**Why Microservices:**

To understand microservice, let's first take a look at monolithic software. In monolithic software, we mainly use three-tier architecture:

* Presentation layer
* Business layer
* Data access layer

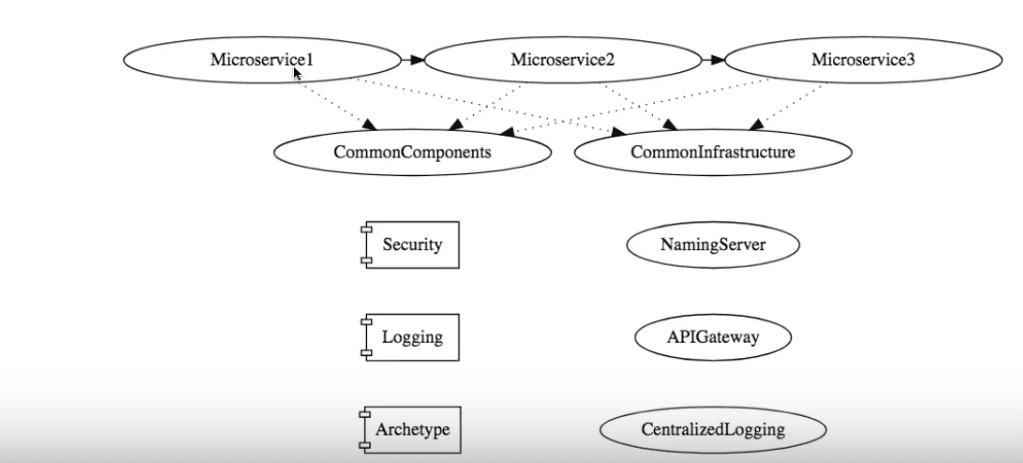
In monolithic application follows multi module project, one module is depended on another module. If we need to changes occurs in one module it will affect another module. Entire project one code base follow like java or .net.

**How many ways to communicate two microservices:**

There are two basic messaging patterns that microservices can use to communicate with other microservices.

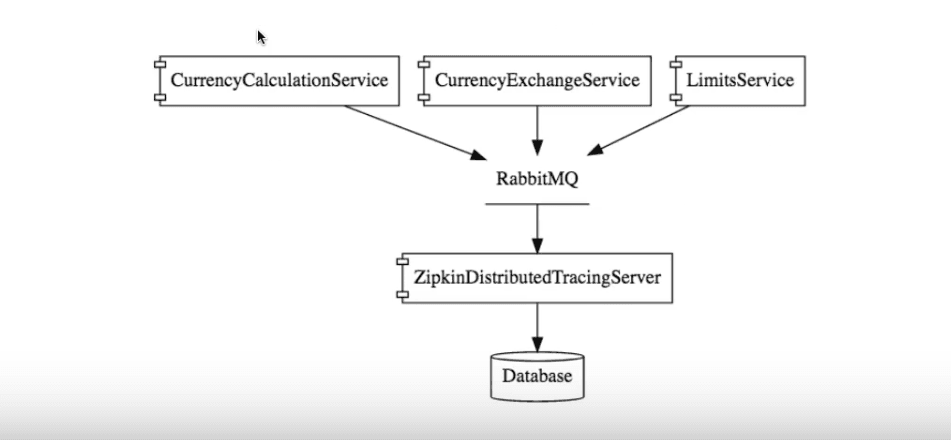
1. **Synchronous communication:** In this pattern, a service calls an API that another service exposes, using a protocol such as HTTP or RPC. This option is a synchronous messaging pattern because the caller waits for a response from the receiver.

Ex: RestTemplate, FeignClient and etc



1. **Asynchronous message passing:** In this pattern, a service sends message without waiting for a response, and one or more services process the message asynchronously.

Ex: RabbitMQ, Apache Kafka, Apache ActiveMQ and etc.



