

.NET Microservices

Dr. Ernesto Lee

Sandrilla Thomas

What is a Microservice?

- A *microservice* is a service with one, and only one, very narrowly focused capability that a remote API exposes to the rest of the system.

What is a Microservice - Example

- Receive stock arriving at the warehouse.
- Determine where new stock should be stored.
- Calculate placement routes inside the warehouse for putting stock into the right storage units.
- Assign placement routes to warehouse employees.
- Receive orders.
- Calculate pick routes in the warehouse for a set of orders.
- Assign pick routes to warehouse employees.



Microservices Architecture

Microservices Architecture

- 1. Definition and Scope:** The term "microservices" refers to both individual services and an architectural style for whole systems comprising multiple services.
- 2. Comparison with SOA:** Microservices are a lightweight form of service-oriented architecture (SOA), where each service is highly specialized.
- 3. Distributed Nature:** Microservices architectures result in distributed systems with potentially a large number of collaborating services.
- 4. Growing Popularity:** The adoption of the microservices architectural style is rapidly increasing due to its advantages for server-side software systems.
- 5. Key Benefits:** Microservices are malleable, scalable, and robust, offering advantages over traditional SOA and monolithic architectures.
- 6. Performance Metrics:** Microservices excel in four key metrics:
 1. Deployment frequency
 2. Lead time for changes
 3. Time to restore service
 4. Change failure rate.

Microservices Characteristics

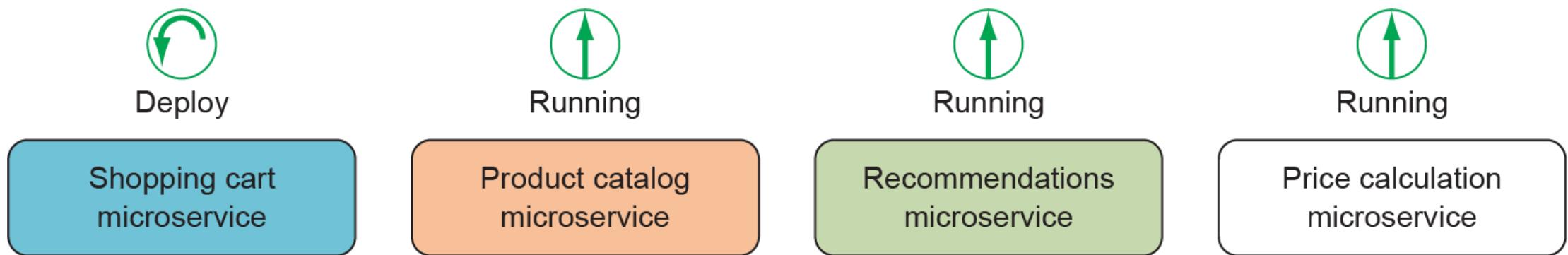
Characteristics of Microservices

- A microservice is responsible for a single capability.
- A microservice is individually deployable.
- A microservice consists of one or more processes.
- A microservice owns its own data store.
- A small team can maintain a few handfuls of microservices.
- A microservice is replaceable.

Responsible for a single capability

- A microservice has a single responsibility.
- That responsibility is for a capability.

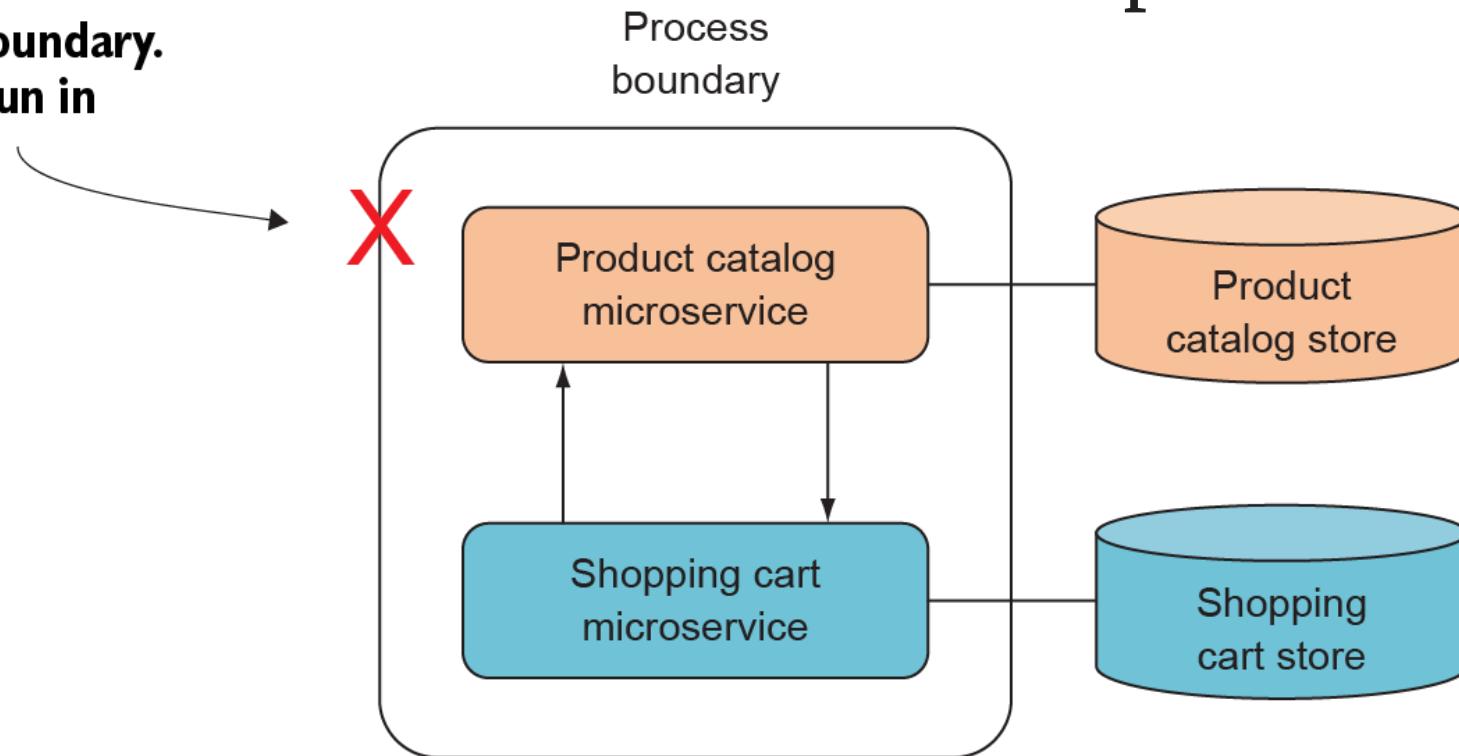
Individually deployable



Consists of one or more processes

- Each microservice must run in separate processes from other microservices.
- Each microservice can have more than one process.

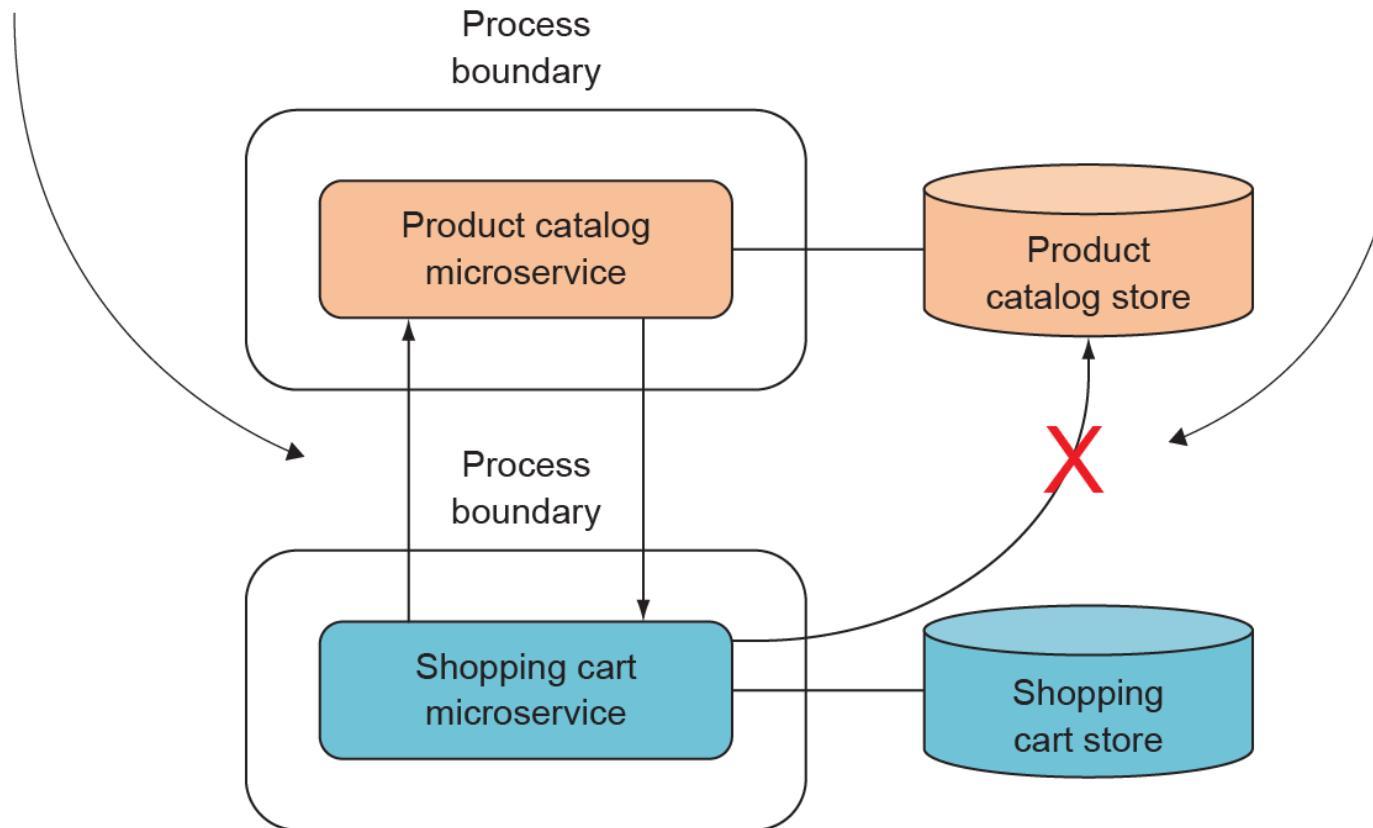
Problematic process boundary.
Microservices should run in separate processes to avoid coupling.



Owns its own data store

All communication with the product catalog microservice must go through the public API.

Direct access to the product catalog store is not allowed. The product catalog microservice owns the product catalog store.



Maintained by a small team

- 1. Size Ambiguity:** The term "micro" in microservices doesn't necessarily refer to the number of lines of code or requirements but rather the complexity of the capability it provides.
- 2. Maintenance Guideline:** A small team (around five members) should be able to maintain "a few handfuls" of microservices, encompassing development, testing, monitoring, and bug fixes. This can range from 10 to 30 microservices, depending on system maturity and automation.
- 3. Team Ownership:** A team should possess a cohesive set of both business and technical capabilities, leading them to own the microservices that represent those functions.

Replaceable

- 1. Replaceability Criterion:** A microservice should be small enough to be rewritten from scratch within a regular workflow, ensuring its size doesn't make it prohibitively expensive or time-consuming to replace.
- 2. Reasons for Rewriting:** Over time, a microservice may become messy, unmaintainable, or underperforming due to evolving requirements, making a complete rewrite more practical than maintenance.
- 3. Informed Rewrites:** If a rewrite is necessary, the team should leverage the knowledge gained from the original implementation and incorporate new requirements to ensure the new microservice is efficient and effective.

Why Microservices

Why Microservices

- Enable continuous delivery
- Allow for an efficient developer workflow because they're highly maintainable
- Are robust by design
- Can scale up or down independently of each other

Enabling continuous delivery

- Can be developed and modified quickly
- Can be comprehensively tested by automated tests
- Can be deployed independently
- Can be operated efficiently

High level of maintainability

- Each well-factored microservice provides *a single capability*. Not two—just one.
- A microservice owns its own data store. No other services can interfere with a microservice’s data store. This, combined with the typical size of the codebase for a microservice, means you can understand a complete service all at once.
- Well-written microservices can (and should) be comprehensively covered by automated tests.

Robust and scalable

- 1. Individual Scalability:** In a microservices architecture, each service can be scaled independently, allowing for targeted scaling based on specific bottlenecks.
- 2. Asynchronous Collaboration:** Microservices prioritize asynchronous, event-based interactions, reducing the dependencies and bottlenecks associated with synchronous communication.
- 3. Fault Tolerance and High Availability:** Emphasizing fault tolerance in synchronous interactions ensures that systems built with microservices are both highly available and scalable.

Costs and downsides of microservices

- 1. Challenges of Distributed Systems:** Microservice architectures, being distributed systems, are inherently more complex, difficult to reason about, and test compared to monolithic systems. Moreover, communication across them is substantially slower.
- 2. Operational Complexity:** With a microservices setup, each service requires its own development, deployment, and management in production, leading to numerous deployments and intricate production configurations.
- 3. Refactoring Difficulties:** Since each microservice has its own codebase, moving code between services is cumbersome, necessitating careful scoping of each microservice from the outset.
- 4. Weighing Costs vs. Complexity:** Before adopting microservices, it's crucial to evaluate if the system's complexity warrants the overhead and challenges associated with a microservices architecture.

Greenfield vs. brownfield

1. Microservices Decision: Deciding between introducing microservices at the start or only for large existing systems is a common dilemma.
2. Monolithic System Evolution: Many systems originate as small applications but evolve into large monolithic structures over time, leading to inherent challenges.
3. Challenges of Monoliths:
 1. High coupling throughout the codebase.
 2. Hidden couplings, like implicit knowledge in code about data formatting or database usage.
 3. Deployment complexities, involving multiple stakeholders and potential downtime.
 4. Over-engineering of simpler components due to a one-size-fits-all architectural approach.
4. Origins of Microservices: The microservices architecture emerged as a solution to the problems faced by monolithic systems.
5. Natural Progression: Breaking down components of a monolith repeatedly can eventually lead to the creation of microservices, allowing for more manageable and modular components.

Code reuse

1. Challenges of Code Reuse:

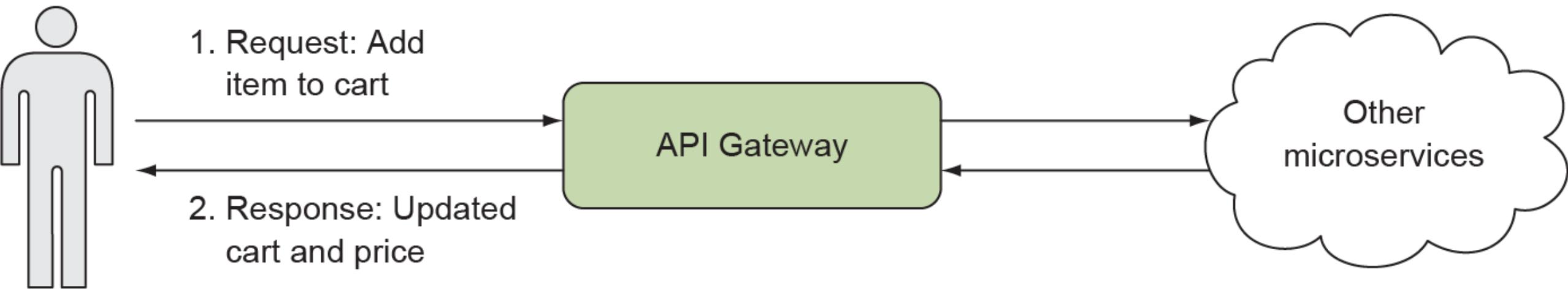
1. Extracting code into a library introduces an additional dependency, complicating service understanding and maintenance.
2. Code in shared libraries must cater to multiple use cases, increasing development effort.
3. Shared libraries can inadvertently introduce harmful coupling between services, leading to complexities during updates and potential errors.
4. Business code should generally not be reused across microservices to avoid unnecessary and detrimental coupling.

2. Business vs. Infrastructure Code: While business code reuse can introduce harmful coupling, there's merit in reusing infrastructure code addressing technical concerns.

