Introduction

Imagine you're a software developer for an online retailer named *eShopOnContainers*. The retailer uses a microservices-based architecture that's native to the cloud. It uses .NET Core for its online storefront. A new project is underway to add support for accepting coupon codes at checkout. Your assignment on the project is to finish writing a containerized ASP.NET Core web API to manage coupon codes. This web API is referred to as the coupon service.

This exercise explores completing the coupon service, adding it to the existing solution.

Learning objectives

In this exercise, you will:

- Examine existing ASP.NET Core microservices.
- Implement a new ASP.NET Core microservice and containerize it.
- Run the solution on local environment using docker-compose.

Prerequisites

- Experience writing C# at the beginner level
- Familiarity with RESTful service concepts and HTTP action verbs, such as GET, POST, PUT, and DELETE
- Conceptual knowledge of containers at the beginner level

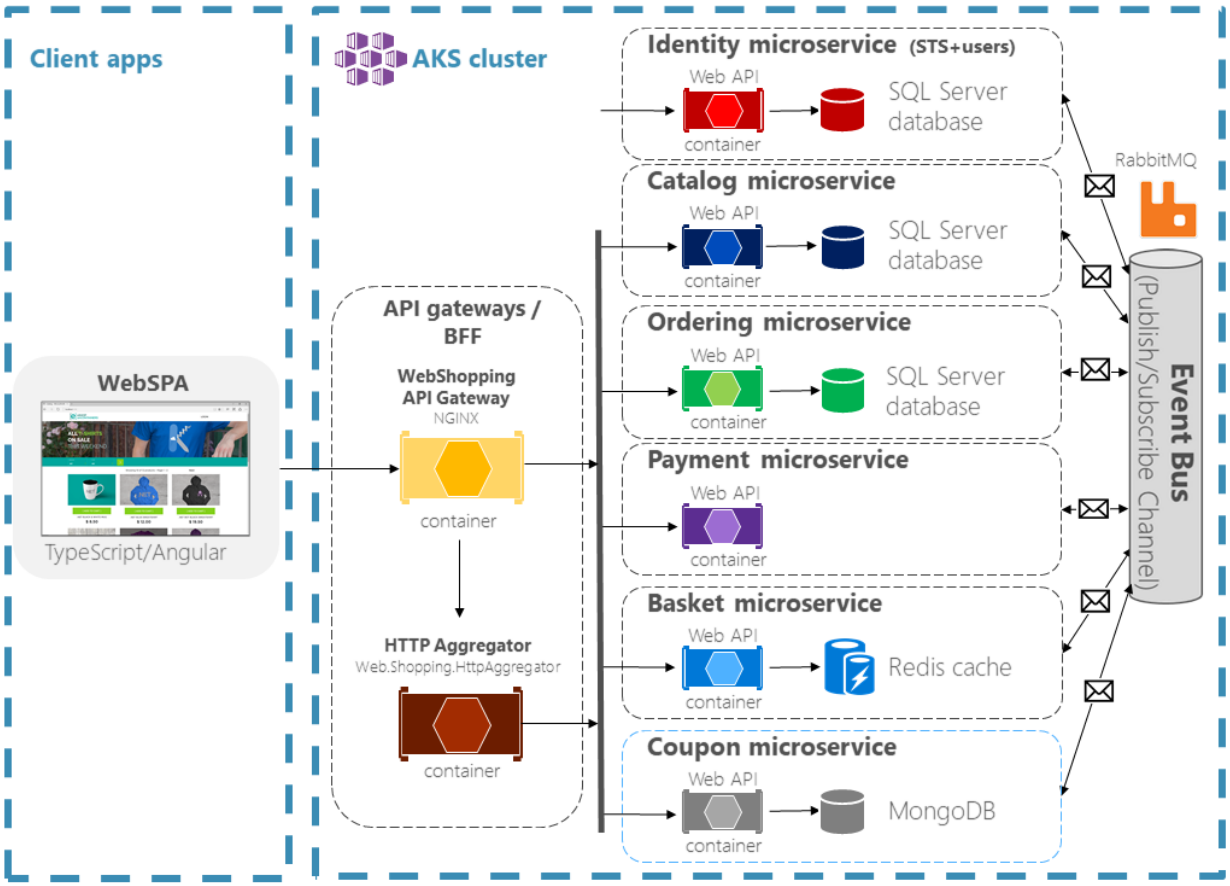
Exercise - Set up the environment

- Installs the required version of the .NET Core SDK.
- Install Docker Desktop, NodeJS and Git
- Clones the *eShopOnContainers* app from a GitHub repository.

Review the solution architecture

In this unit, you'll gain an understanding of the *eShopOnContainers* app and the microservices architecture used.

Solution architecture



The *eShopOnContainers* app is an online store that sells various products, including pins, T-shirts, and coffee mugs. The store includes the following features:

- Catalog management
- Shopping basket
- User management
- Order management
- Payments
- Coupon management (pictured in the diagram, but doesn't exist yet)

You manage each of the preceding features with a distinct microservice. Each microservice is autonomous, independently deployable, and responsible for its own data. This architecture enables each microservice to implement the data store that is best optimized for its workload, storage needs, and read/write patterns. Data store choices include relational, document, key-value, and graph-based.

For example, the *catalog* service stores its data in a SQL Server on Linux database. The *basket* service uses a Redis cache for storage. There's no single master data store with which all services interact. Instead, inter-service communication occurs on an as-needed basis, either via synchronous API calls or asynchronously through messaging. This data isolation gives every service the autonomy to independently apply data schema updates, without breaking other services in the production environment.

