## 



Microservices

**─────────**

Neeraj Saini

[Ditstek Innovations](https://www.ditstek.com/) Pvt. Ltd.

India

# 

# Overview

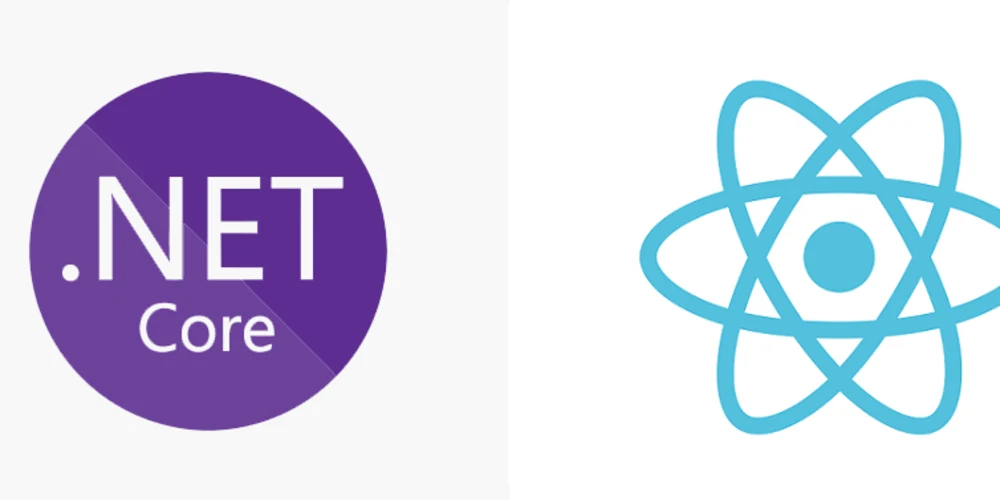
Our application is building with latest technologies including :

* Microservices architecture as backend using **ASP NET CORE** with code first approach to get benefits of multiple services as per modules to enhance performance.
* **React JS** architecture to get benefits of Excellent cross-platform support, Handles dependencies, Extremely competent.

# Goals

**Microservices benefits**

* Different technologies can be used to develop individual processes and services best suited to deliver the business functionality.
* Easier to develop new systems rather than updating existing ones.
* Using microservices, a [particular service can be redesigned and deployed](https://www.thesunflowerlab.com/cloud-deployment/) within a short time without affecting the scope of other processes or services.
* Easier to build and maintain appsOrganized around business capabilities.
* Organized around business capabilities : Microservices facilitate the creation of products rather than projects.
* Each microservice can be developed using a different technology. This simplifies the process of selecting the most appropriate tech stack to fulfill specific needs of each business component.
* Developers get more independence and work autonomously.



## **React JS benefits**

* A ReactJS web application is made up of multiple components, and each component has its own logic and controls.
* These components are responsible for outputting a small, reusable piece of HTML code which can be reused wherever you need them.
* The reusable code helps to make your apps easier to develop and maintain.
* ReactJS is a perfect combination of JavaScript and HTML tags. The usage of the HTML tags and JS codes make it easy to deal with a vast set of data containing the document object model. During this time, ReactJS works as a mediator which represents the DOM and assists to decide which component needs changes to get the exact results.
* ReactJS is extremely intuitive to work with and provides interactivity to the layout of any UI. Plus, it enables fast and quality assured application development that in turn saves time for both - clients and developers.

# 

# Frontend

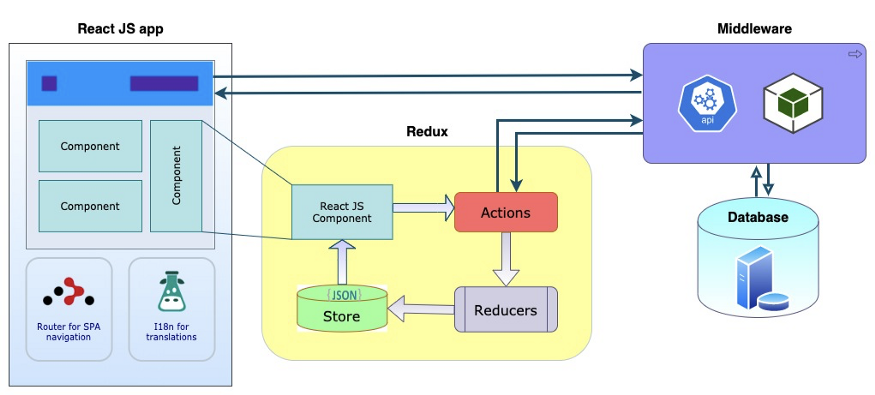
# Technical Specifications

* Frontend : React JS
* React JS Version : 17.0.2
* React-router-dom : 6.2.1
* Semantic -ui-react : 2.1.1
* Styled-components : 4.5.5
* Redux version : 4.1
* Axois : 0.26

**ReactJS**

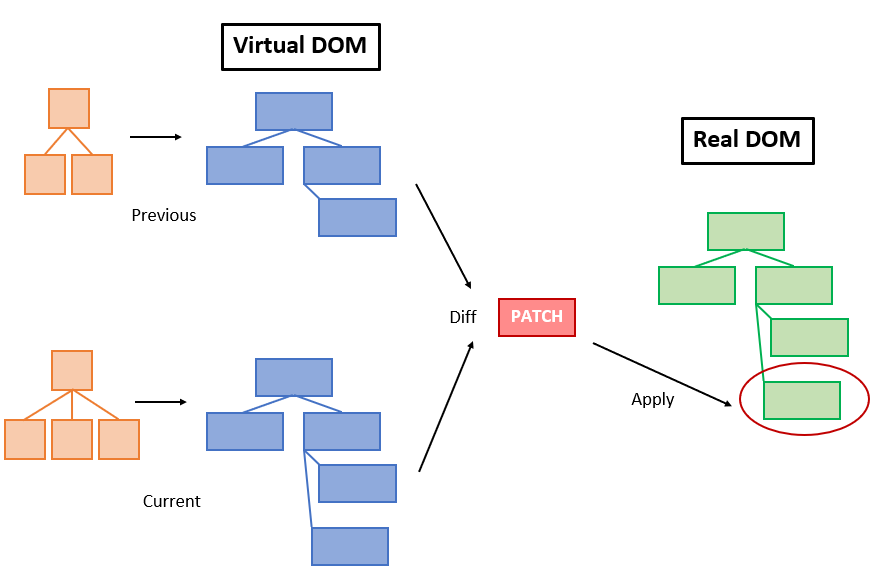
ReactJS is a JavaScript library used for building reusable UI components.

React uses Virtual DOM which is like a lightweight copy of the actual DOM(a virtual representation of the DOM). So for every object that exists in the original DOM, there is an object for that in React Virtual DOM. It is exactly the same, but it does not have the power to directly change the layout of the document. Manipulating DOM is slow, but manipulating Virtual DOM is fast as nothing gets drawn on the screen. So each time there is a change in the state of our application, virtual DOM gets updated first instead of the real DOM.



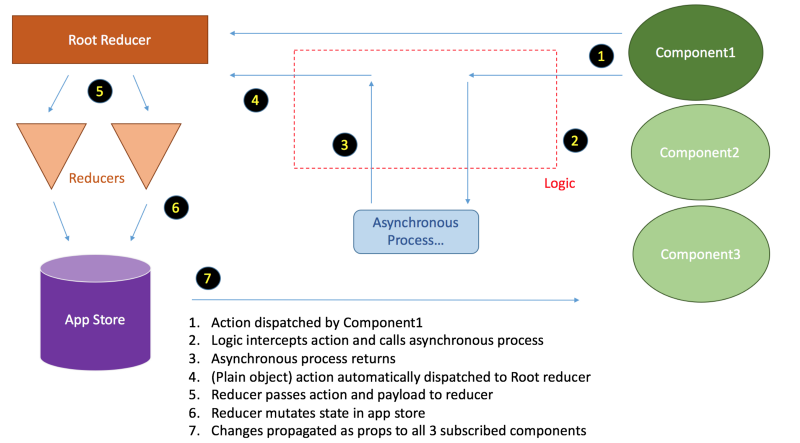
**React Router Dom**

* React Router is a standard library for routing in React.
* It enables the navigation among views of various components in a React Application, allows changing the browser URL, and keeps the UI in sync with the URL.This makes it effortless to handle navigation in SPA.



**Redux**

* Redux is an open-source JavaScript library for managing and centralizing application state.