Advantages:

* **Reduced coupling**. The message sender does not need to know about the consumer.
* **Multiple subscribers**. Using a pub/sub model, multiple consumers can subscribe to receive events. See [Event-driven architecture style](https://docs.microsoft.com/en-us/azure/architecture/guide/architecture-styles/event-driven).
* **Failure isolation**. If the consumer fails, the sender can still send messages. The messages will be picked up when the consumer recovers. This ability is especially useful in microservices architecture, because each service has its own lifecycle. Synchronous APIs, on the other hand, require the downstream service to be available or the operation fails.
* **Responsiveness**. An upstream service can reply faster if it does not wait on downstream services. This is especially useful in microservices architecture. If there is a chain of service dependencies (service A calls B, which calls C, and so on), waiting on synchronous calls can add unacceptable amounts of latency.
* **Load leveling**. A queue can act as a buffer to level the workload, so that receivers can process messages at their own rate.
* **Workflows**. Queues can be used to manage a workflow, by check-pointing the message after each step in the workflow.

**Event-driven architecture with microservices using event sourcing and CQRS (Command Query Responsibility Segregation):**

For example, there are two microservices running in their own containers: 'Order' and 'Customer.'

As part of microservices best practices, each microservice should have its own database.

The Order microservices access the Order database and the Customer microservice access the Customer database.

In this scenario, the relationships among the tables cannot be established, as both tables are in separate databases.

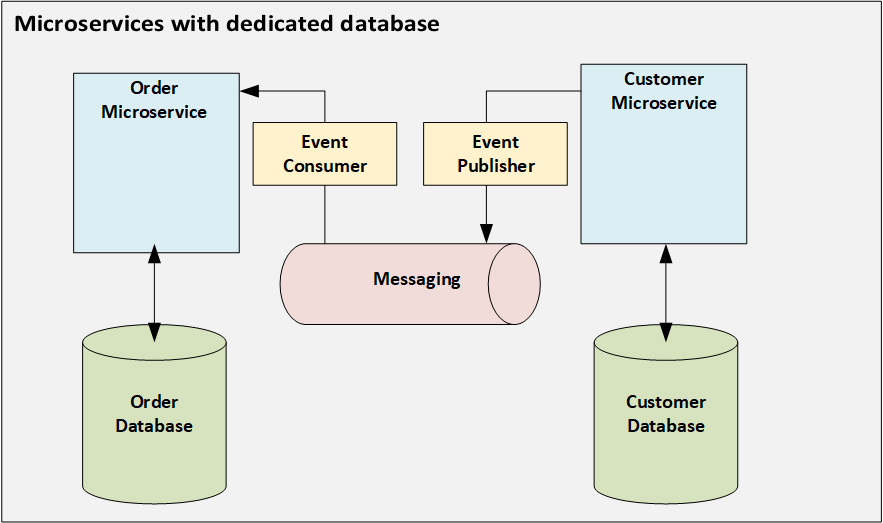
If the Customer microservice wants to update the Order data like address, the Customer microservice can pass the customer id as a request parameter to the HTTP service of the Order microservice to update the Order data for the corresponding customer id in the Order database.

The limitation of this approach is that transaction management cannot be properly handled. If customer data is deleted, the corresponding order also has to be deleted for that customer.

Though this can be achieved with workarounds, like calling a delete service in the Order service, atomicity is not achievable in a straight forward way. This needs to be handled with customization.

To overcome this limitation, we can integrate an event-driven architecture with our microservices components.

As per the below diagram, any change in the customer data will be published as an event to the messaging system, so that the event consumer consumes the data and updates the order data for the given customer changed event.



The limitation of this approach is the atomic updates between the database and publish events to the message queue cannot be handled easily. Though these types of transactions can be handled by distributed transaction management, this is not recommended in a microservices approach, as there might not be support for XA transactions in all scenarios.

To avoid these limitations, event-sourcing can be introduced in this microservices architecture.

In event-sourcing, any event triggered will be stored in an event store. There is no update or delete operations on the data, and every event generated will be stored as a record in the database. If there is a failure in the transaction, the failure event is added as a record in the database. Each record entry will be an atomic operation.