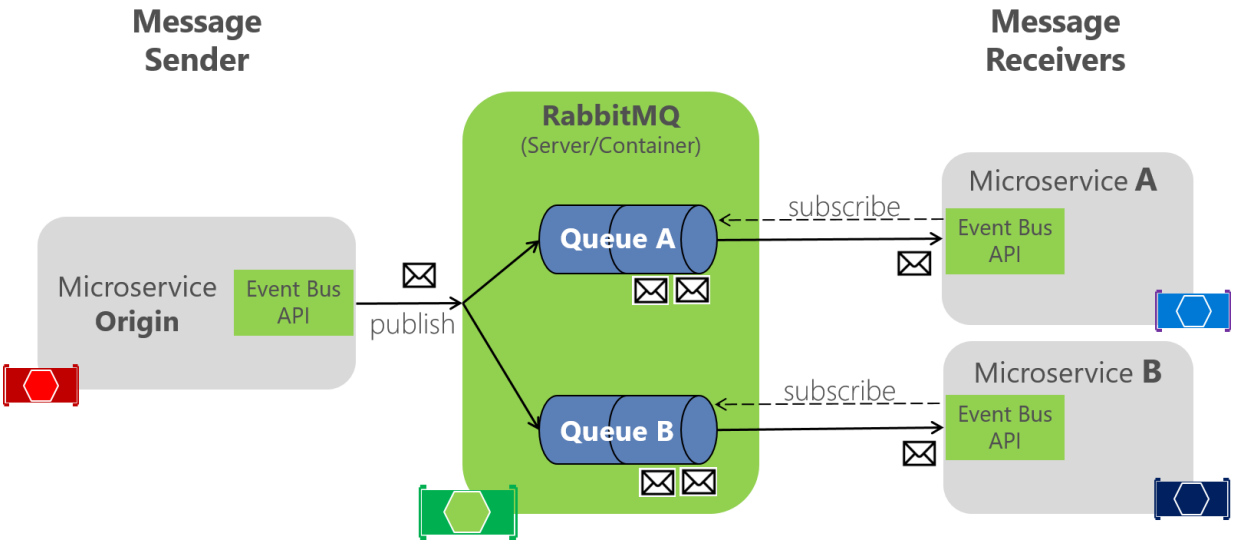
Authentication and authorization

The *WebSPA* client app delegates authentication and authorization concerns to an identity service that also serves as a Security Token Service (STS). The identity service is a containerized ASP.NET Core project that uses IdentityServer 4, a popular OpenID Connect and OAuth 2.0 framework for ASP.NET Core. An alternative to hosting an STS is to use Azure Active Directory (Azure AD). Azure AD offers identity and access management as a service.

The identity service in the diagram is configured to allow direct access. Consequently, the API gateway is bypassed. The full *eShopOnContainers* app, on which this sample is based, uses multiple API gateways separated by business areas. For this smaller implementation, however, another API gateway isn't required.

Event Bus

You use an event bus for asynchronous messaging and event-driven communication. The preceding architecture diagram depicts RabbitMQ in a Docker container deployed to AKS, but a service such as Azure Service Bus would also be appropriate.



The preceding diagram depicts the publish/subscribe (commonly shortened to *pub-sub*) pattern used with the event bus. Any service can publish an event to the event bus. Each service is responsible for subscribing to the messages relevant to its domain. The services each call an `AddEventBus` extension method in the `ConfigureServices` method of *Startup.cs*. This method establishes a connection to the event bus and registers the appropriate event handlers for that service's domain.

API Gateways

The microservices are accessible to clients via the API gateway. Among other advantages, API gateways enhance security and decouple back-end services from individual clients.

The *WebSPA* storefront is an ASP.NET Core MVC and Angular app that is accessible via a public IP address. The HTTP requests from the *WebSPA* app to the microservices are routed through the API gateway, which is an implementation of the Backends-For-Frontends pattern. Implement basic routing configurations by using the NGINX reverse proxy. The ASP.NET Core web API named *Web.Shopping.HttpAggregator* combines multiple requests into a single request. This is a pattern known as *gateway aggregation*.

For real-world scenarios, use managed API gateway services like Azure API Management

Coupon Service

Microservices are small enough for a feature team to independently build, test, and deploy to production multiple times a day, without affecting other systems. In this module, you'll complete and deploy an ASP.NET Core microservice project named *Coupon.API* to the existing *eShopOnContainers* app. While doing so, you'll also learn about:

- Designing microservices by using Domain-Driven Design (DDD).
- Wiring up communications between microservices using event bus.
- Containerizing services by using Docker.
- Debug and run the solution locally using docker-compose.

In the architecture diagram, the coupon service to be added is outlined with a dashed, blue box.

Other Services

There are a few services in the *eShopOnContainers* app that aren't represented in the preceding diagram. Services such as *Seq* (for unified logging) and the *WebStatus* web app are also present in the solution.

