# **What Is a Service Mesh?**

A service mesh can be defined as an infrastructure layer which handles the inter-service communication in a microservice architecture. Service mesh reduces the complexity associated with a microservice architecture and provides lot of the functionalities like:

* Load balancing
* Service discovery
* Health checks
* Authentication
* Traffic management and routing
* Circuit breaking and failover policy
* Security
* Metrics and telemetry
* Fault injection

# **Why Is Service Mesh Necessary?**

In a microservice architecture, handling service to service communication is challenging and most of the time we depend upon third-party libraries or components to provide functionalities like service discovery, load balancing, circuit breaker, metrics, telemetry, and more. Companies like Netflix came up with their own libraries like Hystrix for circuit breakers, Eureka for service discovery, Ribbon for load balancing which are popular and widely used by organizations.

However, these components need to be configured inside your application code and based on the language you are using, the implementation will vary a bit. Anytime these external components are upgraded, you need to update your application, verify it, and deploy the changes. This also creates an issue where now your application code is a mixture of business functionalities and these additional configurations. Needless to say, this tight coupling increases the overall application complexity, since the developer now needs to also understand how these components are configured so that he/she can troubleshoot in case of any issues.

Service Mesh comes to the rescue here. It decouples this complexity from your application and puts it in a service proxy and lets it handle it for you. These service proxies can provide you with a bunch of functionalities like traffic management, circuit breaking, service discovery, authentication, monitoring, security, and much more. Hence from an application standpoint, all it contains is the implementation of business functionalities.

Say, in your microservice architecture, if you have five services talking with each other. Then, instead of building the common necessary functionalities like configuration, routing, telemetry, logging, circuit breaking, etc. inside every microservice, it makes more sense to abstract it into a separate component — called a 'service proxy.'

With the introduction of service mesh, there is a clear segregation of responsibilities. This makes the lives of developers easier. If there is an issue, developers can easily identify the root cause based on whether it is application or network related.

# **How Is Service Mesh Implemented?**

To implement a service mesh, you can deploy a proxy alongside your services. This is also known as the sidecar pattern.

The sidecars abstract the complexity away from the application and handle the functionalities like service discovery, traffic management, load balancing, circuit breaking, etc.

[Envoy](https://www.envoyproxy.io/) from Lyft is the most popular open source proxy designed for cloud native applications. Envoy runs along side every service and provides the necessary features in a platform agnostic manner. All traffic to your service flows through the Envoy proxy.

# **What Is Istio?**

[Istio](https://istio.io/) is an open platform to connect, manage, and secure microservices. It is very popular in the Kubernetes community and is getting widely adopted.

Istio provides additional capabilities in your microservices architecture like intelligent routing, load balancing, service discovery, policy enforcement, in-depth telemetry, circuit breaking and retry functionalities, logging, monitoring, and more.

Istio is one of the best implementations of a service mesh at this point. It enables you to deploy microservices without an in-depth knowledge of the underlying infrastructure.

As more and more organizations start breaking down their monoliths into a microservice architecture, they will reach a point where managing the increasing number of services becomes a burden. Service mesh comes to the rescue in such scenarios and abstracts away all the complexities without the need to make any changes to the application.

# **Design Patterns for Microservices**

Microservice architecture has become the de facto choice for modern application development. Though it solves certain problems, it is not a silver bullet. It has several drawbacks and when using this architecture, there are numerous issues that must be addressed. This brings about the need to learn common patterns in these problems and solve them with reusable solutions. Thus, design patterns for microservices need to be discussed. Before we dive into the design patterns, we need to understand on what principles microservice architecture has been built:

1. Scalability
2. Availability
3. Resiliency
4. Independent, autonomous
5. Decentralized governance
6. Failure isolation
7. Auto-Provisioning
8. Continuous delivery through DevOps

Applying all these principles brings several challenges and issues. Let's discuss those problems and their solutions.

# **1. Decomposition Patterns**

### a. Decompose by Business Capability

#### **Problem**

Microservices is all about making services loosely coupled, applying the single responsibility principle. However, breaking an application into smaller pieces has to be done logically. How do we decompose an application into small services?

#### **Solution**

One strategy is to decompose by business capability. A business capability is something that a business does in order to generate value. The set of capabilities for a given business depend on the type of business. For example, the capabilities of an insurance company typically include sales, marketing, underwriting, claims processing, billing, compliance, etc. Each business capability can be thought of as a service, except it’s business-oriented rather than technical.

### b. Decompose by Subdomain

#### **Problem**

Decomposing an application using business capabilities might be a good start, but you will come across so-called "God Classes" which will not be easy to decompose. These classes will be common among multiple services. For example, the Order class will be used in Order Management, Order Taking, Order Delivery, etc. How do we decompose them?

#### **Solution**

For the "God Classes" issue, DDD (Domain-Driven Design) comes to the rescue. It uses subdomains and bounded context concepts to solve this problem. DDD breaks the whole domain model created for the enterprise into subdomains. Each subdomain will have a model, and the scope of that model will be called the bounded context. Each microservice will be developed around the bounded context.

**Note**: Identifying subdomains is not an easy task. It requires an understanding of the business. Like business capabilities, subdomains are identified by analyzing the business and its organizational structure and identifying the different areas of expertise.

### c. Strangler Pattern

#### **Problem**

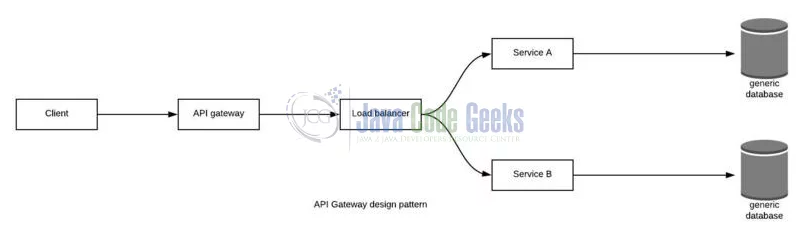
So far, the design patterns we talked about were decomposing applications for greenfield, but 80% of the work we do is with brownfield applications, which are big, monolithic applications. Applying all the above design patterns to them will be difficult because breaking them into smaller pieces at the same time it's being used live is a big task.

