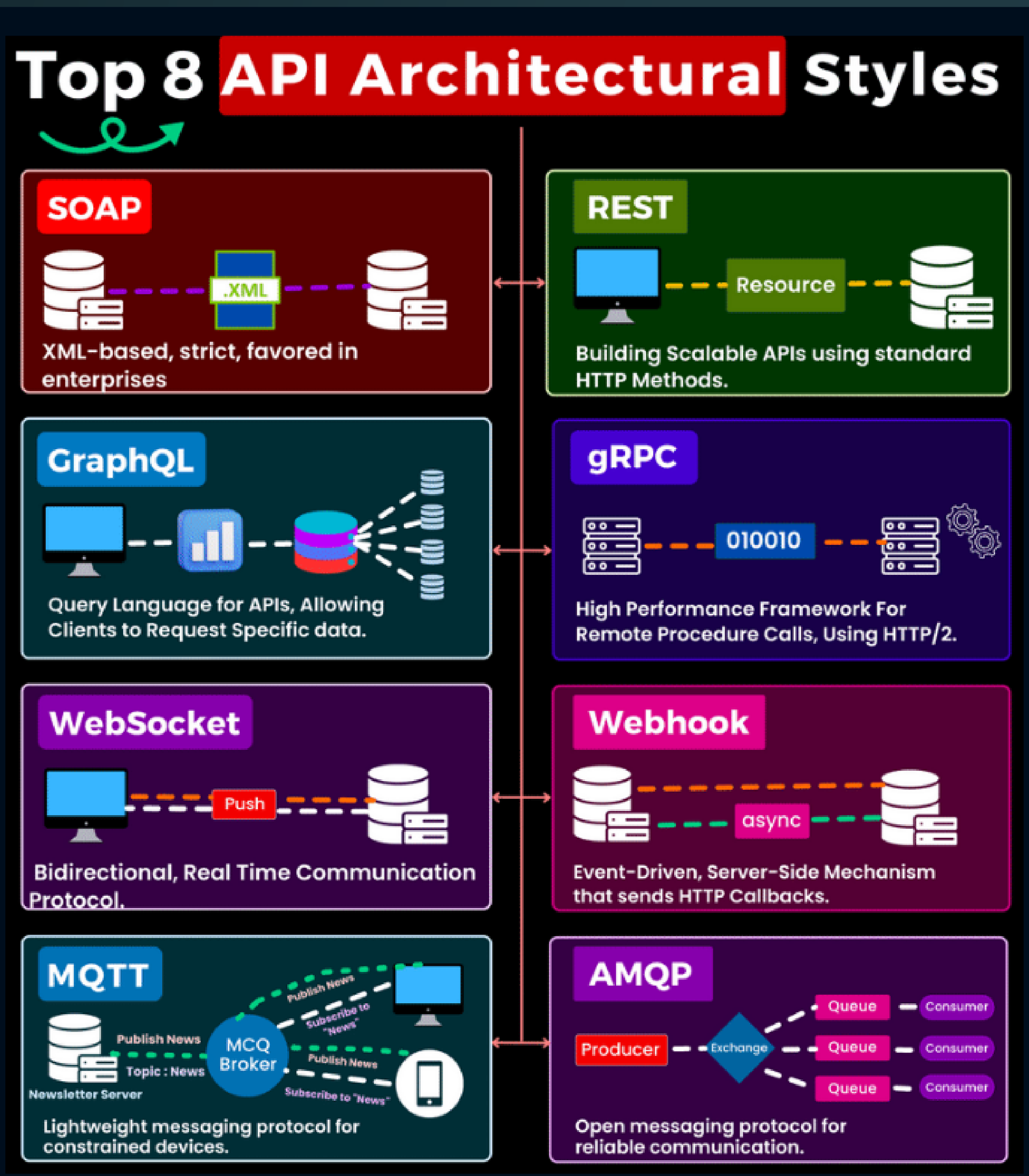