Exercise - Review the code and run the solution locally

Review code

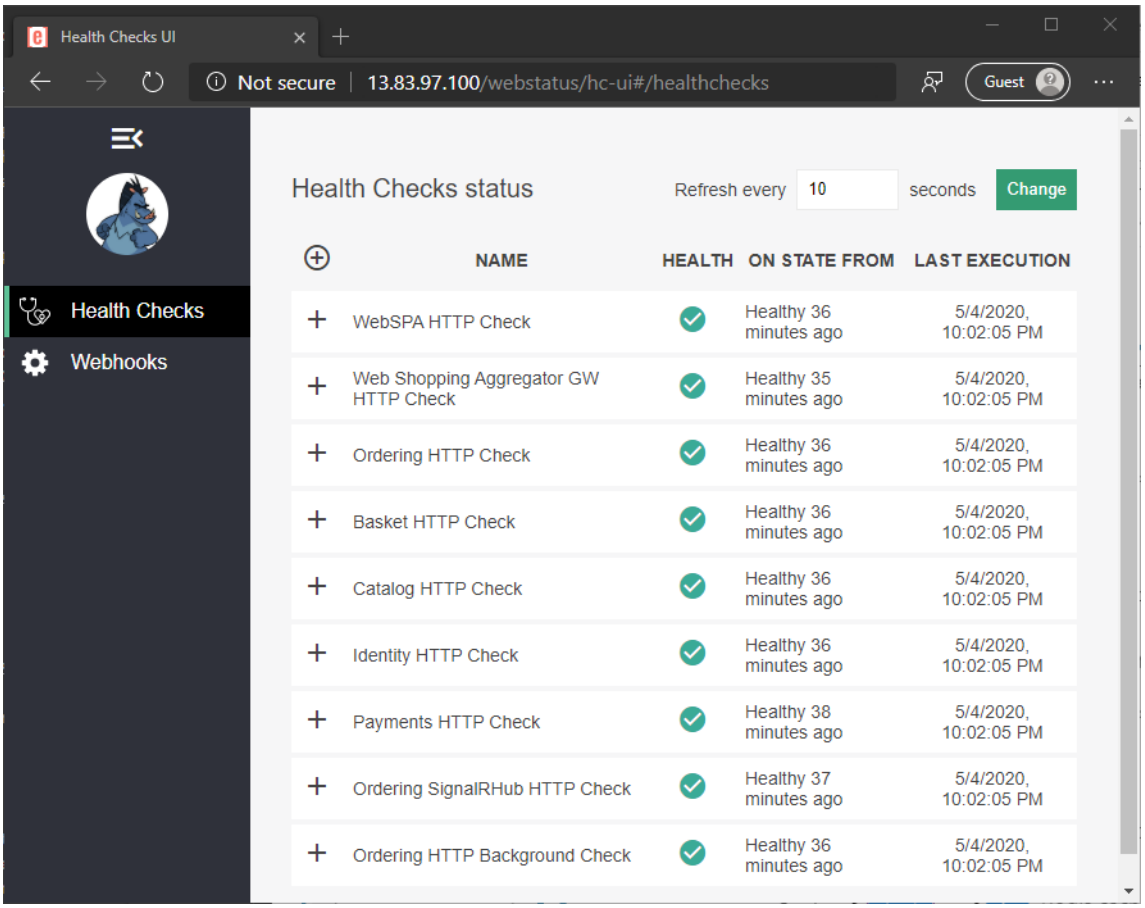
The following *src* subdirectories contain .NET Core projects, each of which is containerized:

Project directory	Description
ApiGateways/	Services to aggregate across multiple microservices for certain cross-service operations. An HTTP aggregator is implemented in the *ApiGateways/Aggregators/Web.Shopping.HttpAggregator* project.
BuildingBlocks/	Services that provide cross-cutting functionality, such as the app's event bus used for inter-service events.
Services/	These projects implement the business logic of the app. Each microservice is autonomous, with its own data store. They showcase different software patterns, including Create-Read-Update-Delete (CRUD), Domain-Driven Design (DDD), and Command and Query Responsibility Segregation (CQRS). The new *Coupon.API* project has been provided, but it's incomplete.
Web/	ASP.NET Core apps that implement user interfaces. *WebSPA* is a storefront UI built with Angular. *WebStatus* is the health checks dashboard for monitoring the operational status of each service.

Run the solution locally

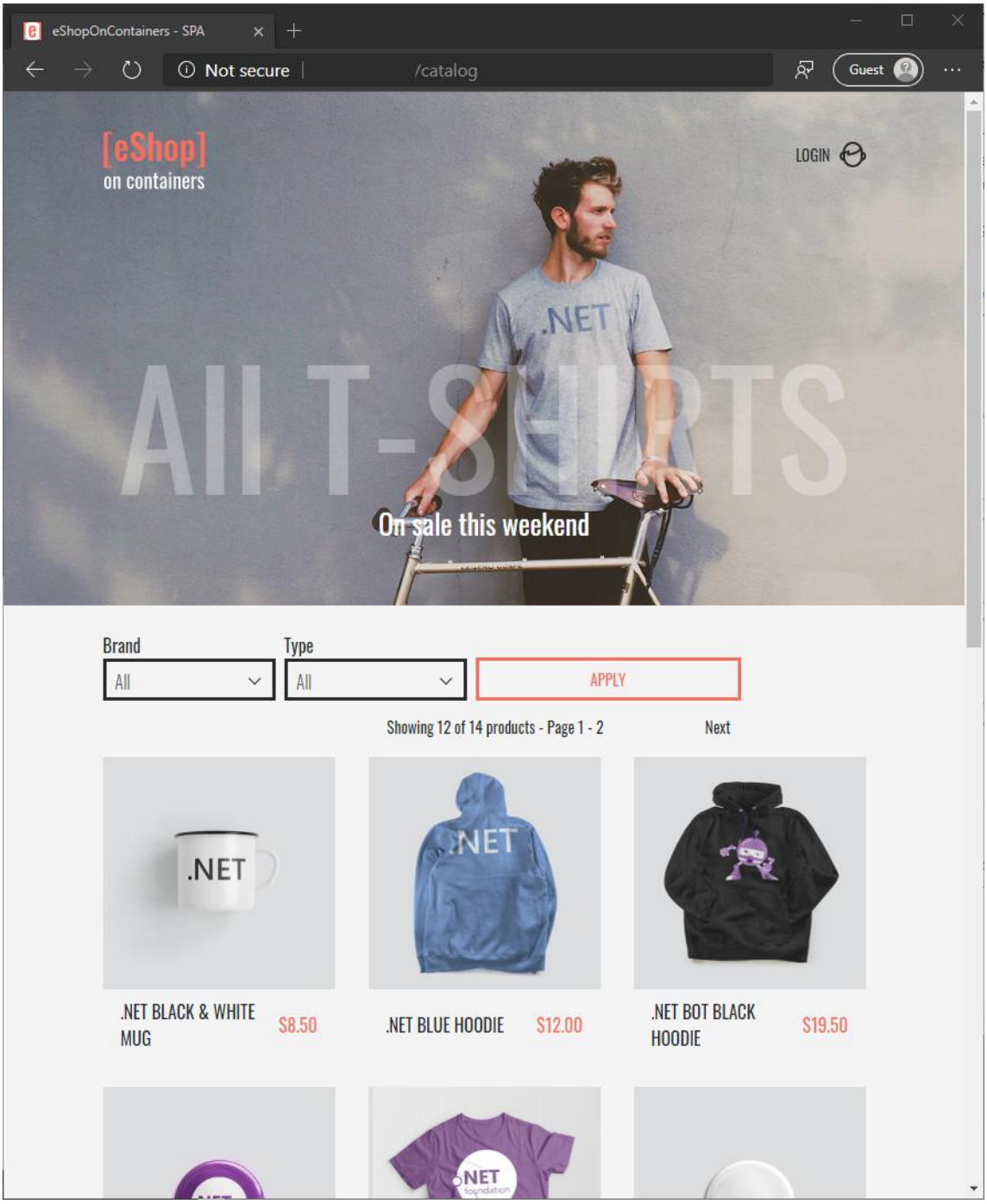
Open solution using Visual Studio, right-click *docker-compose* project and *Set as Startup Project*. Start debugging or press F5.