#### **Solution**

The Strangler pattern comes to the rescue. The Strangler pattern is based on an analogy to a vine that strangles a tree that it’s wrapped around. This solution works well with web applications, where a call goes back and forth, and for each URI call, a service can be broken into different domains and hosted as separate services. The idea is to do it one domain at a time. This creates two separate applications that live side by side in the same URI space. Eventually, the newly refactored application “strangles” or replaces the original application until finally you can shut off the monolithic application.

# **2. Integration Patterns**

### a. API Gateway Pattern



#### **Problem**

When an application is broken down to smaller microservices, there are a few concerns that need to be addressed:

1. How to call multiple microservices abstracting producer information.
2. On different channels (like desktop, mobile, and tablets), apps need different data to respond for the same backend service, as the UI might be different.
3. Different consumers might need a different format of the responses from reusable microservices. Who will do the data transformation or field manipulation?
4. How to handle different type of Protocols some of which might not be supported by producer microservice.

#### **Solution**

An API Gateway helps to address many concerns raised by microservice implementation, not limited to the ones above.

1. An API Gateway is the single point of entry for any microservice call.
2. It can work as a proxy service to route a request to the concerned microservice, abstracting the producer details.
3. It can fan out a request to multiple services and aggregate the results to send back to the consumer.
4. One-size-fits-all APIs cannot solve all the consumer's requirements; this solution can create a fine-grained API for each specific type of client.
5. It can also convert the protocol request (e.g. AMQP) to another protocol (e.g. HTTP) and vice versa so that the producer and consumer can handle it.
6. It can also offload the authentication/authorization responsibility of the microservice.

### b. Aggregator Pattern

#### **Problem**

We have talked about resolving the aggregating data problem in the API Gateway Pattern. However, we will talk about it here holistically. When breaking the business functionality into several smaller logical pieces of code, it becomes necessary to think about how to collaborate the data returned by each service. This responsibility cannot be left with the consumer, as then it might need to understand the internal implementation of the producer application.

#### **Solution**

The Aggregator pattern helps to address this. It talks about how we can aggregate the data from different services and then send the final response to the consumer. This can be done in two ways:

1. A **composite microservice** will make calls to all the required microservices, consolidate the data, and transform the data before sending back.

2. An **API Gateway** can also partition the request to multiple microservices and aggregate the data before sending it to the consumer.

It is recommended if any business logic is to be applied, then choose a composite microservice. Otherwise, the API Gateway is the established solution.

#### Chained:

##### Problem statement

In eCommerce application, a user wants to purchase a product, then there will be a dependency of order microservice to products microservice.

##### Solution

Chained microservice design pattern will help us to provide the consolidated outcome to our request. The request received by a microservice-1, which is then communicating with microservice-2 and it may be communicating with microservice-3. All these services are synchronous calls.

#### Branch:

##### Problem statement

A microservice may need to get the data from multiple sources including other microservices. When we look at any eCommerce application, those needs get the products data from the catalog, latest pricing, and availability from sellers and any other extra information from the product owner.

##### Solution

Branch microservice pattern, we can say it is a mix of Aggregator & Chain design patterns and allows simultaneous request/response processing from two or more microservices. The invoked microservice can be chains of microservices. Brach pattern can also be used to invoke different chains of microservices, or a single chain, based our business needs.

### c. Client-Side UI Composition Pattern

#### **Problem**

When services are developed by decomposing business capabilities/subdomains, the services responsible for user experience have to pull data from several microservices. In the monolithic world, there used to be only one call from the UI to a backend service to retrieve all data and refresh/submit the UI page. However, now it won't be the same. We need to understand how to do it.

#### **Solution:**

With microservices, the UI has to be designed as a skeleton with multiple sections/regions of the screen/page. Each section will make a call to an individual backend microservice to pull the data. That is called composing UI components specific to service. Frameworks like AngularJS and ReactJS help to do that easily. These screens are known as Single Page Applications (SPA). This enables the app to refresh a particular region of the screen instead of the whole page.

# **3. Database Patterns**

### a. Database per Service

#### **Problem**

There is a problem of how to define database architecture for microservices. Following are the concerns to be addressed:

1. Services must be loosely coupled. They can be developed, deployed, and scaled independently.

2. Business transactions may enforce invariants that span multiple services.

3. Some business transactions need to query data that is owned by multiple services.

4. Databases must sometimes be replicated and sharded in order to scale.

5. Different services have different data storage requirements.

#### **Solution**

To solve the above concerns, one database per microservice must be designed; it must be private to that service only. It should be accessed by the microservice API only. It cannot be accessed by other services directly. For example, for relational databases, we can use private-tables-per-service, schema-per-service, or database-server-per-service. Each microservice should have a separate database id so that separate access can be given to put up a barrier and prevent it from using other service tables.

### b. Shared Database per Service

#### **Problem**

We have talked about one database per service being ideal for microservices, but that is possible when the application is greenfield and to be developed with DDD. But if the application is a monolith and trying to break into microservices, denormalization is not that easy. What is the suitable architecture in that case?

#### **Solution**

A shared database per service is not ideal, but that is the working solution for the above scenario. Most people consider this an anti-pattern for microservices, but for brownfield applications, this is a good start to break the application into smaller logical pieces. This should not be applied for greenfield applications. In this pattern, one database can be aligned with more than one microservice, but it has to be restricted to 2-3 maximum, otherwise scaling, autonomy, and independence will be challenging to execute.

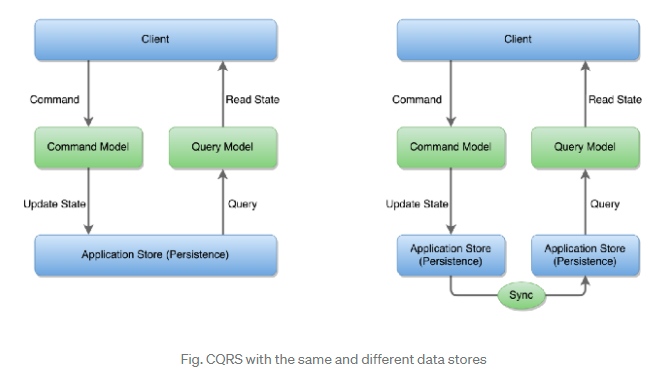
### c. Command Query Responsibility Segregation (CQRS)

#### **Problem**

Once we implement database-per-service, there is a requirement to query, which requires joint data from multiple services — it's not possible. Then, how do we implement queries in microservice architecture?

