

Development of Browse Bay in Microservices Architecture

Man Fortun

Web Developer | Arcanys

Contents

Browse Bay Microservice Architecture	3
The Client Application	3
Basic Functionalities.....	3
Role-based Functionalities.....	4
Definition of Roles.....	4
API Gateway	4
The Catalog Service.....	5
Functionalities	5
The Authorization Service	5
Functionalities.....	5
Message Bus	6
Development and Configuration.....	6
Building the applications	6
Setting up the development environment	7
Creating the Docker images	9
Deploying a service	10
Creating a Persistent Volume Claim	11
Adding a K8S secret.....	12
Setting up the SQL Server	12
Ingress-NGINX	13
Setting up the client configuration.....	13
Inter-service communication	14
HTTP.....	15
Message Queues	15
Final.....	18
References	19

Browse Bay Microservice Architecture

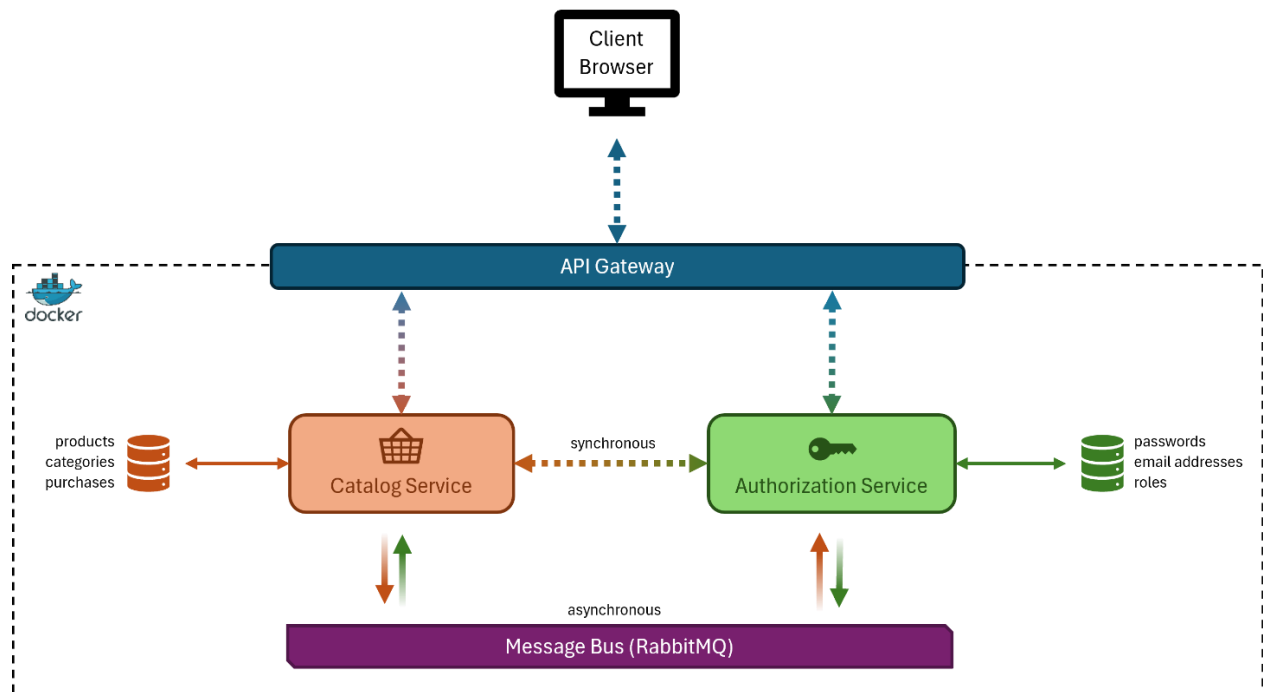


Figure 1. Browse Bay Microservice Architecture

The Client Application

The client application (CA) is the interface for the users where they interact with the system. For this exercise, we are using the ASP.NET Core MVC Framework.

Let us define the functionalities of our client application:

Basic Functionalities

- Users can create their accounts using an email address and a password
- A user can register as any of the roles available. See [Definition of Roles](#).
- Users can log in using a registered account.
- Users can view the products on the home page. Users must be authenticated to see the home page. This means that those who are not logged in cannot view the home page or proceed to any of the bullets mentioned from hereon.
- Users can add a product to their basket in 2 ways:
 - Clicking on the add icon (+) on the bottom of the product card; and
 - Clicking on the product card, then clicking on the add icon on the bottom of the product card.
- Users can view their basket by clicking the basket icon (🛒) from the navigation bar.
- Users can update their basket by clicking on the edit button and then clicking save.
- Users can check out their baskets. This will clear their current baskets.

- Users can change their password by clicking the gear icon (⚙️). First, they must input their current password. Then, they must fill in their desired password on the “New password” and “Confirm new password” inputs. Lastly, if the inputs are correct, they can click the save button.
- Users can log out of the application.

Role-based Functionalities

The bullets defined in this section pertain to the system functionalities that are accessible based on the role of the currently logged-in user. This means that not all users can access these functionalities. To see which role has functionalities, see [Definition of Roles](#).