1. Use your browser and open http://localhost:5017 to view the *WebStatus* health check dashboard. The resulting page displays the status of each microservice in the debugging mode. The page refreshes automatically, every 10 seconds.

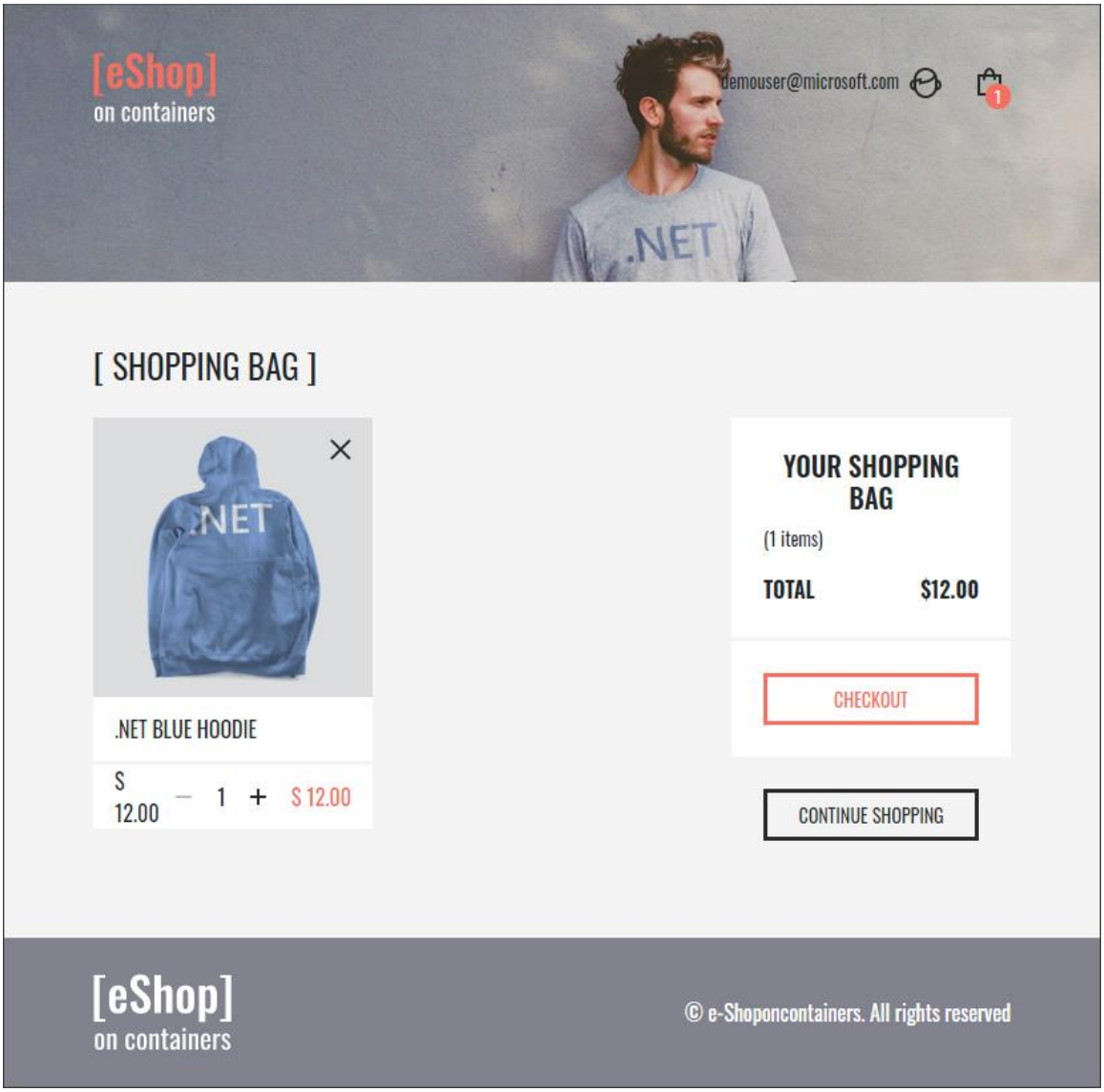


Note:
While the app is starting up, you might initially receive an HTTP 503 response from the server. Retry after a few seconds. The Seq logs, which are viewable at the Centralized logging URL, are available before the other endpoints.