#### **Solution**

CQRS suggests splitting the application into two parts — the command side and the query side. The command side handles the Create, Update, and Delete requests. The query side handles the query part by using the materialized views. The **event sourcing pattern** is generally used along with it to create events for any data change. Materialized views are kept updated by subscribing to the stream of events.



### d. Saga Pattern

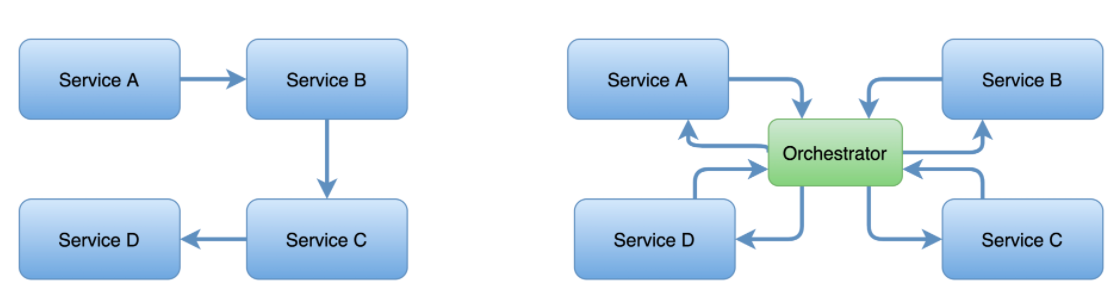
#### **Problem**

When each service has its own database and a business transaction spans multiple services, how do we ensure data consistency across services? For example, for an e-commerce application where customers have a credit limit, the application must ensure that a new order will not exceed the customer’s credit limit. Since Orders and Customers are in different databases, the application cannot simply use a local ACID transaction.

#### **Solution**

A Saga represents a high-level business process that consists of several sub requests, which each update data within a single service. Each request has a compensating request that is executed when the request fails. It can be implemented in two ways:

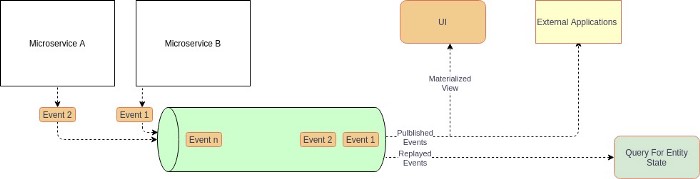
1. Choreography — When there is no central coordination, each service produces and listens to another service’s events and decides if an action should be taken or not.
2. Orchestration — An orchestrator (object) takes responsibility for a saga’s decision making and sequencing business logic.



### e. Event Sourcing

In a Microservice Architecture, especially with **Database per Microservice,** the Microservices need to exchange data. For resilient, highly scalable, and fault-tolerant systems, they should communicate asynchronously by exchanging Events. In such a case, you may want to have Atomic operations, e.g., update the Database and send the message. If you have SQL databases and want to have distributed transactions for a high volume of data, you cannot use the [two-phase locking](https://en.wikipedia.org/wiki/Two-phase_locking) (2PL) as it does not scale. If you use NoSQL Databases and want to have a distributed transaction, you cannot use 2PL as many NoSQL databases do not support two-phase locking.

In such scenarios, use Event based Architecture with Event Sourcing. In traditional databases, the Business Entity with the current “state” is directly stored. In Event Sourcing, any state-changing event or other significant events are stored instead of the entities. It means the modifications of a Business Entity is saved as a series of immutable events. The State of a Business entity is deducted by reprocessing all the Events of that Business entity at a given time. Because data is stored as a series of events rather than via direct updates to data stores, various services can replay events from the event store to compute the appropriate state of their respective data stores.



Event Sourcing by

**Pros**

* Provide atomicity to highly scalable systems.
* Automatic history of the entities, including time travel functionality.
* Loosely coupled and event-driven Microservices.

**Cons**

* Reading entities from the Event store becomes challenging and usually need an additional data store (**CQRS** pattern)
* The overall complexity of the system increases and usually need [Domain-Driven Design](https://en.wikipedia.org/wiki/Domain-driven_design).
* The system needs to handle duplicate events (idempotent) or missing events.
* Migrating the Schema of events becomes challenging.

**When to use Event Sourcing**

* Highly scalable transactional systems with SQL Databases.
* Transactional systems with NoSQL Databases.
* Highly scalable and resilient Microservice Architecture.
* Typical Message Driven or Event-Driven systems (e-commerce, booking, and reservation systems).

**When not to use Event Sourcing**

* Lowly scalable transactional systems with SQL Databases.
* In simple Microservice Architecture where Microservices can exchange data synchronously (e.g., via API).

**Enabling Technology Examples**

*Event Store:* [EventStoreDB](https://www.eventstore.com/), [Apache Kafka](https://kafka.apache.org/), [Confluent Cloud](https://www.confluent.io/confluent-cloud), [AWS Kinesis](https://aws.amazon.com/kinesis/), [Azure Event Hub](https://azure.microsoft.com/en-us/services/event-hubs/), [GCP Pub/Sub](https://cloud.google.com/pubsub), [Azure Cosmos DB](https://docs.microsoft.com/en-us/azure/cosmos-db/introduction), [MongoDB](https://www.mongodb.com/), [Cassandra](https://cassandra.apache.org/). [Amazon DynamoDB](https://aws.amazon.com/dynamodb/?trk=ps_a134p000004f2XeAAI&trkCampaign=acq_paid_search_brand&sc_channel=PS&sc_campaign=acquisition_EMEA&sc_publisher=Google&sc_category=Database&sc_country=EMEA&sc_geo=EMEA&sc_outcome=acq&sc_detail=amazon%20dynamodb&sc_content=DynamoDB_e&sc_matchtype=e&sc_segment=468764879940&sc_medium=ACQ-P|PS-GO|Brand|Desktop|SU|Database|DynamoDB|EMEA|EN|Text|xx|EU&s_kwcid=AL!4422!3!468764879940!e!!g!!amazon%20dynamodb&ef_id=CjwKCAiAq8f-BRBtEiwAGr3DgRRqVmhD5PL323QFmdBJvvOwzxU1nvrGFdbM8ra-DQViD8jjGn-PGBoCWJYQAvD_BwE:G:s&s_kwcid=AL!4422!3!468764879940!e!!g!!amazon%20dynamodb),

# **4. Observability Patterns**

### a. Log Aggregation

#### **Problem**

Consider a use case where an application consists of multiple service instances that are running on multiple machines. Requests often span multiple service instances. Each service instance generates a log file in a standardized format. How can we understand the application behavior through logs for a particular request?