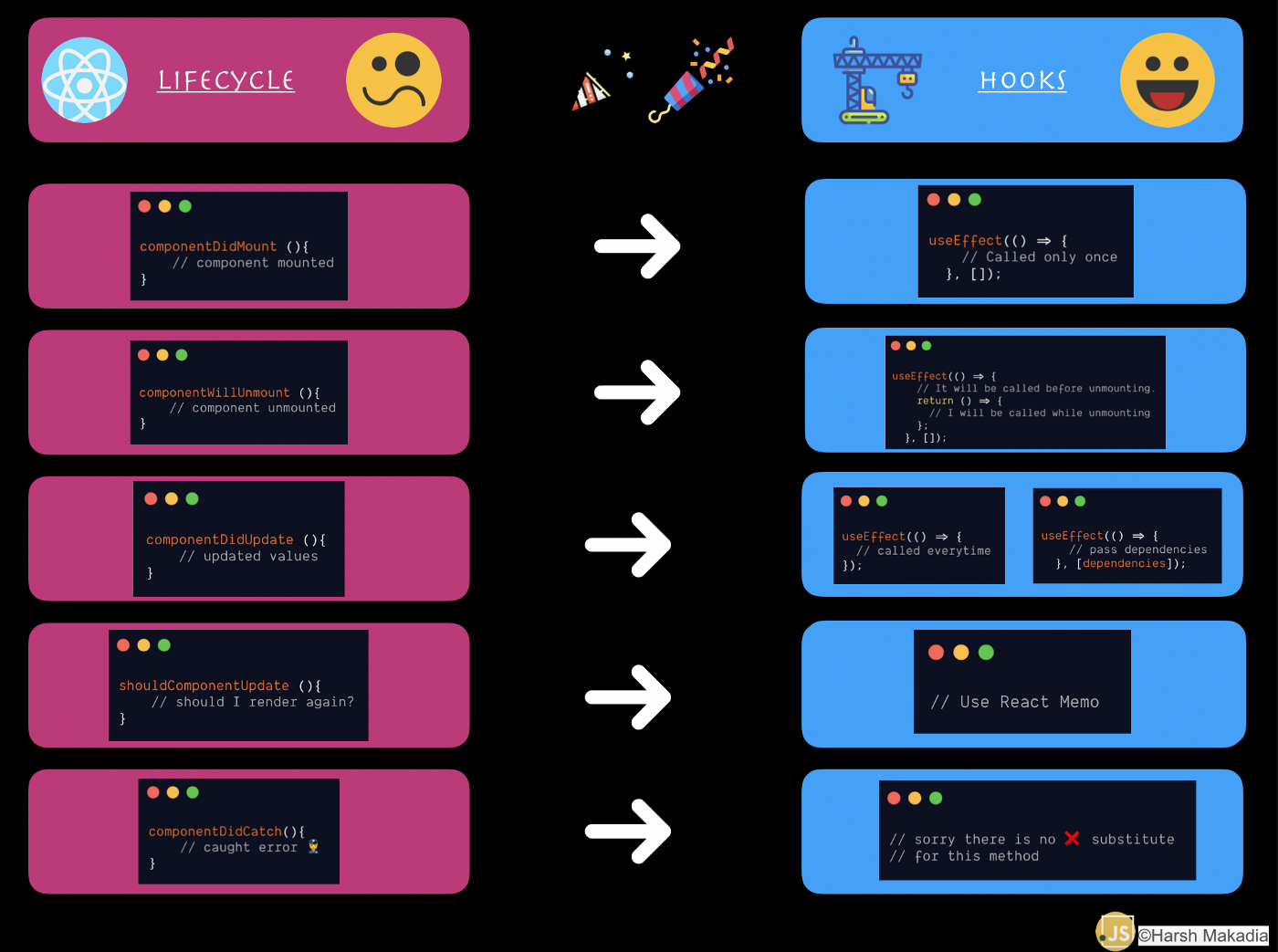


**Axios**

* Axios is a simple data fetching module for REST API connectivity using all the required methods like GET, POST, PUT, DELETE, etc.

**Hooks**

* We are going to use hooks for creating components in the application. Hooks are the new feature introduced in the React 16.8 version.
* It allows you to use state and other React features without writing a class. Hooks are the functions which "hook into" React state and lifecycle features from function components.
* It does not work inside classes.



## 

## **Managing multiple environment**

* We have created an environments folder for managing multiple environments in the application. Suppose, you have a different API URL for each environment.
* We can add the configuration in the particular .env file.
* Configuration here means any environment variable that can be used globally within the application.

### **Development Environment**

##### Command to run application in development mode

**npm run start-dev**

##### Command to build application in development mode

**npm build start-dev**

#### **Production Environment**

##### Command to run application in production mode

**npm run pro-dev**

##### Command to build application in production mode

**npm build pro-dev**

#### **QA Environment**

##### Command to run application in QA mode

**npm run qa-dev**

##### Command to build application in QA mode

**npm build qa-dev**

## **Folder structure (Frontend)**

The source folder in which contains the folders. All the folder structures below.

1. **Assets**. It is a parent directory of
   * **Fonts directory** :- Store all fonts in the application
   * **Images** :- Store all static images used the application
   * **Videos** :- Store all videos used in the application
2. **Components**.

All the shared components in the application will come in this directory. Shared components should be divided in the form of atomic design pattern.all components should have their own directory/folder.

* **Context**:- A context provide a way to pass data or state through the components tree
  + - Without passing props manually.
* **Layout**:- All Layouts that are used in the application will come in this directory like Admin layout, Authlaoyout.
* **Modal**:- All modals that are used in the application will come in this directory.

3. **Config**: This directory is used to store all the configurable things that are used in the application like routes.config, env.config,api.config etc.

* **Api Configuration**:-All API’s that are used in the application store in this file.
* **Message Configuration** - It is used to store all success and failure messages that are used in the application.
* **RoutePath Configuration** - It is used to store all routes used in the application.

5. **Pages**:- All pages that are used in the application will come in this directory. Every new page always comes in a new directory/folder.

6. **Routes** This folder contains the route path.

7. Utils Colors Label and phone number input styling added

## **Store directory**

All the redux related things are done here.

**Types directory :-** All actions that we need to perform to update the store.

Below are the examples of API type action

**Actions directory** : - All the actions that need to be performed will be defined in this directory.

**Reduce directory :-**  All the reducers of the will be defined in this directory.

# Backend

# Technical Specifications

* Backend : ASP NET Core
* ASP NET Core Version : 6.0
* Entity Framework : 6.4.4
* DataBase : SQL Server 2019
* SQL Version : 15.0.2000
* Dapper framework

**Third party tools**

* Swashbuckle.AspNetCore 5.0
* IdentityServer4 4.0.4
* Jaeger
* RabbitMQ
* Logstash
* Kibana
* Open Tracing API

**Solution architecture components**

Our solution architecture has three main components:

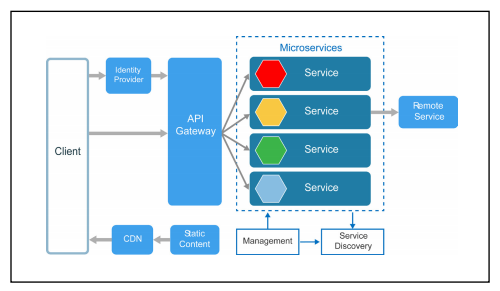
* [SPA client application](https://github.com/mmacneil/AngularASPNETCoreOAuth/tree/master/src/Spa/oauth-client) - React JS
* [Authorization/Identity server](https://github.com/mmacneil/AngularASPNETCoreOAuth/tree/master/src/AuthServer) - ASP.NET Core 5.0.13 and IdentityServer4
* [Resource server](https://github.com/mmacneil/AngularASPNETCoreOAuth/tree/master/src/Resource.Api) - ASP.NET Core Web API

**Microservices API**

We will create Microservices on .net platforms that use Dotnet Core WebAPI, Docker,

RabbitMQ, Ocelot API Gateway, Redis, SqlServer, Dapper Framework, Clean Architecture

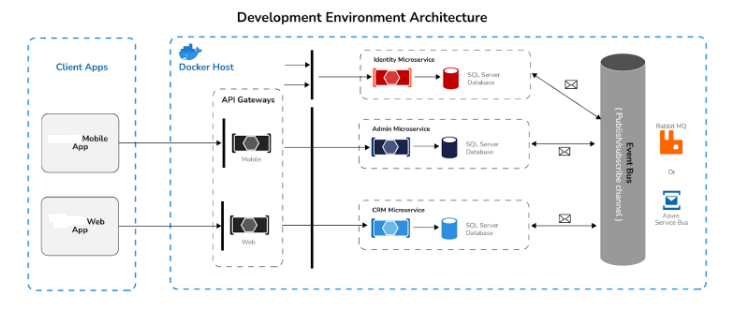
Implementation.