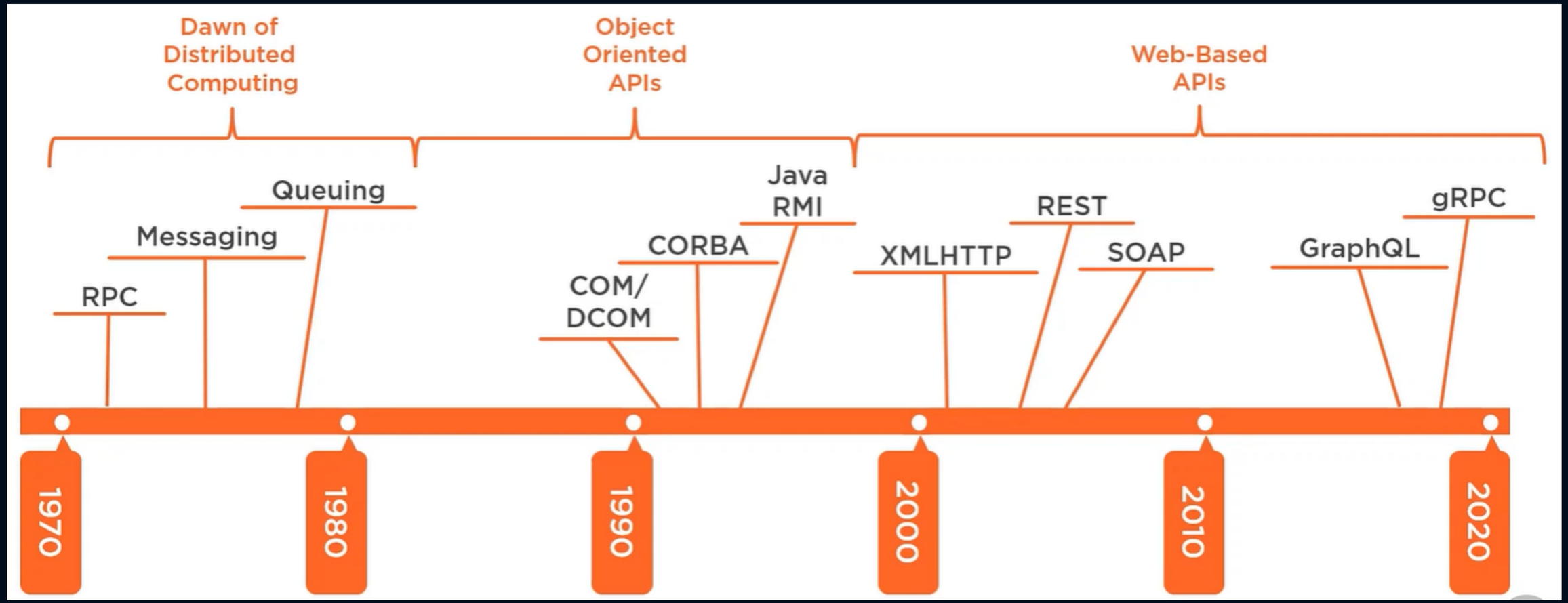
Architectural styles