- Add product – add a product and define the following properties: name, price, description, image, and categories. A product can have multiple categories or none. After a successful add, the product can then be viewed from the home page.
- Edit product – change the following properties of a product: name, price, description, image, and categories.
- Delete product – remove a product from the home page.
- Add category – add a category with a unique name.
- Edit category – edit a category name. The new name must be unique.
- Delete category – remove the category from the category list.

Definition of Roles

The system consists of three roles, each having their respective accessibilities.

1. Admin – has access to all basic and role-based functionalities. For the simplicity of this application, anyone can register as an admin.
2. Seller – has access to all basic functionalities and “Add product”, “Edit product”, and “Delete product” from role-based functionalities.
3. Buyer – has access to all basic functionalities and no access to any role-based functionalities.

API Gateway

The API gateway is the single entry point for client requests to access various microservices within a system. This layer is responsible for routing incoming requests from clients to the appropriate microservices based on predefined rules, endpoints, or paths. API gateways also provide monitoring and analytics capabilities to track request metrics, performance, and usage patterns. This helps in identifying bottlenecks, optimizing performance, and making data-driven decisions.

The Catalog Service

One of our two microservices is the catalog service (CS). This service is responsible for providing the product and basket information, as well as handling edit/delete requests to products and baskets.

Functionalities

- Can handle product CRUD requests. If the request is successful, an HTTP 200 is returned. Otherwise, an HTTP 400 is returned.
- Can handle category CRUD requests. If the request is successful, an HTTP 200 is returned. Otherwise, an HTTP 400 is returned.
- Can handle basket CRUD requests. To process this request, the user's login token must be provided on the HTTP request header. This token is then sent to authorization service, where the service replies with the ID of the owner. If the token cannot be authenticated, CS will return an HTTP 401 reply. This is how the service determines whose basket will be modified. If the request is successful, an HTTP 200 is returned. Otherwise, an HTTP 400 is returned.
- Other services can test their connectivity to this service. Once a GET request is received, an HTTP 200 is returned. There is no logic in this functionality.

The Authorization Service

The next and final microservice that we have is the authorization service (AS). This service is responsible for identity authentication and registration, as well as password changes. Here are its functionalities:

Functionalities

- The application accepts a request for account registration. If the registration is successful, HTTP 200 is returned. Otherwise, HTTP 400 is returned with the error description.
- Attempts to sign in a user using the username and password. If the sign-in is successful, a JWT token is generated and returned along with an HTTP 200. Otherwise, HTTP 400 is returned with the error description.
- Changes the password of a user in the database. The next time a login is requested, the new password must be provided. The old password is not stored in the database. If the request is successful, HTTP 200 is returned. Otherwise, HTTP 400 is returned.
- This service can also return the user ID from a JWT token. If the token is not valid, an HTTP 401 is returned.
- Other services can test their connectivity to this service. Once a GET request is received, an HTTP 200 is returned. There is no logic in this functionality.

Message Bus

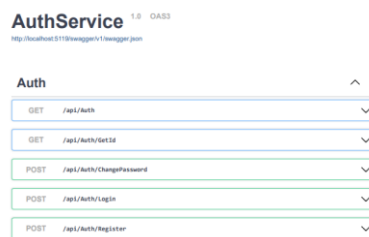
The message bus is a communication infrastructure that facilitates message-based communication between different components or services. It serves as a middleware layer that allows components to send and receive messages asynchronously, decoupling the sender and receiver of messages. Messages are sent and received asynchronously, meaning that the sender does not have to wait for an immediate response from the receiver.

Development and Configuration

Building the applications

First, let us begin by developing the microservices applications. In this document, we are focused on how we connected the layers within a distributed system. Hence, the logic for each functionality will not be discussed here.

We will begin by defining our controller actions.



AuthService 1.0 OAS3
<http://localhost:5173/swagger-ui/index.html>

Auth

GET	/api/Auth
GET	/api/Auth/GetId
POST	/api/Auth/ChangePassword
POST	/api/Auth/Login
POST	/api/Auth/Register

Figure 2. Authorization Service Controller Actions



CatalogService 1.0 OAS3
<http://localhost:5173/swagger-ui/index.html>

Catalog

GET	/api/Catalog
GET	/api/Catalog/GetProducts
GET	/api/Catalog/GetProductById
POST	/api/Catalog/CreateProduct
POST	/api/Catalog/AddCategory
GET	/api/Catalog/GetCategories
POST	/api/Catalog/GetCategories
GET	/api/Catalog/GetCategories
DELETE	/api/Catalog/DeleteCategory
POST	/api/Catalog/GetProduct
POST	/api/Catalog/UpdateProductCategory
DELETE	/api/Catalog/DeleteProduct
GET	/api/Catalog/GetProduct
GET	/api/Catalog/GetProduct
POST	/api/Catalog/GetProduct
POST	/api/Catalog/GetProduct
POST	/api/Catalog/GetProduct
DELETE	/api/Catalog/DeleteProduct

Figure 3. Catalog Service Controller Actions

Then, let us create the CA.