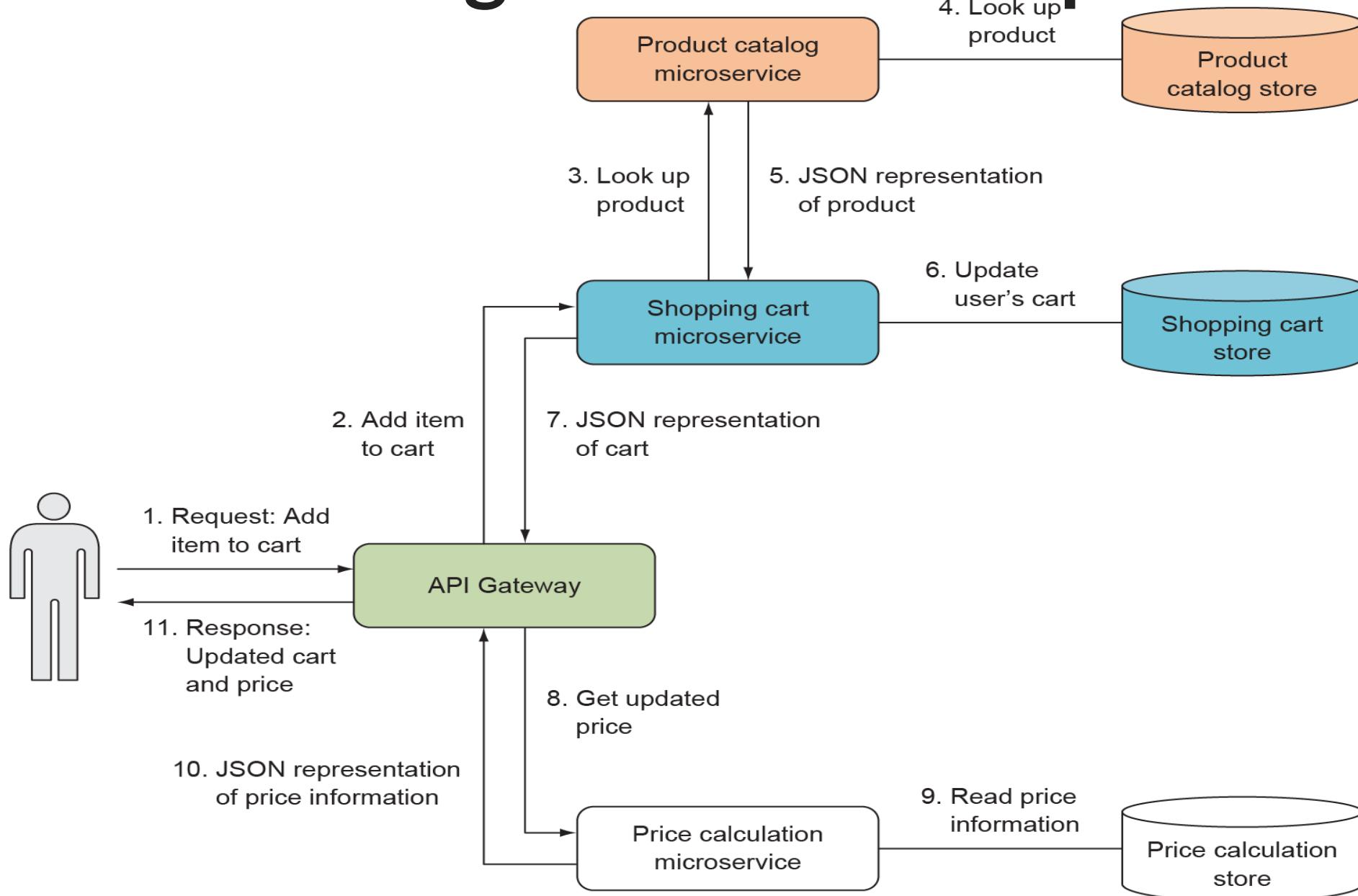
3. Maintaining Service Purity: To ensure a microservice remains focused on its core capability, it's often better to create a new service than to add functionalities to an existing one.

4. Efficient Service Creation: Reusable platforms addressing common technical concerns can streamline the creation of new services, ensuring consistency and reducing the development effort.

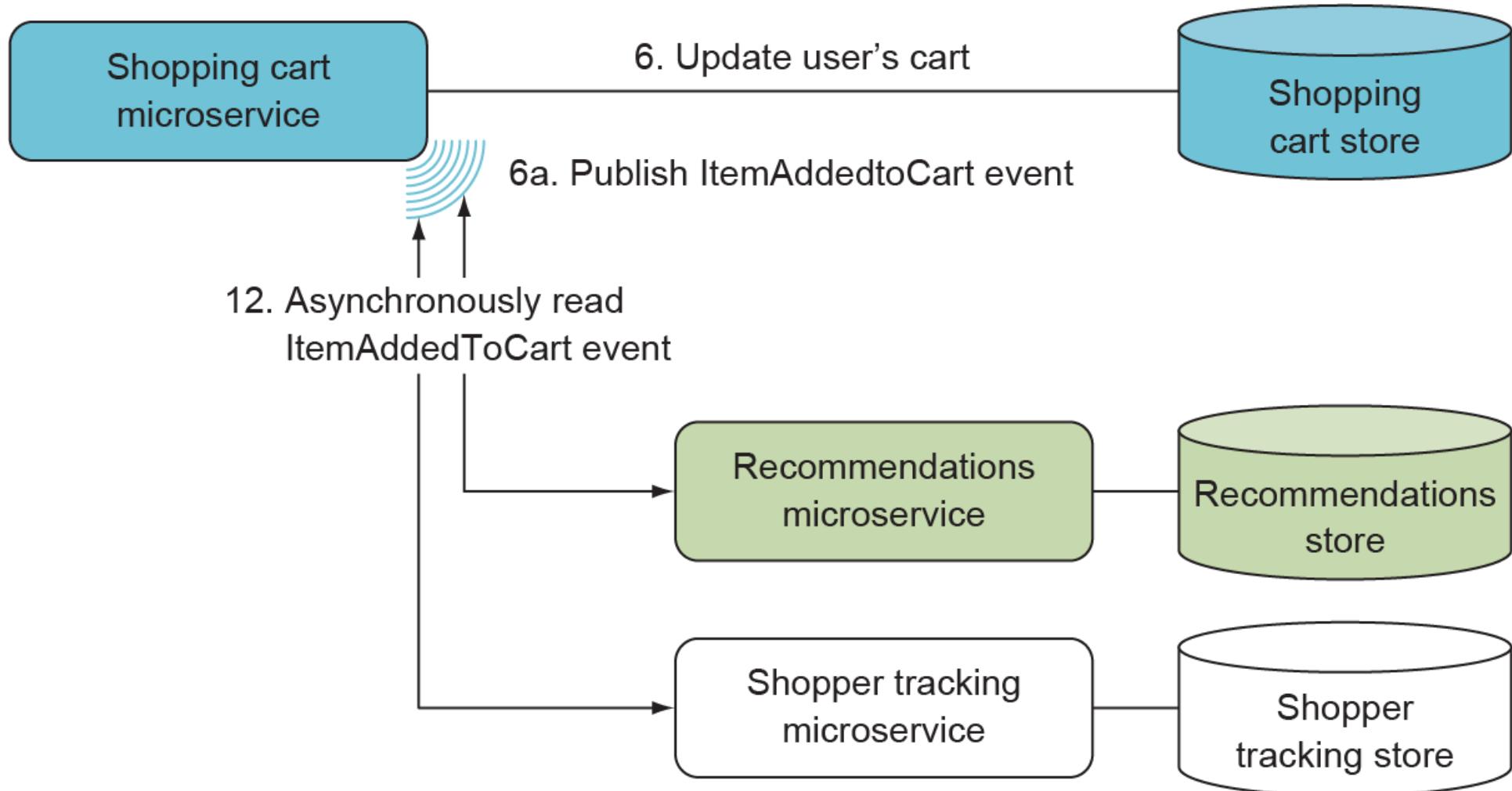
Serving a user request: An example of how microservices work in concert



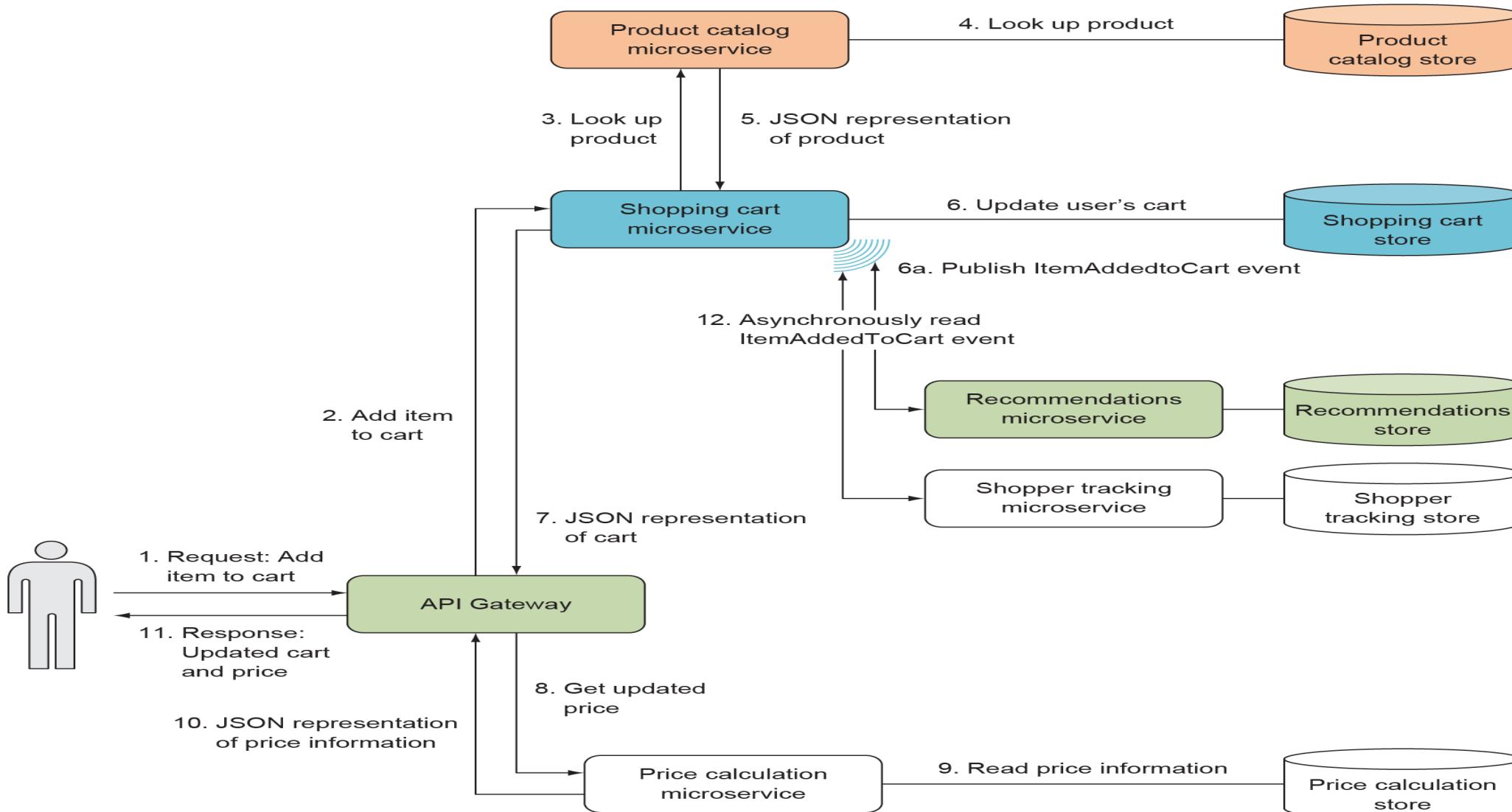
Main handling of the user request



Side effects of the user request

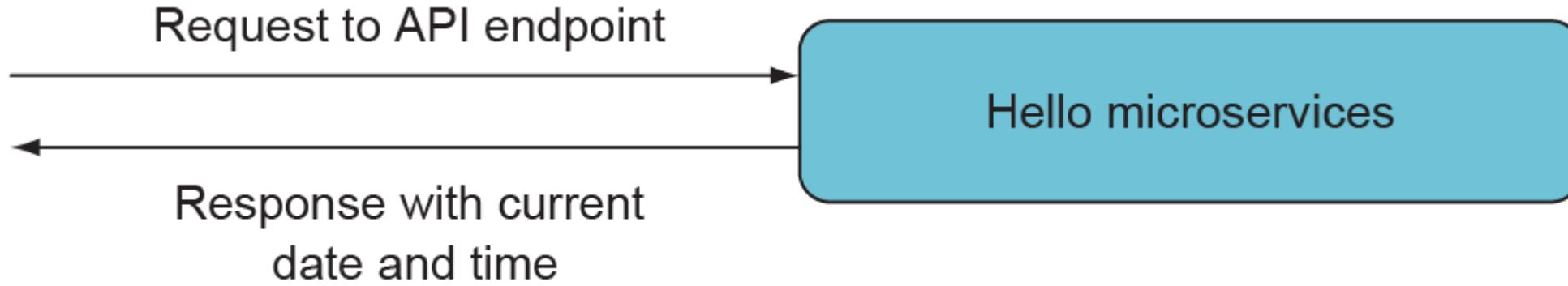


The complete picture



A .NET microservices technology stack

Let's see some code!



To implement this example, you'll follow these three steps:

- 1.Create an empty ASP.NET application.
- 2.Add ASP.NET MVC to the application.
- 3.Add an MVC controller with an implementation of the endpoint.

Summary

1. Definition and Scope:

1. "Microservices" refers to both an architectural style and individual units within a system.
2. It's a specialized form of SOA, focusing on delivering a single business capability.

2. Characteristics of a Microservice:

1. Offers one capability.
2. Independently deployable.
3. Runs in separate processes.
4. Manages its own data.
5. A small team can manage multiple microservices.
6. Can be rewritten quickly if necessary.

3. Relation to Continuous Delivery:

1. Simplifies the continuous delivery process.
2. Facilitates reliable and rapid deployment.

4. Benefits of Microservices:

1. Promotes scalability and resilience.
2. Offers flexibility and adaptability to business needs.
3. Each unit is maintainable and can be quickly developed.
4. Microservices work collaboratively to deliver user functionality.