Arch styles

API Styles Evoluation



API Styles Use Cases

Matching Communication to Technology

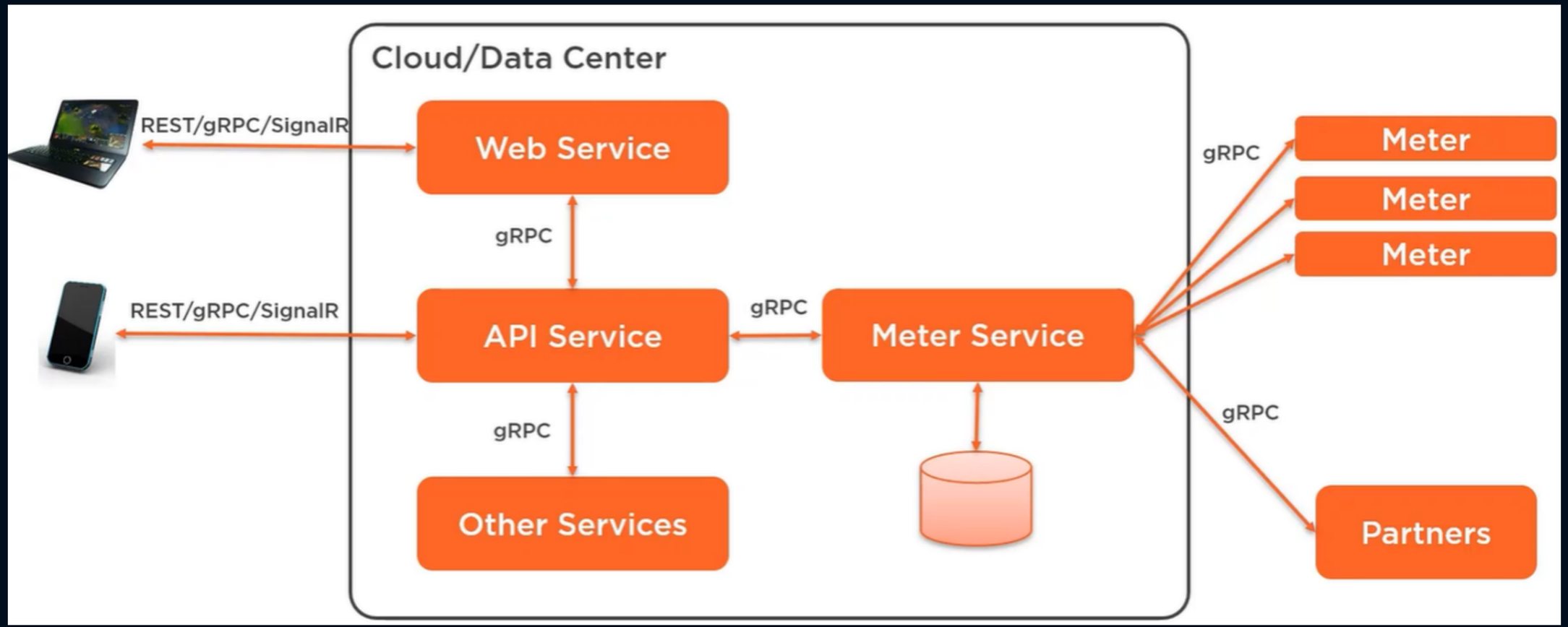
REST: CRUD and purely web apps

SignalR: Multicasting and web messaging

GraphQL: Open querying of large datasets

gRPC: Streaming, low resource clients, and inter-data center

API Styles Example



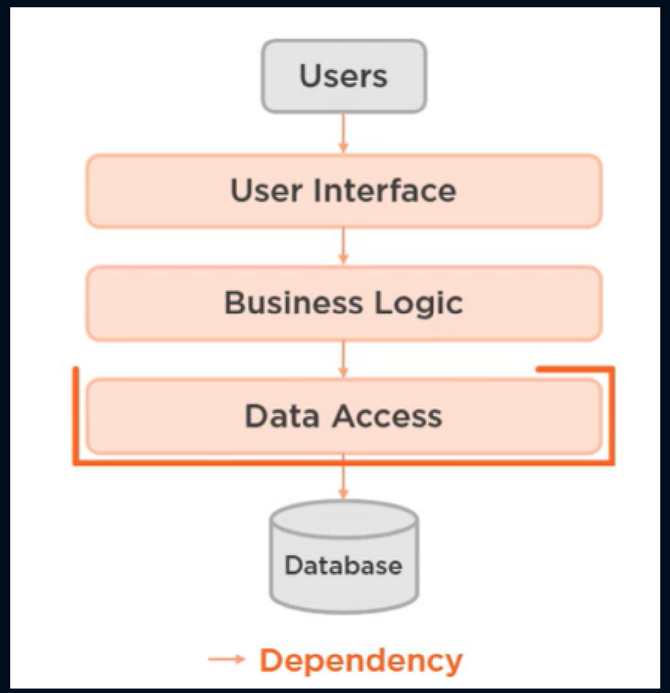
GraphQL vs Rest vs gRPC

Feature	GraphQL	REST	gRPC
Data fetch	Efficient fetching of data. Only required data will be fetched from server.	Extra data might be returned by the server unless new endpoints/query filters are defined on server side.	Extra data might be returned by server unless new endpoints/filters are defined on server side.
HTTP 1.1 vs HTTP 2	Follows request-response model. It can work with either HTTP version but is typically built with HTTP 1.1.	Follows request-response model. It can work with either HTTP version but is still typically built with HTTP 1.1.	Follows client-response model and is based on HTTP 2. Some servers have workarounds to make it work with HTTP 1.1 but it is not the default.
Browser support	Works everywhere.	Works everywhere.	Limited support. Need to use gRPC-Web, which is an extension of gRPC for the web and is based on HTTP 1.1.
Payload data structure	Uses JSON-based payloads to send/receive data.	Mostly uses JSON- and XML-based payloads to send/receive data.	Uses Protocol Buffers by default to serialize payload data.