****

Here are the steps in brief details

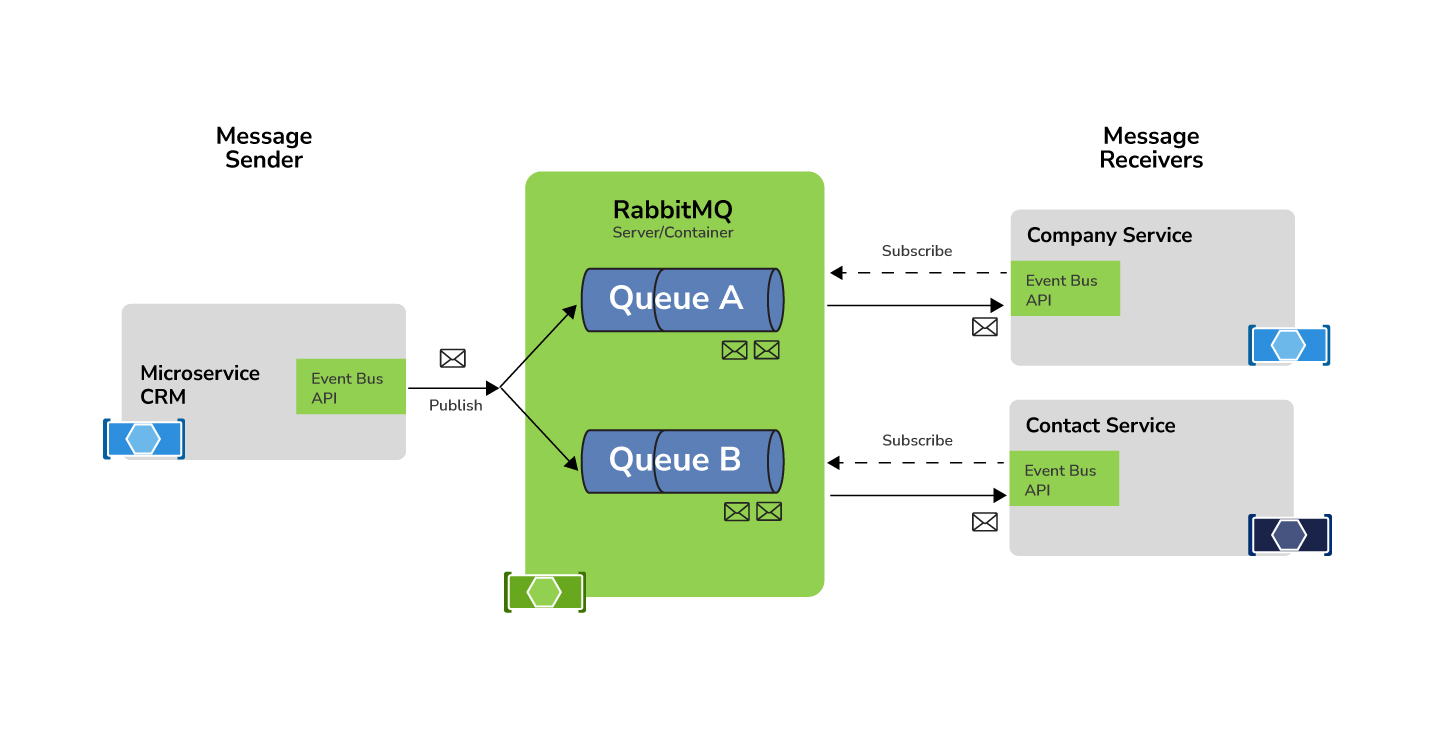
1. API controllers will receive requests and create a domain model from request data
2. In the next step, the controller will call Repository service that will be particular for that controller and forward domain models to Repository service
3. Repository service will then call Generic Repository if required to call generic function else code is written in repository class.
4. Generic Repository service will communicate with the database and return database records back to the repository service and later repository service will return those records to the API controller.

**Inter-Service Communication Of CRM Workflow**

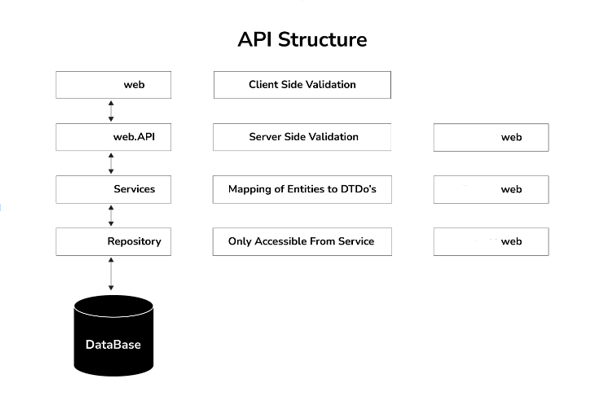
****

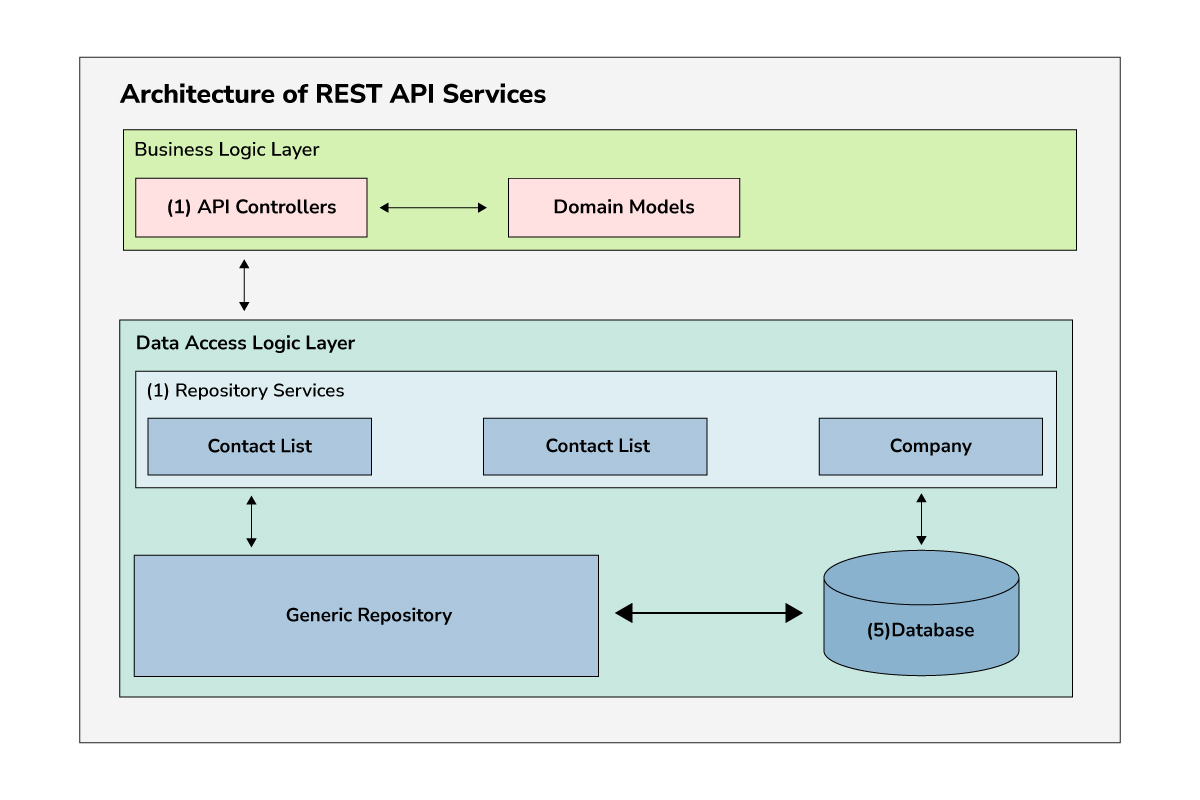
**Consider our consists of following modules :**

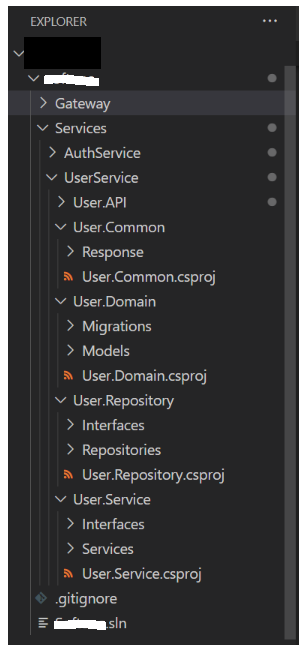
* **Contacts**
* **Companies**

****

**Microservice Folder Structure**

* **Solution Architecture**
* **API solution**
  + - An ASP.NET WebAPI project that supports all API calls necessary for the react app. This layer provides validation before passing on requests to the service layer and returning the response. Implement Swagger for standardization and documentation.
* **Domain** 
  + - **Models Folder**
      * Contain all classes auto generated from tables in the database.
      * Entities that contain models of all DB objects.
      * **DBContext Folder**
        + Contain DbContext class. DbContext is the primary class that is responsible for interacting with the database.
* **Service** 
  + - A library project that contains all application logic and acts as a midlayer between Repository and Web.API.
* **Repository :**
  + A project that uses all data access. We will implement the Unit of Work Pattern (Design Pattern) so we are not opening multiple DB connections.
* **Common**
  + **Auto Mapper Folder**
    - Define Mapping b/w two entities.
  + **Shared Folder**
    - It is used for storing any shared items like Helper methods, Static methods etc.
      * **Enums Folder**
        + Any enumerators used on the site should be contained here.