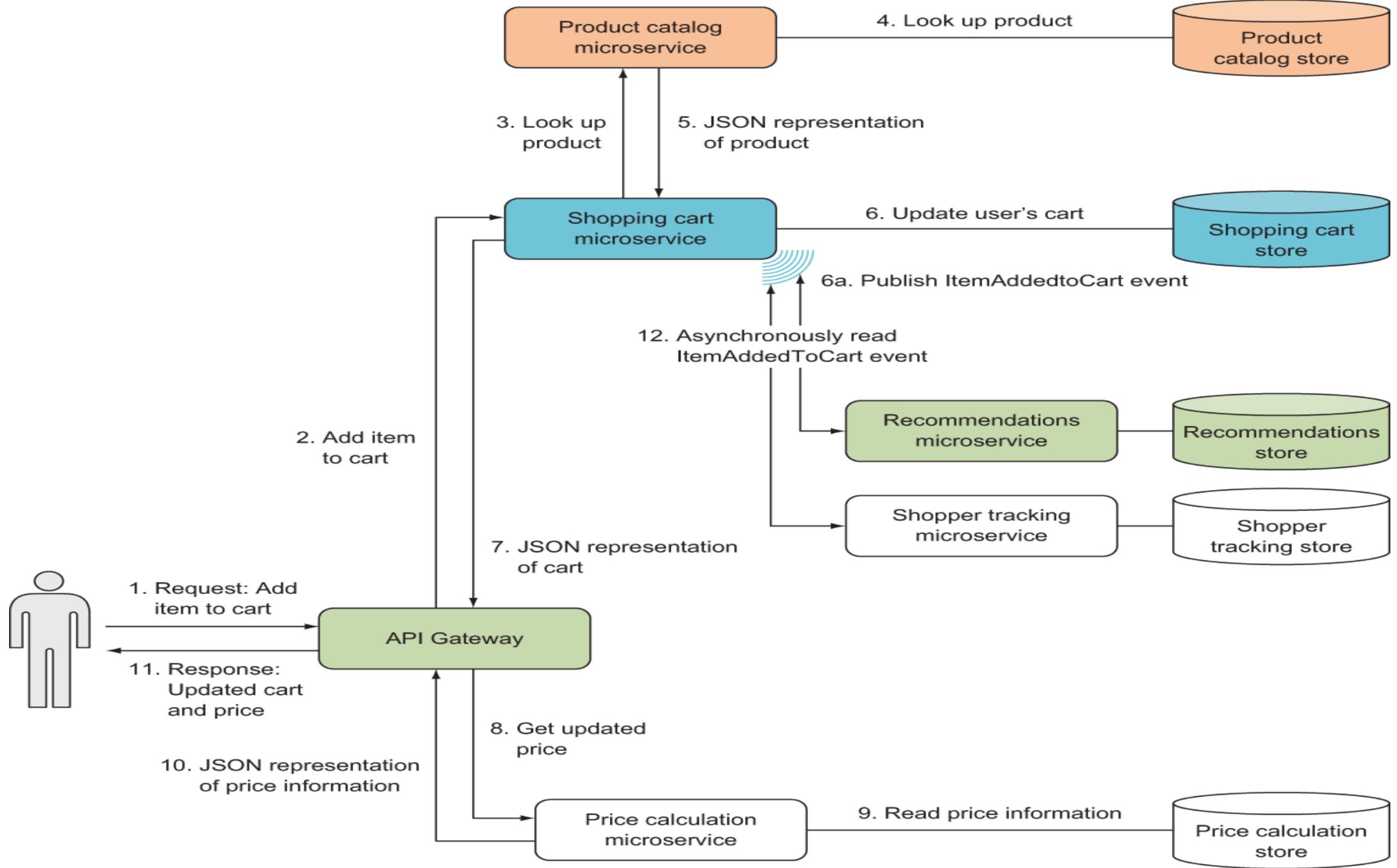
5. Interaction Mechanisms:

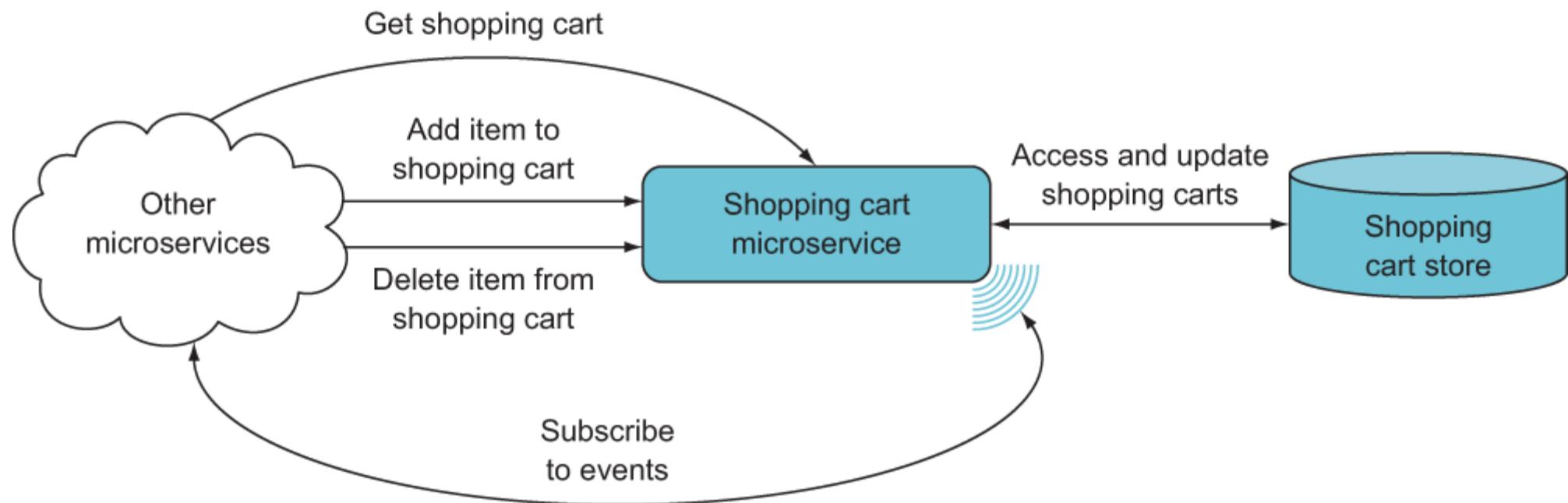
1. Provides a remote public API.
2. Can publish event feeds for other services to subscribe to, ensuring fast asynchronous reactions.

6. Implementation Tools:

1. ASP.NET is a viable platform for creating microservices.
2. Kubernetes is the chosen container orchestrator for the course.
3. Microservices typically serve data (like JSON) instead of HTML. Tools like REST client, Postman, and Fiddler are useful for testing these endpoints.

A Basic Service





HTTP API: Accessible from other microservices

Shopping cart HTTP API

- Endpoint to get a shopping cart
- Endpoint to add items to shopping cart
- Endpoint to delete items from shopping cart

Event feed HTTP API

- Endpoint to get events

Shopping cart domain model

Shopping cart store

`ProductCatalogClient`

`EventStore`

Shopping cart database

Implementing the Shopping Cart microservice

- System.Net.Http.HttpClient
- Polly
- Scrutor

dotnet cli

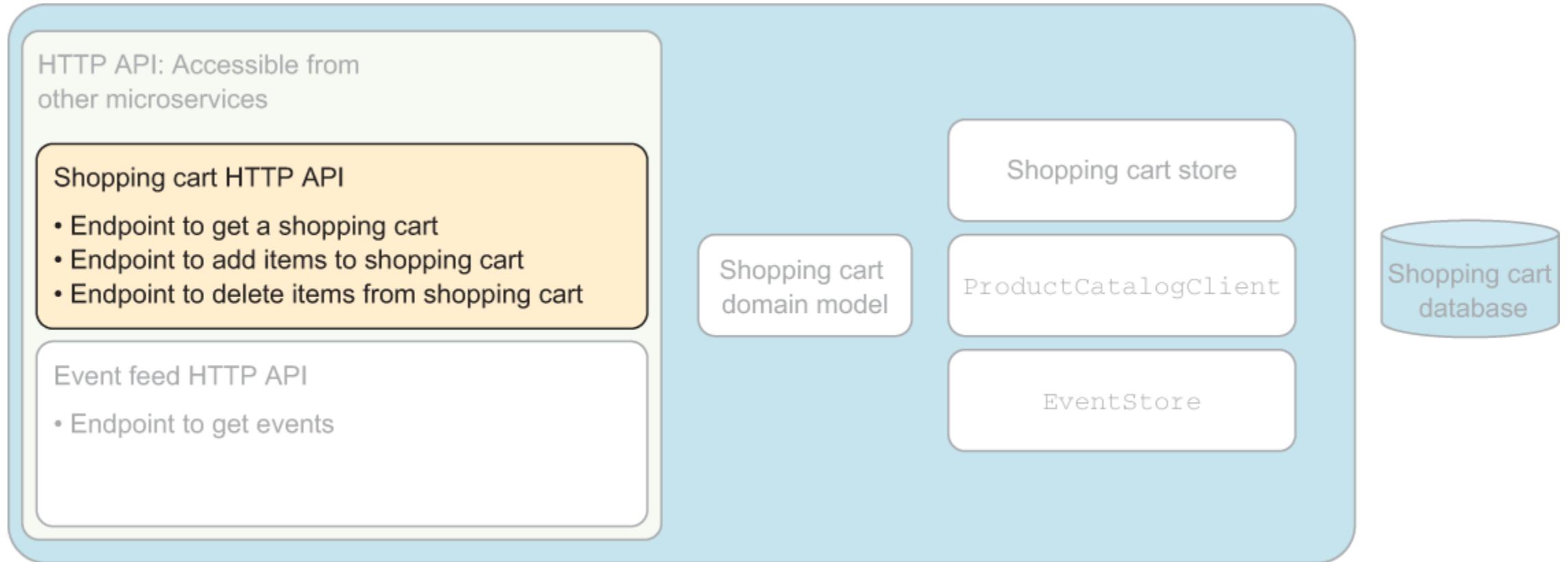
- dotnet new web –n ShoppingCart

Create an Empty Project

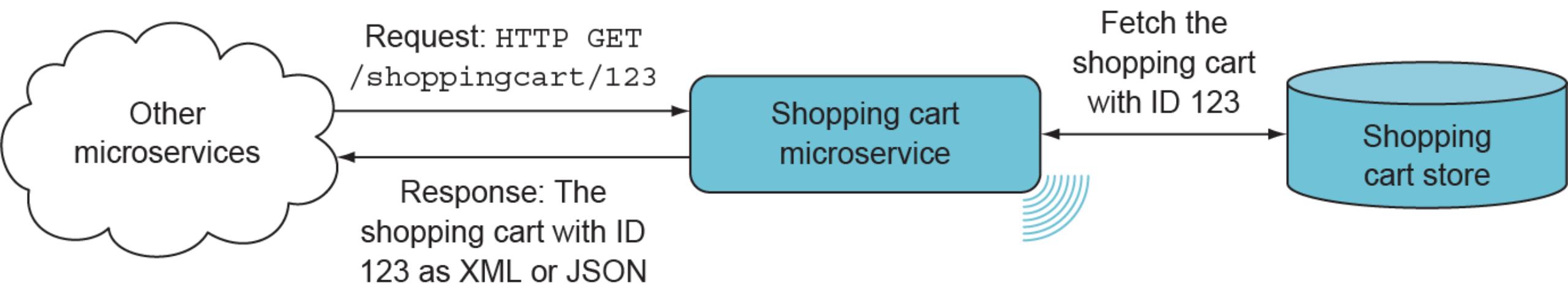
```
namespace ShoppingCart
```

```
{  
    using Microsoft.AspNetCore.Builder;  
    using Microsoft.Extensions.DependencyInjection;  
  
    public class Startup  
    {  
        public void ConfigureServices(IServiceCollection services)  
        {  
            services.AddControllers();  
        }  
  
        public void Configure(IApplicationBuilder app)  
        {  
            app.UseHttpsRedirection();  
            app.UseRouting();  
            app.UseEndpoints(endpoints =>  
                endpoints.MapControllers());  
        }  
    }  
}
```

The Shopping Cart microservice's API for other services



Getting a shopping cart



Getting a shopping cart

For example, the API Gateway may need the shopping cart for a user with ID 123. To get that, it sends this HTTP request:

HTTP GET /shoppingcart/123 HTTP/1.1

Host: shoppingcart.my.company.com

Accept: application/json

ShoppingCart

| appsettings.Development.json

| appsettings.json

| Program.cs

| ShoppingCart.csproj

| Startup.cs

|

└─ ShoppingCart

 ShoppingCart.cs

 ShoppingCartController.cs

 ShoppingCartStore.cs

- The ShoppingCart domain model is simple and looks like this:

```

namespace ShoppingCart.ShoppingCart
{
    using System.Collections.Generic;
    using System.Linq;

    public class ShoppingCart
    {
        private readonly HashSet<ShoppingCartItem> items = new();

        public int UserId { get; }
        public IEnumerable<ShoppingCartItem> Items => this.items;

        public ShoppingCart(int userId) => this.UserId = userId;

        public void AddItems(IEnumerable<ShoppingCartItem> shoppingCartItems)
        {
            foreach (var item in shoppingCartItems)
                this.items.Add(item);
        }

        public void RemoveItems(int[] productCatalogueIds) =>
            this.items.RemoveWhere(i => productCatalogueIds.Contains(
                i.ProductCatalogueId));
    }

    public record ShoppingCartItem(
        int ProductCatalogueId,
        string ProductName,
        string Description,
        Money Price)
    {
        public virtual bool Equals(ShoppingCartItem? obj) =>
            obj != null && this.ProductCatalogueId.Equals(obj.ProductCatalogueId);

        public override int GetHashCode() =>
            this.ProductCatalogueId.GetHashCode();
    }

    public record Money(string Currency, decimal Amount);
}

```

```
namespace ShoppingCart.Shoppingcart
{
    using Microsoft.AspNetCore.Mvc;
    using ShoppingCart;

    [Route("/shoppingcart")]
    public class ShoppingCartController : ControllerBase
    {
        private readonly IShoppingCartStore shoppingCartStore;

        public ShoppingCartController(IShoppingCartStore shoppingCartStore)
        {
            this.shoppingCartStore = shoppingCartStore;
        }

        [HttpGet("{userId:int}")]
        public ShoppingCart Get(int userId) =>
            this.shoppingCartStore.Get(userId);
    }
}
```

- The `HttpGet` attribute is followed by a method:

```
public ShoppingCart Get(int userId) =>  
    this.shoppingCartStore.Get(userId);
```

- The action method uses a `shoppingCartStore` object that the `ShoppingCartController` constructor takes as an argument and assigns to an instance variable:

```
public ShoppingCartController(IShoppingCartStore shoppingCartStore)  
{  
    this.shoppingCartStore = shoppingCartStore;  
}
```

First, we add Scrutor to the shopping cart project by running this dotnet command:

```
PS> dotnet add package scrutor
```

- This installs the Scrutor NuGet package, which we can see in the shopping cart project file—ShoppingCart.csproj—where there now is a package reference to Scrutor:

```
<Project Sdk="Microsoft.NET.Sdk.Web">

<PropertyGroup>
  <TargetFramework>netcoreapp3.0</TargetFramework>
</PropertyGroup>

<ItemGroup>
  <PackageReference Include="Scrutor" Version="3.1.0" />
</ItemGroup>

</Project>
```

- With Scrutor installed, we can add code to the ConfigureServices method in the Startup class, so it becomes

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllers();
    services.Scan(selector =>
        selector
            .FromAssemblyOf<Startup>()
            .AddClasses()
            .AsImplementedInterfaces());
}
```

```
public interface IShoppingCartStore
{
    ShoppingCart Get(int userId);
    void Save(ShoppingCart shoppingCart);
}

public class ShoppingCartStore : IShoppingCartStore
{
    private static readonly Dictionary<int, ShoppingCart>
        Database = new Dictionary<int, ShoppingCart>();

    public ShoppingCart Get(int userId) =>
        Database.ContainsKey(userId)
            ? Database[userId]
            : new ShoppingCart(userId);

    public void Save(ShoppingCart shoppingCart) =>
        Database[shoppingCart.UserId] = shoppingCart;
}
```

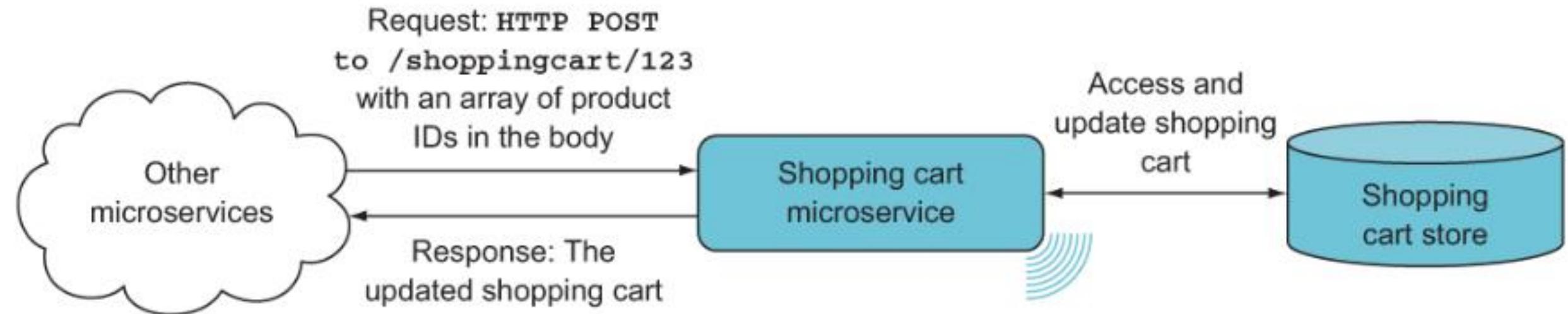
- Returning attention to the ShoppingCartController, the action method Get returns a ShoppingCart object that it gets back from shoppingCartStore:

```
this.shoppingCartStore.Get(userId);
```

HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8

539

```
{  
    "userId": 42,  
    "items": [  
        {  
            "productcatalogId": 1,  
            "productName": "Basic t-shirt",  
            "description": "a quiet t-shirt",  
            "price": {  
                "currency": "eur",  
                "amount": 40  
            }  
        },  
        {  
            "productcatalogId": 2,  
            "productName": "Fancy shirt",  
            "description": "a loud t-shirt",  
            "price": {  
                "currency": "eur",  
                "amount": 50  
            }  
        }  
    ]  
}
```



Adding two items to a shopping cart

POST /shoppingcart/123/items HTTP/1.1

Host: shoppingcart.my.company.com

Accept: application/json

Content-Type: application/json

[1, 2]

```
[Route("/shoppingcart")]
public class ShoppingCartController : Controller
{
    private readonly IShoppingCartStore shoppingCartStore;
    private readonly IProductCatalogClient productCatalogClient;
    private readonly IEventStore eventStore;

    public ShoppingCartController(
        IShoppingCartStore shoppingCartStore,
        IProductCatalogClient productCatalogClient,
        IEventStore eventStore)
    {
        this.shoppingCartStore = shoppingCartStore;
        this.productCatalogClient = productCatalogClient;
        this.eventStore = eventStore;
    }

    [HttpGet("{userId:int}")]
    public ShoppingCart Get(int userId) =>
        this.shoppingCartStore.Get(userId);

    [HttpPost("{userId:int}/items")]
    public async Task<ShoppingCart> Post(
        int userId,
        [FromBody] int[] productIds)
    {
        var shoppingCart = shoppingCartStore.Get(userId);
        var shoppingCartItems =
            await this.productcatalogClient
                .GetShoppingCartItems(productIds);
        shoppingCart.AddItems(shoppingCartItems, eventStore);
        shoppingCartStore.Save(shoppingCart);
        return shoppingCart;
    }
}
```

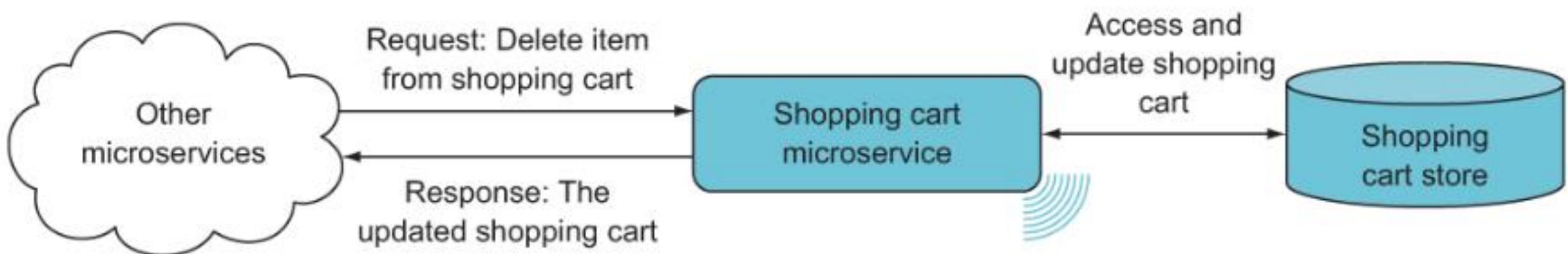
- Second, the body of the request contains a JSON array of product IDs. These are the items that should be added to the shopping cart. The action uses MVC model binding to read these into a C# array:

```
[FromBody] int[] productIds
```

- You once again rely on ASP.NET to provide them through constructor arguments.

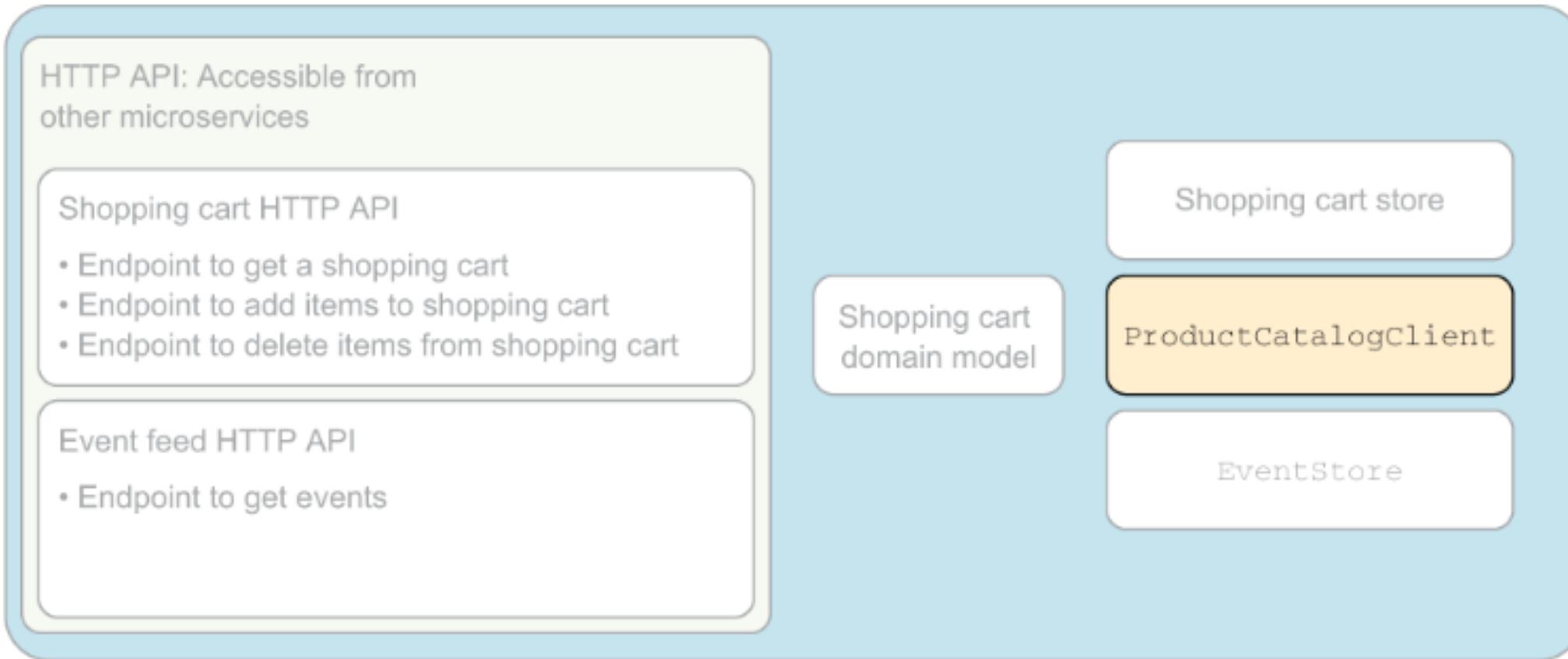
```
public ShoppingCartController(  
    IShoppingCartStore shoppingCartStore,  
    IProductCatalogClient productCatalogClient,  
    IEventStore eventStore)
```

- The remainder of the method is queued up for execution when the awaitable—in this case, the Task returned from `Task.Delay(1000)`—completes. When the awaitable completes, the rest of the method is executed, possibly on a new thread but with same state as before the await re-established.
- The current thread of execution returns from the `async` method and continues in the caller.



```
[HttpDelete("{userid:int}/items")]
public ShoppingCart Delete(
    int userId,
    [FromBody] int[] productIds)
{
    var shoppingCart =
        this.shoppingCartStore.Get(userId);
    shoppingCart.RemoveItems(
        productIds,
        this.eventStore);
    this.shoppingCartStore.Save(shoppingCart);
    return shoppingCart;
}
```

Fetching product information



You need to follow these three steps to implement the HTTP request to the product catalog microservice:

- Implement the HTTP GET request.
- Parse the response from the endpoint at the product catalog microservice and translate it to the domain of the shopping cart microservice.
- Implement a policy for handling failed requests to the product catalog microservice.

Implementing the HTTP GET

- For example, the following request fetches the information for product IDs 1 and 2:

HTTP GET /products?productIds=[1,2] HTTP/1.1

Host: productcatalog.my.company.com

Accept: application/json

```
public class ProductCatalogClient : IProductCatalogClient
{
    private readonly HttpClient client;
    private static string productCatalogBaseUrl =
        @"https://git.io/JeHiE";
    private static string getProductPathTemplate = "?productIds=[{0}]";

    public ProductCatalogClient(HttpClient client)
    {
        client.BaseAddress =
            new Uri(productCatalogBaseUrl);
        client
            .DefaultRequestHeaders
            .Accept
            .Add(new MediaTypeWithQualityHeaderValue("application/json"));
        this.client = client;
    }

    public async Task<IEnumerable<ShoppingCartItem>>
        GetShoppingCartItems(int[] productCatalogIds)
    {
        ....
    }

    private async Task<HttpResponseMessage>
        RequestProductFromProductCatalog(int[] productCatalogIds)
    {
        var productsResource =
            string.Format(getProductPathTemplate,
                string.Join(",", productCatalogIds));
        return await
            this.client.GetAsync(productsResource);
    }
}
```

- For ASP.NET to be able to inject the HttpClient in ProductCatalogClient we need to register ProductCatalogClient as typed http client, which we do by adding this to the ConfigureServices method in Startup:

```
services.AddHttpClient<IProductCatalogClient, ProductCatalogClient>();
```

Parsing the product response

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8

543
[
  {
    "productId": "1",
    "productName": "Basic t-shirt",
    "productDescription": "a quiet t-shirt",
    "price": { "amount" : 40, "currency": "eur" },
    "attributes" : [
      {
        "sizes": [ "s", "m", "l" ],
        "colors": [ "red", "blue", "green" ]
      }
    ],
    {
      "productId": "2",
      "productName": "Fancy shirt",
      "productDescription": "a loud t-shirt",
      "price": { "amount" : 50, "currency": "eur" },
      "attributes" : [
        {
          "sizes": [ "s", "m", "l", "xl" ],
          "colors": [ "ALL", "Batique" ]
        }
      ]
    }
  ]
]
```

```
private static async Task<IEnumerable<ShoppingCartItem>>
    ConvertToShoppingCartItems(HttpResponseMessage response)
{
    response.EnsureSuccessStatusCode();
    var products = await
        JsonSerializer.DeserializeAsync<List<ProductCatalogProduct>>(
            await response.Content.ReadAsStreamAsync(),
            new JsonSerializerOptions
            {
                PropertyNameCaseInsensitive = true
            }) ?? new();
    return products
        .Select(p =>
            new ShoppingCartItem(
                p.ProductId,
                p.ProductName,
                p.ProductDescription,
                p.Price
            ));
}

private record ProductCatalogProduct(
    int ProductId,
    string ProductName,
    string ProductDescription,
    Money Price);
```

The following listing combines the code that requests the product information and the code that parses the response.

```
public async Task<IEnumerable<ShoppingCartItem>>
    GetShoppingCartItems(int[] productCatalogIds)
{
    using var response =
        await RequestProductFromProductCatalogue(productCatalogIds);
    return await ConvertToShoppingCartItems(response);
}
```

Adding a failure-handling policy

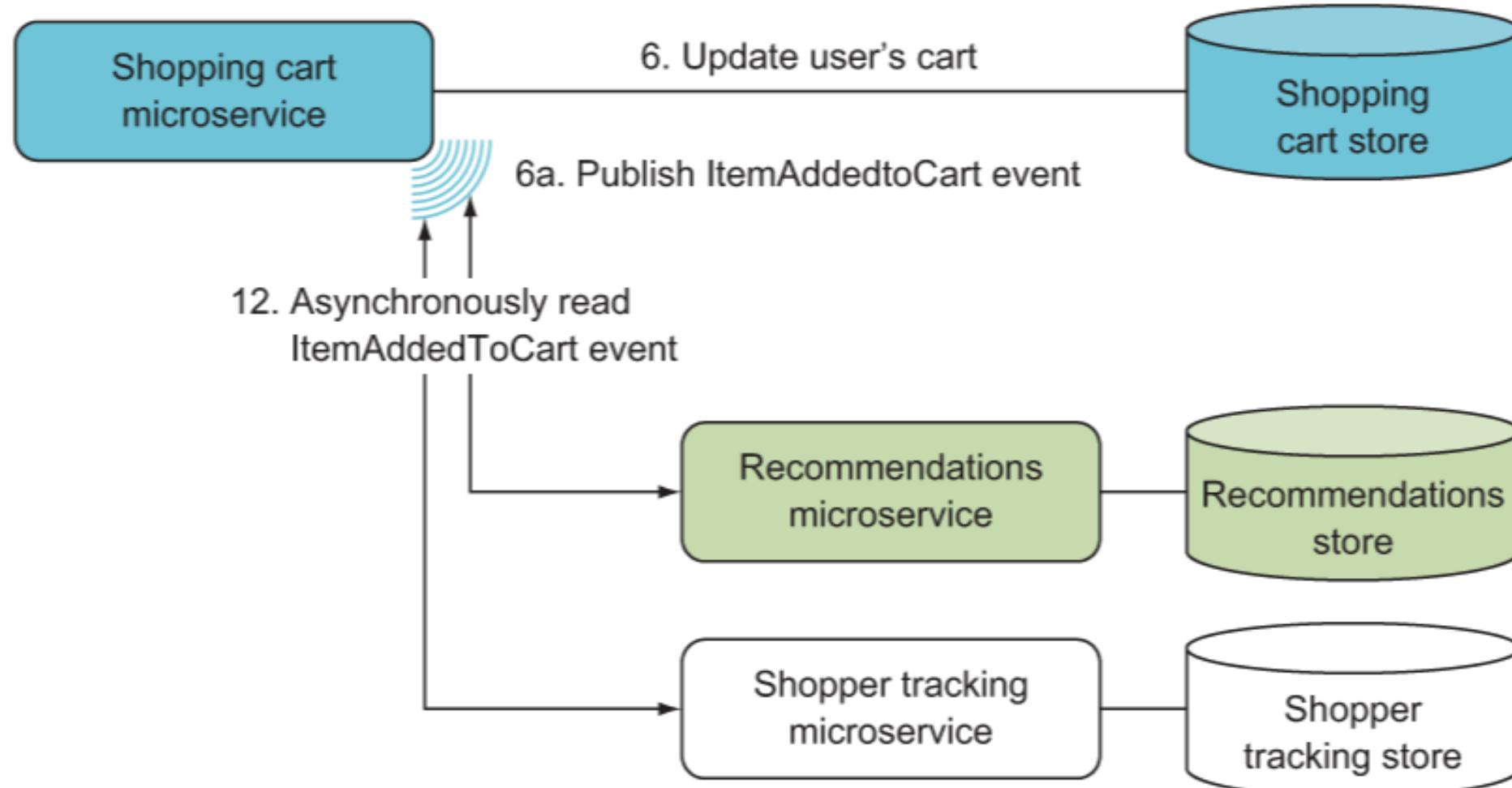
Caching product information has some significant advantages:

- It makes the shopping cart more resilient to failures in product catalog.
- The shopping cart microservice will perform better when the product information is present in the cache.
- Fewer calls made from the shopping cart microservice mean less stress is put on the product catalog microservice.

