Asynchronous communication

(Demo prep)

Using Asynchronous communication

It’s time to update our Catalog microservice so that it starts publishing messages any time an item is created, updated or deleted.

1. dotnet add package MassTransit.AspNetCore

2. dotnet add package MassTransit.RabbitMQ

3. Open ItemsController

4. **Collapse navigation pane**

5. Inject IPublishEndpoint to ItemsController:

```
private readonly IPublishEndpoint publishEndpoint;

public ItemsController(IItemsRepository itemsRepository, IPublishEndpoint publishEndpoint)
{
    this.itemsRepository = itemsRepository;
    this.publishEndpoint = publishEndpoint;
}
```

6. Update PostAsync:

```
public async Task<ActionResult<ItemDto>> PostAsync(CreateItemDto createItemDto)
{
    var item = new Item
    {
        Name = createItemDto.Name,
        Description = createItemDto.Description,
        Price = createItemDto.Price,
        CreatedDate = DateTimeOffset.UtcNow
    };

    await itemsRepository.CreateAsync(item);

    await publishEndpoint.Publish(new CatalogItemCreated(item.Id, item.Name, item.Description));

    return CreatedAtAction(nameof(GetByIdAsync), new { Id = item.Id }, item);
}
```

7. Update PutAsync:

```
public async Task<IActionResult> PutAsync(Guid id, UpdateItemDto updateItemDto)
{
    var existingItem = await itemsRepository.GetAsync(id);

    if (existingItem == null)
    {
        return NotFound();
    }

    existingItem.Name = updateItemDto.Name;
    existingItem.Description = updateItemDto.Description;
    existingItem.Price = updateItemDto.Price;

    await itemsRepository.UpdateAsync(existingItem);

    await publishEndpoint.Publish(new CatalogItemUpdated(existingItem.Id, existingItem.Name,
existingItem.Description));

    return NoContent();
}
```

8. Update DeleteAsync:

```
public async Task<IActionResult> DeleteAsync(Guid id)
{
    var item = await itemsRepository.GetAsync(id);

    if (item == null)
    {
        return NotFound();
    }

    await itemsRepository.RemoveAsync(item.Id);

    await publishEndpoint.Publish(new CatalogItemDeleted(item.Id));

    return NoContent();
}
```

9. Update appsettings.json:

```
"RabbitMQSettings": {
  "Host": "localhost"
},
```

10. Create a Settings dir

11. Add RabbitMQSettings class under Settings dir:

```
public class RabbitMQSettings
{
    public string Host { get; set; }
}
```

12. Open Startup.cs

13. Collapse navigation pane

14. Add MassTransit services to Startup → ConfigureServices:

```
services.AddMassTransit(x =>
{
    x.UsingRabbitMq((context, configurator) =>
    {
        var rabbitMqSettings = Configuration.GetSection(nameof(RabbitMQSettings)).Get<RabbitMQSettings>();
        configurator.Host(rabbitMqSettings.Host);
        configurator.ConfigureEndpoints(context, new
KebabCaseEndpointNameFormatter(serviceSettings.ServiceName, false));
    });
});

services.AddMassTransitHostedService(); // Starts the RabbitMQ bus so messages can be published to the
exchanges
```

15. Update Default log level to **Debug** in **appsettings.Development.json**

At this point the Catalog microservice is ready to publish messages to RabbitMQ but we don't have a RabbitMQ instance available yet. In the next lesson we will expand our Docker compose file to also stand up a RabbitMQ docker container that Catalog can communicate with.