# Micro Services Communication:

One of the biggest challenges when moving to microservices-based application is changing the communication mechanism. Because microservices are distributed and microservices communicate with each other by inter-service communication on network level. Each microservice has its own instance and process. Therefore, services must interact using an inter-service communication protocols like HTTP, gRPC or message brokers AMQP protocol.

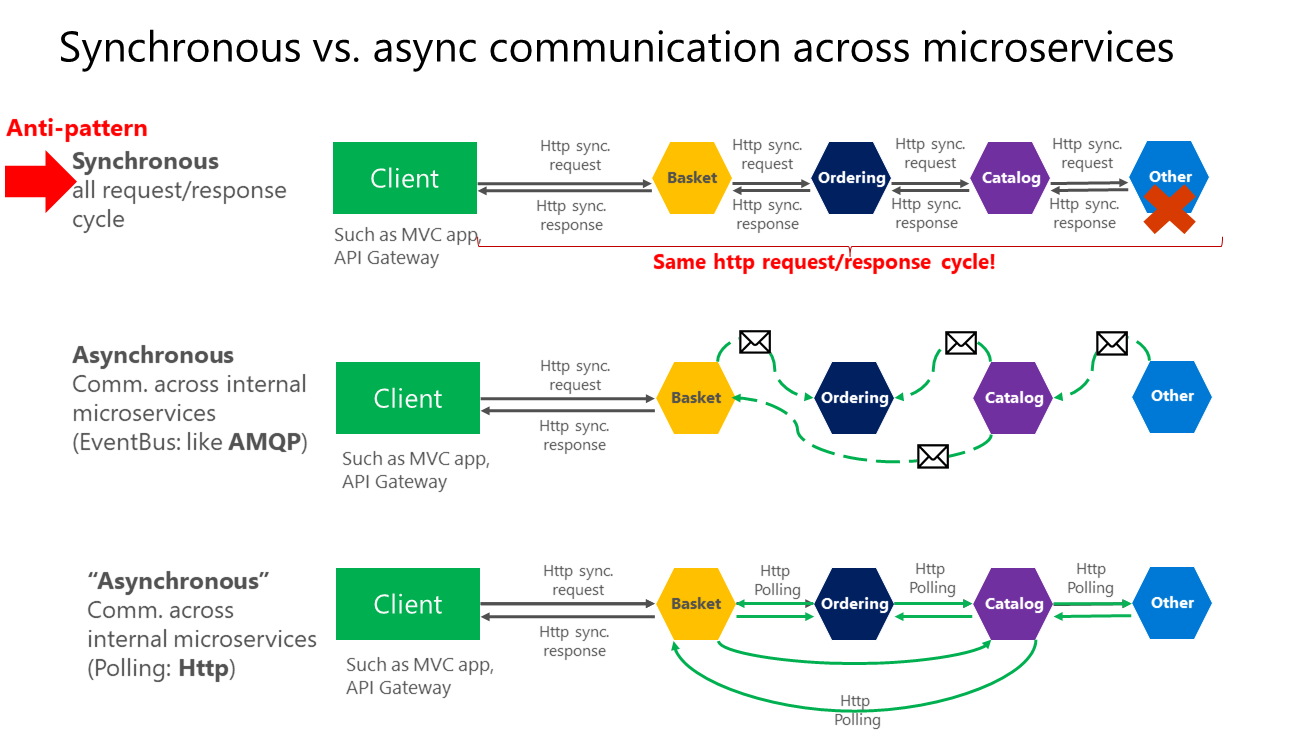
**Communication types:**

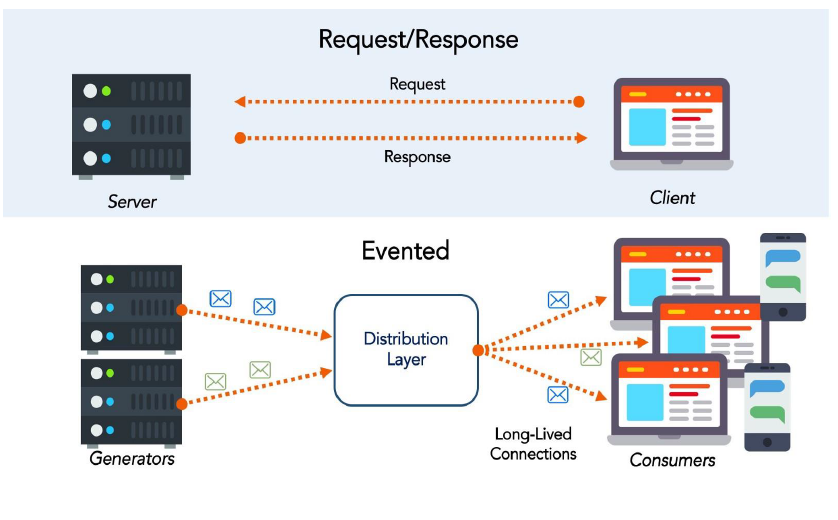
Client and services can communicate through many different types of communication, each one targeting a different scenario and goals. Initially, those types of communications can be classified in two axes.