2. After all the services are healthy, select the Web SPA application link http://localhost:5014 to test the *eShopOnContainers* web app. The following page appears:



3. Complete a purchase as follows:
   a. Select the **LOGIN** link in the upper right to sign into the app. The credentials are provided on the page.
   b. Add the **.NET BLUE HOODIE** to the shopping bag by selecting the image.
   c. Select the shopping bag icon in the upper right.
   d. Select **CHECKOUT**, and then select **PLACE ORDER** to complete the purchase.



In this unit, you've seen the *eShopOnContainers* app's existing checkout process. You'll review the design of the new coupon service in the next unit.

Review the coupon service design

In this unit, you learn about the business requirements for the requested coupon code feature. Additionally, you'll learn about the Domain-Driven Design (DDD) design pattern, and the selected technology stack for the solution.

Business requirements

There are many ways to implement a coupon code feature in an e-commerce app. For simplicity, let's assume the following business requirements:

- To obtain a discount, the user can apply a coupon code from the checkout page. All coupon codes are prefixed with *DISC-* and are suffixed with an unsigned integer. The integer indicates the US dollar (USD) amount to be deducted from the order total. For example, *DISC-30* deducts 30 USD.
- The coupon service must validate that the coupon code is available before allowing it to be used.
- After the order stock is confirmed, the ordering service should request validation for the coupon during the order process.
- Upon validation, the coupon should be assigned to the order and won't be available for any other order.
- If an order is canceled, the assigned coupon should be released for any other order to use.

Domain model

DDD is a design pattern whereby the structure and language of your code, including class names, methods, and variables, matches the business domain. The pattern describes independent problem areas as bounded contexts and emphasizes a common language to describe these problems. Determining where to draw the boundaries is an essential task when designing and defining a microservices-based solution. For each bounded context, you must identify and define the entities, value objects, and aggregates that model your domain.

You implement the coupon service like a CRUD service, and you clearly define the boundaries within the coupon domain. The service represents a `Coupon` as a Coupon object, using the class defined at *src/Services/Coupon/Coupon.API/Infrastructure/Models/Coupon.cs*. This class encapsulates the attributes of a coupon.

Property	Description
`Id`	The unique identifier of the coupon.
`Discount`	The discount amount represented in USD currency.
`Code`	The coupon code.
`Consumed`	A flag that indicates whether the coupon code has been used.
`OrderId`	The unique identifier of the associated order to which the coupon code has been applied.

These design choices might seem obvious but note that the `Coupon` model is a centerpiece of all business logic in the *Coupon.API* project. The coupon service:

- Only concerns itself with the domain of coupons.
- Relies on the other services to interact with other domains, such as determining whether an order is valid.

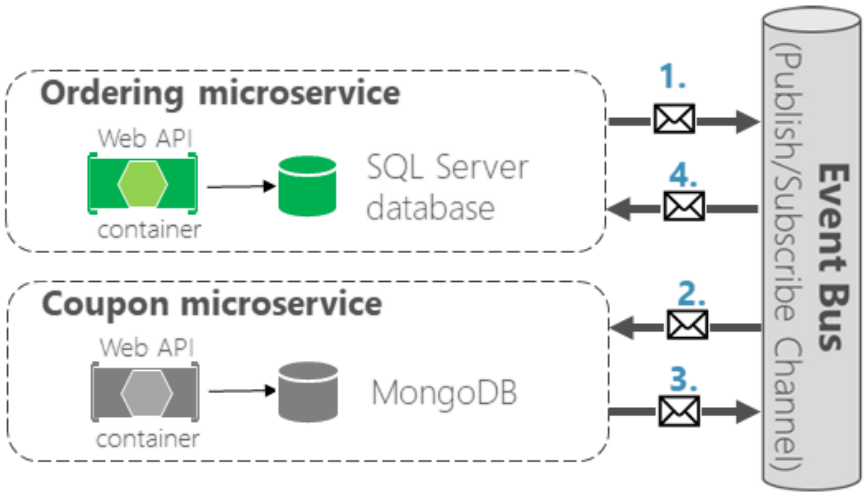
Technology stack

Microservice architectures are technology agnostic. This characteristic gives development teams the flexibility to select the technology stack for each service independently. The following table outlines the relevant technologies used by the coupon service.

Technology	Description
ASP.NET Core web API	The RESTful services for managing discounts are implemented in ASP.NET Core. A web API uses controllers to handle HTTP requests.
MongoDB	The NoSQL database that stores the coupons and their usage data. In a real-world scenario, it's common for services to use a managed database, like Azure Cosmos DB, instead of running them in a container.
Docker	The web API project and the MongoDB database, along with their dependencies, are packaged into respective container images by using Docker.

Integration events in the ordering service

You route orders submitted through the UI to the ordering service via the API gateway. The ordering service then validates the coupon with the coupon service.



1. The ordering service raises an event of type `OrderStatusChangedToAwaitingCouponValidationIntegrationEvent`. The event:

    - Indicates that there's an order awaiting coupon validation.
    - Is serialized as a message on the event bus.

2. The coupon service receives the event from the event bus and invokes the appropriate event handler to determine the status of the coupon.
3. The coupon service raises an event to indicate the status of the coupon.

    - If the coupon is valid, the coupon service:
        - Marks the coupon as "consumed" in the database.
        - Raises an event of type `OrderCouponConfirmedIntegrationEvent`. This event sends a message to the event bus to inform the ordering service that the order can be processed.
    - If the coupon isn't valid, an event of type `OrderCouponRejectedIntegrationEvent` is raised.