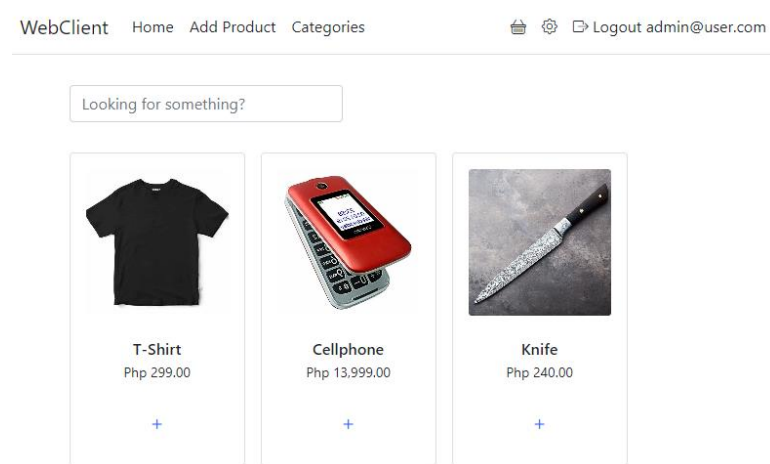


Figure 4. Client application home page

Setting up the development environment

To allow the CA to communicate to the services, we can, for the meantime, define their hosts in our application settings. Locate your `appSettings.Development.json`, and write the application URL of each service.



Figure 5. Setting up the connection strings in development

In this example, it would make more sense (to me, at least) to include the route (`api/controllerName`) so that during development, we would not have to specify the route repeatedly. This, however, is based on how the developer designs the CA.

During development, the CA directly communicates with the microservices. In this scenario, the CA and the microservices will have to be running on the same machine.

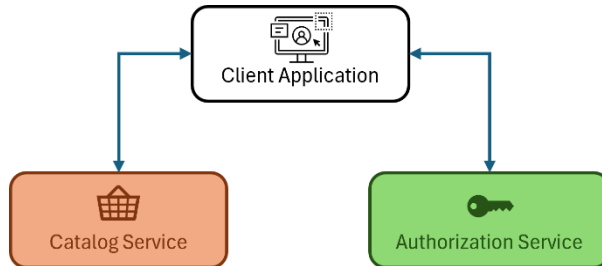


Figure 6. Client application directly communicates with the services during development.

It is important to note that we are doing this on the `appSettings.Development.json`. What this means is that the step above is solely for development purposes. In production, the CA does not and should not have any idea on the IP of the microservices. This should be managed by the API gateway.

Creating the Docker images

A Docker Image is an executable package of software that includes everything needed to run an application. This image informs how a container should instantiate, determining which software components will run and how. Docker Container is a virtual environment that bundles application code with all the dependencies required to run the application. The application runs quickly and reliably from one computing environment to another.^[1]

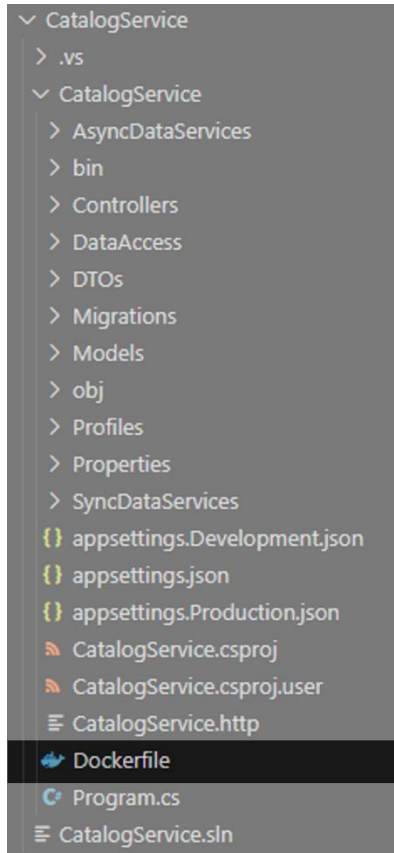


Figure 7. The Dockerfile in Catalog Service directory

To create a docker image, create a text file called “Dockerfile” in your product directory. Note that this file does not have an extension. This file contains instructions for building the Docker image.

We would then need to define the base image. In our application, our base image is `mcr.microsoft.com/dotnet/sdk:8.0`.

```
1 FROM mcr.microsoft.com/dotnet/sdk:8.0 AS build-env
```

Then, we must setup the environment as follows:

```
1 FROM mcr.microsoft.com/dotnet/sdk:8.0 AS build-env
2 WORKDIR /app
3
4 COPY *.csproj ./
5 RUN dotnet restore
6
7 COPY . ./
8 RUN dotnet publish -c Release -o out
9
10 FROM mcr.microsoft.com/dotnet/aspnet:8.0
11 WORKDIR /app
12 COPY --from=build-env /app/out .
13 ENTRYPOINT [ "dotnet", "CatalogService.dll" ]
```

Modify the entry point respectively based on the service DLL. Then build the image using the following command:

```
docker build -t your-image-name .
```

Note: Do not forget the ‘.’ on the end of the command.