Code generation	Need to use third-party tools like GraphQL Code Generator to generate client code. Docs and an interactive playground can be natively generated by using GraphiQL.	Need to use third-party tools like Swagger to generate client code.	gRPC has native support for code generation for various target languages.
Request caching	Hard to cache requests by default as every request is sent to the same endpoint.	Easy to cache requests on the client and server sides. Most clients/servers natively support it.	Doesn't support request/response caching by default.

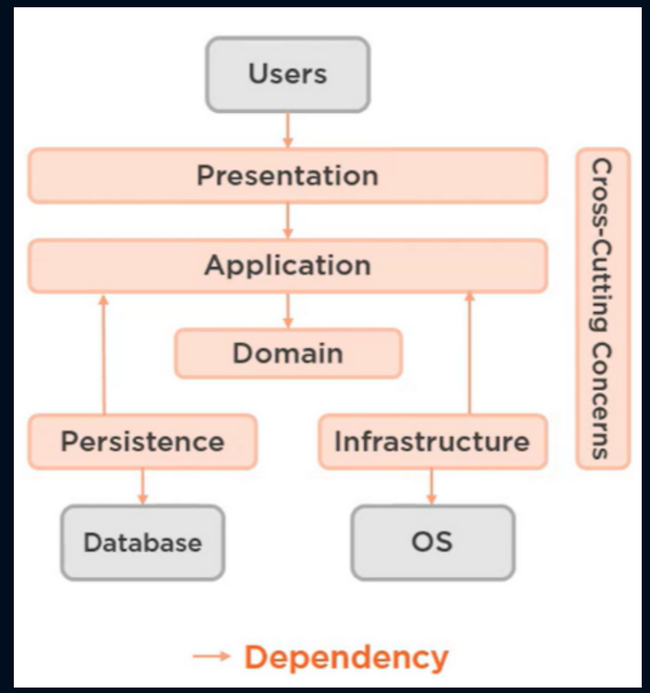
GraphQL vs gRPC vs Rest

Layered architecture

Are common software architectural patterns that organizes an application into logical layers, with each layer having a specific responsibility and level of abstraction. The primary goal of Layered Architecture is to promote separation of concerns and modularity, making the application more maintainable, scalable, and testable.



3 tier layered architecture



4 tier layered architecture

4-tier layered architecture

The goal of the 4-Tier Layered Architecture is to enhance the separation of concerns and improve modularity in larger and more complex applications.

The four-layer domain-centric architecture includes a presentation layer, an application layer embedding the use cases of the application as executable code and abstractions, a domain layer containing only the domain logic of the application, and an infrastructure layer consisting of the persistence and the remaining infrastructure.