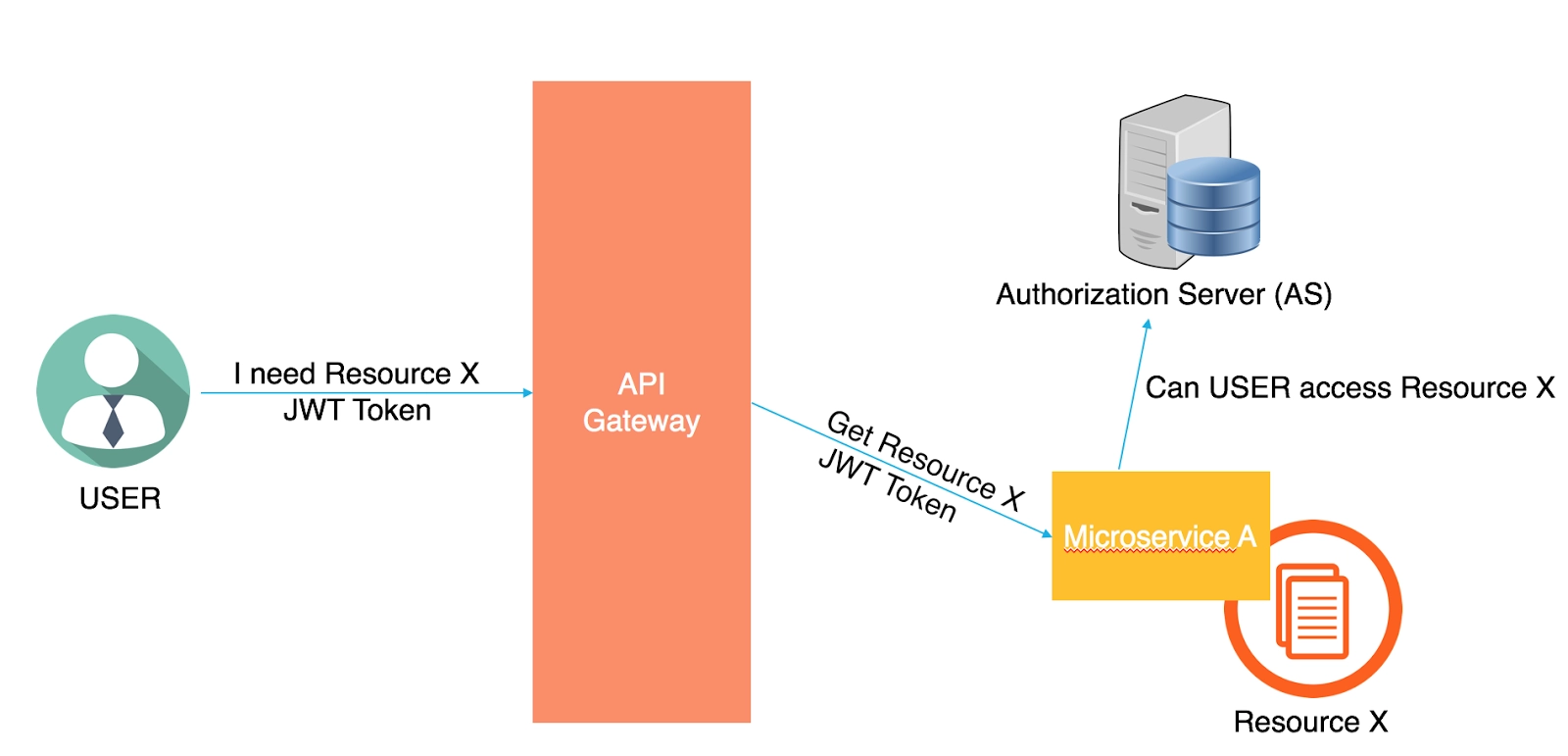
****

**Critical parts of microservices**

* **Api Gateway**
  + Different service to define policies
  + Connector between different services
    - Timeout policies
    - Retry policies
    - Request limit policies

**Authentication and Authorization by API Gateway**

* From the security point of view, API Gateways usually handle the authentication and authorization from the external callers to the microservice level.
* A user starts by sending his credentials to the API gateway which will forward the credentials to the Authorization Server (AS) or the OAuth Server. The AS will generate a JSON Web Token (JWT) and will return it back to the user.

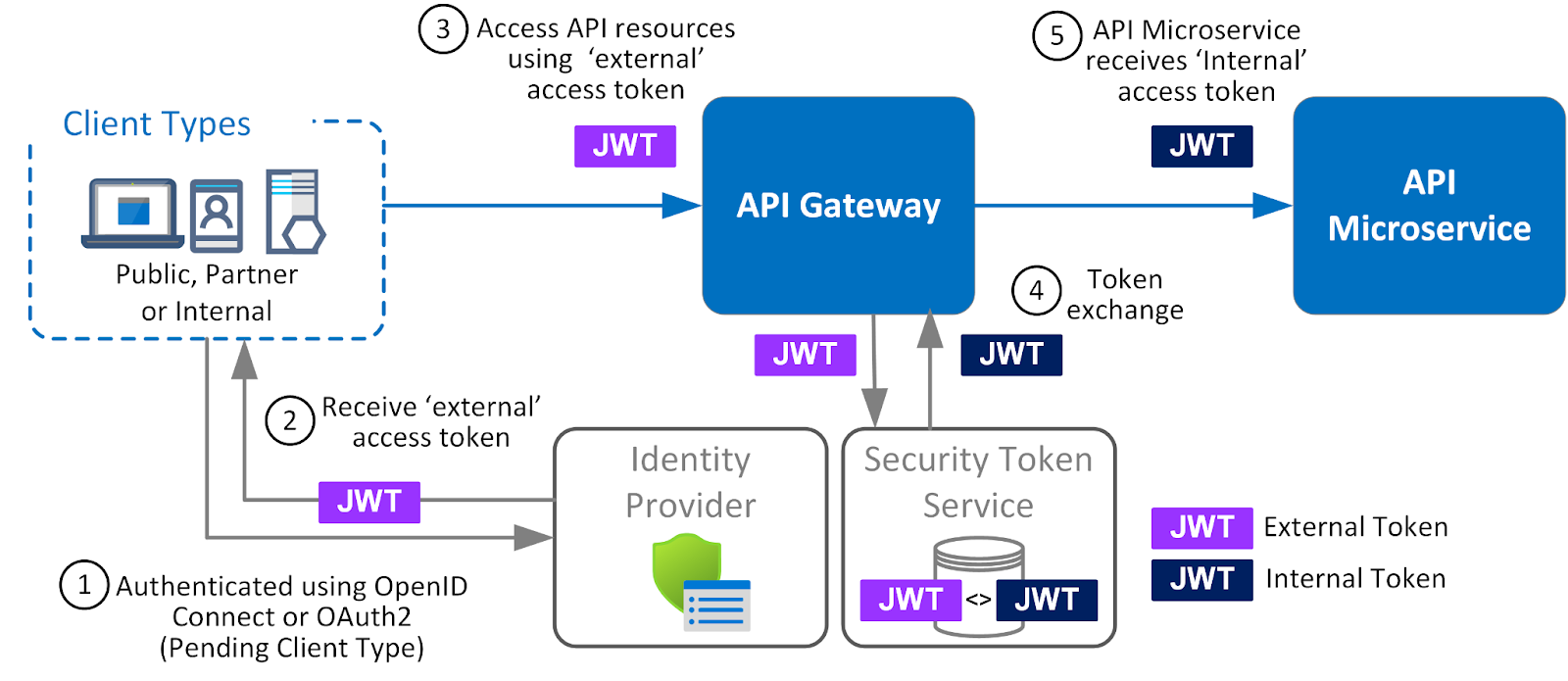


* Whenever the user wants to access a certain resource, he’ll request it from the API Gateway and will send the JWT along with his request.
* The API Gateway will forward the request with the JWT to the microservice that owns this resource. The microservice will then decide to either grant the user the resource (if the user has the required permissions) or not.
* Based on the implementation, the microservice can make this decision by itself (if it knows the permissions of this user over this resource) or simply forward the request to one of the Authorization Servers within the environment to determine the user’s permissions.