You can also upload this image to your Docker Hub using the following command:

```
docker push your-username/your-image-name
```

We will not be directly running a container from the images that we built. Instead, we'll be deploying and managing them through a Kubernetes service.

Deploying a service

Kubernetes (K8S) is a portable, extensible, open source platform for managing containerized workloads and services that facilitates both declarative configuration and automation. It has a large, rapidly growing ecosystem. Kubernetes services, support, and tools are widely available.

The name Kubernetes originates from Greek, meaning helmsman or pilot. K8s as an abbreviation results from counting the eight letters between the "K" and the "s".^[2]

First, we need to create a YAML (YAML Ain't Markup Language) file to deploy a K8S service. Below is an image of a YAML file. Let's discuss through each section.

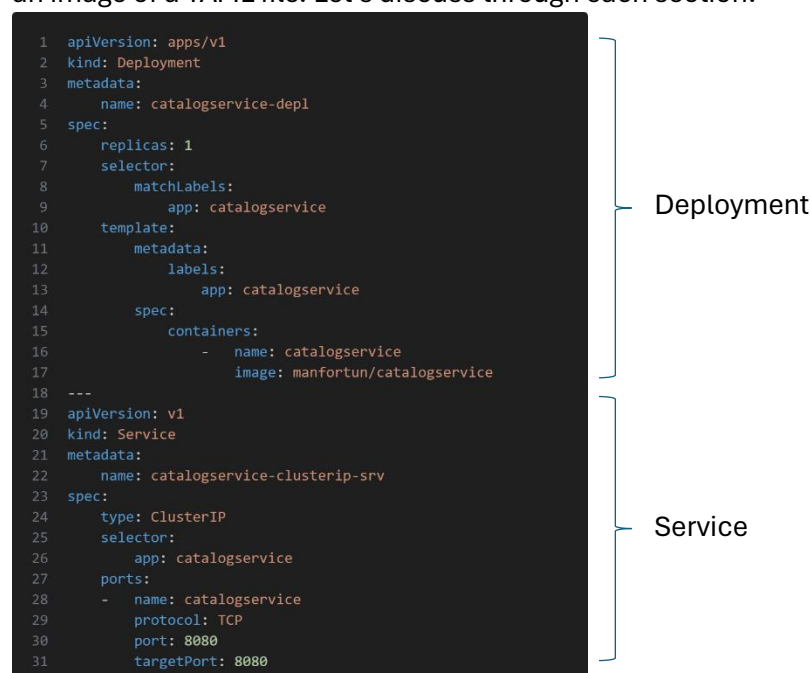


Figure 8. YAML file for Catalog Service

The YAML file above contains two sections: deployment and service for CS.

1. Deployment for Catalog Service
 - a. This section defines a deployment resource for the catalog service.
 - b. It specifies the number of replicas, selector labels, and container details.
 - c. The metadata section contains information such as the name of the deployment.
2. Service for Catalog Service
 - a. This section defines a service resource for the catalog service.
 - b. It specifies the type of service (ClusterIP), selector labels, and port configuration.
 - c. The metadata section contains information such as the name of the service.

Save this file and perform the following command:

```
kubectl apply -f name-of-yaml-file.yaml
```

If done correctly, containers should be running in Docker for the service.









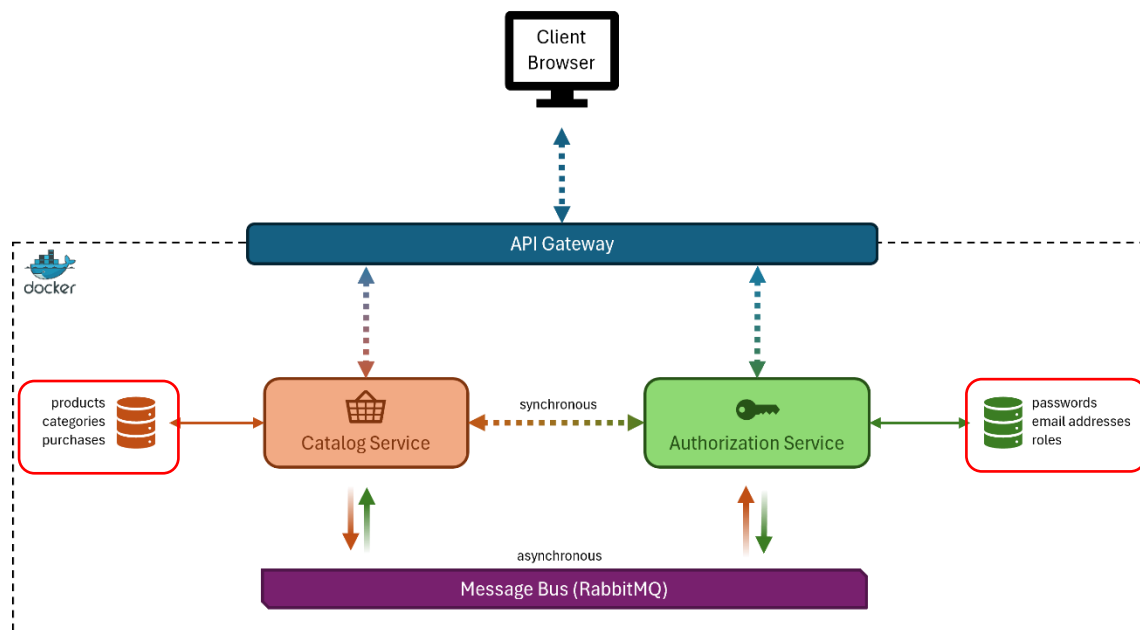
Name	Image	Status	CPU (%)	Port(s)	Last started	Actions
 k8s_POD_catalogservice-depl-5 d0995b23338f	registry.k8s.io/pause:3.9	Running	0%		2 hours ago	  
 k8s_catalogservice_catalogservi f2fc3d1f3315	manfortun/catalogservice	Running	0.04%		2 hours ago	  