#### **Solution**

We need a centralized logging service that aggregates logs from each service instance. Users can search and analyze the logs. They can configure alerts that are triggered when certain messages appear in the logs. For example, PCF does have Loggeregator, which collects logs from each component (router, controller, diego, etc...) of the PCF platform along with applications. AWS Cloud Watch also does the same.

### b. Performance Metrics

#### **Problem**

When the service portfolio increases due to microservice architecture, it becomes critical to keep a watch on the transactions so that patterns can be monitored and alerts sent when an issue happens. How should we collect metrics to monitor application perfomance?

#### **Solution**

A metrics service is required to gather statistics about individual operations. It should aggregate the metrics of an application service, which provides reporting and alerting. There are two models for aggregating metrics:

* Push — the service pushes metrics to the metrics service e.g. NewRelic, AppDynamics
* Pull — the metrics services pulls metrics from the service e.g. Prometheus

### c. Distributed Tracing

#### **Problem**

In microservice architecture, requests often span multiple services. Each service handles a request by performing one or more operations across multiple services. Then, how do we trace a request end-to-end to troubleshoot the problem?

#### **Solution**

We need a service which

* Assigns each external request a unique external request id.
* Passes the external request id to all services.
* Includes the external request id in all log messages.
* Records information (e.g. start time, end time) about the requests and operations performed when handling an external request in a centralized service.

Spring Cloud Slueth, along with Zipkin server, is a common implementation.

### d. Health Check

#### **Problem**

When microservice architecture has been implemented, there is a chance that a service might be up but not able to handle transactions. In that case, how do you ensure a request doesn't go to those failed instances? With a load balancing pattern implementation.

#### **Solution**

Each service needs to have an endpoint which can be used to check the health of the application, such as /health. This API should o check the status of the host, the connection to other services/infrastructure, and any specific logic.

Spring Boot Actuator does implement a /health endpoint and the implementation can be customized, as well.

# **5. Cross-Cutting Concern Patterns**

### a. External Configuration

#### **Problem**

A service typically calls other services and databases as well. For each environment like dev, QA, UAT, prod, the endpoint URL or some configuration properties might be different. A change in any of those properties might require a re-build and re-deploy of the service. How do we avoid code modification for configuration changes?

#### **Solution**

Externalize all the configuration, including endpoint URLs and credentials. The application should load them either at startup or on the fly.

Spring Cloud config server provides the option to externalize the properties to GitHub and load them as environment properties. These can be accessed by the application on startup or can be refreshed without a server restart.

### b. Service Discovery Pattern

#### **Problem**

When microservices come into the picture, we need to address a few issues in terms of calling services:

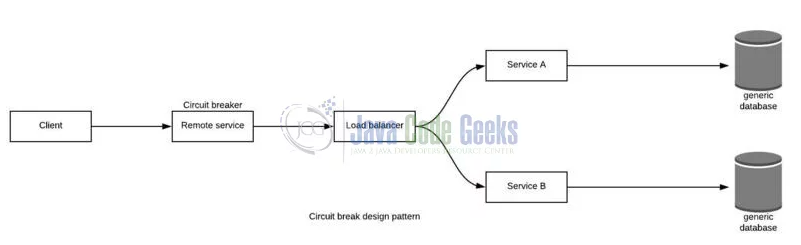
1. With container technology, IP addresses are dynamically allocated to the service instances. Every time the address changes, a consumer service can break and need manual changes.
2. Each service URL has to be remembered by the consumer and become tightly coupled.

So how does the consumer or router know all the available service instances and locations?

#### **Solution**

A service registry needs to be created which will keep the metadata of each producer service. A service instance should register to the registry when starting and should de-register when shutting down. The consumer or router should query the registry and find out the location of the service. The registry also needs to do a health check of the producer service to ensure that only working instances of the services are available to be consumed through it. There are two types of service discovery: client-side and server-side. An example of client-side discovery is Netflix Eureka and an example of server-side discovery is AWS ALB.

### c. Circuit Breaker Pattern



#### **Problem**

A service generally calls other services to retrieve data, and there is the chance that the downstream service may be down. There are two problems with this: first, the request will keep going to the down service, exhausting network resources and slowing performance. Second, the user experience will be bad and unpredictable. How do we avoid cascading service failures and handle failures gracefully?

#### **Solution**

The consumer should invoke a remote service via a proxy that behaves in a similar fashion to an electrical circuit breaker. When the number of consecutive failures crosses a threshold, the circuit breaker trips, and for the duration of a timeout period, all attempts to invoke the remote service will fail immediately. After the timeout expires the circuit breaker allows a limited number of test requests to pass through. If those requests succeed, the circuit breaker resumes normal operation. Otherwise, if there is a failure, the timeout period begins again.

Netflix Hystrix is a good implementation of the circuit breaker pattern. It also helps you to define a fallback mechanism which can be used when the circuit breaker trips. That provides a better user experience.

# **Deployment patterns**

* [Multiple service instances per host](https://microservices.io/patterns/deployment/multiple-services-per-host.html) - deploy multiple service instances on a host
* [Service instance per host](https://microservices.io/patterns/deployment/single-service-per-host.html) - deploy a single service instance on each host
* [Service instance per VM](https://microservices.io/patterns/deployment/service-per-vm.html) - a specialization of the Service Instance per Host pattern where the host is a VM
* [Service instance per Container](https://microservices.io/patterns/deployment/service-per-container.html) - a specialization of the Service Instance per Host pattern where the host is a container

### **Multiple Service Instances per Host (Physical or VM)**

This is one of the most traditional a widely used approach to deploy an application in the Multiple Service Instances per Host pattern. In this method, developers provision one or more physical or virtual hosts and run multiple service instances on each of them. One of the major benefits of this pattern is the efficient usage of the storage since the different service instances uses the same server and OS. Deployment procedures will be relatively faster since we may just have to copy the service to a host and run it. SInce there is no overhead, the initiation of service in this pattern is also quick and seamless. The memory of host will be acquired more if little or complete lack of control on service instance occurs. Since lack of isolation between the instances exists in this case, a small interruption in one of the service interrupts other services rapidly. These are some of the common challenges of this pattern. Efficient information exchanges between the development team and the operations is necessary to avoid further complexities.