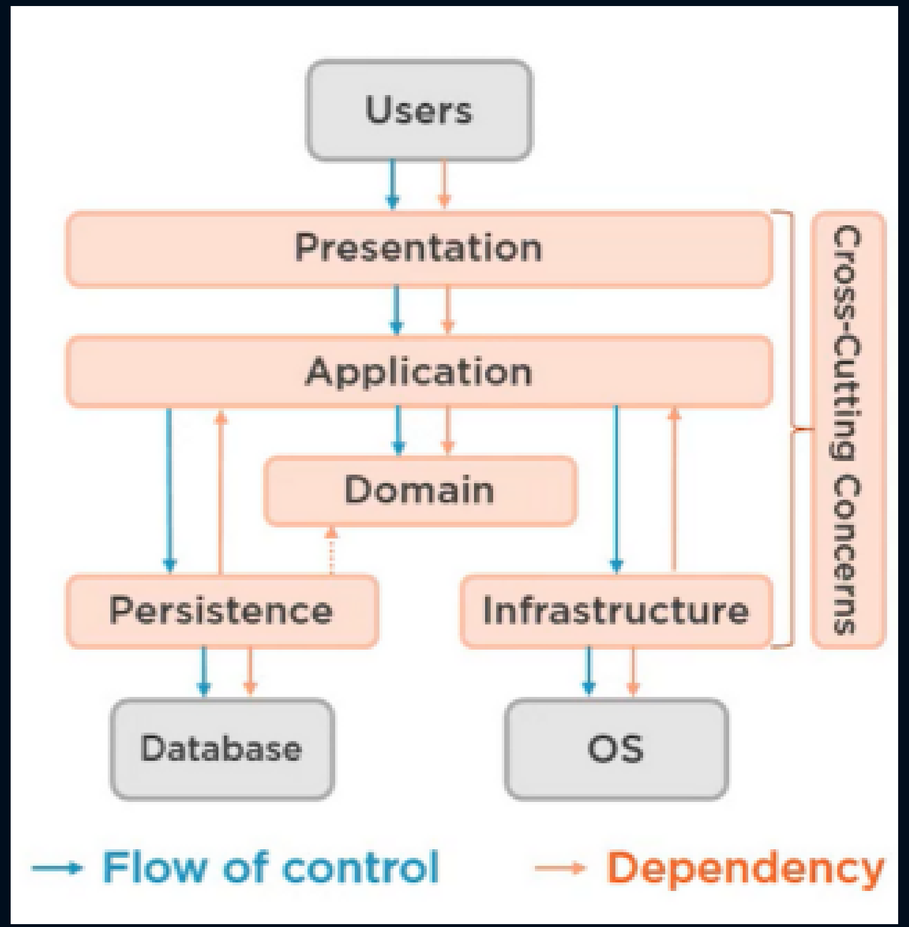
In the application layer, use cases are implemented as executable code structured as high-level representations of application logic.

The application layer knows about the domain layer but does not know about the presentation, persistence, or infrastructure layers.