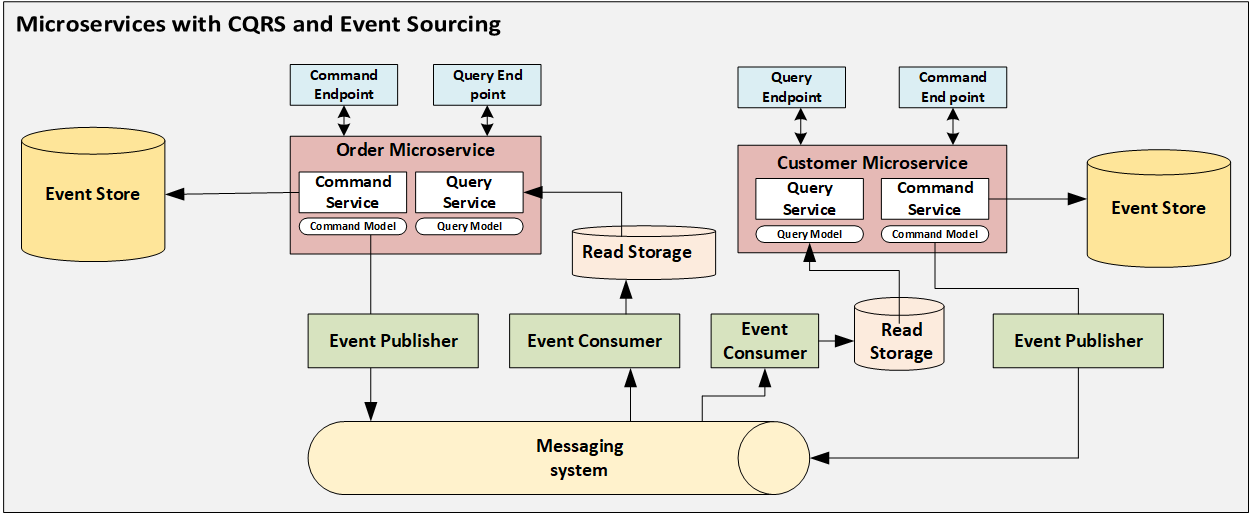
The advantages of event-sourcing are as follows:

* Solves atomicity issues.
* Maintains history and audit of records.
* Can be integrated with data analytics as historical records are maintained.

There are a few limitations, which are:

* Queries on the latest data or a particular piece of data in the event store involve complex handlings.
* To make the data eventually consistent, this involves asynchronous operations because the data flow integrates with messaging systems.
* The model that involves inserting and querying the data is the same and might lead to complexity in the model for mapping with the event store.
* The event store capacity has to be larger in storing all the history of records.

Now we integrate CQRS (Command Query Responsibility Segregation) with event sourcing to overcome the above limitations.



CQRS is another design pattern used in microservices architecture which will have a separate service, model, and database for insert operations in the database. This acts as a command layer and separate service, model, and database for query data that acts as a query layer.

The read database can store a denormalized model where databases like NoSQL (that are horizontally scalable) can be leveraged.

The command layer is used for inserting data into a data store. The query layer is used for querying data from the data store.

In the Customer microservice, when used as a command model, any event change in customer data, like a customer name being added or a customer address being updated, will generate events and publish to the messaging queue. This will also log events in the database in parallel.

The event published in the message queue will be consumed by the event consumer and update the data in the read storage.

The Customer microservice, when used as a query model, needs to retrieve customer data that invokes a query service, which gets data from read storage.

Similarly, events published across microservices also have to be passed through a message queue.

The advantages of CQRS integrated with event sourcing and microservices are:

* Leveraging microservices for modularity with separate databases.
* Leveraging event sourcing for handling atomic operations.
* Maintain historical/audit data for analytics with the implementation of event sourcing.
* CQRS having separate models and services for read and insert operations.
* Request load can be distributed between read and insert operations.
* Read operations can be faster as the load is distributed between read and insert services.
* Read model or DTO need not have all the fields as a command model, and a read model can have required fields by the client view which can save the capacity of the read store.

The limitations of this approach are:

* Additional maintenance of infrastructure, like having separate databases for command and query requests.
* Models should be designed in an optimal way, or this will lead to complexity in handling and troubleshooting.