**Identity Server :**

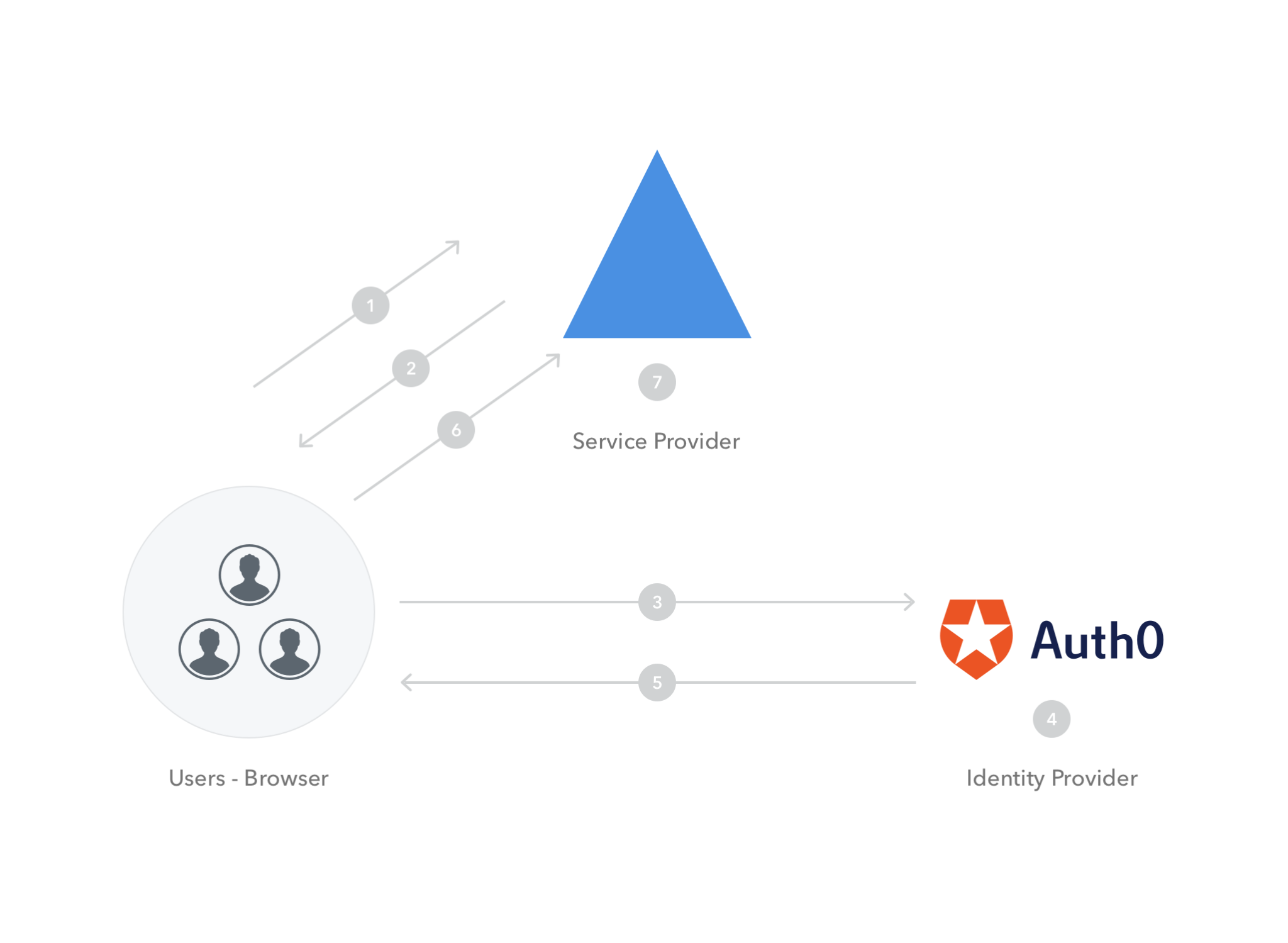
The user (resource owner) initiates an authentication request with the authorization server.

* If the credentials are valid and everything checks out the authorization server obtains end-user consent and grants the client application an access token.
* The access token is attached to subsequent requests made to the protected resource server.
* The authorization server validates the access token; if successful the request for protected resources is granted, and a response sent back to the client application.

****

**Identity Server with SAML(2.0)**

* The **Security Assertion Markup Language** (SAML) is a protocol used to communicate authentication data between two parties.
* This external service holds the credentials and we send them SAML requests.



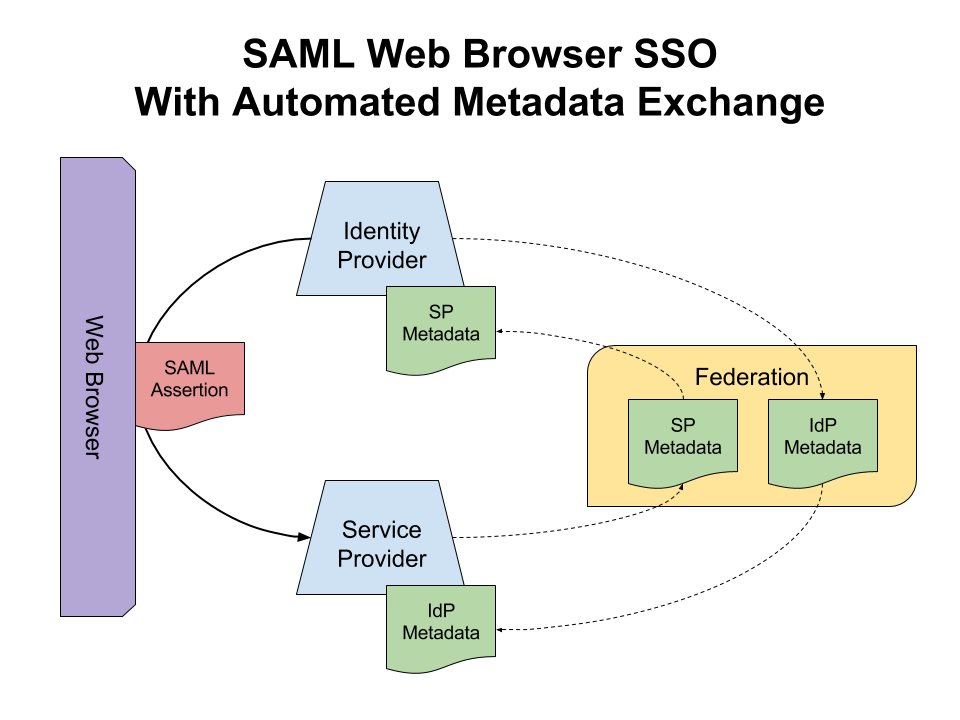
**Working**

* The client tries to log in to our application from a browser.
* Our application responds by generating a SAML request.
* The browser redirects the client to an SSO URL, Auth0
* Auth0 parses the SAML request and authenticates the client. Once the client is authenticated, Auth0 generates a SAML response.
* Auth0 returns the encoded SAML response to the browser.
* The browser sends the SAML response to our application for verification.
* If the verification is successful, the client will be logged in to our application and granted access to the authorized resources.

If any client is using **SAML** authentication then it does not need different credentials to access the different modules of the application. It will provide **single-sign-in**.

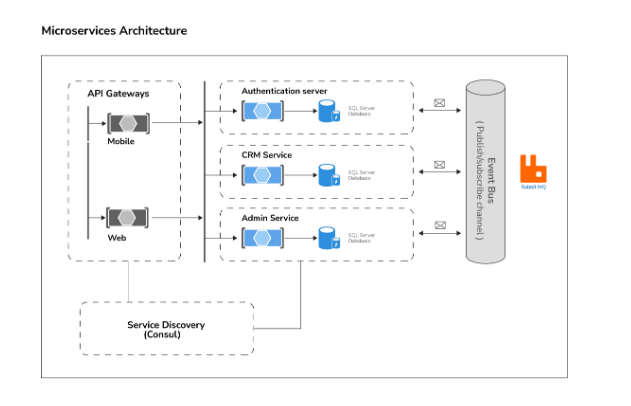
**Handling of metadata into SAML**

* It is an XML document which contains information necessary for interaction with SAML-enabled identity or service providers. It contains URLs of endpoints, information about supported bindings, identifiers and public keys.