### **Service Instance Per Host (Physical or VM)**

In this method instances are run separately on its host. The two specializations of this method are Service Instance per Virtual Machine and Service Instance per Container. Service Instance per Virtual Machine Pattern permits you to package each service as a virtual machine image. Each instance is a virtual machine that runs using a VM image. The major advantage of using this pattern is that it utilizes limited memory and its impossible to steal resources from multiple services since it runs in isolation. It enables systems to leverage sophisticated cloud infrastructure such as AWS and enjoy the benefits of load balancing and auto-scaling. The deployment procedure is simpler because once its packaged as VM the service becomes a black box and this helps with the implemented technology too. Most VMs are usually delivered in fixed sizes in a typical public IaaS, the possibility of incomplete utilization is higher. Less efficient resource utilization eventually channels to a higher cost for deployment since IaaS providers commonly charge for VMs. It is better to use efficient tools to build and manage the VMs since this pattern can often be time-consuming for the team.

### **Service Instance per Container**

In this model, each specific service instance operates in its corresponding container, which is a virtualization device or mechanism at the operating system level. Some conventional container technologies are Docker and Solaris Zones. In this pattern, the services are packaged as a file image comprising the applications and libraries needed to execute the service, commonly known as a container image. Once it is packaged, it’s required to launch one or more containers and can run several containers on a physical or virtual host. For container management cluster managers such as Kubernetes or Marathon can be utilized. ThisThis pattern also works in isolation like Service Instance per Container Pattern. Containers are lightweight and are easier to build. They also can be initiated quickly since there is no OS booting mechanism. It is essential to administer the container infrastructure and presumably the VM infrastructure if the system doesn’t own a hosted solution such as Amazon EC2 Container Service (ECS). Since the majority of the containers are stationed on an infrastructure that is priced per VM, extra deployment cost and over-provisioning of VMs are to be catered if an unexpected spike in the load occurs.

### **Server-less Deployment**

This model supports Java, Node.js, and Python services. In this pattern, the services are packaged as a zip file and uploaded to the Lambda function, a stateless service. The function runs micro-services instances to handle requests automatically. Organizations are billed for each request based on the execution time and memory used. You will be charged based on the work your server performs. This makes the model cost-efficient compared to others. The most consequential challenge of server-less deployment is that it cannot be applied for long-running services. All applications have to be performed within 300 seconds and your services has to be stateless and must in one of the supported languages.

Without the appropriate strategy, the deployment of microservices can end up being quite upsetting. Understanding the right deployment, scaling and administering requirements are mandatory since each of these services might be written in a variety of frameworks and languages. Studying and examining the pattern thoroughly before choosing is a mandatory procedure to be practiced before selecting the deployment strategy.

### Blue-Green Deployment Pattern

#### **Problem**

With microservice architecture, one application can have many microservices. If we stop all the services then deploy an enhanced version, the downtime will be huge and can impact the business. Also, the rollback will be a nightmare. How do we avoid or reduce downtime of the services during deployment?

#### **Solution**

The blue-green deployment strategy can be implemented to reduce or remove downtime. It achieves this by running two identical production environments, Blue and Green. Let's assume Green is the existing live instance and Blue is the new version of the application. At any time, only one of the environments is live, with the live environment serving all production traffic. All cloud platforms provide options for implementing a blue-green deployment. For more details on this topic, check out [this article](https://dzone.com/articles/blue-green-deployment-for-cloud-native-application).

## Canary deployments

Named after the historical practice of sending actual birds into coal mines to see whether the air quality was safe for humans, canary deployments are a way to test actual production deployments with minimal impact or risk. The so-called canary is a candidate version of a service that catches some subset percentage of incoming requests (say, 1%) to try out new features or builds. Teams can then examine the results, and if things go smoothly, gradually increase deployment to 100% of servers or nodes. And if not? Traffic can be quickly redirected from the canary deployments while the offending code is reviewed and debugged.

Canary deployments can be implemented via integrations with edge routing components responsible for processing inbound user traffic. For example, in a Kubernetes environment, a canary deployment can tap the ingress controller configuration to assign specified percentages of traffic requests to the stable and canary deployments. Routing traffic this way ensures that new services have a chance to prove themselves before receiving a full rollout. If they don’t, they’re sent back to have issues remediated and then put through another round of canary deployment testing when ready.

Volume 0%

This video file cannot be played.(Error Code: 232011)

## A/B testing

A/B testing is similar to canary deployments, with one important difference. While canary deployments tend to focus on identifying bugs and performance bottlenecks, A/B testing focuses on gauging user acceptance of new application features. For example, developers might want to know if new features are popular with users, if they’re easy to discover, or if the UI functions properly.

This pattern uses software routing to activate and test specific features with different traffic segments, exposing new features to a specified percentage of traffic, or to limited groups. The A and B routing segments might send traffic to different builds of the software, or the service instances might even be using the same software build but with different configuration attributes (as specified in the orchestrator or elsewhere).

# Microservice-to-Microservice Communication

## Asynchronous

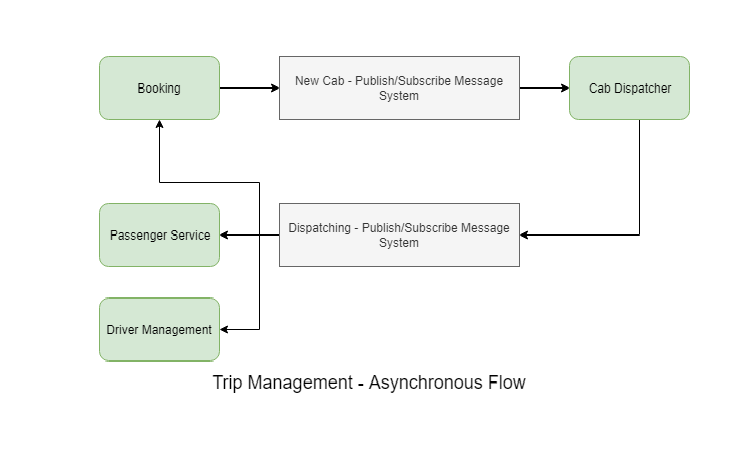
When we talk about asynchronous communication, it means the client makes a call to the server, receives acknowledgment of the request, and forgets about it. The server will process the request and complete it.