4. The ordering service handles the event that was raised by the coupon service.

Exercise - Add the coupon service

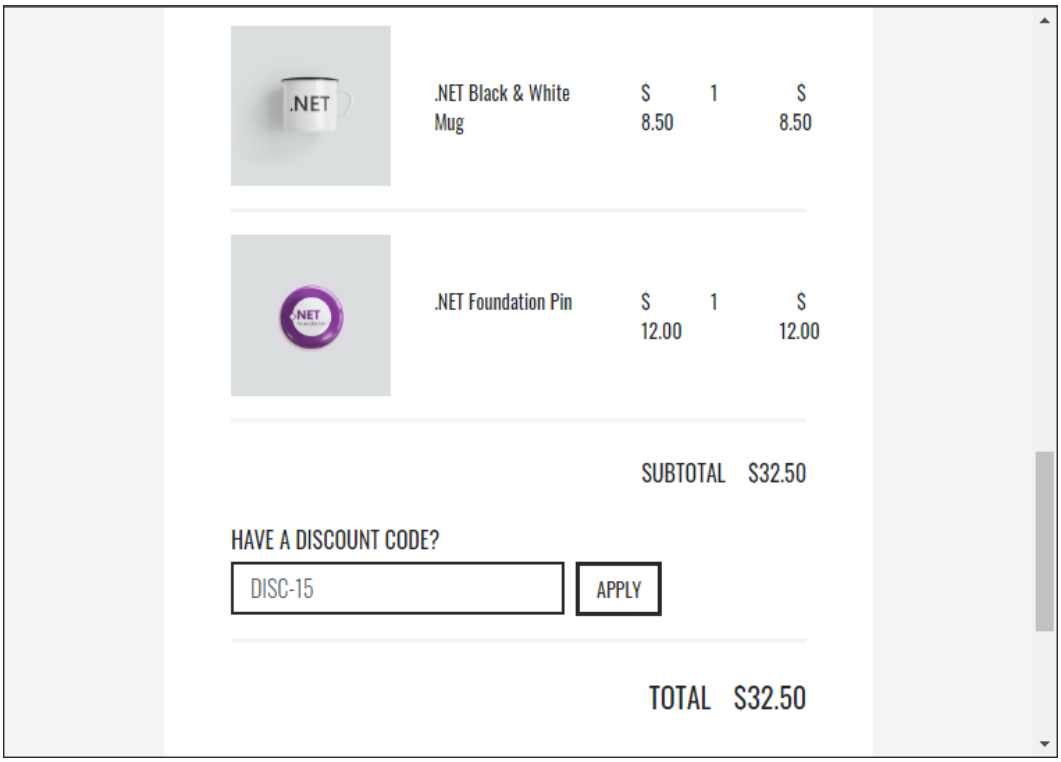
In this unit, you complete the *Coupon.API* project as well as some parts of other services impacted by this new feature that need to be changed. Luckily you don't have to do all required changes by yourself as assume other teams have made some of the changes for you.

Changes have been done for you

The following are the changes that have been done by other teams:

1. *WebSPA* project

Assume another frontend developer has made all required changes on *WebSPA* project so you don't need to make any changes at all on it. One of the UI changes look something like this:



2. ***Docker-compose*** **project**

Assume the DevOps engineer has provided the changes on *docker-compose* project so you no need to make any changes on it.

3. ***Catalog.API*** **project**

The required changes also have been done on this project by other team who own the service.

4. ***Identity.API*** **project**

Security and operational team also have made all required changes on *Identity.API* project so you can leave it.

Changes you need to do

The only changes that you need to do are as follows:

1. ***API Gateways*** **configuration**
   Register *Coupon.API* into API Gateways by adding necessary configuration into **envoy.yml** file under *src/ApiGateways/Envoy/config/webshopping* directory.
   Note: API Gateways use Envoyproxy https://www.envoyproxy.io/

2. ***Coupon.API*** **project**

Most of the infrastructure and functionalities have been developed for you:

- Coupon read-write functionality to the database (mongodb) using repository pattern
- Model, DTOs and DataSeed
- Swagger
- Authorization using Identity.API
- etc.

The rest you need to complete!

3. ***Basket.API*** **project**

Order checkout process start within Basket.API service, so you need to make a change for the following files:

3.1. On *UserCheckoutAcceptedIntegrationEvent.cs* file located under *src/Services/Basket/Basket.API/IntegrationEvents/Events* directory, add the following new read-only public properties:

Property Name	**Type**	**Description**
DiscountCode	string	Coupon code
Discount	decimal	Amount of the discount in USD

3.2. Still on the same file (*UserCheckoutAcceptedIntegrationEvent.cs*) change the class constructor to include values for the above newly created properties.

3.3. On *BasketCheckout.cs* file *src/Services/Basket/Basket.API/Model* directory, add the following read-write public properties:

Property Name	**Type**	**Description**
Coupon	string	Coupon code
Discount	int	Amount of the discount in USD

3.4. On *BasketController.cs* file located under *src/Services/Basket/Basket.API/Controllers* directory, make some necessary changes accordingly to satisfy the above changes.

4. **Ordering Service**

4.1. *Ordering.Domain* project

Some changes you need to do on Ordering.Domain project are as follows:

4.1.1. On *OrderStatus.cs* file located under *src/Services/Ordering/Ordering.Domain/AggregatesModel/OrderAggregate* directory:

a. Add new **order status** called "`AwaitingCouponValidation`"
b. Change/rename the **existing order** status called "`AwaitingValidation`" to "`AwaitingStockValidation`" to make it more explicit.
c. Make another necessarily adjustments inside the file due to the above changes.