Figure 9. Docker containers

Creating a Persistent Volume Claim



In the figure above, we can see that CS and AS each have their own database.

To request for storage resources from the underlying storage system in a K8S cluster, we need to create a Persistence Volume Claim (PVC).

When a containerized application needs access to persistent storage, such as a database or file system, a PVC can be created to request storage resources with specific characteristics, such as size, access mode, and storage class. Once created, K8S dynamically provisions the requested storage from the available storage resources in the cluster.

```
1 apiVersion: v1
2 kind: PersistentVolumeClaim
3 metadata:
4   name: mssql-claim
5 spec:
6   accessModes:
7     - ReadWriteMany
8   resources:
9     requests:
10       storage: 200Mi
```

On the image on the left, we can see how we defined a PersistentVolumeClaim named “mssql-claim”, with access mode read/write, and with 200 megabytes of storage.

CS and AS will be sharing this storage, although they will have a separate database. Save this file and run the K8S apply command.

Adding a K8S secret

When setting up a SQL Server in the K8S cluster, K8S will require a system administrator password. Perform the command below to create your secret key:

```
kubectl create secret generic mssql --from-literal=SA_PASSWORD="your-pass"
```

Setting up the SQL Server

To setup a server, we need to create a deployment and service for the PVC.

This file contains two sections:

```
1 apiVersion: apps/v1
2 kind: Deployment
3 metadata:
4   name: mssql-depl
5 spec:
6   replicas: 1
7   selector:
8     matchLabels:
9       app: mssql
10  template:
11    metadata:
12      labels:
13        app: mssql
14    spec:
15      containers:
16        - name: mssql
17          image: mcr.microsoft.com/mssql/server:2022-latest
18          ports:
19            - containerPort: 1433
20          env:
21            - name: MSSQL_PID
22              value: "Express"
23            - name: ACCEPT_EULA
24              value: "Y"
25            - name: SA_PASSWORD
26              valueFrom:
27                secretKeyRef:
28                  name: mssql
29                  key: SA_PASSWORD
30          volumeMounts:
31            - mountPath: /var/opt/mssql/data
32              name: mssqldb
33          volumes:
34            - name: mssqldb
35              persistentVolumeClaim:
36                claimName: mssql-claim
37 ---
38 apiVersion: v1
39 kind: Service
40 metadata:
41   name: mssql-clusterip-srv
42 spec:
43   type: ClusterIP
44   selector:
45     app: mssql
46   ports:
47     - name: mssql
48       protocol: TCP
49       port: 1433
50       targetPort: 1433
51 ---
```

Deployment – the deployment defines a single replica of a container running MSSQL Server. It exposes port 1433 within the container, and sets environment variables such as MSSQL_PID and ACCEPT_EULA. It also mounts persistent storage using PVC named “mssql-claim”.

Service (mssql-clusterip-srv) – exposes the SQL Server deployment within the K8S cluster using a ClusterIP type, which provides internal connectivity.

Save this file and apply using the K8S apply command.

If done correctly, we should be able to see the containers running for these services.









Name	Image	Status	CPU (%)	Port(s)	Last started	Actions
 k8s_POD_mssql-depl-7c4944d5 cd542d6b4c09	registry.k8s.io/pause:3.9	Running	0%		2 hours ago	  
 k8s_mssql_mssql-depl-7c4944d dc1e79c58c6f	sha256:ffdd6981a89eaaa114321a1fb33	Running	1.21%		2 hours ago	  

Figure 10. Docker containers for MSSQL

In our `appsettings.Production.json`, we can set the connection string using the name of our MSSQL Server ClusterIP name. In the example above, we have `mssql-clusterip-srv`. Hence, to create a connection, we set the following:

```
1 "ConnectionStrings": {  
2   "DefaultConnection": "Server=mssql-clusterip-srv,1433;Initial Catalog=authservice;TrustServerCertificate=True;User ID=sa;Password=pa55w0rd!;"  
3 }
```

The password is the system administrator key that we set up in [this section](#).