Now let's talk about when you need the asynchronous style. If you have an application which is read-heavy, the synchronous style might be a good fit, especially when it needs live data. However, when you have write-heavy transactions and you can't afford to lose data records, you may want to choose asynchronous because, if a downstream system is down and you keep sending synchronous calls to it, you will lose the requests and business transactions. The rule of thumb is to never ever use async for live data read and never ever use sync for business-critical write transactions unless you need the data immediately after write. You need to choose between availability of the data records and strong consistency of the data.

There are different ways we can implement the asynchronous style:

## ****Messaging****

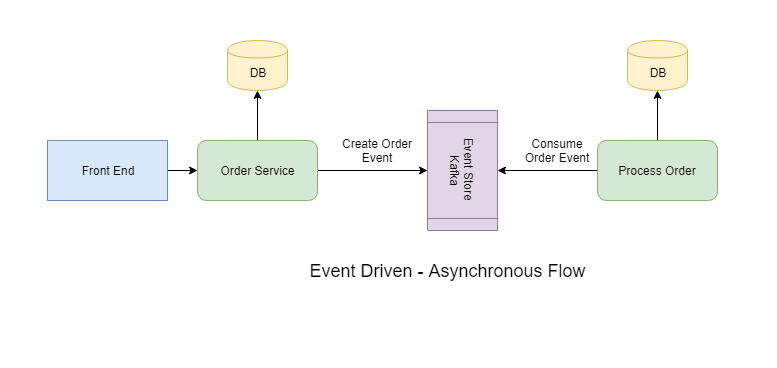
In this approach, the producer will send the messages to a message broker and he consumer can listen to the message broker to receive the message and process it accordingly. There are two patterns within messaing: one-to-one and one-to-many. We talked about some of the complexity synchronous style brings, but some of it is eliminated by default in the messaging style. For example, service discovery becomes irrelevant as the consumer and producer both talk only to the message broker. Load balancing is handled by scaling up the messaging system. Failure handling is in-built, mostly by the message broker. RabbitMQ, ActiveMQ, and Kafka are the best-known solutions in cloud platforms for messaging.



## ****Event-Driven****

The event-driven method looks similar to messaging, but it serves a different purpose. Instead of sending messages, it will send event details to the message broker along with the payload. Consumers will identify what the event is and how to react to it. This enables more loose coupling. There are different types of payloads that can be passed:

* Full payload — This will have all the data related to the event required by the consumer to take further action. However, this makes it more tightly coupled.
* Resource URL — This will be just a URL to a resource that represents the event.
* Only event — No payload will be sent. The consumer will know based on on the event name how to retrieve relevant data from other sources, like databases or queues.



There are other styles, like choreography style, but I personally don't like that. It is too complicated to be implemented. This can only be done with the synchronous style.

That's all for this blog. Let me know your experience with microservice-to-microservice communication.

# **Communication in a microservice architecture**

In a monolithic application running on a single process, components invoke one another using language-level method or function calls. These can be strongly coupled if you're creating objects with code (for example, new ClassName()), or can be invoked in a decoupled way if you're using Dependency Injection by referencing abstractions rather than concrete object instances. Either way, the objects are running within the same process. The biggest challenge when changing from a monolithic application to a microservices-based application lies in changing the communication mechanism. A direct conversion from in-process method calls into RPC calls to services will cause a chatty and not efficient communication that won't perform well in distributed environments. The challenges of designing distributed system properly are well enough known that there's even a canon known as the [Fallacies of distributed computing](https://en.wikipedia.org/wiki/Fallacies_of_distributed_computing) that lists assumptions that developers often make when moving from monolithic to distributed designs.

There isn't one solution, but several. One solution involves isolating the business microservices as much as possible. You then use asynchronous communication between the internal microservices and replace fine-grained communication that's typical in intra-process communication between objects with coarser-grained communication. You can do this by grouping calls, and by returning data that aggregates the results of multiple internal calls, to the client.

A microservices-based application is a distributed system running on multiple processes or services, usually even across multiple servers or hosts. Each service instance is typically a process. Therefore, services must interact using an inter-process communication protocol such as HTTP, AMQP, or a binary protocol like TCP, depending on the nature of each service.

The microservice community promotes the philosophy of "[smart endpoints and dumb pipes](https://simplicable.com/new/smart-endpoints-and-dumb-pipes)" This slogan encourages a design that's as decoupled as possible between microservices, and as cohesive as possible within a single microservice. As explained earlier, each microservice owns its own data and its own domain logic. But the microservices composing an end-to-end application are usually simply choreographed by using REST communications rather than complex protocols such as WS-\* and flexible event-driven communications instead of centralized business-process-orchestrators.

The two commonly used protocols are HTTP request/response with resource APIs (when querying most of all), and lightweight asynchronous messaging when communicating updates across multiple microservices. These are explained in more detail in the following sections.

## Communication types

Client and services can communicate through many different types of communication, each one targeting a different scenario and goals. Initially, those types of communications can be classified in two axes.

The first axis defines if the protocol is synchronous or asynchronous:

* Synchronous protocol. HTTP is a synchronous protocol. The client sends a request and waits for a response from the service. That's independent of the client code execution that could be synchronous (thread is blocked) or asynchronous (thread isn't blocked, and the response will reach a callback eventually). The important point here is that the protocol (HTTP/HTTPS) is synchronous and the client code can only continue its task when it receives the HTTP server response.
* Asynchronous protocol. Other protocols like AMQP (a protocol supported by many operating systems and cloud environments) use asynchronous messages. The client code or message sender usually doesn't wait for a response. It just sends the message as when sending a message to a RabbitMQ queue or any other message broker.

The second axis defines if the communication has a single receiver or multiple receivers:

* Single receiver. Each request must be processed by exactly one receiver or service. An example of this communication is the [Command pattern](https://en.wikipedia.org/wiki/Command_pattern).
* Multiple receivers. Each request can be processed by zero to multiple receivers. This type of communication must be asynchronous. An example is the [publish/subscribe](https://en.wikipedia.org/wiki/Publish%E2%80%93subscribe_pattern) mechanism used in patterns like [Event-driven architecture](https://microservices.io/patterns/data/event-driven-architecture.html). This is based on an event-bus interface or message broker when propagating data updates between multiple microservices through events; it's usually implemented through a service bus or similar artifact like [Azure Service Bus](https://azure.microsoft.com/services/service-bus/) by using [topics and subscriptions](https://docs.microsoft.com/en-us/azure/service-bus-messaging/service-bus-dotnet-how-to-use-topics-subscriptions).

