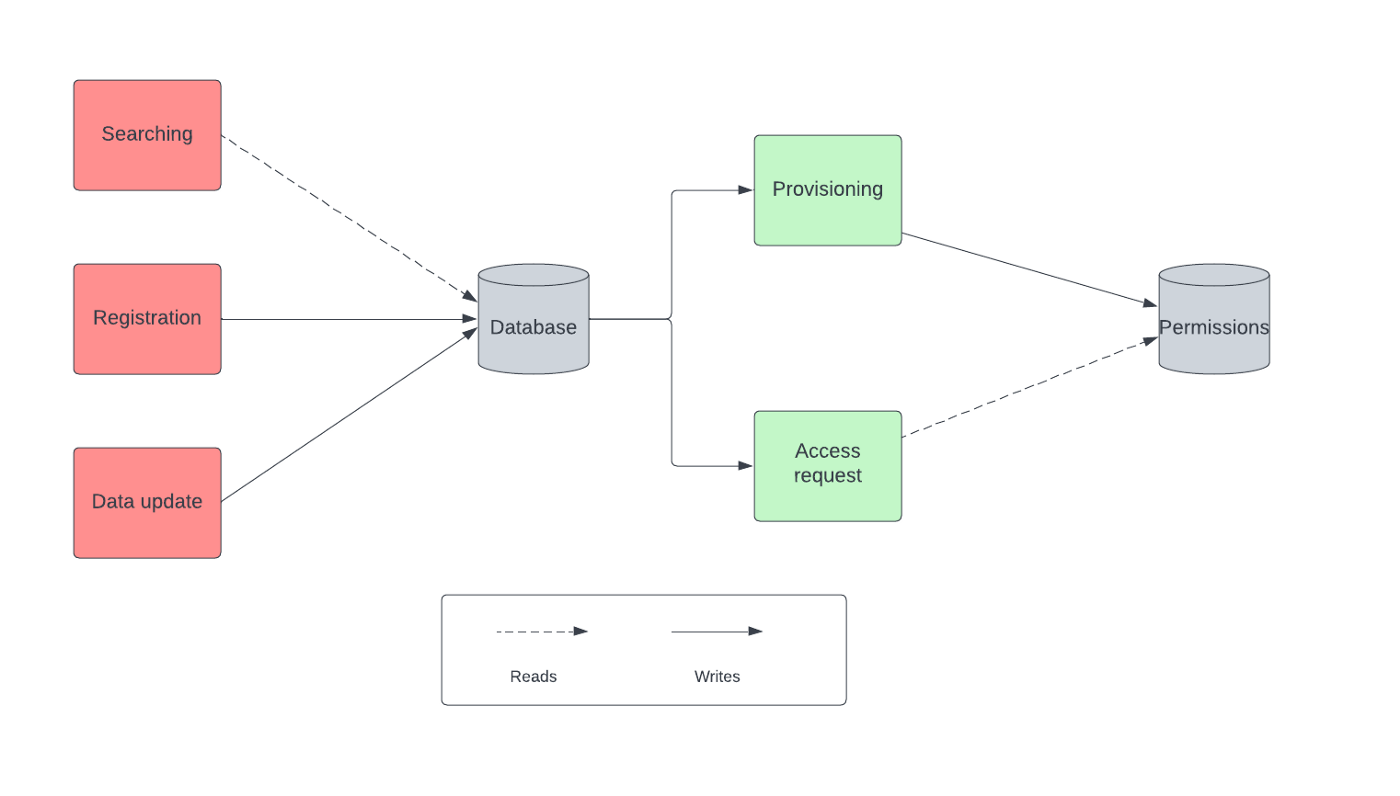


Publish Subscribe Example

This architecture promotes reusability and performance in the exchange of data due to the ease of access to them, optimizing how they are produced and consumed in the event bus. However, it is difficult to think about the performance of these systems given the asynchronous nature of communication. Ultimately, the event bus is the bottleneck in the good or bad performance of the system.

**Shared Data Pattern**

In the Shared Data pattern, multiple components access a set of data through a common data store. No component is entirely responsible for the data or its storage, rather it is a common medium for all consumers. This pattern is particularly useful when multiple components require a large amount of data.