Ingress-NGINX

Ingress-NGINX is an Ingress controller designed for K8S, utilizing NGINX (pronounced engine x) as a reverse proxy and load balancer. This controller is responsible for handling external traffic and routing it to the correct services.

```
1 apiVersion: networking.k8s.io/v1  
2 kind: Ingress  
3 metadata:  
4   name: ingress-srv  
5   annotations:  
6     kubernetes.io/ingress.class: nginx  
7     nginx.ingress.kubernetes.io/use-regex: 'true'  
8 spec:  
9   rules:  
10    - host: acme.com  
11      http:  
12        paths:  
13          - path: /api/Auth  
14            pathType: Prefix  
15            backend:  
16              service:  
17                name: authservice-clusterip-srv  
18                port:  
19                  number: 8080  
20          - path: /api/Catalog  
21            pathType: Prefix  
22            backend:  
23              service:  
24                name: catalogservice-clusterip-srv  
25                port:  
26                  number: 8080
```

This Ingress configuration defines routing rules for incoming HTTP traffic based on the host name and path prefixes, directing requests to specific backend services within the K8S cluster.

In this file, it means that accessing `acme.com/api/Auth` would direct us to the authorization service, and accessing `acme.com/api/Catalog` would direct us to the catalog service.

How does it know the IP for each service? Using the ClusterIP service that we setup on [this section](#).

Note: If somehow you can't access the API gateway, make sure that you are not being redirected to the default website in the IIS. If you are, you can disable this in the IIS Manager.

Save this file and apply the K8S apply command.

Setting up the client configuration

Our client application should now be able to access the microservices using the API gateway. We just need to setup the connection string on the `appsettings.Production.json` as follows:

```

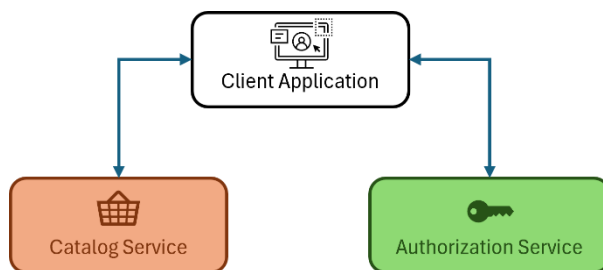
1  "ConnectionStrings": {
2    "AuthService": "http://acme.com/api/Auth",
3    "CatalogService": "http://acme.com/api/Catalog"
4  },

```

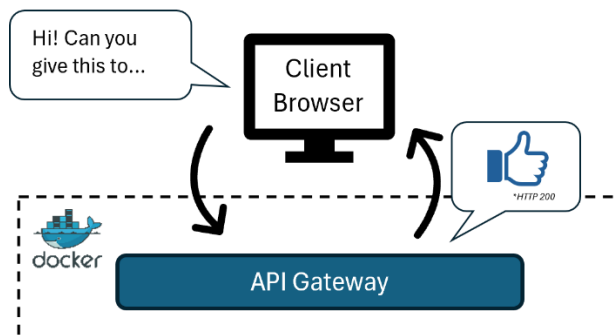
Figure 11. Setting up the connection strings on development

With this configured, we now have the appropriate architecture.

Development:



Production:



All that's left to do is to setup the service-to-service communication.

Inter-service communication

Inter-service communication refers to the mechanisms and protocols used by individual services within a distributed system to communicate with one another. In our application, we will use two different mechanisms: HTTP (synchronous) and Messaging Queues (asynchronous).

HTTP

One of the most common mechanisms for inter-service communication is HTTP (Hypertext Transfer Protocol). Services expose HTTP-based APIs that other services can call to perform actions or retrieve data.

In order to allow synchronous communication between services, the **sender** service needs to know the IP of the **receiver** service. Because our service is running on the K8S cluster, we can introduce the receiver to the sender using its ClusterIP.

From our example, CS (sender) will send a GET request to AS (receiver). Therefore, we will need to configure the application settings (`appsettings.Production.json`) of CS as follows:

```
1 "ConnectionStrings": {  
2   "DefaultConnection": "Server=mssql-clusterip-srv.1433;Initial_Catalog=catalogservicedb;TrustServerCertificate=True;User_ID=sa;Password=pa55w0rd!";  
3   "AuthServiceConnection": "http://authservice-clusterip-srv/api/Auth"  
4 },
```

Figure 12. Setting up a service for HTTP inter-service communication

Now, CS can synchronously communicate with AS.

Message Queues

Messaging queues provide asynchronous communication between services by allowing them to send and receive messages via a **message bus**. Among the popular messaging queue

implementations is the RabbitMQ. Services can **publish** messages to a queue, and other services can **subscribe** to the queue to receive and process messages asynchronously.

With the same example, CS will send an asynchronous message to the message broker, while AS will subscribe to receive that message.

First, we need to setup our message bus in the K8S cluster. In the YAML file configuration on the left image, we setup a RabbitMQ with a single replica, exposing management and messaging ports internally (via ClusterIP service). The ClusterIP service enables communication between RabbitMQ and other services within the K8S cluster.