- As you can see in the following listing, Polly's API and integration with ASP.NET makes both these steps easy. Replace the current registration of ProductCatalogClient in Startup with this:

```
services.AddHttpClient<IProductCatalogClient, ProductCatalogClient>()  
    .AddTransientHttpErrorPolicy(p =>  
        p.WaitAndRetryAsync(  
            3,  
            attempt => TimeSpan.FromMilliseconds(100 * Math.Pow(2, attempt))));
```

Implementing a basic event feed



HTTP API: Accessible from other microservices

Shopping cart HTTP API

- Endpoint to get a shopping cart
- Endpoint to add items to shopping cart
- Endpoint to delete items from shopping cart

Event feed HTTP API

- Endpoint to get events

Shopping cart domain model

Shopping cart store

ProductCatalogClient

EventStore

Shopping cart database

Implementing the event feed involves these steps:

- *Raise events.*
- *Store events.*
- *Publish events.*

- The first file contains the model type for events, shown in the following listing.

```
namespace ShoppingCart.EventFeed
```

```
{
```

```
using System;
```

```
public record Event(
```

```
    long SequenceNumber,
```

```
    DateTimeOffset OccuredAt,
```

```
    string Name,
```

```
    object Content);
```

```
}
```

Using this interface, the ShoppingCart domain object can raise events, as shown next.

```
public void AddItems(  
    IEnumerable<ShoppingCartItem> shoppingCartItems,  
    IEventStore eventStore)  
{  
    foreach (var item in shoppingCartItems)  
        if (this.items.Add(item))  
            eventStore.Raise(  
                "ShoppingCartItemAdded",  
                new { UserId, item });  
}
```

Storing an event

```
public void Raise(string eventName, object content)
{
    var seqNumber = database.NextSequenceNumber();
    database.Add(
        new Event(
            seqNumber,
            DateTimeOffset.UtcNow,
            eventName,
            content));
}
```

A simple event feed

- A subscriber can, for example, issue the following request to get all events newer than event number 100:

GET /events?start=100 HTTP/1.1

Host: shoppingcart.my.company.com

Accept: application/json

- Or, if the subscriber wants to limit the number of incoming events per call, it can add an end argument to the request:

GET /events?start=100&end=200 HTTP/1.1

Host: shoppingcart.my.company.com

Accept: application/json

```
namespace ShoppingCart.EventFeed
{
    using System.Linq;
    using Microsoft.AspNetCore.Mvc;

    [Route("/events")]
    public class EventFeedController : Controller
    {
        private readonly IEventStore eventStore;

        public EventFeedController(IEventStore eventStore) =>
            this.eventStore = eventStore;

        [HttpGet("")]
        public Event[] Get(
            [FromQuery] long start,
            [FromQuery] long end = long.MaxValue)
        =>
            this.eventStore
                .GetEvents(start, end)
                .ToArray();
    }
}
```

- The following simple implementation illustrates it well.

```
public IEnumerable<Event> GetEvents(  
    long firstEventSequenceNumber,  
    long lastEventSequenceNumber) =>  
    database  
        .Where(e =>  
            e.SequenceNumber >= firstEventSequenceNumber &&  
            e.SequenceNumber <= lastEventSequenceNumber)  
        .OrderBy(e => e.SequenceNumber);
```

Running the code

- Now that all the code for the shopping cart microservice is in place, you can run it.
- You can test all the endpoints with RestClient or a similar tool.

Summary

- Implementing a complete microservice doesn't take much code.
- The Polly library is useful for implementing failure-handling policies and wrapping them around remote calls.
- You should always expect that other microservices may be down. To prevent errors from propagating, each remote call should be wrapped in a policy for handling failure.

Deploying a microservice to Kubernetes

Introduction

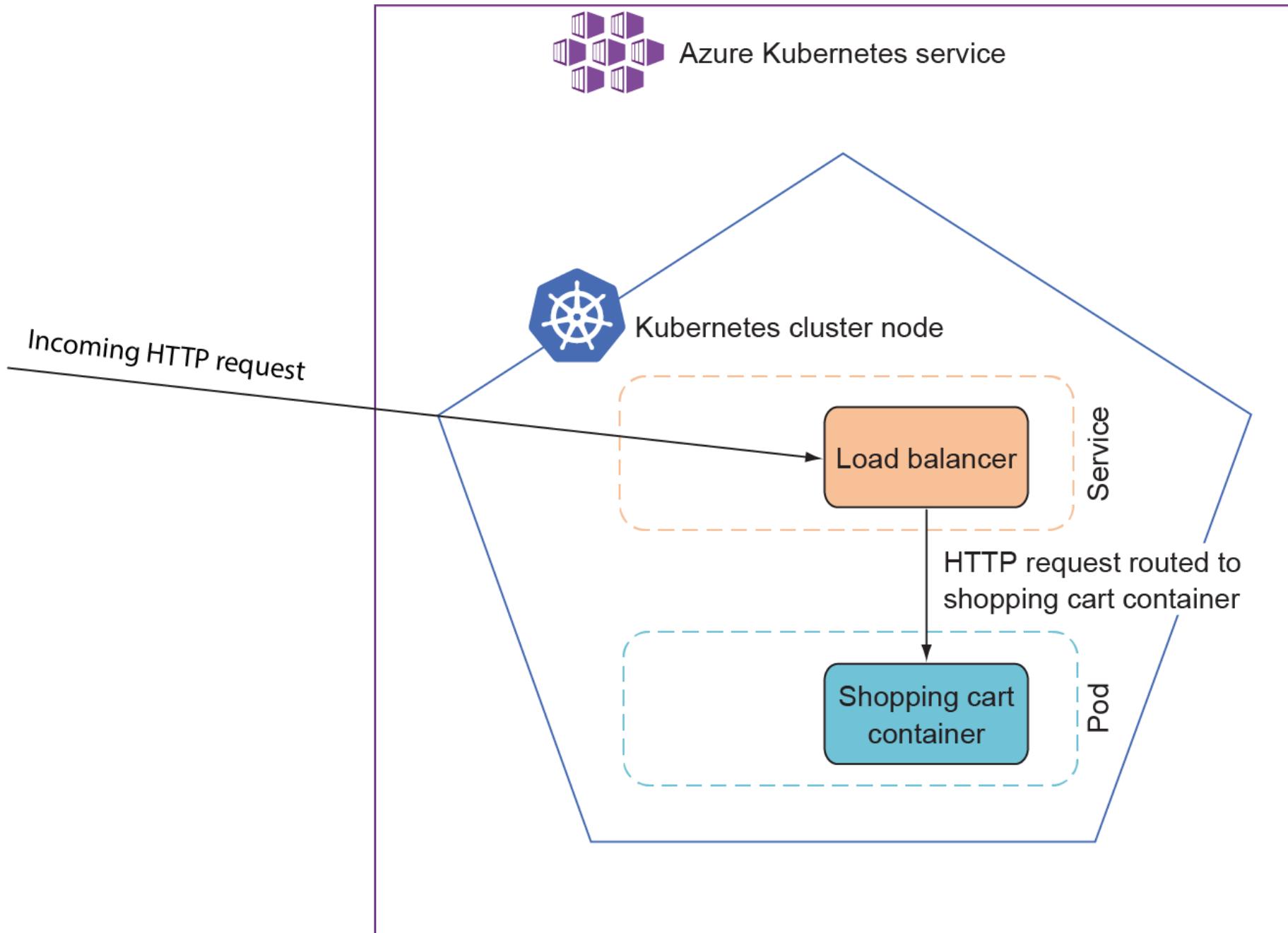
This Module covers:

- Packaging a microservice in a Docker container
- Deploying a microservice container to Kubernetes on localhost
- Creating a basic Kubernetes cluster on Azure's AKS (Azure Kubernetes Service)
- Deploying a microservice container to a Kubernetes cluster on AKS

Choosing a production environment

Simply running our microservices on localhost isn't very interesting. We need them to run somewhere our end users can get to them and use them. There are a number of options for doing that:

- Running the microservices on your own Windows or Linux servers on-premise.
- Using a Platform as a Service (PaaS) cloud option that supports .
- Putting microservices into containers and deploying them to a cloud-specific container service like Azure's ACS or Amazon's ECS Platform as a Service (PaaS) cloud option that supports
- Using cloud-agnostic container orchestrators like Kubernetes, Apache Mesos, or RedHat OpenShift.



Putting the Shopping Cart microservice in a container

Adding a Dockerfile to the Shopping Cart microservice

```
FROM mcr.microsoft.com/dotnet/core/sdk:3.1 AS build  
  
WORKDIR /src  
  
COPY ["ShoppingCart/ShoppingCart.csproj", "ShoppingCart/"]  
  
RUN dotnet restore "ShoppingCart/ShoppingCart.csproj"  
  
COPY ..  
  
WORKDIR "/src/ShoppingCart"  
  
RUN dotnet build "ShoppingCart.csproj" -c Release -o /app/build
```

```
FROM build AS publish  
  
RUN dotnet publish "ShoppingCart.csproj" -c Release -o /app/publish
```

```
FROM mcr.microsoft.com/dotnet/core/aspnet:3.1 AS final  
  
WORKDIR /app  
  
EXPOSE 80  
  
COPY --from=publish /app/publish .  
  
ENTRYPOINT ["dotnet", "ShoppingCart.dll"]
```

Steps in the Dockerfile

The steps in the Dockerfile are as follows:

- Build the shopping cart code
- Publish the shopping cart microservice
- Create a container image based on ASP.NET

Adding a Dockerfile to the Shopping Cart microservice

- To make sure the Dockerfile runs a clean build, we add a `.dockerignore` with these lines that make sure any bin and obj folders are not copied into the container:

[B|b]in/

[O|o]bj/

Building and running the shopping cart container

- The next step is to build a shopping cart container image from the Dockerfile we just added.
- First, make sure you have Docker running and then open a command line and go to the root of shopping cart—where the Dockerfile is.
- Then issue this Docker command:

```
docker build . -t shopping-cart
```

Building and running the shopping cart container

- Subsequent builds will be faster.
- The output of the docker build is rather long.
- When it is successful the last few lines of the output are similar to

Successfully built 8d448ba53088

Successfully tagged shopping-cart:latest

Building and running the shopping cart container

- Possibly followed by this warning if you are on Windows:

SECURITY WARNING: You are building a Docker image from Windows against a non-Windows Docker host. All files and directories added to build context will have '`-rwxr-xr-x`' permissions. It is recommended to double-check and reset permissions for sensitive files and directories.

Building and running the shopping cart container

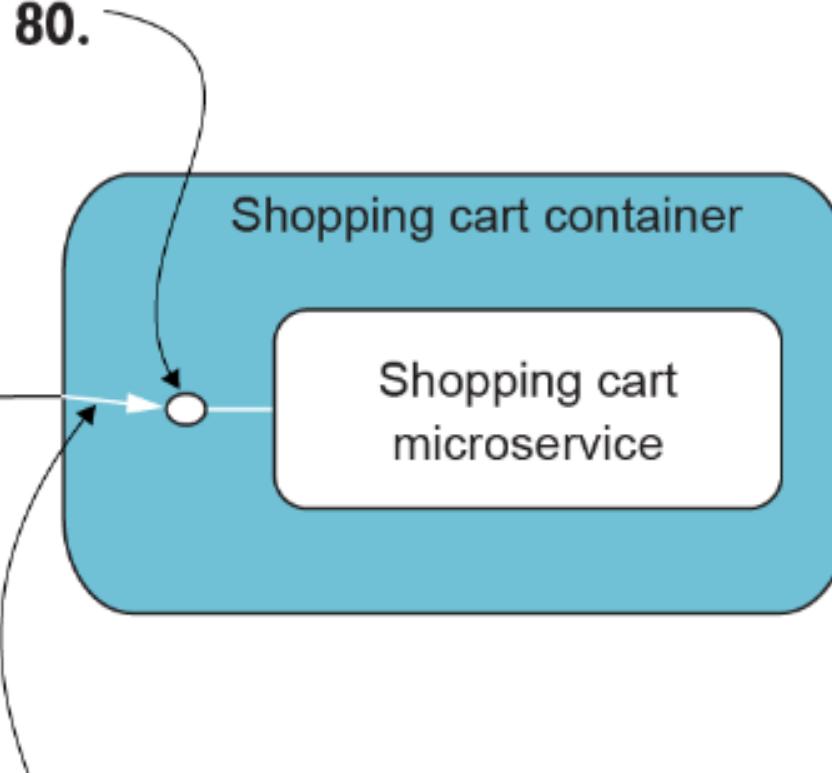
- You are now ready to run the newly built container image with this command:

```
> docker run --name shopping-cart --rm -p 5000:80 shopping-cart
```

**The shopping cart microservice
inside the container listens on
port 80.**

**The container listens
on port 5000.**

**The container forwards traffic to
port 5000 and then to the shopping
cart microservice on port 80.**

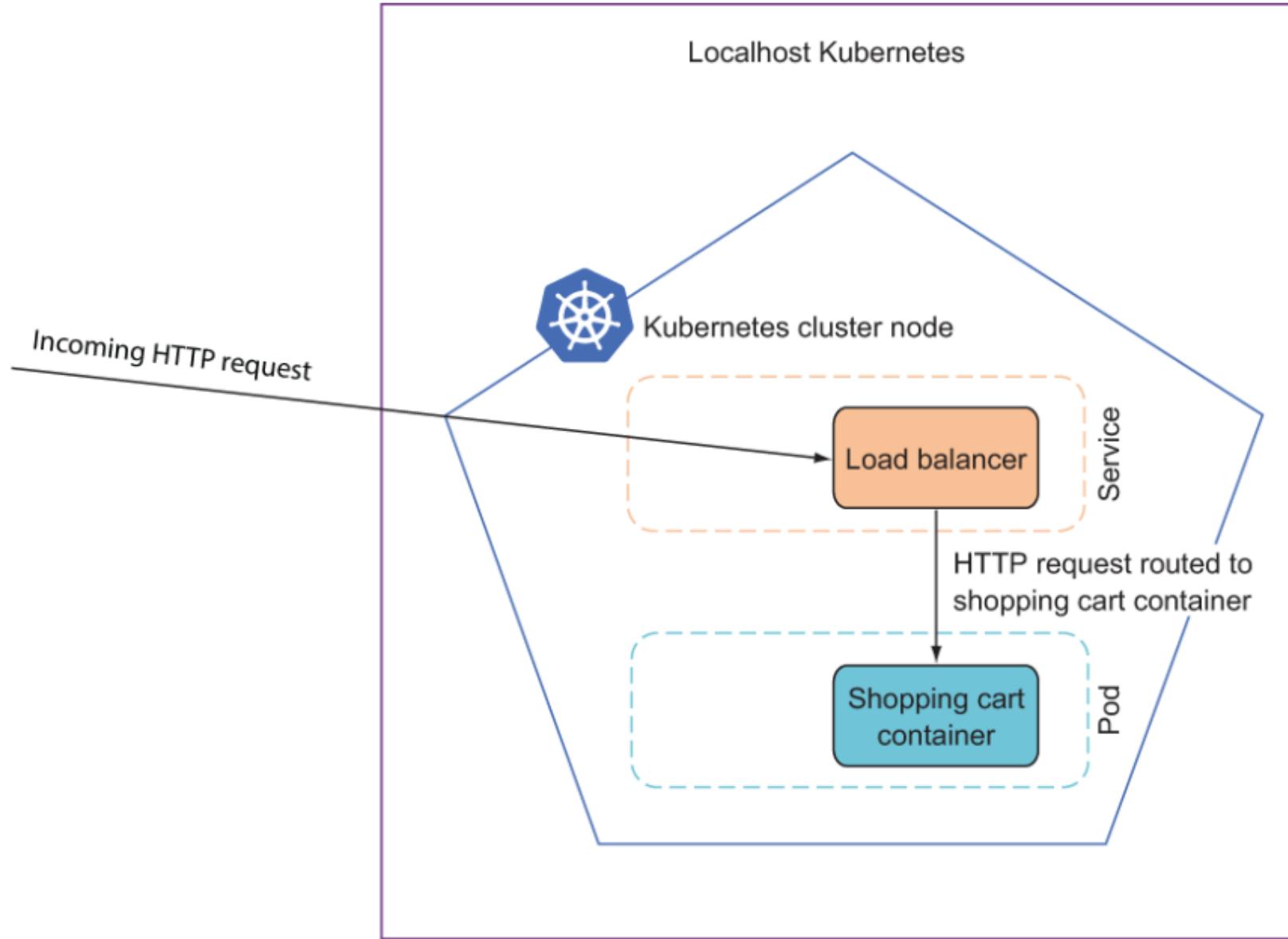


Building and running the shopping cart container

- To stop the shopping cart container, you can use the docker stop command:

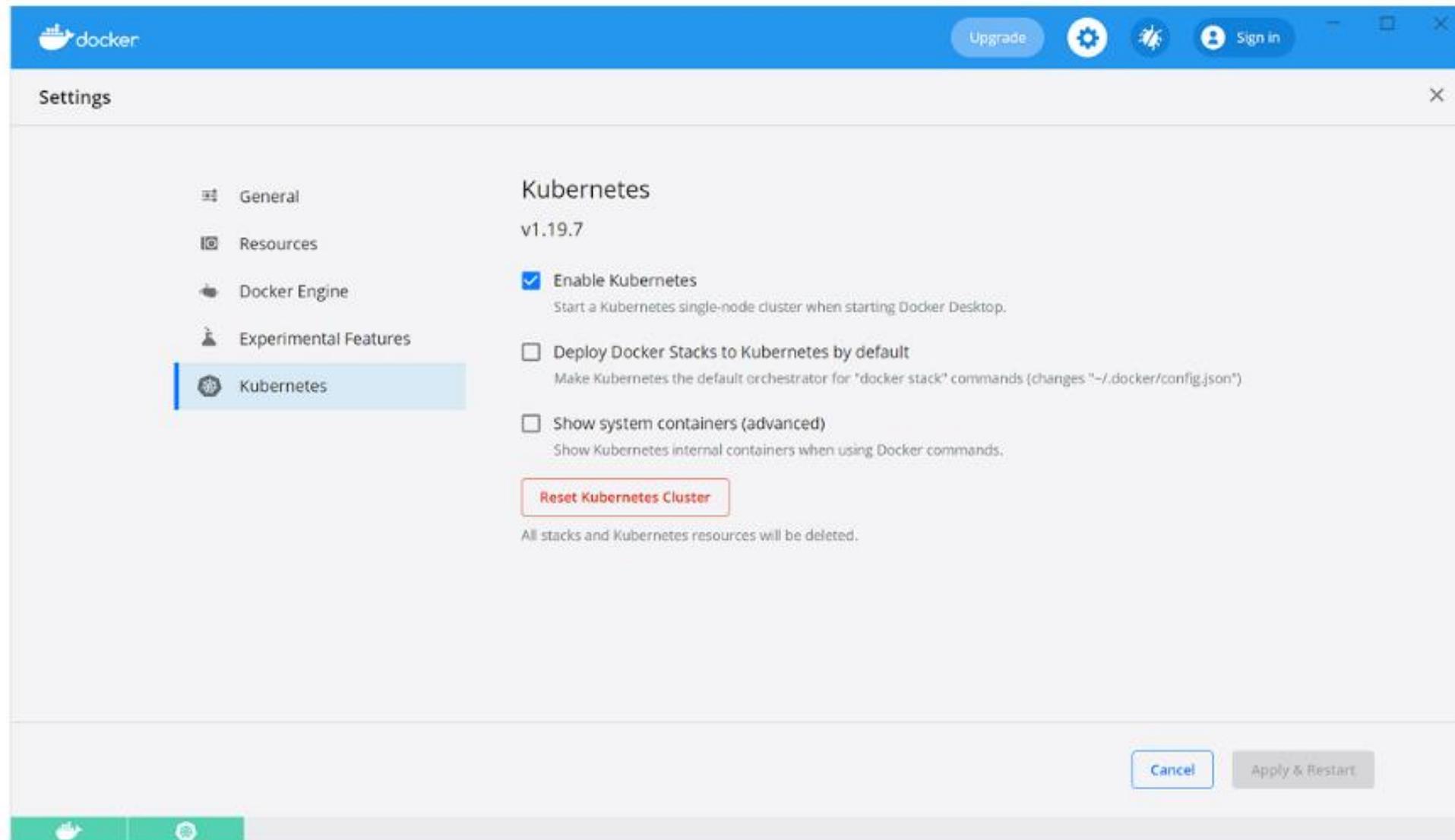
```
> docker stop shopping-cart
```

Running the shopping cart container in Kubernetes



Setting up Kubernetes localhost

Docker Desktop Kubernetes settings



- To install MicroK8S on a Linux machine, simply run this command:

```
sudo snap install microk8s --classic
```

- This will install and start the Kubernetes cluster. Furthermore, this installs the microk8s command-line interface, which includes the kubectl command we are going to use

```
sudo snap alias microk8s.kubectl kubectl
```

- When Kubernetes is running, you can go to the command line and check that Kubernetes is indeed running, using this command

```
kubectl cluster-info
```

- Which should give you a response similar to this:

Kubernetes master is running at <https://kubernetes.docker.internal:6443>

KubeDNS is running at <https://kubernetes.docker.internal:6443/api/v1/namespaces/kube-system/services/kube-dns:dns/proxy>

To further debug and diagnose cluster problems, use 'kubectl cluster-info dump'.

Creating Kubernetes deployment for the shopping cart

- Next, we want to deploy the shopping cart to the Kubernetes cluster we just installed and started. To do that, we need add a manifest file describing the to the shopping cart code base called `shopping-cart.yaml`. This file contains two major sections:
 1. A deployment section
 2. A service section

```
1 kind: Deployment
2 apiVersion: apps/v1
3 metadata:
4   name: shopping-cart
5 spec:
6   replicas: 1
7   selector:
8     matchLabels:
9       app: shopping-cart
10  template:
11    metadata:
12      labels:
13        app: shopping-cart
14  spec:
15    containers:
16      - name: shopping-cart
17        image: shopping-cart
18        imagePullPolicy: IfNotPresent
19        ports:
20          - containerPort: 80
21 ---
22 kind: Service
23 apiVersion: v1
24 metadata:
25   name: shopping-cart
26 spec:
27   type: LoadBalancer
28   ports:
29     - name: shopping-cart
30       port: 5000
31       targetPort: 80
32   selector:
33     app: shopping-cart
```

- Using this manifest to deploy and run the shopping cart in Kubernetes is as simple as running this from the command line:

```
kubectl apply -f shopping-cart.yaml
```

- If the deployment went well, the output from kubectl get all should be similar to this:

| NAME | READY | STATUS | RESTARTS | AGE |
|---|--------------|-------------|-------------|----------------|
| pod/shopping-cart-f4c8f4b94-4j48v | 1/1 | Running | 0 | 15h |
| NAME | TYPE | CLUSTER-IP | EXTERNAL-IP | PORT(S) |
| service/kubernetes | ClusterIP | 10.96.0.1 | <none> | 443/TCP |
| service/shopping-cart | LoadBalancer | 10.103.8.64 | localhost | 5000:31593/TCP |
| NAME | READY | UP-TO-DATE | AVAILABLE | AGE |
| deployment.apps/shopping-cart | 1/1 | 1 | 1 | 6d20h |
| NAME | DESIRED | CURRENT | READY | AGE |
| replicaset.apps/shopping-cart-f4c8f4b94 | 1 | 1 | 1 | 15h |

- If you prefer to have a UI, you can install and start the Kubernetes dashboard.
- First install the Kubernetes dashboard using this command:

```
> kubectl apply -f https://raw.githubusercontent.com/kubernetes/dashboard/  
v2.2.0/aio/deploy/recommended.yaml
```

Kubernetes Dashboard

localhost:8001/api/v1/namespaces/kubernetes-dashboard/services/https:kubernetes-dashboard/proxy/#/overview?namespace=default

kubernetes

Search

Overview

Cluster

- Cluster Roles
- Namespaces
- Nodes
- Persistent Volumes
- Storage Classes

Namespace

| default |
|------------------------------|
| overview |
| Workloads |
| Cron Jobs |
| Daemon Sets |
| Deployments |
| Jobs |
| Pods |
| Replica Sets |
| Replication Controllers |
| Stateful Sets |
| Discovery and Load Balancing |
| Ingresses |
| Services |
| Config and Storage |
| Config Maps |
| Persistent Volume Claims |
| Secrets |
| Custom Resource Definitions |
| Settings |

Workloads

Deployments

| Name | Namespace | Labels | Pods | Age | Images |
|---------------|-----------|--------|------|--------|--|
| shopping-cart | default | - | 1/1 | 6 days | microservicesindotnetregistry.azurecr.io/shopping-cart:1.0.0 |

1 - 1 of 1 | < < > >|

Pods

| Name | Namespace | Labels | Node | Status | Restarts | CPU Usage (cores) | Memory Usage (bytes) | Age |
|-------------------------------|-----------|--|----------------|---------|----------|-------------------|----------------------|----------|
| shopping-cart-f4c8f4b94-4j6bv | default | app: shopping-cart pod-template-hash: f4c8f4b94 | docker-desktop | Running | 0 | - | - | 15 hours |

1 - 1 of 1 | < < > >|

Replica Sets

| Name | Namespace | Labels | Pods | Age | Images |
|-------------------------|-----------|--|------|----------|--|
| shopping-cart-f4c8f4b94 | default | app: shopping-cart pod-template-hash: f4c8f4b94 | 1/1 | 15 hours | microservicesindotnetregistry.azurecr.io/shopping-cart:1.0.0 |

1 - 1 of 1 | < < > >|

Discovery and Load Balancing

Services

| Name | Namespace | Labels | Cluster IP | Internal Endpoints | External Endpoints | Age |
|---------------|-----------|--|-------------|---|--------------------|---------|
| shopping-cart | default | - | 10.103.8.64 | shopping-cart:5000 TCP shopping-cart:31593 TCP | localhost:5000 TCP | 6 days |
| kubernetes | default | component: apiserver provider: kubernetes | 10.96.0.1 | kubernetes:443 TCP kubernetes:0 TCP | - | a month |

1 - 2 of 2 | < < > >|

- Let's add a couple of items to a cart:

POST http://localhost:5000///shoppingcart/15/items

Accept: application/json

Content-Type: application/json

[1, 2]

- Next, let's read the same cart back:

GET http://localhost:5000///shoppingcart/15

- The body of the response from the GET request should be the list of items in the cart:

```
{  
  "userId": 15,  
  "items": [  
    {  
      "productCatalogueId": 1,  
      "productName": "Basic t-shirt",  
      "description": "a quiet t-shirt",  
      "price": {  
        "currency": "eur",  
        "amount": 40  
      }  
    },  
    {  
      "productCatalogueId": 2,  
      "productName": "Fancy shirt",  
      "description": "a loud t-shirt",  
      "price": {  
        "currency": "eur",  
        "amount": 50  
      }  
    }  
  ]  
}
```

- Once done with the shopping cart we can clean up by removing the shopping cart from Kubernetes using this command:

```
kubectl delete -f shopping-cart.yaml
```

Running the shopping cart container on Azure Kubernetes Service

- Set up AKS. We need to create all the Azure resources for a Kubernetes cluster in AKS. This includes the cluster itself, networking, and a private container registry where we will store the container images for our microservices.
- Push the shopping cart container image to our private container registry.
- Deploy the shopping cart to the AKS cluster using the shopping cart's Kubernetes manifest.

Setting up AKS

The setup we need in order to be ready to work with a Kubernetes cluster in AKS consists of four parts:

- Creating a resource group in Azure.
- Creating a private container registry.
- Creating a Kubernetes cluster in AKS.
- Logging our local Kubernetes command line—`kubectl`—into the newly created AKS cluster.

Azure resource group



Azure Kubernetes service



Kubernetes cluster node

Container
registry

```
az group create --name MicroservicesInDotnet --location northeurope
```

```
az acr create --resource-group MicroservicesInDotnet --name  
YOUR_UNIQUE_REGISTRY_NAME --sku Basic
```

```
az aks create --resource-group MicroservicesInDotnet --name  
MicroservicesInDotnetAKSCluster --node-count 1 --enable-addons monitoring  
--generate-ssh-keys --attach-acr YOUR_UNIQUE_REGISTRY_NAME
```

```
az aks get-credentials --resource-group MicroservicesInDotnet --name  
MicroservicesInDotnetAKSCluster  
kubectl get nodes
```

- The last lines of the output are the output from the line `kubectl get nodes` and should look similar to this:

| NAME | STATUS | ROLES | AGE | VERSION |
|----------------------------------|--------|-------|------|---------|
| aks-nodepool1-32786309-vmss00000 | Ready | agent | 107s | v1.14.8 |

- When you are done with the Kubernetes cluster in AKS, you can delete the cluster as well as the container registry with this command:

```
> az group delete --name MicroservicesInDotnet --yes --no-wait
```

- This command will give the dashboard access by telling Kubernetes to assign the role cluster-admin to the account kube-system:kubernetes-dashboard:

```
> kubectl create clusterrolebinding kubernetes-dashboard  
--clusterrole=cluster-admin  
--serviceaccount=kube-system:kubernetes-dashboard
```

- With the access in place we can start the Kubernetes dashboard with the Azure command line like this:

```
>az aks browse --resource-group MicroservicesInDotnet --name  
MicroservicesInDotnetAKSCluster
```

Overview - Kubernetes Dashboard

127.0.0.1:8001#!/overview?namespace=default

kubernetes

Search

+ CREATE

Overview

Cluster

- Namespaces
- Nodes
- Persistent Volumes
- Roles
- Storage Classes

Namespace

- default

Overview

Workloads

- Cron Jobs
- Daemon Sets
- Deployments
- Jobs
- Pods
- Replica Sets

Discovery and Load Balancing

Services

| Name | Labels | Cluster IP | Internal endpoints | External endpoints | Age |
|------------|-------------------------------------|------------|--------------------|--------------------|------------|
| kubernetes | component: a... provider: kub... | 10.0.0.1 | kubernetes:4... | - | 29 minutes |

Config and Storage

Secrets

| Name | Type | Age |
|---------------------|-------------------------------------|------------|
| default-token-4d4ft | kubernetes.io/service-account-to... | 28 minutes |

Running the shopping cart in AKS

The remaining steps to deploying and running the shopping cart in AKS are as follows:

- Tag the shopping cart container image, so we have a precise identification of the container image.
- Push the tagged container image to our container registry in Azure.
- Modify the shopping cart's Kubernetes manifest to refer to the tagged container.
- Apply the modified manifest to AKS.
- Test that the shopping cart runs correctly.

- We set the tag as follows:

```
>docker tag shopping-cart your_unique_registry_name.azurecr.io/shopping-cart:1.0.0
```

- This authentication is done using the Azure CLI like this:

```
> az acr login --name YOUR_UNIQUE_REGISTRY_NAME
```

- Now the tagged container image can be pushed to our private Docker registry in Azure with this Docker command:

```
> docker push your_unique_registry_name.azurecr.io/shopping-cart:1.0.0
```

```
kind: Deployment
apiVersion: apps/v1
metadata:
  name: shopping-cart
spec:
  replicas: 1
  selector:
    matchLabels:
      app: shopping-cart
template:
  metadata:
    labels:
      app: shopping-cart
spec:
  containers:
    - name: shopping-cart
      image: your_unique_registry_name.azurecr.io/
        shopping-cart:1.0.0
      imagePullPolicy: IfNotPresent
      ports:
        - containerPort: 80
---
apiVersion: v1
kind: Service
metadata:
  name: shopping-cart
spec:
  type: LoadBalancer
  ports:
    - name: shopping-cart
      port: 5000
      targetPort: 80
  selector:
    app: shopping-cart
```

- The only change is the image name, which now refers to the image in our private container registry in Azure.
- All that remains is to apply this manifest to the Kubernetes cluster in AKS:

```
> kubectl apply -f shopping-cart.yaml
```

- Using the command line the command kubectl get all will show the information about everything running in the cluster.
- The output should be similar to this:

| NAME | | READY | STATUS | RESTARTS | AGE |
|---|--------------|--------------|---------------|----------------|------|
| pod/shopping-cart-f4c8f4b94-vnmn9 | | 1/1 | Running | 0 | 107s |
| NAME | TYPE | CLUSTER-IP | EXTERNAL-IP | PORT(S) | AGE |
| service/kubernetes | ClusterIP | 10.0.0.1 | <none> | 443/TCP | 15m |
| service/shopping-cart | LoadBalancer | 10.0.100.183 | 52.142.83.184 | 5000:31552/TCP | 107s |
| NAME | | READY | UP-TO-DATE | AVAILABLE | AGE |
| deployment.apps/shopping-cart | | 1/1 | 1 | 1 | 107s |
| NAME | | DESIRED | CURRENT | READY | AGE |
| replicaset.apps/shopping-cart-f4c8f4b94 | | 1 | 1 | 1 | 107s |

Services - Kubernetes Dashboard + New

127.0.0.1:8001#!/service?namespace=default

... ⋮

kubernetes Search + CREATE

☰ Discovery and load balancing > Services

Cluster

- Namespaces
- Nodes
- Persistent Volumes
- Roles
- Storage Classes

Namespace

- default ⋮

Overview

Workloads

- Cron Jobs
- Daemon Sets
- Deployments
- Jobs
- Pods
- Replica Sets
- Replication Controllers
- Stateful Sets

Discovery and Load Balancing

- Ingresses
- Services**

Config and Storage ⋮

Services

| Name | Labels | Cluster IP | Internal endpoints | External endpoints | Age |
|---------------|---|--------------|---|--------------------|------------|
| shopping-cart | - | 10.0.100.183 | shopping-cart:5000 ... shopping-cart:3155... | 52.142.83.184:5000 | 2 minutes |
| kubernetes | component: apiserve provider: kubernetes | 10.0.0.1 | kubernetes:443 TCP | - | 16 minutes |

- To verify that the shopping cart is indeed running, we can test its endpoint by, for instance, adding items to a cart using the shopping cart's POST endpoint

POST http://52.142.83.184:5000//shoppingcart/15/items

Accept: application/json

Content-Type: application/json

[1, 2]

- And then reading the same cart using the GET endpoint:

GET http://52.142.83.184:5000//shoppingcart/15

Summary

- The microservices we develop can be deployed to many different environments.
- .NET-based microservices are easily put into containers and run as containers.
- Dockerfiles for .NET-based microservices follow a common template.
- Deploying our microservices to Kubernetes gives us a highly scalable environment and a versatile container orchestrator.
- Kubernetes works the same on localhost and in the cloud. This means we can easily run the exact containers on localhost for development and in the cloud for production.
- Kubernetes can run everything we are going to develop in the upcoming Modules while providing tools for scaling, monitoring, and debugging microservices.
- Azure Kubernetes Services is an easy-to-set-up managed Kubernetes offering that enables us to get up and running with Kubernetes quickly.