**Synchronous** **or asynchronous protocol:**

1. 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.
2. Asynchronous protocol. Other protocols like AMQP (**A**dvanced **M**essage **Q**ueuing **P**rotocol -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.

**Single or multiple receivers:**

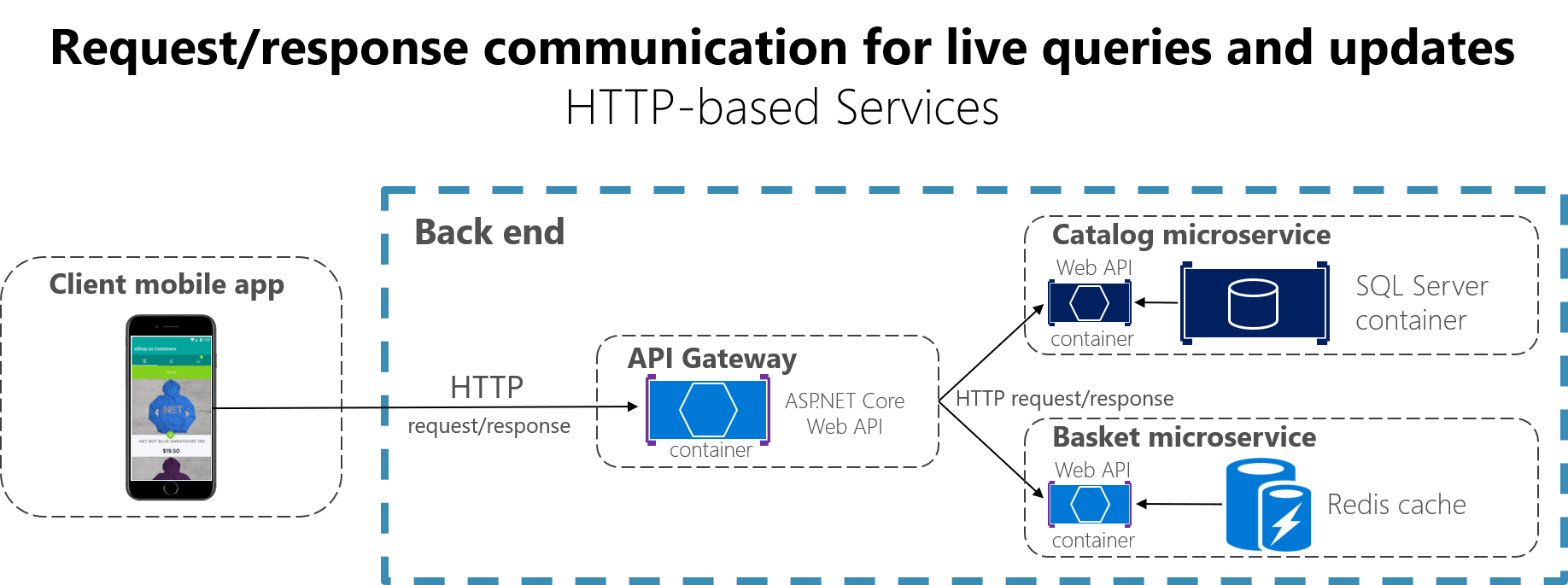
1. Single receiver. Each request must be processed by exactly one receiver or service. An example of this communication is the Command pattern.
2. Multiple receivers. Each request can be processed by zero to multiple receivers. This type of communication must be asynchronous. An example is the publish/subscribe mechanism used in patterns like Event-driven architecture. 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 by using topics and subscriptions.



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.

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.