Save and apply this using K8S apply command.

Then, we need to configure the application settings for both publisher and subscriber:

```
1 apiVersion: apps/v1  
2 kind: Deployment  
3 metadata:  
4   name: rabbitmq-depl  
5 spec:  
6   replicas: 1  
7   selector:  
8     matchLabels:  
9       app: rabbitmq  
10  template:  
11    metadata:  
12      labels:  
13        app: rabbitmq  
14    spec:  
15      containers:  
16        - name: rabbitmq  
17          image: rabbitmq:3-management  
18          ports:  
19            - containerPort: 15672  
20              name: rbmq-mgmt-port  
21            - containerPort: 5672  
22              name: rbmq-msg-port  
23  ---  
24 apiVersion: v1  
25 kind: Service  
26 metadata:  
27   name: rabbitmq-clusterip-srv  
28 spec:  
29   type: ClusterIP  
30   selector:  
31     app: rabbitmq  
32   ports:  
33     - name: rbmq-mgmt-port  
34       protocol: TCP  
35       port: 15672  
36       targetPort: 15672  
37     - name: rbmq-msg-port  
38       protocol: TCP  
39       port: 5672  
40       targetPort: 5672
```

```
1 "RabbitMQHost": "rabbitmq-clusterip-srv",  
2 "RabbitMQPort": "5672"
```

Publisher

A publisher is any service that sends a message to the message bus. In our example, CS is the publisher.

The class `MessageBusClient.cs` defines the logic in sending a message to the message bus:

```
1 public class MessageBusClient
2 {
3     private readonly IConfiguration _config;
4     private readonly IConnection _connection;
5     private readonly IModel _channel;
6
7     public MessageBusClient(IConfiguration config)
8     {
9         _config = config;
10        var factory = new ConnectionFactory()
11        {
12            HostName = _config["RabbitMQHost"],
13            Port = int.Parse(_config["RabbitMQPort"])
14        };
15
16        try
17        {
18            _connection = factory.CreateConnection();
19            _channel = _connection.CreateModel();
20
21            _channel.ExchangeDeclare(exchange: "trigger", type: ExchangeType.Fanout);
22
23            _connection.ConnectionShutdown += RabbitMQ_ConnectionShutdown;
24        }
25        catch (Exception ex)
26        {
27            Console.WriteLine($"Could not connect to the message bus: {ex.Message}");
28        }
29    }
30
31    public void SendMessage(string message)
32    {
33        MessageDto dto = new MessageDto
34        {
35            Message = message,
36            Event = "Message"
37        };
38
39        var httpContent = JsonSerializer.Serialize(dto);
40
41        if (_connection.IsOpen)
42        {
43            Console.WriteLine("RabbitMQ Connection Open; Sending message...");
44
45            var body = Encoding.UTF8.GetBytes(httpContent);
46
47            _channel.BasicPublish(
48                exchange: "trigger",
49                routingKey: "",
50                basicProperties: null,
51                body: body);
52
53            Console.WriteLine($"We have sent \"{httpContent}\"");
54        }
55        else
56        {
57            Console.WriteLine("RabbitMQ connection is closed; Unable to send message.");
58        }
59    }
60
61    public void Dispose()
62    {
63        Console.WriteLine("MessageBus disposed.");
64
65        if (_channel.IsOpen)
66        {
67            _channel.Close();
68            _connection.Close();
69        }
70    }
71
72    private void RabbitMQ_ConnectionShutdown(object sender, ShutdownEventArgs e)
73    {
74        Console.WriteLine("RabbitMQ Connection Shutdown.");
75    }
76 }
77
```


Subscriber

A subscriber is any service that is listening for messages from a message queue. Subscribers are also referred to as consumers. In our example, AS is the subscriber.