Identifying and scoping microservices

Introduction

This Module covers:

- Scoping microservices for business capability
- Scoping microservices to support technical capabilities
- Scoping microservices to support efficient development work
- Managing when scoping microservices is difficult
- Carving out new microservices from existing ones

The primary driver for scoping microservices: Business capabilities

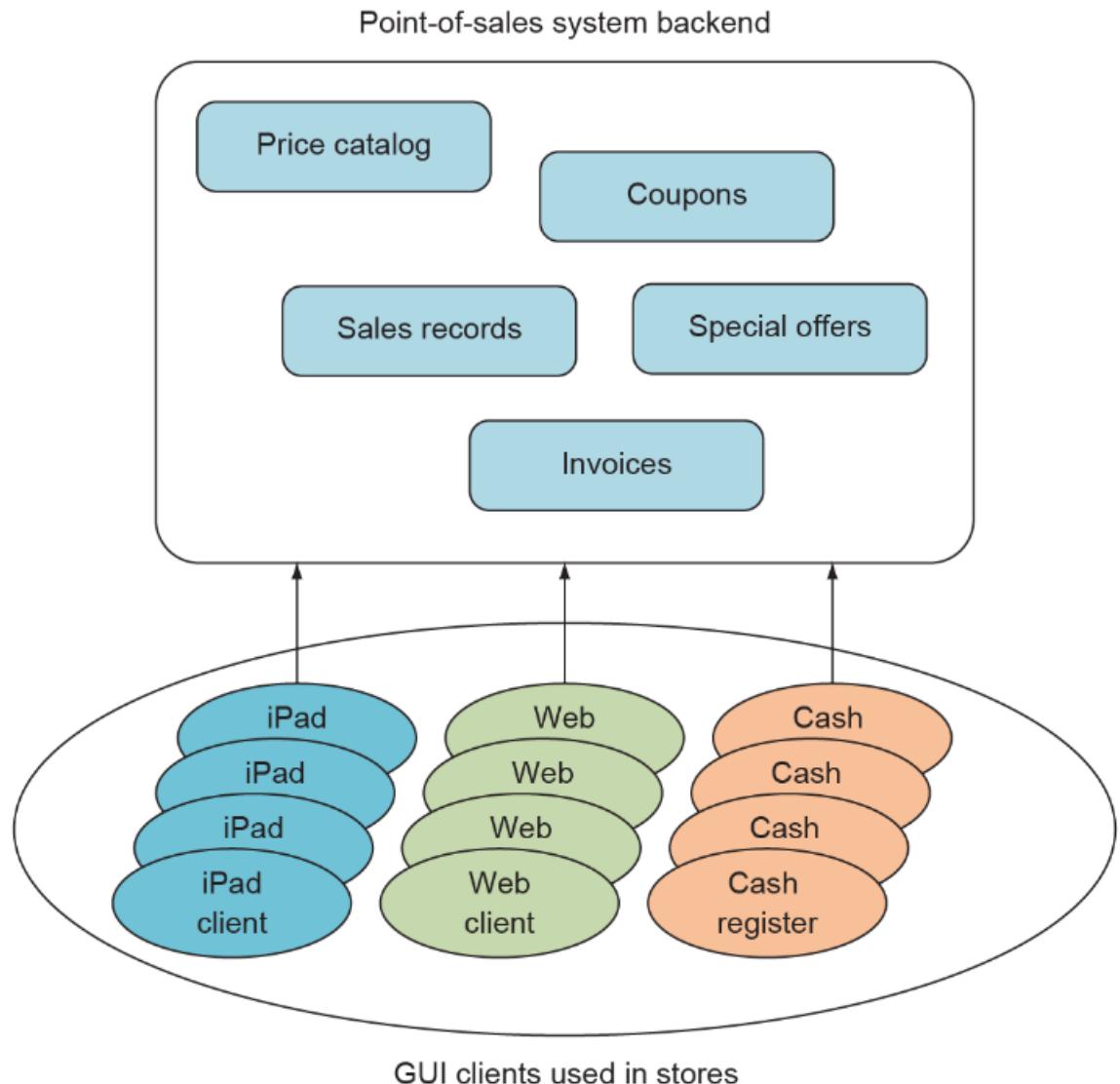
What is a business capability?

- A *business capability* is something an organization does that contributes to business goals.
- For instance, handling a shopping cart on an e-commerce website is a business capability that contributes to the broader business goal of allowing users to purchase items.
- A given business will have a number of business capabilities that together make the overall business function.

Identifying business capabilities

- A good understanding of the domain will enable you to understand how the business functions.
- Understanding how the business functions means you can identify the business capabilities that make up the business and the processes involved in delivering the capabilities.
- In other words, the way to identify business capabilities is to learn about the business's domain.
- You can gain this type of knowledge by talking with the people who know the business domain best: business analysts, the end users of your software, and so on—all the people directly involved in the day-to-day work that drives the business.

Example: Point-of-sale system

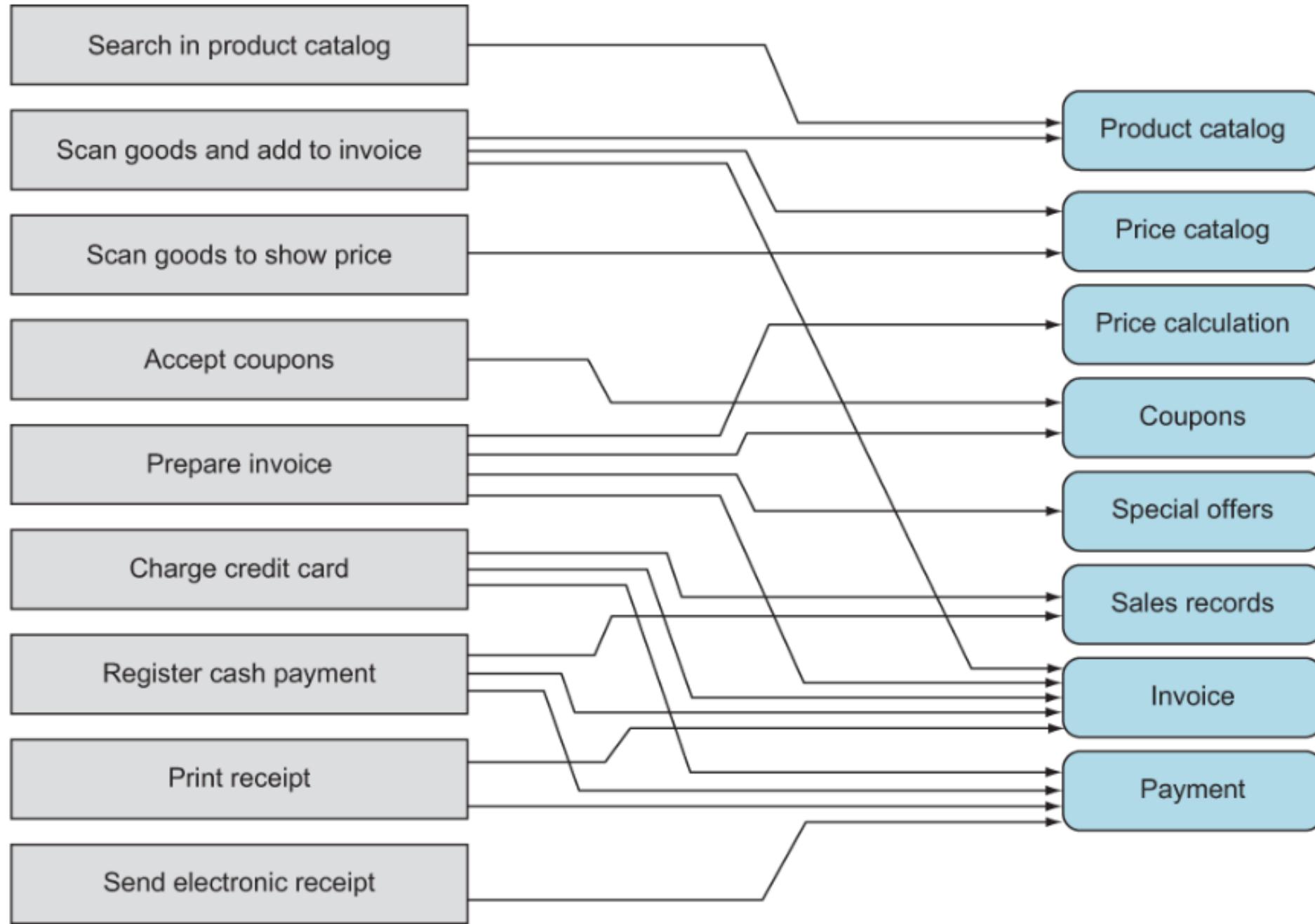


The system offers cashiers a variety of functions:

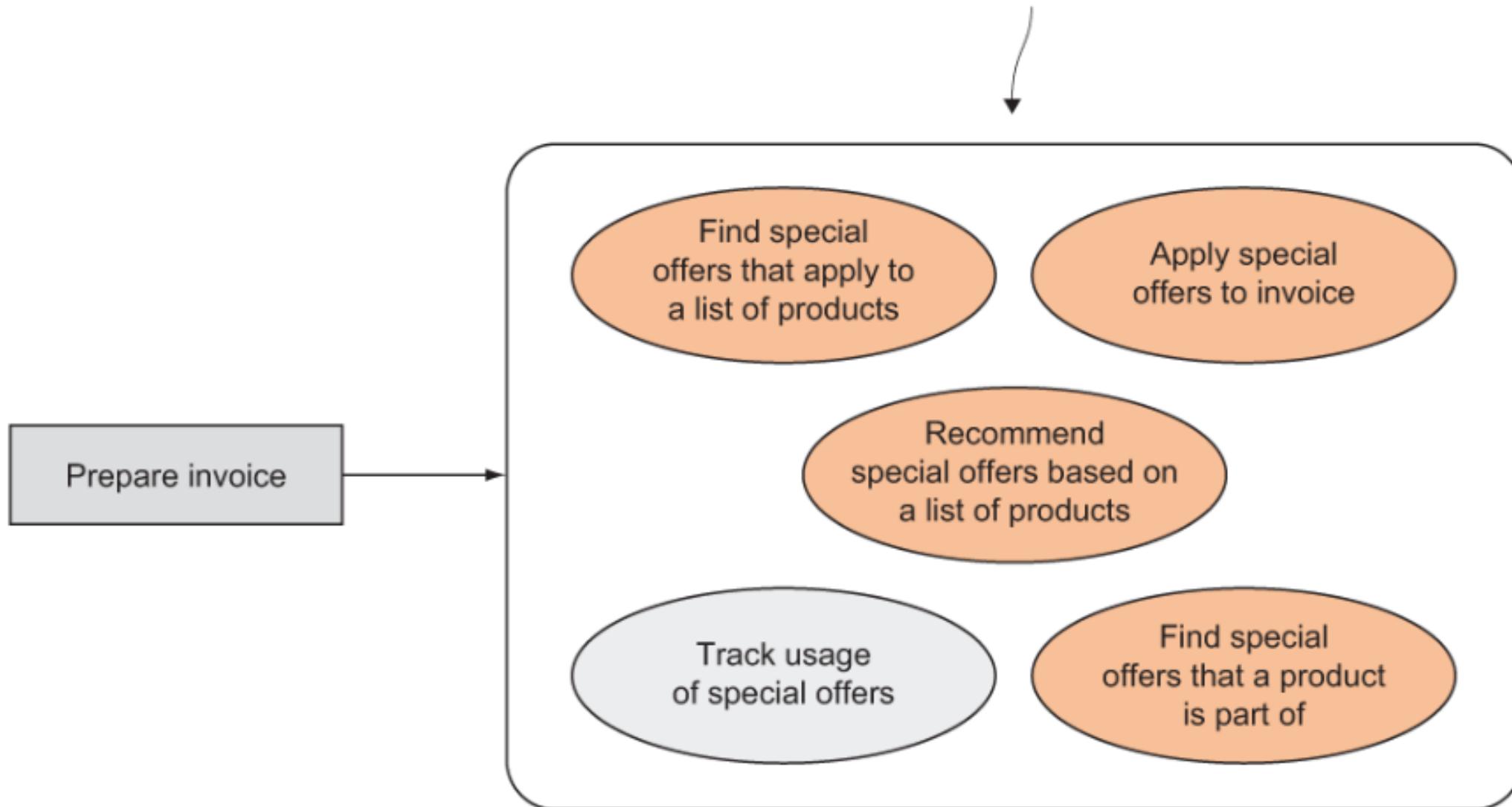
- Scan products and add them to the invoice.
- Prepare an invoice.
- Charge a credit card via a card reader attached to the client.
- Register a cash payment.
- Accept coupons.
- Print a receipt.
- Send an electronic receipt to the customer.
- Search in the product catalog.
- Scan one or more products to show prices and special offers related to the products.

If you continued the hunt for business capabilities in the POS system, you might end up with this list:

- Product catalog
- Price catalog
- Price calculation
- Special offers
- Coupons
- Sales records
- Invoice
- Payment



The special offers business capability



HTTP API: Accessible from other microservices

Special offers HTTP API

- GetApplicableSpecialOffers (list of products)
- ApplySpecialOffers (invoice, special offers)
- GetPotentialOffers (product)
- GetRecommendedSpecialOffers (list of products)

Event feed HTTP API

- Endpoint to get events

Special offers store

Special offers domain model

EventStore

Special offers database

HTTP API: Accessible from other microservices

Shopping cart HTTP API

- Endpoint to get a shopping cart
- Endpoint to add items to shopping cart
- Endpoint to delete items from shopping cart

Event feed HTTP API

- Endpoint to get events

Shopping cart domain model

Shopping cart store

ProductCatalogClient

EventStore

Shopping cart database

The secondary driver for scoping microservices: Supporting technical capabilities

- The secondary way to identify scopes for microservices is to look at supporting technical capabilities.
- A *supporting technical capability* is something that doesn't directly contribute to a business goal but supports other microservices, such as integrating with another system or scheduling an event to happen some time in the future.