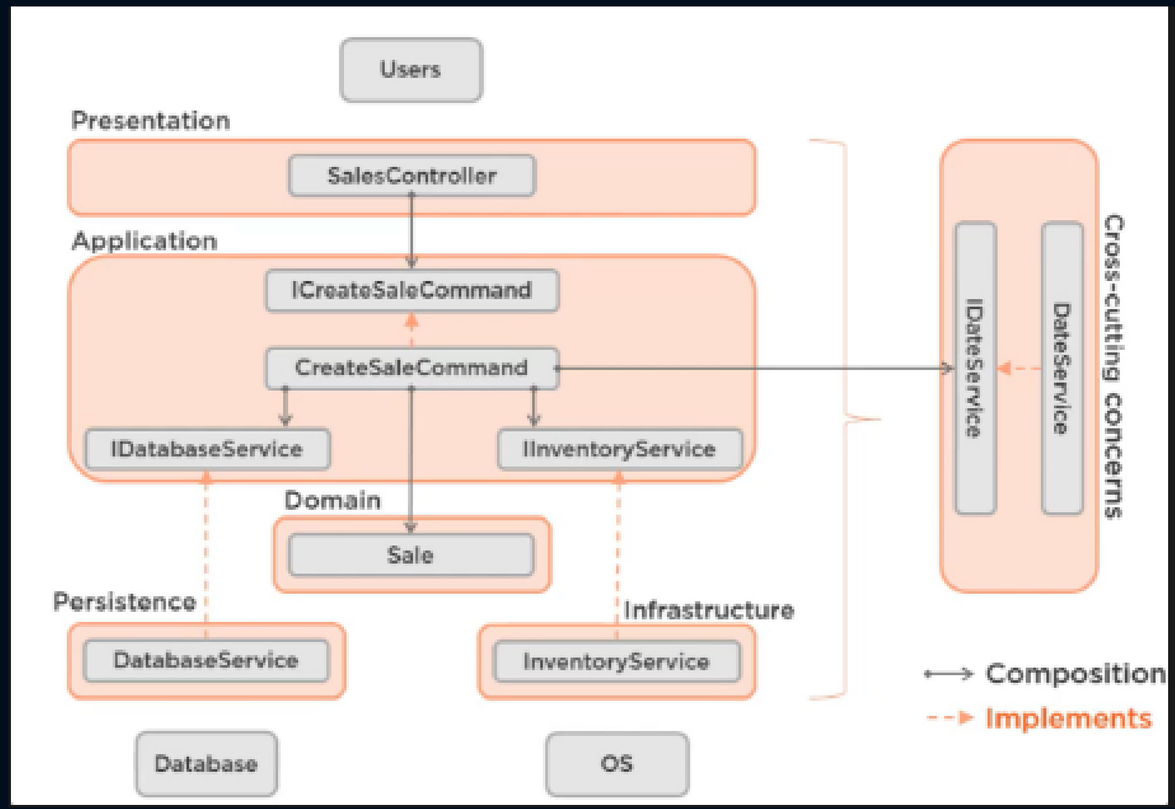
Dependencies in the persistence and infrastructure layers oppose the flow of control in the application due to the dependency inversion principle.

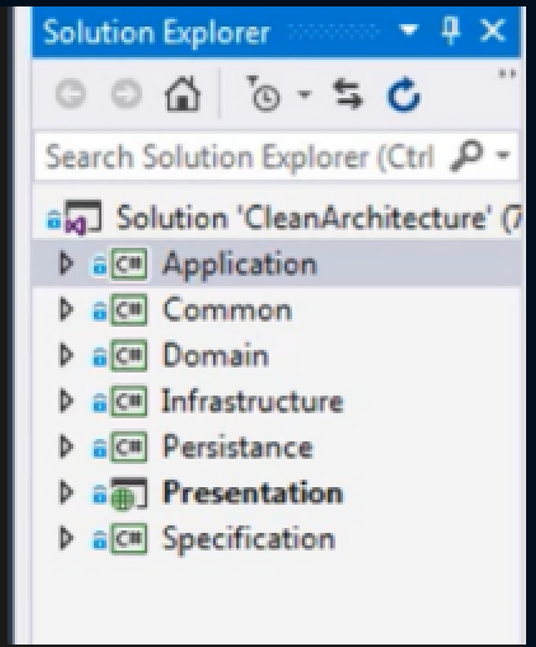
An ORM may require an additional dependency from the persistence project directly to the domain project to map domain entities to tables in the database.

The use of an IoC framework and dependency injection allows the wiring up of all the interfaces and their implementations at run time.



Example





Layered architecture

Task

Quick Exercises

Using the gRPC API Project from C# fundamentals repository. refactor following a Layered architecture (3 tier or 4 tier)