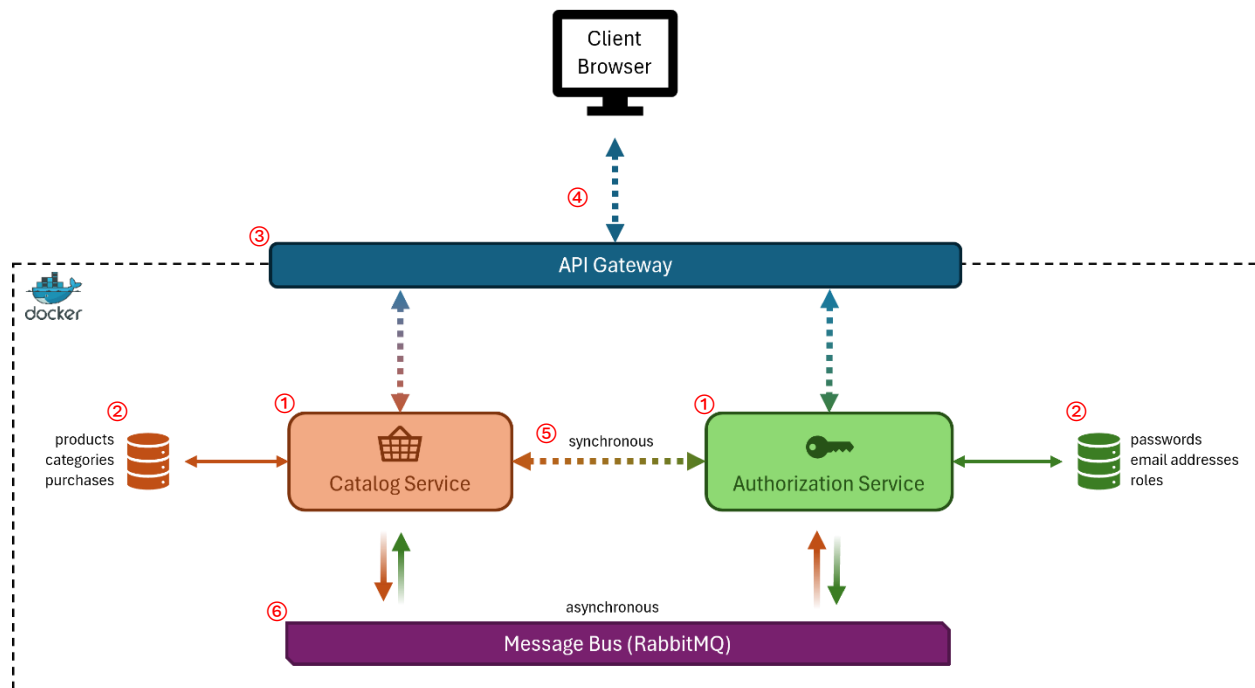
The class `MessageBusSubscriber.cs` defines the logic in subscribing to the message bus. The `IEventProcessor` contains the logic for processing the event:

```
1 public class MessageBusSubscriber : BackgroundService
2 {
3     private readonly IConfiguration _config;
4     private readonly IEventProcessor _eventProcessor;
5     private IConnection _connection;
6     private IModel _channel;
7     private string _queueName;
8
9     public MessageBusSubscriber(IConfiguration config, IEventProcessor eventProcessor)
10    {
11        _config = config;
12        _eventProcessor = eventProcessor;
13
14        InitializeRabbitMQ();
15    }
16
17    private void InitializeRabbitMQ()
18    {
19        var factory = new ConnectionFactory()
20        {
21            HostName = _config["RabbitMQHost"],
22            Port = int.Parse(_config["RabbitMQPort"])
23        };
24
25        _connection = factory.CreateConnection();
26        _channel = _connection.CreateModel();
27        _channel.ExchangeDeclare(exchange: "trigger", type: ExchangeType.Fanout);
28        _queueName = _channel.QueueDeclare().QueueName;
29        _channel.QueueBind(
30            queue: _queueName,
31            exchange: "trigger",
32            routingKey: "");
33
34        Console.WriteLine("Listening on the message bus.");
35
36        _connection.ConnectionShutdown += RabbitMQ_ConnectionShutdown;
37    }
38
39    protected override Task ExecuteAsync(CancellationToken stoppingToken)
40    {
41        stoppingToken.ThrowIfCancellationRequested();
42
43        var consumer = new EventingBasicConsumer(_channel);
44
45        consumer.Received += (ModuleHandle, ea) =>
46        {
47            Console.WriteLine("Event received.");
48
49            var body = ea.Body;
50            var message = Encoding.UTF8.GetString(body.ToArray());
51
52            _eventProcessor.ProcessEvent(message);
53        };
54
55        _channel.BasicConsume(queue: _queueName, autoAck: true, consumer: consumer);
56
57        return Task.CompletedTask;
58    }
59
60    private void RabbitMQ_ConnectionShutdown(object? sender, ShutdownEventArgs e)
61    {
62        Console.WriteLine("Connection shutdown");
63    }
64
65    public override void Dispose()
66    {
67        if (_channel.IsOpen)
68        {
69            _channel.Close();
70            _connection.Close();
71        }
72
73        base.Dispose();
74    }
75 }
76
```

Final

Your microservice should be up and running now, and the client application should be able to communicate to the microservices that are running in the Kubernetes cluster.

See the numbered items below if you want to see how we built each layer and connection.



1. [Deploying a service](#)
2. Database
 - a. [Creating a Persistent Volume Claim](#)
 - b. [Adding a K8S secret](#)
 - c. [Setting up the SQL Server](#)
3. [Ingress-NGINX](#)
4. [Setting up the client configuration](#)
5. [HTTP](#)
6. [Message Queues](#)

The code can be found in this GitHub repository: [manfortun/Microservices \(github.com\)](https://github.com/manfortun/Microservices)

References

- [1] [Docker Image \(geeksforgeeks.org\)](https://www.geeksforgeeks.org/what-is-docker-image/), <https://www.geeksforgeeks.org/what-is-docker-image/>
- [2] [Overview | Kubernetes](https://kubernetes.io/docs/concepts/overview/), <https://kubernetes.io/docs/concepts/overview/>