More information: <https://blog.theodo.com/2019/08/event-driven-architectures-rabbitmq/>

**Saga design pattern:**

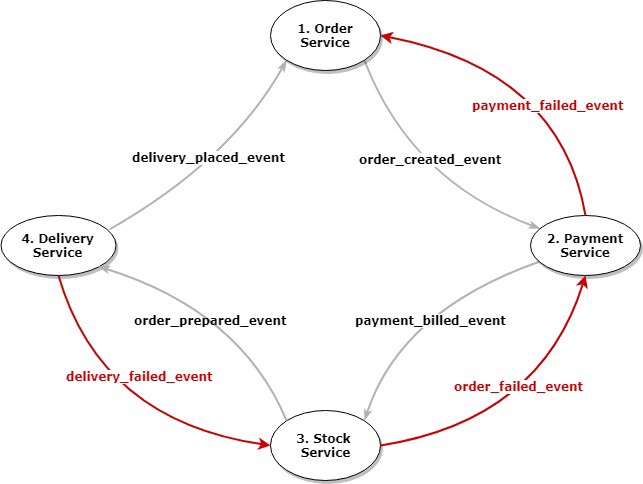
Saga desing pattern better which is an alternative to Two-Face commit. To manage distributed transactions saga and be considered a sequence of local transactions in different micro services each local transaction updates the database of the micro service and then publishes a message or event to trigger the next local transaction in the saga. If one local transaction fails the saga executes a series of compensating transactions lateral exchanges of local transactions forming part of the distributive transaction that has already been executed.

There are too many different types of Saga implementations.

**Choreographic base:**

In this approach, there would be no orchestrator in the system. Each service performs its own process and if the result is successful, it fires a success event for the next step to continue with. In case of a failure it fires a failure event for the previous step. So the services that have worked before this step can sequentially perform rollbacks.

This approach is useful for the cases which consist of only a few steps. As it includes more steps, the events and the design could get more complex to manage.

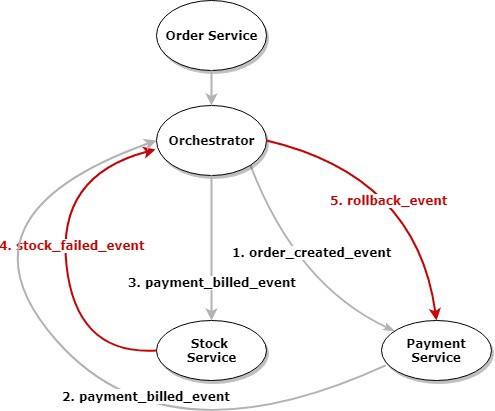


* Order Service creates the order and sends “order\_created\_event” into the message queue.
* Payment Service first receives the message and creates the payment and then sends “payment\_billed\_event” into the message queue.
* Stock Service receives this message from Payment Service, performs required processes and sends “stock\_prepared\_event” to the message queue.
* Delivery Service runs into an error when performing this event and sends back “delivery\_failed\_event” message to roll back the entire process.
* Stock Service receives this failure message and using the transaction\_id provided in it, performs a return process and sends back “stock\_failed\_event”.
* Finally Payment Service receives the failure message and performs its own compensation process and sends “payment\_failed\_event” for Order Service.
* Order Service for such cases can use a retry mechanism with some delay and if the error still persists, it can use some warning mechanisms for a manual check.

**Orchestration-Based Saga:**

In this approach on the other hand, there would be an orchestrator to manage the entire operation from one canter. An orchestrator receives a start command from a source and begins calling related services sequentially. After each successful response, it makes the next call to the following service. If one of the steps fails and the service returns a failure message, the orchestrator makes rollback calls for each previous step/service.

As it also brings along some scalability issues and the risk of a single point of failure, developers should consider necessary recovery actions or simply go for Choreography-Based architecture if they can.



* Order Service creates the order and employs Orchestrator.
* Orchestrator sends “order\_created\_event” for Payment Service.
* Payment Service creates the payment and sends “payment\_billed\_event” for Orchestrator.
* Orchestrator this time sends “payment\_billed\_event” for Stock Service.
* Stock Service runs into an error when performing this event and sends “stock\_failed\_event” for Orchestrator.
* Orchestrator starts rollback cycle and sends “rollback\_event” for the previous service which is Payment Service in this scenario.
* Payment Service performs a compensation process after receiving this message.