**Api Gateway**

* An API Gateway is a server that is the single entry point into the system.
* The API Gateway encapsulates the internal system architecture and provides an API that is tailored to each client.
* It might have other responsibilities such as authentication, monitoring, load balancing, caching, request shaping and management, and static response handling.
* All requests from clients first go through the API Gateway. It then routes requests to the appropriate microservice. The API Gateway will often handle a request by invoking multiple microservices and aggregating the results.
* The API Gateway provides each kind of client with a specific API. This reduces the number of round trips between the client and application
* Architecture
* Interaction with auth service in order to validate JWT token
* Setting-up routes and aggregate routes
* Deployment of auth service with (docker Azure App Service) OR (Docker+Kubernete)
* Implementation of CI/CD pipelines

****

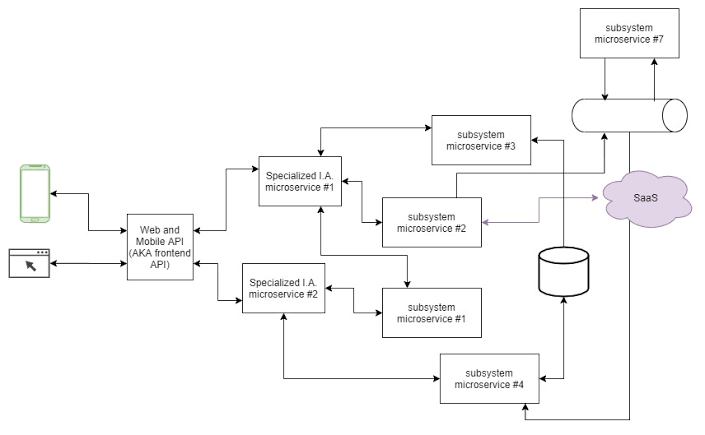
**Interservice Communication**

**Service Mesh**

* **Service Mesh** provides support for service discovery and load balancing.
* Service-to-service communication S[ervice mesh](https://www.nginx.com/blog/what-is-a-service-mesh/) is a configurable infrastructure layer with built-in capabilities to handle service-to-service communication, resiliency, and many cross-cutting concerns.
* A service mesh manages traffic, communication, and networking concerns at the application level. It understands messages and requests.
* A service mesh typically integrates with a container orchestrator. Kubernetes supports an extensible architecture in which a service mesh can be added.

### **Multi-tenancy**

* + The multi-tenancy deployment pattern isolates groups of microservices from each other when a tenant exclusively uses those groups. A typical use case for multi-tenancy is to isolate the services between two different departments within an organization or to isolate entire organizations altogether.

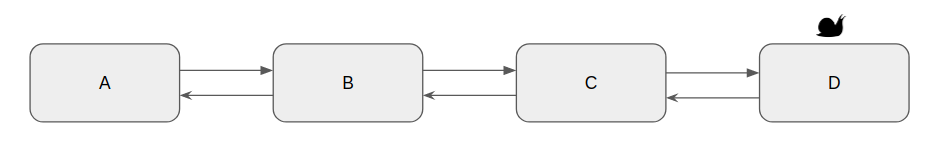


**Timeout Pattern**

In Microservice architecture, when there are multiple services (A, B, C & D), one service (A) might depend on the other service (B) which in turn might depend on C and so on.

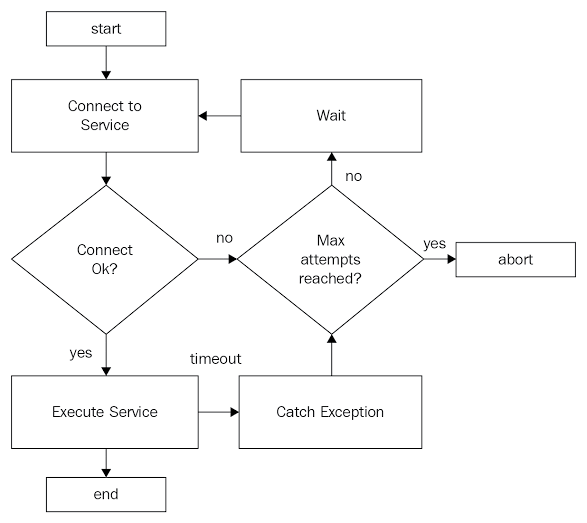
Sometimes due to some network issue, Service D might not respond as expected.

This slowness could affect the downstream services – all the way up to Service A & block the threads in the individual services.

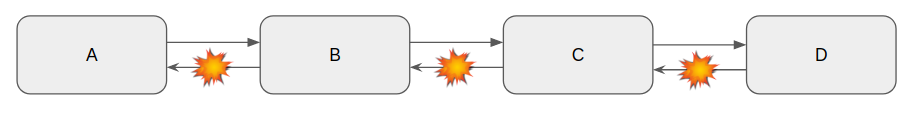


* It is better to take this service slowness/unavailability issues into consideration while designing your Microservices by setting a timeout for any network call.
* So that we could have the core services working as expected & responsive even when the dependent services are NOT available.

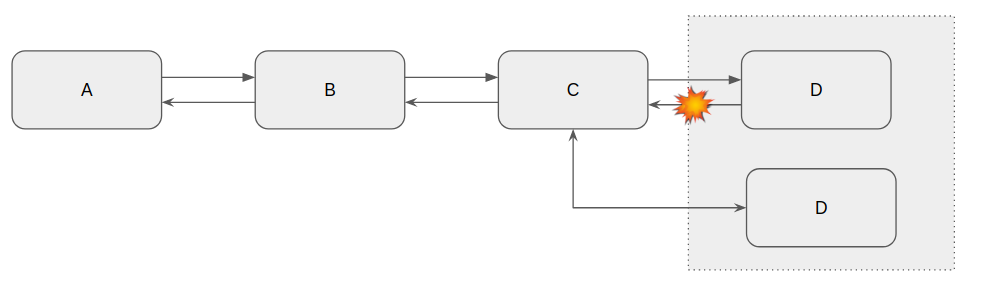
**Retry Pattern**

****

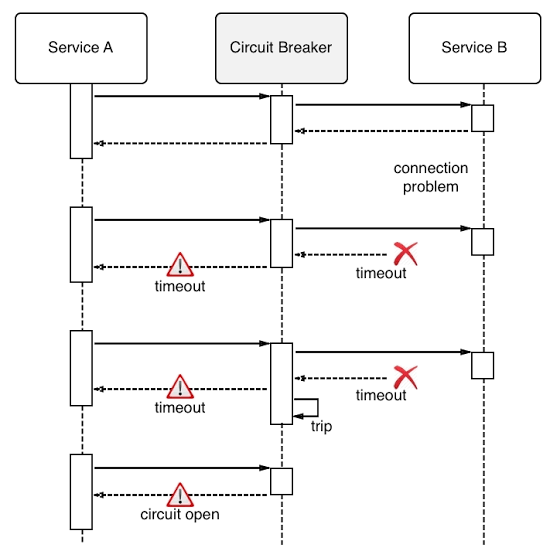
The idea behind the retry pattern is quite simple. If the caller service receives an unexpected response for a request, the caller service sends the request to the callee service again.



* Do only the limited numbers of repeats (count++ on each retry)
* Make a timeout after the retry
* Make an exponential timeout after the retry

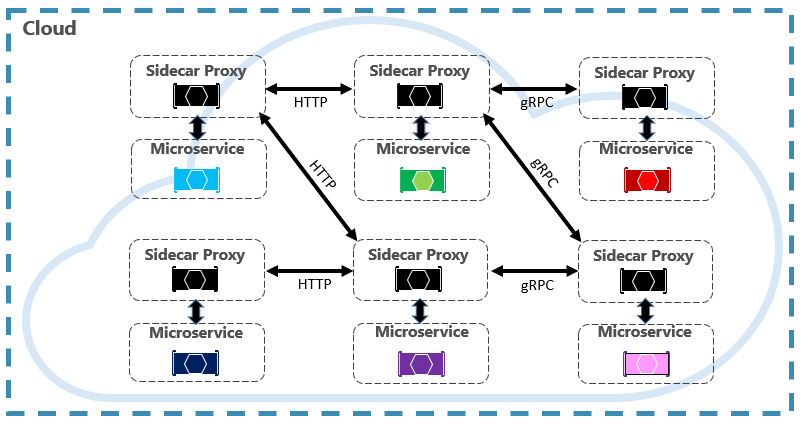


* In the Microservices world, we might be running multiple instances of the same Service D for high availability and load balancing.
* If one of the instances could be having the issue and it does not respond properly to our request, If we retry the request, the load balancer could send the request to a healthy node and get the response properly.
* So with the Retry option, we have more chances for getting the proper response.