A microservice-based application will often use a combination of these communication styles. The most common type is single-receiver communication with a synchronous protocol like HTTP/HTTPS when invoking a regular Web API HTTP service. Microservices also typically use messaging protocols for asynchronous communication between microservices.

These axes are good to know so you have clarity on the possible communication mechanisms, but they're not the important concerns when building microservices. Neither the asynchronous nature of client thread execution nor the asynchronous nature of the selected protocol are the important points when integrating microservices. What is important is being able to integrate your microservices asynchronously while maintaining the independence of microservices, as explained in the following section.

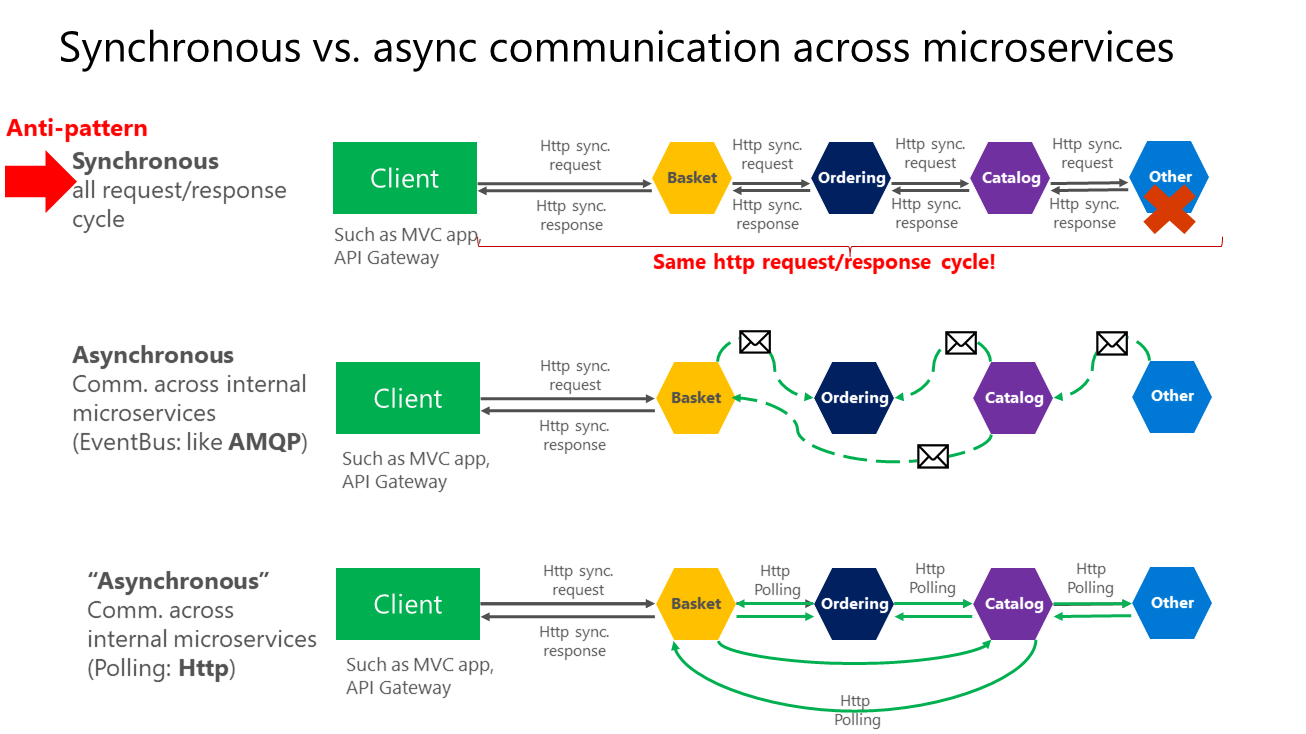
## Asynchronous microservice integration enforces microservice's autonomy

As mentioned, the important point when building a microservices-based application is the way you integrate your microservices. Ideally, you should try to minimize the communication between the internal microservices. The fewer communications between microservices, the better. But in many cases, you'll have to somehow integrate the microservices. When you need to do that, the critical rule here is that the communication between the microservices should be asynchronous. That doesn't mean that you have to use a specific protocol (for example, asynchronous messaging versus synchronous HTTP). It just means that the communication between microservices should be done only by propagating data asynchronously, but try not to depend on other internal microservices as part of the initial service's HTTP request/response operation.

If possible, never depend on synchronous communication (request/response) between multiple microservices, not even for queries. The goal of each microservice is to be autonomous and available to the client consumer, even if the other services that are part of the end-to-end application are down or unhealthy. If you think you need to make a call from one microservice to other microservices (like performing an HTTP request for a data query) to be able to provide a response to a client application, you have an architecture that won't be resilient when some microservices fail.

Moreover, having HTTP dependencies between microservices, like when creating long request/response cycles with HTTP request chains, as shown in the first part of the Figure 4-15, not only makes your microservices not autonomous but also their performance is impacted as soon as one of the services in that chain isn't performing well.

The more you add synchronous dependencies between microservices, such as query requests, the worse the overall response time gets for the client apps.



**Figure 4-15**. Anti-patterns and patterns in communication between microservices

As shown in the above diagram, in synchronous communication a "chain" of requests is created between microservices while serving the client request. This is an anti-pattern. In asynchronous communication microservices use asynchronous messages or http polling to communicate with other microservices, but the client request is served right away.

If your microservice needs to raise an additional action in another microservice, if possible, do not perform that action synchronously and as part of the original microservice request and reply operation. Instead, do it asynchronously (using asynchronous messaging or integration events, queues, etc.). But, as much as possible, do not invoke the action synchronously as part of the original synchronous request and reply operation.

And finally (and this is where most of the issues arise when building microservices), if your initial microservice needs data that's originally owned by other microservices, do not rely on making synchronous requests for that data. Instead, replicate or propagate that data (only the attributes you need) into the initial service's database by using eventual consistency (typically by using integration events, as explained in upcoming sections).