## 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.

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 and messaging technologies.

Services that frequently communicate with each other synchronously (e.g., via REST requests) introduce latency, increase failure risk, and reduce overall system performance.

**Client – server asynchronous communication:**  
  
Clients can use either web sockets or polling methods to communicate asynchronously.  
  
**Polling:**

Client send periodic requests until answered.

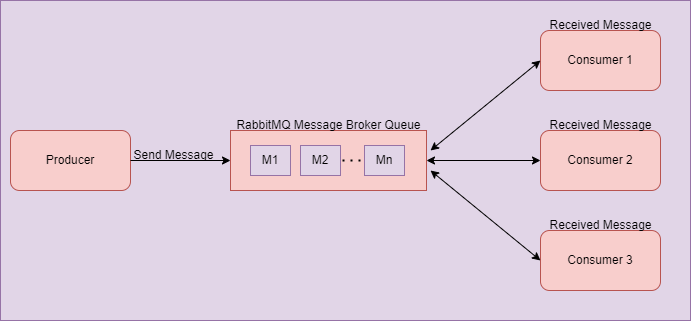
**Web sockets (signal -R):**

1. Client Request: The WebSocket connection begins as a normal HTTP request. The client sends a request to the server, indicating that it wants to establish a WebSocket connection. This request is an HTTP Upgrade request, which asks the server to switch the protocol to WebSocket.
2. Server Response: If the server supports WebSockets, it responds with an HTTP 101 status code (Switching Protocols), agreeing to upgrade the connection to WebSocket.
3. Connection Established: Once the handshake is successful, the WebSocket connection is established, and both the client and server can send and receive messages at any time, without needing to wait for the other to finish.
4. Communication: Data sent through WebSockets is typically in the form of messages (which can be text or binary data). These messages are transmitted as packets of data over the open connection, allowing low-latency communication.
5. Closing the Connection: Either the client or the server can initiate closing the WebSocket connection. The protocol supports a graceful closing process, where both parties agree to end the connection.

## Asynchronous communication using RabbitMQ:

**Introduction of RabbitMQ:**

* Rabbit MQ is the message broker that acts as a middleware while using multiple microservices.
* RabbitMQ is an open-source message broker software. It is sometimes also called message-oriented middleware.
* RabbitMQ is written in the Erlang programming language.
* RabbitMQ is used to reduce the load and delivery time of a web application when some of the resources have taken a lot of time to process the data.



As you can see in the diagram above, there is one producer who sends a message to the RabbitMQ server. The server will store that message inside the queue in a FIFO manner.

Once the producer has sent the message to the queue, there may be multiple consumers that want the message produced by the producer. In that case, consumers subscribe to the message and get that message from the Message Queue as you see in the above diagram.

In this section, we will use one eCommerce Site as an example to understand more fully.

There are multiple microservices running in the background while we are using the eCommerce website. There is one service that takes care of order details, and another service that takes care of payment details and receipts.

Suppose we placed one order. At that time, the order service will start and process our order. After taking the order details, it will send data to the payment service, which takes the payment and sends the payment receipt to the end-users.

In this case, there may be a chance of some technical issue occurring in the payment service. If the user did not receive the payment receipt due to this, the user will be impacted and connected with the support team to try to learn the status of the order.

There may be another scenario on the user(consumer) side. Perhaps due to some technical issue, the user is exited from the application when the payment is in process. But