4.1.2. Under *src/Services/Ordering/Ordering.Domain/Events* directory, create new domain event class called "`OrderStatusChangedToAwaitingCouponValidationDomainEvent`" and make sure it contains the following read-only public properties, and don't forget to change the class constructor to supply its values:

Property Name	Type	Description
OrderId	int	Order Id
Code	string	Coupon code

Or you can just copy the file form:
assets/Services/Ordering/Ordering.Domain/Events/OrderStatusChangedToAwaitingCouponValidationDomainEvent.cs

4.1.3. On *Order.cs* file located under *src/Services/Ordering/Ordering.Domain/AggregatesModel/OrderAggregate* directory:

a. Add the following read-only public properties:

Property Name	Type	Description
DiscountConfirmed	bool?	Nullable flag indicates if discount has been confirmed
DiscountCode	string	It contains Coupon Code
Discount	decimal?	Amount of discount in USD

b. Change its class constructor to supply the values for the above newly created properties.

c. Change/rename the existing public method called "`SetAwaitingValidationStatus()`" with "`SetAwaitingStockValidationStatus()`" and make sure the *_orderStatusId* class field inside the method set to "`OrderStatus.AwaitingStockValidation.Id`"

d. Replace the public method "`SetStockConfirmedStatus()`" with the following code:

```
public void SetStockConfirmedStatus()
    {
        if (_orderStatusId == OrderStatus.AwaitingStockValidation.Id)
        {
            if (DiscountCode == null)
            {
                AddDomainEvent(new
OrderStatusChangedToStockConfirmedDomainEvent(Id));

                _orderStatusId = OrderStatus.StockConfirmed.Id;
                _description = "All the items were confirmed with
available stock.";
            }
            else
            {
                AddDomainEvent(new
OrderStatusChangedToAwaitingCouponValidationDomainEvent(Id,
DiscountCode));

                _orderStatusId =
OrderStatus.AwaitingCouponValidation.Id;
                _description = "Validate discount code.";
            }
        }
    }
```

e. Add a new public method called "`ProcessCouponConfirmed()`" without any arguments and put the following code inside it:

```
if (_orderStatusId == OrderStatus.AwaitingCouponValidation.Id)
{
    AddDomainEvent(new
OrderStatusChangedToStockConfirmedDomainEvent(Id));

    _orderStatusId = OrderStatus.StockConfirmed.Id;
    _description = "Discount coupon validated.";
    DiscountConfirmed = true;
}
```

f. Make sure you replace all the "`OrderStatus.AwaitingValidation.Id`" with "`OrderStatus.AwaitingStockValidation.Id`" inside that class.

g. Change formula to get the total order inside public method "`GetTotal()`" with the following new formula:

```
return _orderItems.Sum(o => o.GetUnits() * o.GetUnitPrice()) -
(Discount ?? 0);
```

4.2. *Ordering.API* project

Some changes you need to do on *Ordering.API* project are as follows:

4.3.1. On *OrderingContextSeed.cs* file located under *src/Services/Ordering/Ordering.API/Infrastructure* directory, inside private function "`GetPredefinedOrderStatus()`" change existing order status "`OrderStatus.AwaitingValidation`" with "`OrderStatus.AwaitingStockValidation`" and also add new order status "`OrderStatus.AwaitingCouponValidation`"

4.3.2. On *src/Services/Ordering/Ordering.API/Application/Commands* directory, create a new command class called "`CouponConfirmedOrderStatusCommand`" by copying the following file: *assets/Services/Ordering/Ordering.API/Application/Commands/ CouponConfirmedOrderStatusCommand.cs*

4.3.3. On *src/Services/Ordering/Ordering.API/Application/Commands* directory, create a new command handler class called "`CouponConfirmedOrderStatusCommandHandler`" by copying the following file: *assets/Services/Ordering/Ordering.API/Application/Commands/ CouponConfirmedOrderStatusCommandHandler.cs*

4.3.4. On existing *CreateOrderCommand.cs* file located under *src/Services/Ordering/Ordering.API/Application/Commands* directory, add the following read-only public properties:

Property Name	Type	Description
DiscountCode	string	It contains Coupon Code
Discount	decimal	Amount of discount in USD

Decorate the above properties with "`[DataMember]`" attribute and make sure you change the class constructor to include values for the above newly added properties.

4.3.5. On existing *CreateOrderCommandHandler.cs* file located under *src/Services/Ordering/Ordering.API/Application/Commands* directory, modify the *order* variable assignment inside public method "`Handle()`" to satisfy new "`Order`" constructor.

4.3.6. On *SetAwaitingValidationOrderStatusCommandHandler.cs* file located under *src/Services/Ordering/Ordering.API/Application/Commands* directory, inside public method "`Handle()`" replace the following code:

```
orderToUpdate.SetAwaitingValidationStatus();
```

with:

```
orderToUpdate.SetAwaitingStockValidationStatus();
```

4.3.7. Under *src/Services/Ordering/Ordering.API/Application/DomainEventHandlers* directory, create new folder called "OrderCoupon". Inside that folder create new domain event handler class called "`OrderStatusChangedToAwaitingCouponValidationDomainEventHandler`" by copying from the following file: *Assets/Services/Ordering/Ordering.API/Application/DomianEventHandlers/OrderCoupon/OrderStatusChangedToAwaitingCouponValidationDomainEventHandler.cs*

4.3.8. Under *src/Services/Ordering/Ordering.API/Application/IntegrationEvents/Events* directory, rename *OrderStatusChangedToAwaitingValidationIntegrationEvent.cs* file (including its class name) to *OrderStatusChangedToAwaitingStockValidationIntegrationEvent.cs* file to make it more explicit.

4.3.9. Under *src/Services/Ordering/Ordering.API/Application/DomainEventHandlers/OrderGracePeriodConfirmed* directory, rename *OrderStatusChangedToAwaitingValidationDomainEventHandler.cs* file (including its class name) to *OrderStatusChangedToAwaitingStockValidationDomainEventHandler.cs* file to make it more explicit and make.