As noted earlier in the [Identifying domain-model boundaries for each microservice](https://docs.microsoft.com/en-us/dotnet/architecture/microservices/architect-microservice-container-applications/identify-microservice-domain-model-boundaries) section, duplicating some data across several microservices isn't an incorrect design—on the contrary, when doing that you can translate the data into the specific language or terms of that additional domain or Bounded Context. For instance, in the [eShopOnContainers application](https://github.com/dotnet-architecture/eShopOnContainers) you have a microservice named identity-api that's in charge of most of the user's data with an entity named User. However, when you need to store data about the user within the Ordering microservice, you store it as a different entity named Buyer. The Buyer entity shares the same identity with the original User entity, but it might have only the few attributes needed by the Ordering domain, and not the whole user profile.

You might use any protocol to communicate and propagate data asynchronously across microservices in order to have eventual consistency. As mentioned, you could use integration events using an event bus or message broker or you could even use HTTP by polling the other services instead. It doesn't matter. The important rule is to not create synchronous dependencies between your microservices.

The following sections explain the multiple communication styles you can consider using in a microservice-based application.

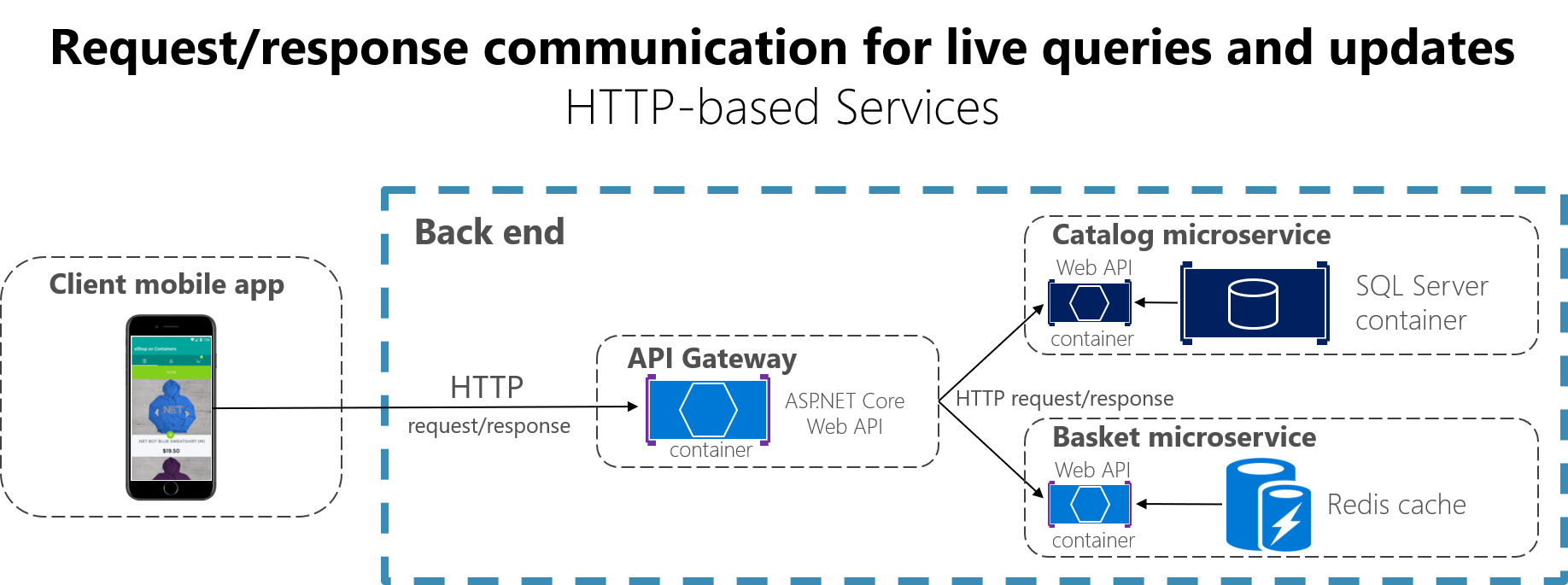
## Communication styles

There are many protocols and choices you can use for communication, depending on the communication type you want to use. If you're using a synchronous request/response-based communication mechanism, protocols such as HTTP and REST approaches are the most common, especially if you're publishing your services outside the Docker host or microservice cluster. If you're communicating between services internally (within your Docker host or microservices cluster), you might also want to use binary format communication mechanisms (like WCF using TCP and binary format). Alternatively, you can use asynchronous, message-based communication mechanisms such as AMQP.

There are also multiple message formats like JSON or XML, or even binary formats, which can be more efficient. If your chosen binary format isn't a standard, it's probably not a good idea to publicly publish your services using that format. You could use a non-standard format for internal communication between your microservices. You might do this when communicating between microservices within your Docker host or microservice cluster (for example, Docker orchestrators), or for proprietary client applications that talk to the microservices.

### Request/response communication with HTTP and REST

When a client uses request/response communication, it sends a request to a service, then the service processes the request and sends back a response. Request/response communication is especially well suited for querying data for a real-time UI (a live user interface) from client apps. Therefore, in a microservice architecture you'll probably use this communication mechanism for most queries, as shown in Figure 4-16.



**Figure 4-16**. Using HTTP request/response communication (synchronous or asynchronous)

When a client uses request/response communication, it assumes that the response will arrive in a short time, typically less than a second, or a few seconds at most. For delayed responses, you need to implement asynchronous communication based on [messaging patterns](https://docs.microsoft.com/en-us/azure/architecture/patterns/category/messaging) and [messaging technologies](https://en.wikipedia.org/wiki/Message-oriented_middleware), which is a different approach that we explain in the next section.

A popular architectural style for request/response communication is [REST](https://en.wikipedia.org/wiki/Representational_state_transfer). This approach is based on, and tightly coupled to, the [HTTP](https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol) protocol, embracing HTTP verbs like GET, POST, and PUT. REST is the most commonly used architectural communication approach when creating services. You can implement REST services when you develop ASP.NET Core Web API services.

There's additional value when using HTTP REST services as your interface definition language. For instance, if you use [Swagger metadata](https://swagger.io/) to describe your service API, you can use tools that generate client stubs that can directly discover and consume your services.