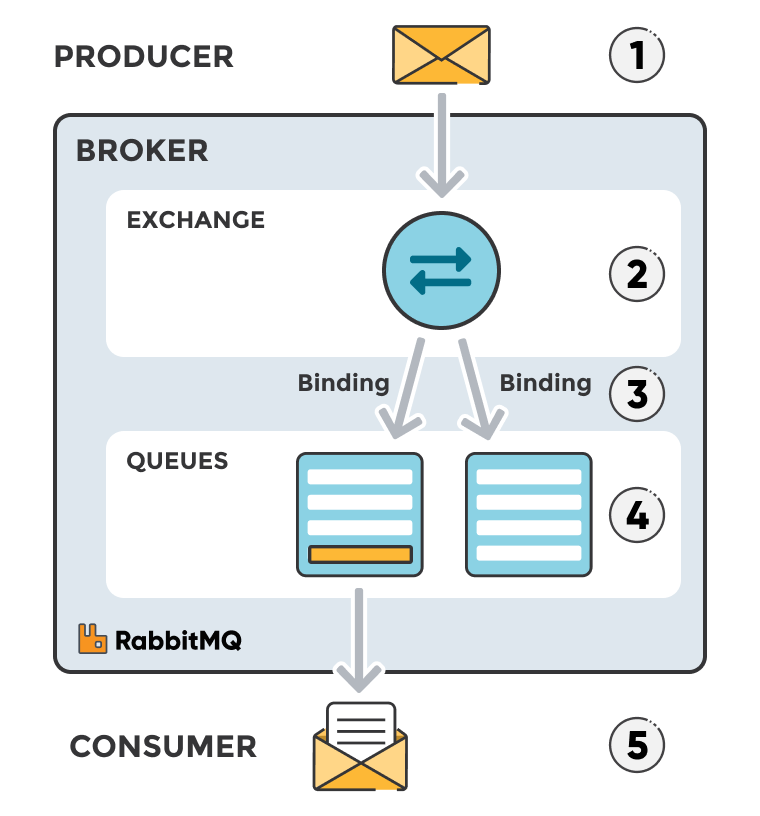
he will not get any receipt details after payment is successfully processed from backend services.

In these scenarios, the RabbitMQ plays an essential role to process messages in the message queue. So, when the consumer gets online, he will receive that order receipt message from the message queue, produced by the producer without impacting the web application.

All these examples are just for understanding purpose. There are a lot of scenarios in which RabbitMQ may play an important role while using multiple microservices. Sometimes RabbitMQ is used fully to load balancing between multiple services, or for many other purposes.

**EXCHANGES**

Messages are not published directly to a queue; instead, the producer sends messages to an exchange. An exchange is responsible for routing the messages to different queues with the help of bindings and routing keys. A binding is a link between a queue and an exchange.



##### **Message flow in RabbitMQ**

1. The producer publishes a message to an exchange. When creating an exchange, the type must be specified.
2. The exchange receives the message and is now responsible for routing the message. The exchange takes different message attributes into account, such as the routing key, depending on the exchange type.
3. Bindings must be created from the exchange to queues. In this case, there are two bindings to two different queues from the exchange. The exchange routes the message into the queues depending on message attributes.
4. The messages stay in the queue until they are handled by a consumer
5. The consumer handles the message.

#### **RABBITMQ AND SERVER CONCEPTS**

Some important concepts need to be described before we dig deeper into RabbitMQ. The default virtual host, the default user, and the default permissions are used in the examples, so let’s go over the elements and concepts:

* **Producer:** Application that sends the messages.
* **Consumer:** Application that receives the messages.
* **Queue:** Buffer that stores messages.
* **Message:** Information that is sent from the producer to a consumer through RabbitMQ.
* **Connection:** A TCP connection between your application and the RabbitMQ broker.
* **Channel:** A virtual connection inside a connection. When publishing or consuming messages from a queue - it's all done over a channel.
* **Exchange:** Receives messages from producers and pushes them to queues depending on rules defined by the exchange type. To receive messages, a queue needs to be bound to at least one exchange.
* **Binding:** A binding is a link between a queue and an exchange.
* **Routing key:** A key that the exchange looks at to decide how to route the message to queues. Think of the routing key like an *address for the message.*
* **AMQP:** Advanced Message Queuing Protocol is the protocol used by RabbitMQ for messaging.
* **Users:** It is possible to connect to RabbitMQ with a given username and password. Every user can be assigned permissions such as rights to read, write and configure privileges within the instance. Users can also be assigned permissions for specific virtual hosts.
* **Vhost, virtual host:** Provides a way to segregate applications using the same RabbitMQ instance. Different users can have different permissions to different vhost and queues and exchanges can be created, so they only exist in one vhost.

**Benefits of using RabbitMQ:**

1. **High Availability**

When multiple microservices are used by the application, if one of the microservices is stopped due to technical reasons at that time, the message will never be lost. Instead, it persists in the RabbitMQ server. After some time, when our service starts working, it will connect with RabbitMQ and take the pending message easily.

1. **Scalability**

When we use RabbitMQ, at that time our application does not depend on only one server and virtual machine to process a request. If our server is stopped at that time, RabbitMQ will transfer our application load to another server that has the same services running in the background.