Shared Data Pattern Example

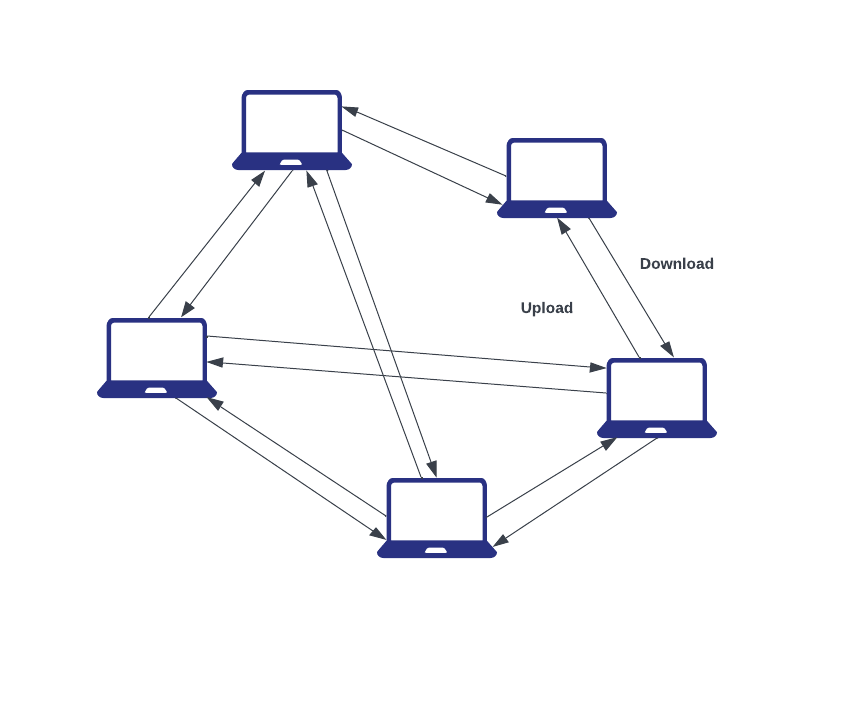
Although this scheme has its own name, we commonly see it as part of other larger systems, for example when in a SOA architecture different services access a common database. Then, we can define access types in reads and writes with different rules and permissions that optimize and protect data access. Today the complexity of this architecture is facilitated, for example, by cloud services such as AWS RDS, which is responsible for provisioning a database with reading replicas, scalability and backups facilitating the management and provisioning of such characteristics

This pattern promotes reliability through data consistency as well as scalability and performance if the data is partitioned correctly. On the other hand, there is also a single point of failure in case the system is not managed correctly.

**p2p Pattern**

Peer-to-peer architectural patterns belong to the category of symmetric client-server patterns. Symmetric in this context refers to the fact that there is no need for a strict division in terms of client, server, and so on in the network of systems. In a peer-to-peer pattern, a single system acts as both client and server

Each system, also called a peer, sends requests to other peers in the network and at the same time receives and services requests from other peers, which are part of the network. This is a great difference when compared to a traditional client server network where a client must only send a request and wait for the server to process.



p2p Network Example

We can find examples of this architecture in file-sharing networks such as [**Gnutella**](https://en.wikipedia.org/wiki/Gnutella), cryptocurrency protocolos such [**Blockchain**](https://www.blockchain.com/)and its[**Bitcoin**](https://bitcoin.org/en/)implementation.

**Service Broker Pattern**

Three main components are involved in a broker system: the broker, the server, and the client.This pattern is used to structure distributed systems with decoupled components. It is, at a certain level, an extension of the Client-Server approach for more complex scenarios.

The *broker* is the message-routing component of your system. It passes messages from client to server and from server to client. These messages are requests for services and replies to those requests, as well as messages about exceptions that have occurred. The requests are coded as calls to the broker’s API. The broker is responsible for error handling in response to these exception reports.

Servers publish their capabilities (services and characteristics) to a broker. Clients request a service from the broker, and the broker then redirects the client to a suitable service from its registry. Good examples of Message broker softwares are [Apache ActiveMQ](https://en.wikipedia.org/wiki/Apache_ActiveMQ), [Apache Kafka](https://en.wikipedia.org/wiki/Apache_Kafka), [RabbitMQ](https://en.wikipedia.org/wiki/RabbitMQ).

Here’s an example on how this pattern would look in a diagram:

Diagram

Description automatically generated

Broker Pattern Example

This pattern is recommended if the relation client-server is not fixed because there are many suitable servers or the availability of the servers changes over time. Also if the choice of server depends on some criterion that is complex enough to be delegated to a separate component. By the contrary setting up or building the broker piece is a challenging task which is usually taken over one of the providers mentioned above.

**Discover your architecture**

The list of patterns could go on, in fact these are the ones that in my personal opinion I consider to be the most commonly found out there. Certainly patterns are born from new experiences, some may apply to a variety of systems and teams while others may apply to a single organization. It is important to observe the reality of our system, look for other people and experiences that are familiar with the problem that is being addressed. It is likely that your solution will require the combination of several of these patterns, complexity is an unavoidable factor in all successful software systems.

**Bonus: Architecture missmatch**

Architecture missmatch is a phenomenon that occurs very commonly when designing an architecture where it is assumed that a component will be used in a way that conflicts with the actual use of that component. This will cause the architecture in general to be difficult to develop and mainly to maintain, making quality attributes not achieved.

This situation can happen at the conceptual level when an architectural scheme is not aligned with the most important attributes of the desired system or when the selected technology is not the right one. For example, if there are indications that the architecture is associated with a pub-sub model, using a relational database as the main mechanism to exchange messages will notoriously affect the final result.

Studying these patterns, and fundamentally taking the time required to define an architecture properly, can save us from many problems, so we should not underestimate this activity at the beginning of each new project that we have to face. Even the selection of technologies is a decision that should come after having the architecture defined and not the other way around as often happens.