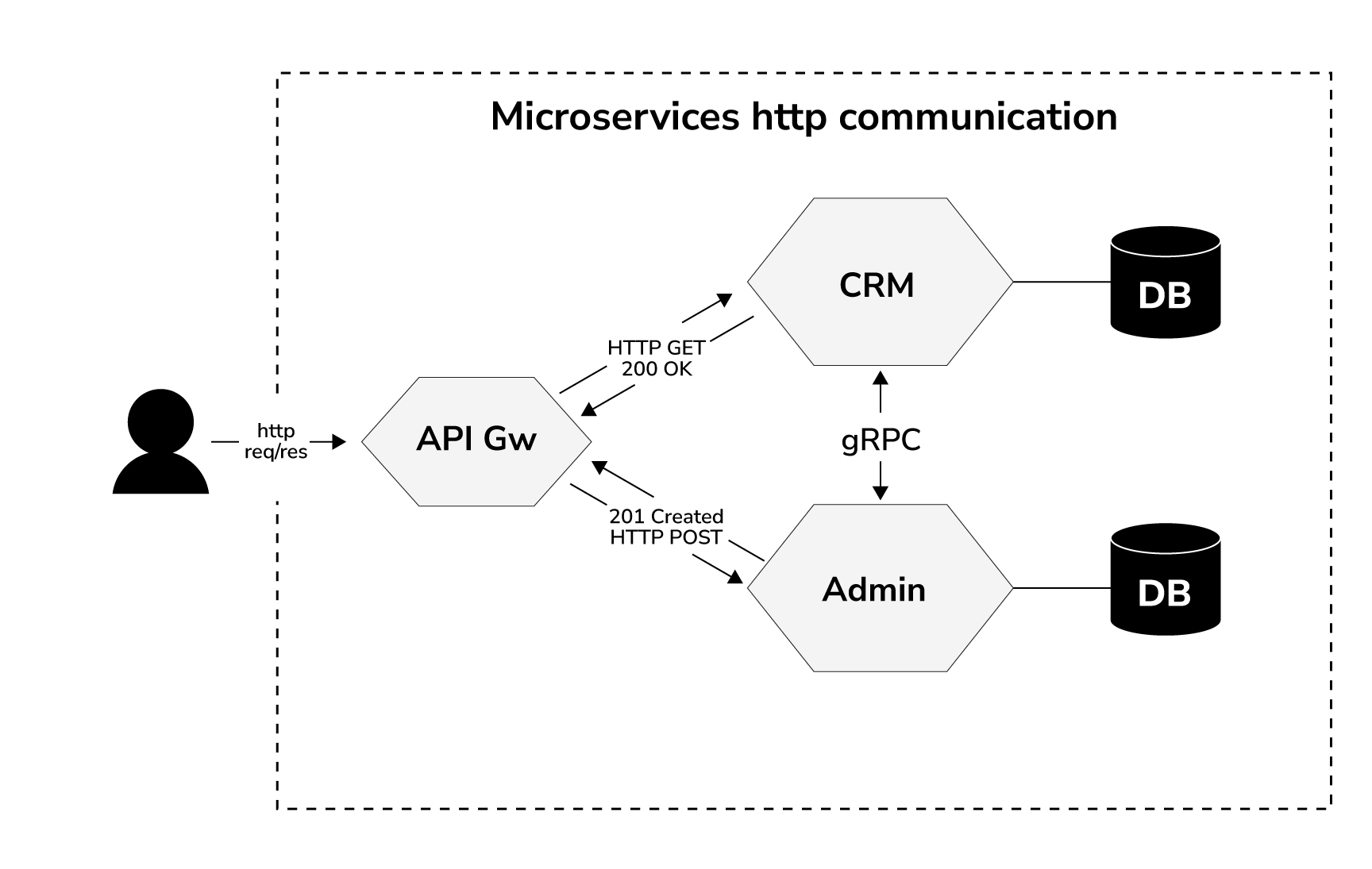
**Rate Limit Pattern**

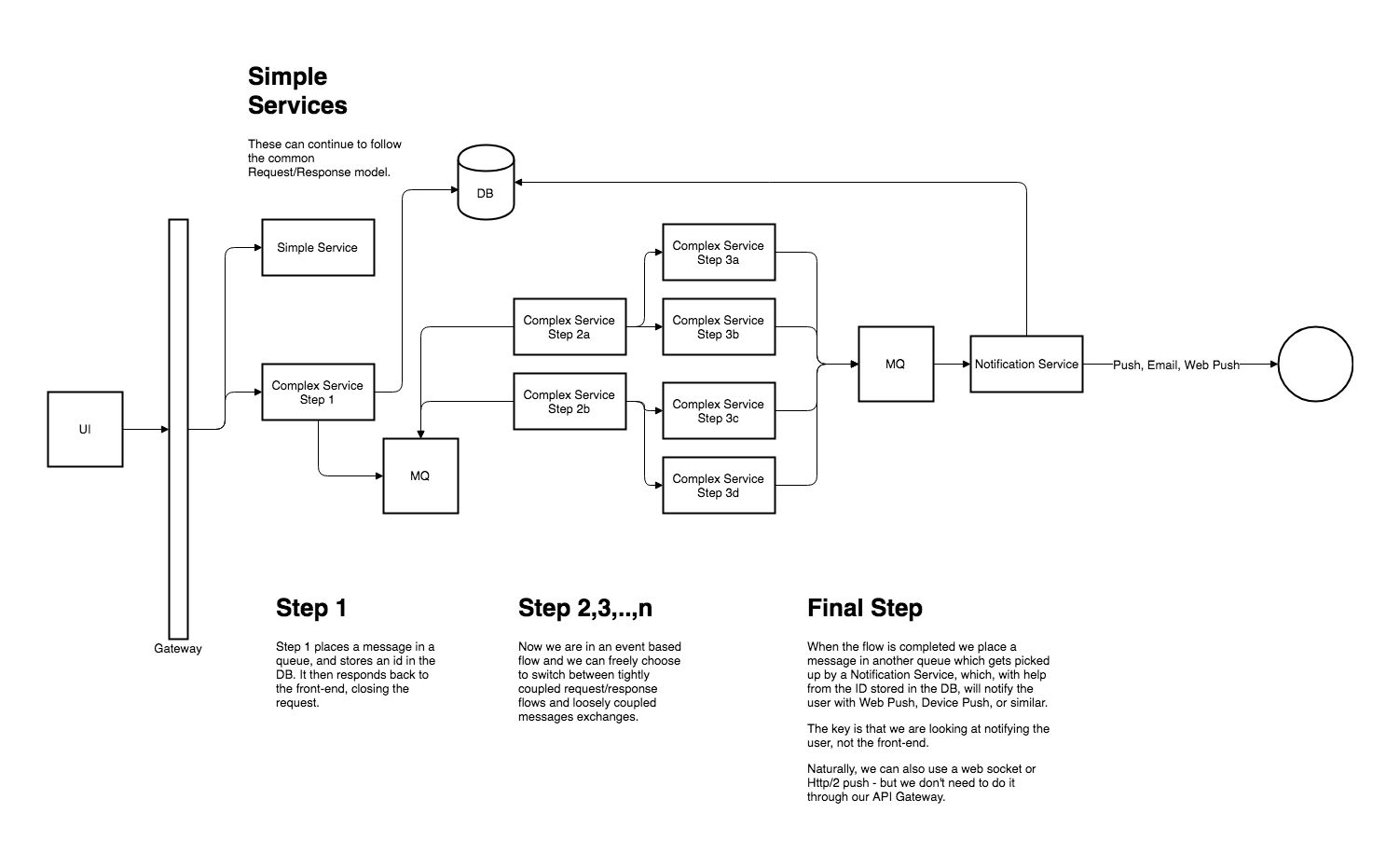
* Rate limiting is generally put in place as a defensive measure for services. Shared services need to protect themselves from excessive use—whether intended or unintended—to maintain service availability.
* Even highly scalable systems should have limits on consumption at some level. For the system to perform well, clients must also be designed with rate limiting in mind to reduce the chances of [cascading failure](https://landing.google.com/sre/sre-book/chapters/addressing-cascading-failures/).



