Asynchronous communication

(Demo prep)

Consuming messages

1. The whole point of moving Inventory to using Asynchronous communication is to have its own database of catalog items that it can use without ever having to get in touch with the Catalog service API. Therefore, let's start by defining the entity that represents Catalog items in the Inventory microservice.

2. Add the CatalogItem entity:

```
namespace Play.Inventory.Service.Entities
{
    public class CatalogItem : IEntity
    {
        public Guid Id { get; set; }

        public string Name { get; set; }

        public string Description { get; set; }

    }
}
```

3. dotnet add package Play.Catalog.Contracts

4. Bump the version of **Play.Common** to **1.0.1**

5. Create a **Consumers** directory

6. Add the CatalogItemCreatedConsumer:

```
namespace Play.Inventory.Service.Consumers
{
    public class CatalogItemCreatedConsumer : IConsumer<CatalogItemCreated>
    {
        private readonly IRepository<CatalogItem> repository;

        public CatalogItemCreatedConsumer(IRepository<CatalogItem> repository)
        {
            this.repository = repository;
        }
```

```
    public async Task Consume(ConsumeContext<CatalogItemCreated> context)
    {
      var message = context.Message;

      var item = await repository.GetAsync(message.ItemId);

      if (item != null)
      {
        return;
      }

      item = new CatalogItem
      {
        Id = message.ItemId,
        Name = message.Name,
        Description = message.Description
      };

      await repository.CreateAsync(item);
    }
  }
}
```

7. Add the CatalogItemUpdatedConsumer:

```
namespace Play.Inventory.Service.Consumers
{
  public class CatalogItemUpdatedConsumer : IConsumer<CatalogItemUpdated>
  {
    private readonly IRepository<CatalogItem> repository;

    public CatalogItemUpdatedConsumer(IRepository<CatalogItem> repository)
    {
      this.repository = repository;
    }

    public async Task Consume(ConsumeContext<CatalogItemUpdated> context)
    {
      var message = context.Message;

      var item = await repository.GetAsync(message.ItemId);

      if (item == null)
      {
        item = new CatalogItem
        {
          Id = message.ItemId,
          Name = message.Name,
```

```
                Description = message.Description
            };

            await repository.CreateAsync(item);
        }
        else
        {
            item.Name = message.Name;
            item.Description = message.Description;

            await repository.UpdateAsync(item);
        }
    }
  }
}
```

8. Add the CatalogItemDeletedConsumer:

```
namespace Play.Inventory.Service.Consumers
{
  public class CatalogItemDeletedConsumer : IConsumer<CatalogItemDeleted>
  {
    private readonly IRepository<CatalogItem> repository;

    public CatalogItemDeletedConsumer(IRepository<CatalogItem> repository)
    {
      this.repository = repository;
    }

    public async Task Consume(ConsumeContext<CatalogItemDeleted> context)
    {
      var message = context.Message;

      var item = await repository.GetAsync(message.ItemId);

      if (item == null)
      {
        return;
      }

      await repository.RemoveAsync(message.ItemId);
    }
  }
}
```

9. Update appsettings.json:

```
 "RabbitMQSettings": {
```

```
  "Host": "localhost"
 },
```

10. Update Startup:

```
public void ConfigureServices(IServiceCollection services)
{
   services.AddMongo()
         .AddMongoRepository<InventoryItem>("inventoryitems")
         .AddMongoRepository<CatalogItem>("catalogitems")
         .AddMassTransitWithRabbitMq();

   AddCatalogClient(services);

   services.AddControllers();
   services.AddSwaggerGen(c =>
   {
      c.SwaggerDoc("v1", new OpenApiInfo { Title = "Play.Inventory.Service", Version = "v1" });
   });
}

private static void AddCatalogClient(IServiceCollection services)
{
   Random jitterer = new Random();

   services.AddHttpClient<CatalogClient>(client =>
   {
      client.BaseAddress = new Uri("https://localhost:5001");
   })
   .AddTransientHttpErrorPolicy(builder => builder.Or<TimeoutRejectedException>().WaitAndRetryAsync(
      5,
      retryAttempt => TimeSpan.FromSeconds(Math.Pow(2, retryAttempt))
                + TimeSpan.FromMilliseconds(jitterer.Next(0, 1000)),
      onRetry: (outcome, timespan, retryAttempt) =>
      {
         var serviceProvider = services.BuildServiceProvider();
         serviceProvider.GetService<ILogger<CatalogClient>>()?
            .LogWarning($"Delaying for {timespan.TotalSeconds} seconds, then making retry {retryAttempt}");
      }
   ))
   .AddTransientHttpErrorPolicy(builder => builder.Or<TimeoutRejectedException>().CircuitBreakerAsync(
      3,
      TimeSpan.FromSeconds(15),
      onBreak: (outcome, timespan) =>
      {
         var serviceProvider = services.BuildServiceProvider();
         serviceProvider.GetService<ILogger<CatalogClient>>()?
            .LogWarning($"Opening the circuit for {timespan.TotalSeconds} seconds...");
```

```
      },
      onReset: () =>
      {
         var serviceProvider = services.BuildServiceProvider();
         serviceProvider.GetService<ILogger<CatalogClient>>()?
            .LogWarning($"Closing the circuit...");
      }
   ))
   .AddPolicyHandler(Policy.TimeoutAsync<HttpResponseMessage>(1));
}
```

11. Switch default **LogLevel** to **debug** appsettings.Development.json

12. **DELETE THE DATABASES**

13. Start Catalog and Inventory via **dotnet run**

14. Notice in RabbitMQ that queues have been created

15. Create a new catalog item in Postman:

```
{
   "name": "Potion",
   "description": "Restores a small amount of HP",
   "price": 5
}
```

16. Notice logs and verify Inventory DB is now there with catalogitems collection

17. Stop Inventory

18. Create a new catalog item:

```
{
   "name": "Antidote",
   "description": "Cures poison",
   "price": 7
}
```

19. Notice the inventory-catalog-item-created queue has 1 queued item

20. Start Inventory

21. Notice item is dequeued in RabbitMQ portal, logs and DB

Our Inventory microservice now have a nice way to stay in sync with any updates to the items managed by the Catalog microservice. In the next lesson we will update Inventory so that it starts taking advantage of this new capability any time it needs to present data that is partially coming from Catalog.