### RabbitMQ message broker to enable communication between your distributed systems:

**RabbitMQ**, also known as Open-Source Message Broker, supports multiple message protocols and can be deployed on distributed systems. It is lightweight and makes it easy to deploy applications. It mainly acts as a queue where a message which is entered can first be operated on. RabbitMQ runs on many operating systems and cloud environments and provides a wide range of developer tools for most popular languages. It is a Producer-Consumer style pattern where the producer sends a message and the consumer consumes it. The main features of RabbitMQ are mentioned below:

1. Asynchronous Messaging
2. Distributed Deployment
3. Management & Monitoring
4. Enterprise- and Cloud-Ready

These queue patterns can help to scale your application by communicating between various microservices. We can use these queues for various purposes, like

1. Interaction between core microservices.

2. Decoupling of microservices.

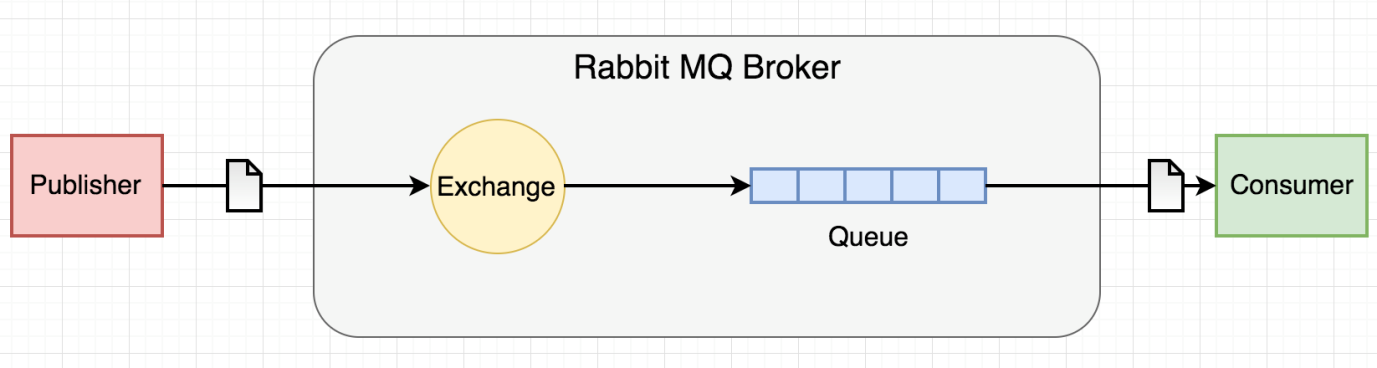
3. Implementing failover mechanisms of microservices.

4. Sending email notifications via message brokers etc.

Wherever two or more core modules need communicate with each other, we should not make direct HTTP calls, as they can make the core layer tightly coupled and it will be difficult to manage when there are more instances of each core module. Also, whenever a service is down, the HTTP call pattern will fail, as after a restart, there is no way to track old HTTP request calls. This results in the need for RabbitMQ.

**Installation process:**

* Download the latest Erlang from the [official website](https://www.erlang.org/downloads) for me it is OTP 22.0.



For example sending email notifications via various core microservices. In this pattern, we will have a producer microservice generate the email content and pass it on to the queue. Then this email content is taken by the consumer, who is always listening to the queue for new messages.

**1) Producer:**This layer is responsible for the generation of email content and passing on this content to the message broker in RabbitMQ.

Example code looks here: <https://dzone.com/articles/rabbitmq-in-microservices>

**2) Consumer:**This layer is responsible for consuming the message from the RabbitMQ message broker with the FIFO method, then performing email-related operations.

Brief explanation here: <https://blog.theodo.com/2019/08/event-driven-architectures-rabbitmq/>

Example code looks here: <https://dzone.com/articles/rabbitmq-in-microservices>

Installation process looks here: <https://www.onlinetutorialspoint.com/windows/how-to-install-rabbitmq-on-windows-10.html>

<https://codenotfound.com/rabbitmq-download-install-windows.html>

* 1. Install Docker.
  2. Check docker is proper installation or not using command prompt.

Docker --version.