**RabbitMQ Installation:**

1. Download and install [Erlang](https://www.erlang.org/downloads).
2. [Download RabbitMQ installer.](https://www.rabbitmq.com/install-windows.html#installer)
3. On cmd:

* Cd C:\Program Files\RabbitMQ Server\rabbitmq\_server-3.12.0\sbin
* rabbitmq-plugins enable rabbitmq\_management
* restart RabbitMQ service

**RabbitMQ Implementation:**

1. Install RabbitMQ.Client nuget package.
2. Create IRabitMQProducer.cs and RabitMQProducer.cs classes for the message queue inside the RabbitMQ folder:

namespace RabitMqProductAPI.RabitMQ {

public interface IRabitMQProducer {

public void SendProductMessage < T > (T message);

}

}

1. Create a RabbitMQProducer.cs class:

public class RabbitMQProducer: IRabitMQProducer {

public async Task SendProductMessageAsync<T>(T message)

{

var factory = new ConnectionFactory

{

HostName = "localhost"

};

var connection = await factory.CreateConnectionAsync();

using

var channel = await connection.CreateChannelAsync();

await channel.QueueDeclareAsync("product", exclusive: false);

var json = JsonConvert.SerializeObject(message);

var body = Encoding.UTF8.GetBytes(json);

await channel.BasicPublishAsync(exchange: "", routingKey: "product", body: body);

} }

d

* Specify the Rabbit MQ Server
* Create the RabbitMQ connection using connection factory.
* Create channel with session and model
* Declare the queue after mentioning name and a few properties related to that.
* Serialize the message
* Put the data on to the product queue

1. Create Controller:

[Route("api/[controller]")]

[ApiController]

public class ProductController: ControllerBase {

private readonly IProductService productService;

private readonly IRabitMQProducer \_rabitMQProducer;

public ProductController(IProductService \_productService,

IRabitMQProducer rabitMQProducer) {

productService = \_productService;

\_rabitMQProducer = rabitMQProducer;

}

[HttpGet("productlist")]

public IEnumerable < Product > ProductList() {

var productList = productService.GetProductList();

return productList;

}

[HttpGet("getproductbyid")]

public Product GetProductById(int Id) {

return productService.GetProductById(Id);

}

[HttpPost("addproduct")]

public Product AddProduct(Product product) {

var productData = productService.AddProduct(product);

//send the inserted product data to the queue and consumer will listening this data from queue

await \_rabitMQProducer.SendProductMessage(productData);

return productData;

}

[HttpPut("updateproduct")]

public Product UpdateProduct(Product product) {

return productService.UpdateProduct(product);

}

[HttpDelete("deleteproduct")]

public bool DeleteProduct(int Id) {

return productService.DeleteProduct(Id);

}

}

1. Register a few services inside the Program.cs class:

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.

builder.Services.AddScoped < IProductService, ProductService > ();

builder.Services.AddDbContext < DbContextClass > ();

builder.Services.AddScoped < IRabitMQProducer, RabitMQProducer > ();

builder.Services.AddControllers();

builder.Services.AddEndpointsApiExplorer();

builder.Services.AddSwaggerGen();

var app = builder.Build();

// Configure the HTTP request pipeline.

if (app.Environment.IsDevelopment()) {

app.UseSwagger();

app.UseSwaggerUI();

}

app.UseHttpsRedirection();

app.UseAuthorization();

app.MapControllers();

app.Run();

1. Create New Console project as a receiver:

using RabbitMQ.Client;

using RabbitMQ.Client.Events;

using System.Threading.Channels;

var factory = new ConnectionFactory

{

HostName = "localhost"

};

var connection = await factory.CreateConnectionAsync();

using

var channel = await connection.CreateChannelAsync();

await channel.QueueDeclareAsync("product", exclusive: false);

var consumer = new AsyncEventingBasicConsumer(channel);

//var consumer = new EventingBasicConsumer(channel);

consumer.ReceivedAsync +=async (model, ea) =>

{

var body = ea.Body.ToArray();

var message = System.Text.Encoding.UTF8.GetString(body);

Console.WriteLine($" [x] Received {message}");

};

await channel.BasicConsumeAsync(queue: "product",

autoAck: true,

consumer: consumer);

Console.WriteLine(" Press [enter] to exit.");

Console.ReadLine();

1. Rabbit management tool: http://localhost:15672/

User: guest

Password:guest