4.3.10. Under *src/Services/Ordering/Ordering.API/Application/IntegrationEvents/EventHandling* directory, create new integration event handler class called "`OrderCouponConfirmedIntegrationEventHandler`" by copying the following file: *assets/Services/Ordering/Ordering.API/Application/IntegrationEvents/EventHandling/OrderCouponConfirmedIntegrationEventHandler.cs*

4.3.11. Under *src/Services/Ordering/Ordering.API/Application/IntegrationEvents/EventHandling* directory, create new integration event handler class called "`OrderCouponRejectedIntegrationEventHandler`" by copying the following file: *assets/Services/Ordering/Ordering.API/Application/IntegrationEvents/EventHandling/ OrderCouponRejectedIntegrationEventHandler.cs*

4.3.12. Under *src/Services/Ordering/Ordering.API/Application/IntegrationEvents/EventHandling* directory, modify *UserCheckoutAcceptedIntegrationEventHandler.cs* file to satisfy the changes of *CreateOrderCommand* command class.

4.3.13. Under *src/Services/Ordering/Ordering.API/Application/IntegrationEvents/Events* directory, create new integration event class called "`OrderCouponConfirmedIntegrationEvent`" by copying from the following file:
assets/Services/Ordering/Ordering.API/Application/IntegrationEvents/Events/OrderCouponConfirmedIntegrationEvent.cs

4.3.14. Under *src/Services/Ordering/Ordering.API/Application/IntegrationEvents/Events* directory, create new integration event class called "`OrderCouponRejectedIntegrationEvent`" by copying from the following file:
assets/Services/Ordering/Ordering.API/Application/IntegrationEvents/Events/OrderCouponRejectedIntegrationEvent.cs

4.3.15. Under *src/Services/Ordering/Ordering.API/Application/IntegrationEvents/Events* directory, create new integration event class called "`OrderStatusChangedToAwaitingCouponValidationIntegrationEvent`" by copying from the following file:
assets/Services/Ordering/Ordering.API/Application/IntegrationEvents/Events/OrderStatusChangedToAwaitingCouponValidationIntegrationEvent.cs

4.3.16. Under *src/Services/Ordering/Ordering.API/Application/IntegrationEvents/Events* directory, modify integration event class called "`UserCheckoutAcceptedIntegrationEvent`" by adding the following read-write public properties and don't forget to change the class constructor to include values for those properties:

Property Name	Type	Description
DiscountCode	string	It contains Coupon Code
Discount	decimal	Amount of discount in USD

4.3.17. Under *src/Services/Ordering/Ordering.API/Application/Queries* directory, modify the code of class "`OrderQueries`" by copying the code form the following file:
assets/Services/Ordering/Ordering.API/Application/Queries/OrderQueries.cs

4.3.18. Under *src/Services/Ordering/Ordering.API/Application/Queries* directory, modify the code of class "`OrderViewModel`" by copying the code form the following

file: *assets/Services/Ordering/Ordering.API/Application/Queries/OrderViewModel.cs*

4.3.19. Inside file *src/Services/Ordering/Ordering.API/Startup.cs*, modify private method "`ConfigureEventBus()`" to include the following code:

```
eventBus.Subscribe<OrderCouponRejectedIntegrationEvent,
IIntegrationEventHandler<OrderCouponRejectedIntegrationEvent>>();

eventBus.Subscribe<OrderCouponConfirmedIntegrationEvent,
IIntegrationEventHandler<OrderCouponConfirmedIntegrationEvent>>();
```

4.3. *Ordering.SignalrHub* project

Some changes on *Ordering.SignalrHub* project are as follows:

4.3.1. Under *src/Services/Ordering/Ordering.SignalrHub/IntegrationEvents/Events* directory, rename *OrderStatusChangedToAwaitingValidationIntegrationEvent.cs* file (including its class name) to *OrderStatusChangedToAwaitingStockValidationIntegrationEvent.cs* file to make it more explicit.

4.3.2. Under *src/Services/Ordering/Ordering.SignalrHub/IntegrationEvents/Events* directory, create new integration event class called "`OrderStatusChangedToAwaitingCouponValidationIntegrationEvent`" by copying from the following file: *Assets/Services/Ordering/Ordering.SIgnalrHub/IntegrationEvents/Events/OrderStatusChangedToAwaitingCouponValidationIntegrationEvent.cs*

4.3.3. Under *src/Services/Ordering/Ordering.SignalrHub/IntegrationEvents/EventHandling* directory, rename *OrderStatusChangedToAwaitingValidationIntegrationEvent.cs* file (including its class name) to *OrderStatusChangedToAwaitingStockValidationIntegrationEvent.cs* file to make it more explicit.

4.3.4. Under *src/Services/Ordering/Ordering.SignalrHub/IntegrationEvents/EventHandling* directory, create new integration event handler class called "`OrderStatusChangedToAwaitingCouponValidationIntegrationEventHandler`" by copying from the following file: *Assets/Services/Ordering/Ordering.SIgnalrHub/IntegrationEvents/EventHandling/OrderStatusChangedToAwaitingCouponValidationIntegrationEventHandler.cs*

4.3.5. Inside file *src/Services/Ordering/Ordering.SignalrHub/Startup.cs*, modify private method "`ConfigureEventBus()`" to include the following code:

```
eventBus.Subscribe<OrderStatusChangedToAwaitingCouponValidationIntegration
Event,
OrderStatusChangedToAwaitingCouponValidationIntegrationEventHandler>();
```

Run and debug the solution

In your Visual Studio, right-click *docker-compose* project and *Set as Startup Project* and start debugging or press F5.

See if all services can run properly and discount feature works as expected.