**Request - Response flow in microservices**

* Services communicate using HTTP or RPC
* **“Service Orchestration”** since there is one service to manage the flow and instruct other services to perform actions.





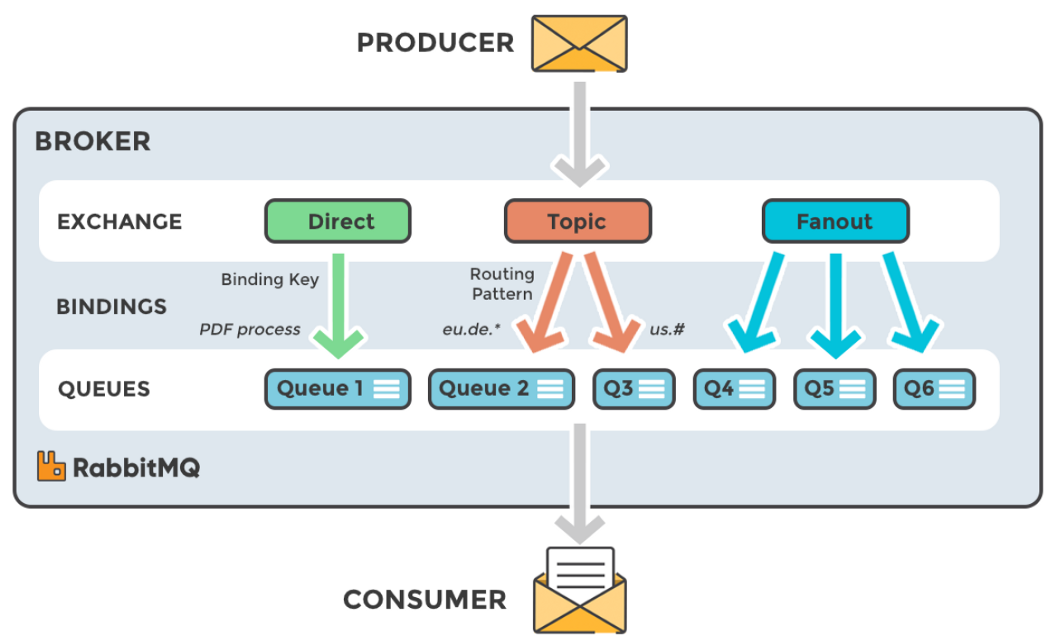
* For complicated, multi-step flows, the first step should respond immediately after placing the request into a message queue, and a notification identifier in a database.
* The initial request completes and both the front-end application and the users are freed.

**RabbitMQ**

* **All transactions can be listed in a queue until the source to be transmitted gets up**
* RabbitMQ instance with docker
* Business logic for events
* Commands
  + Events
  + Query
  + Setup Sega pattern

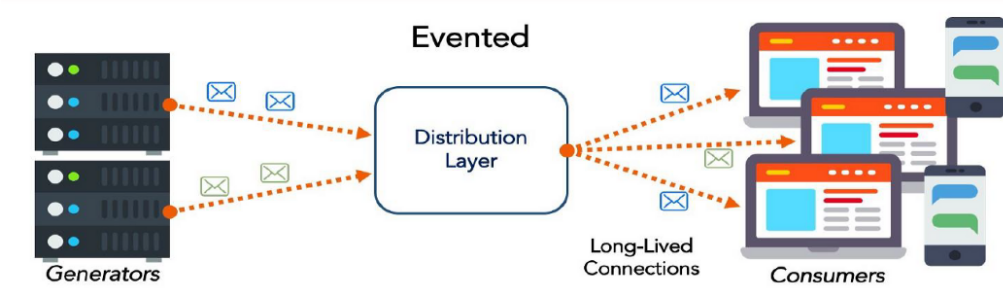
**Main Logic of RabbitMQ**

* **Producer**: The source of the message is the application.
* **Queue**: Where messages are stored. The sent messages are put in a queue before they are received. All incoming messages are stored in Queue, that is memory.
* **Consumer**: It is the server that meets the sent message. It is the application that will receive and process the message on the queue.
* **Message**: The data we send on the queue.
* **Exchange**: It is the structure that decides which queues to send the messages. It makes the decision according to routing keys.
* **Binding**: The link between exchange and queue.
* **FIFO**: The order of processing of outgoing messages in RabbitMQ is first in first out.

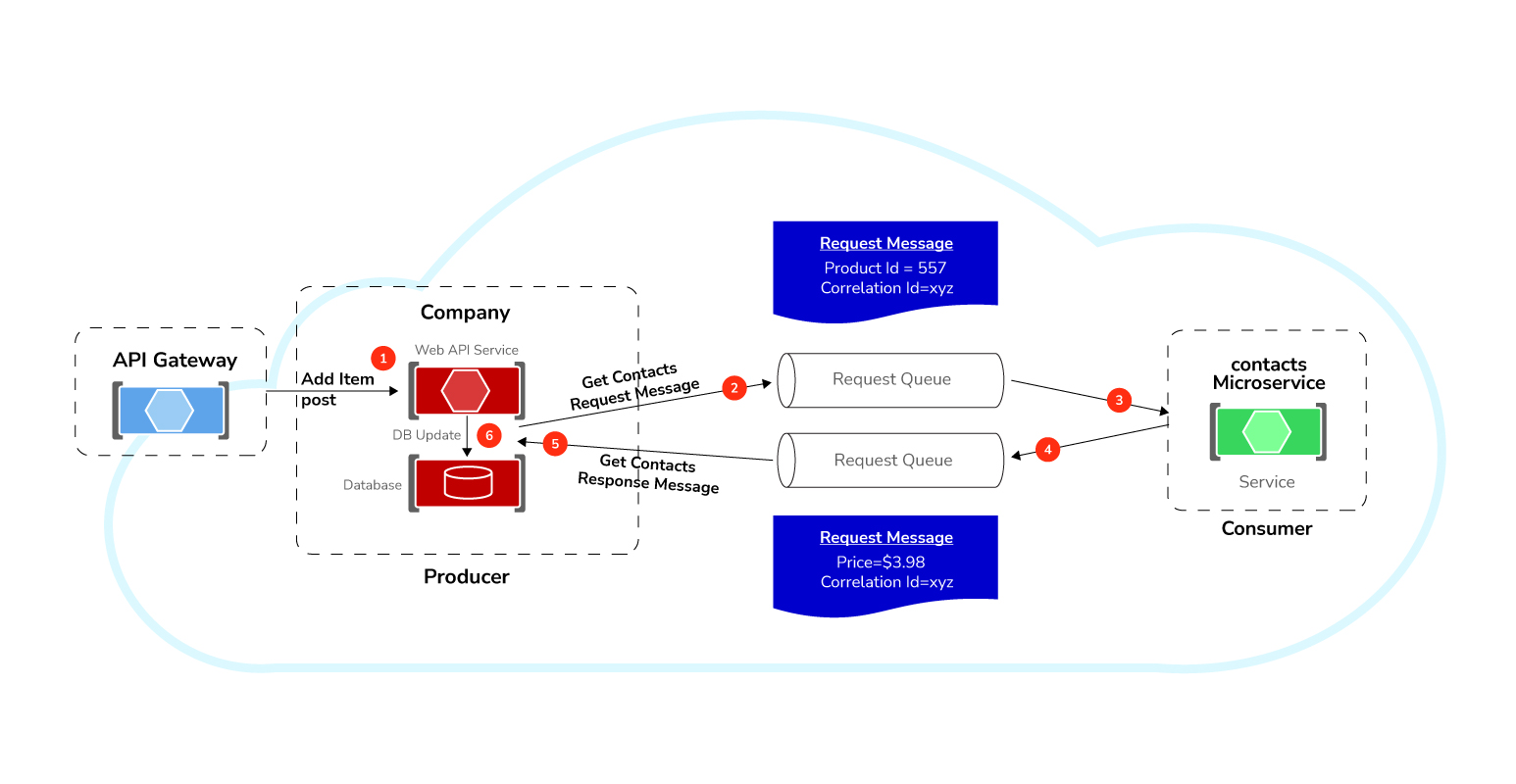


**Event-Driven Architecture**

These communication type microservices don't call each other, instead they create events and consume events from message broker systems in an async way. In this section we will follow this type of communication for



* Company microservice
* Publish Get Contacts Queue with using MassTransit and RabbitMQ
* Contacts Microservice
* Consuming RabbitMQ Company event queue with using MassTransit-RabbitMQ Configuration



**RabbitMQ on Api Gateway, Admin and CRM**

* While communicating with multiple services, suppose one of them is not able to succeed the operation inside the transaction.