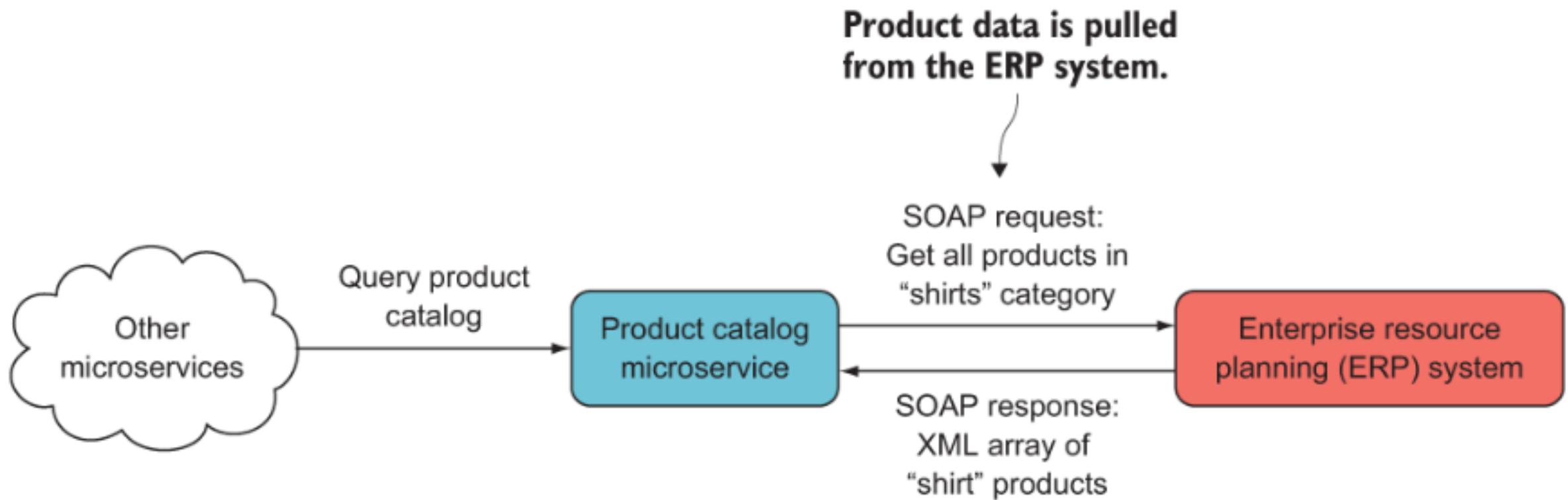
What is a technical capability?

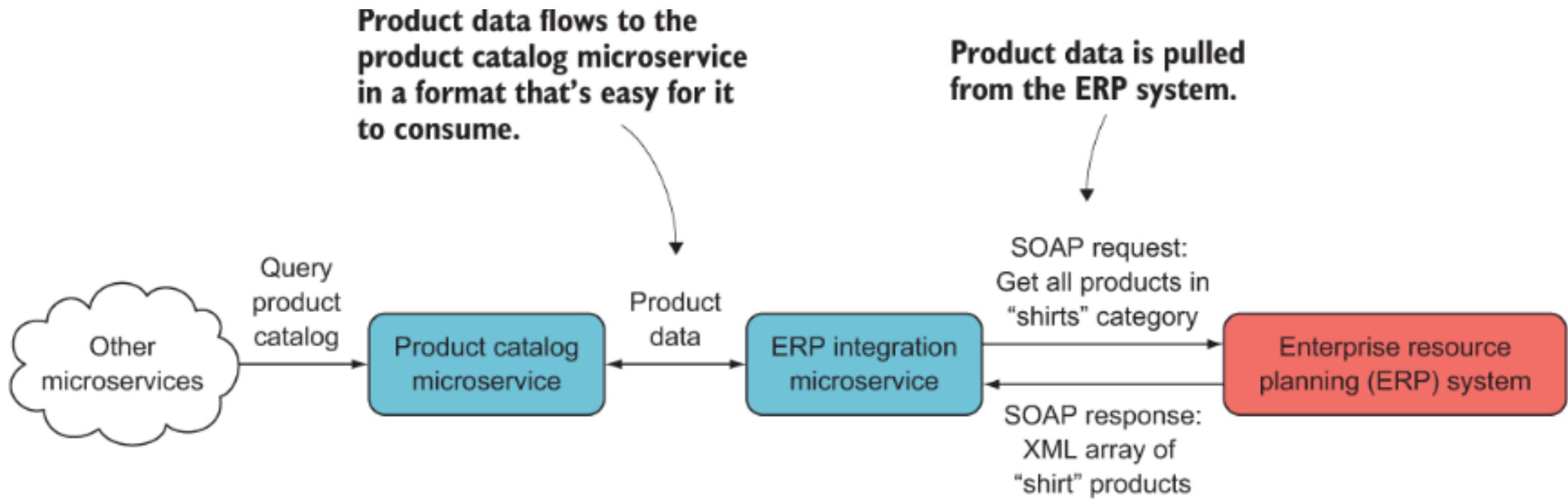
- Supporting technical capabilities are secondary drivers in scoping microservices because they don't directly contribute to the system's business goals.
- They exist to simplify and support the other microservices that implement business capabilities.

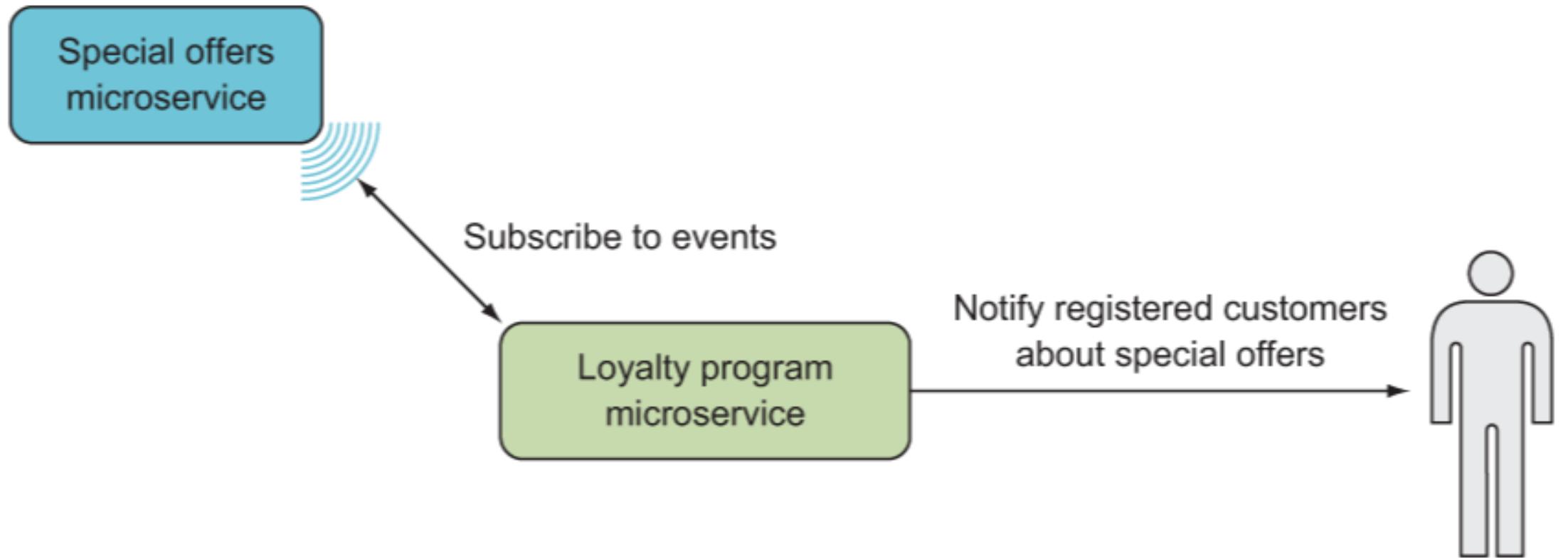
Examples of supporting technical capabilities

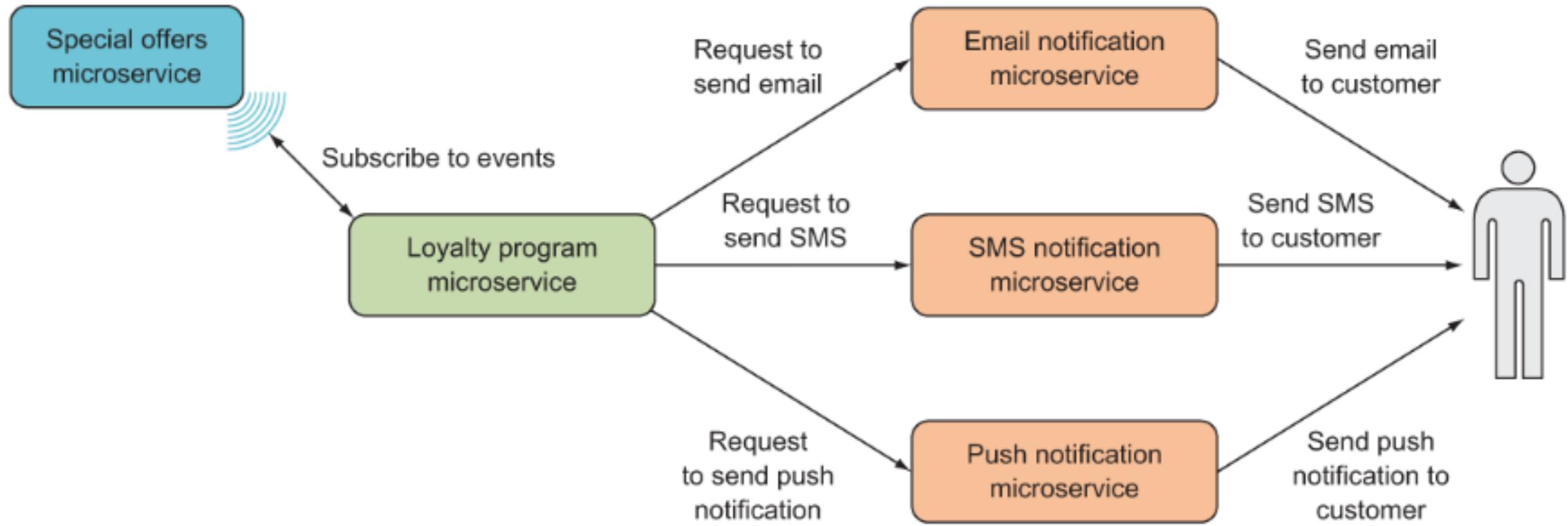
To give you a feel for what I mean by supporting technical capabilities, let's consider two examples:

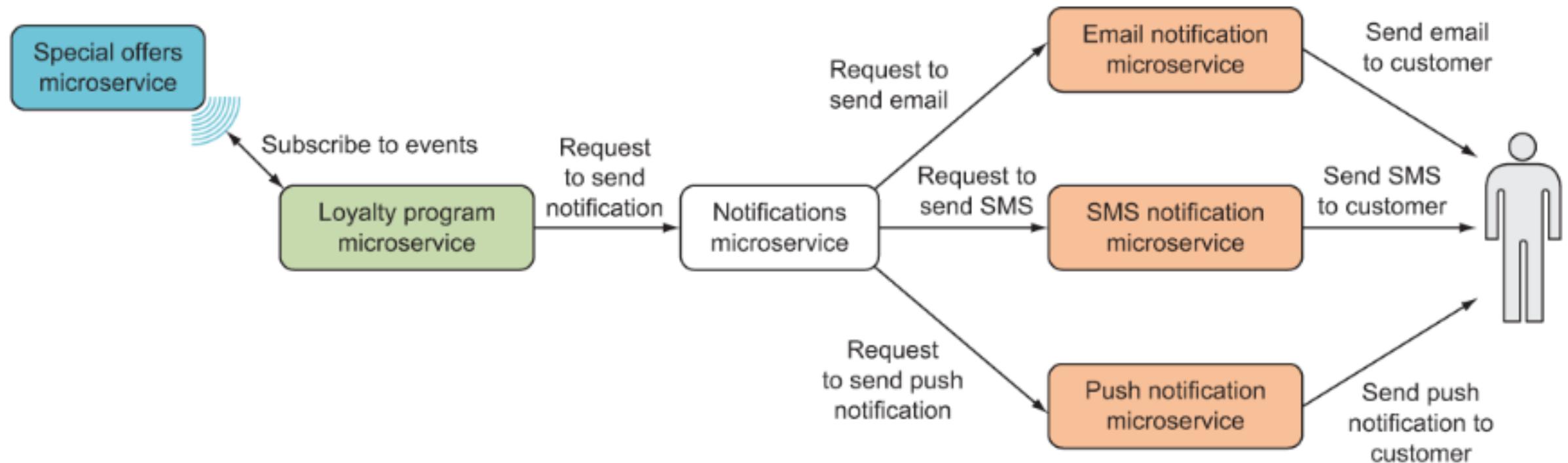
- An integration with another system.
- The ability to send notifications to customers.

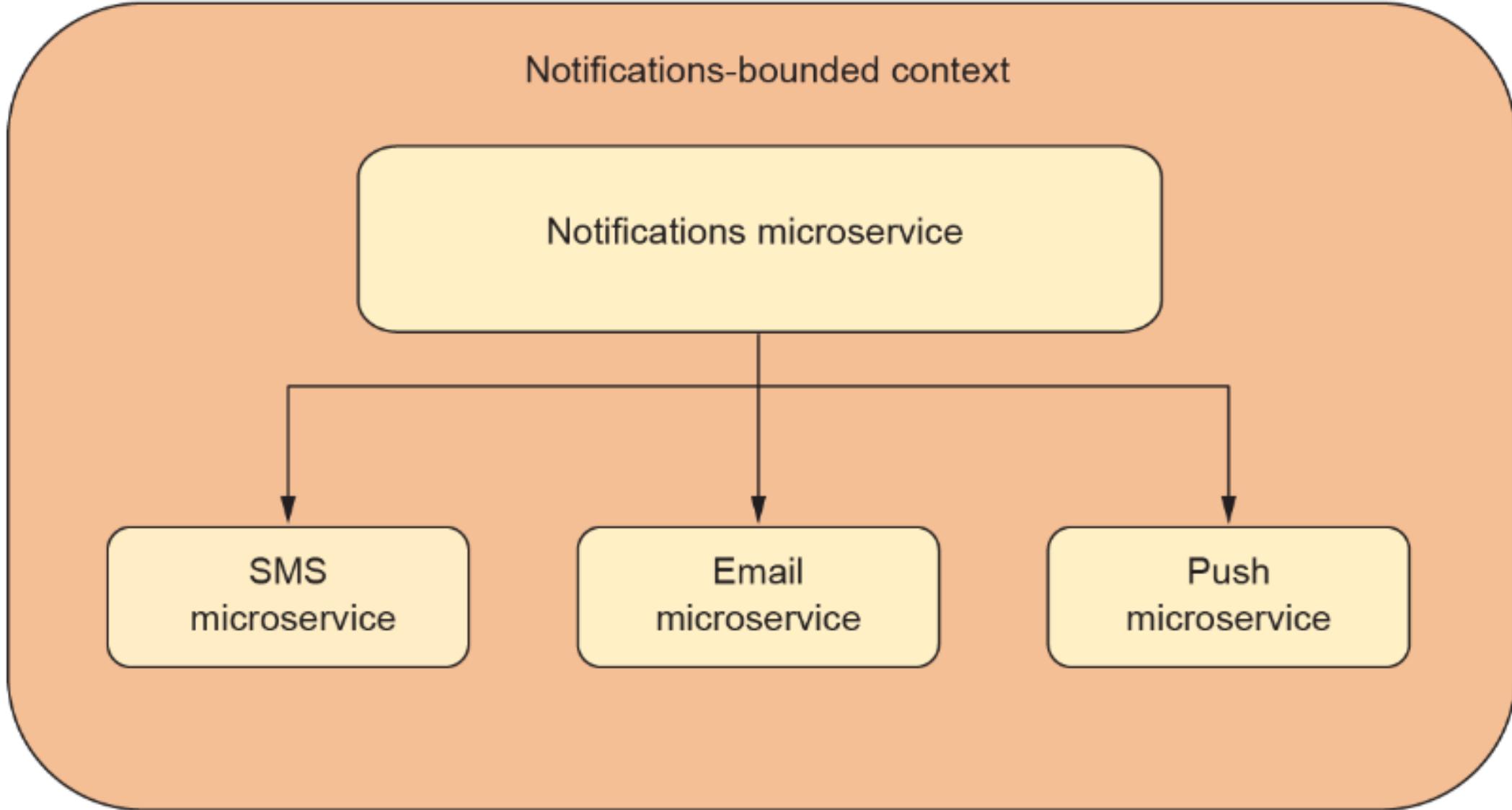












Identifying technical capabilities

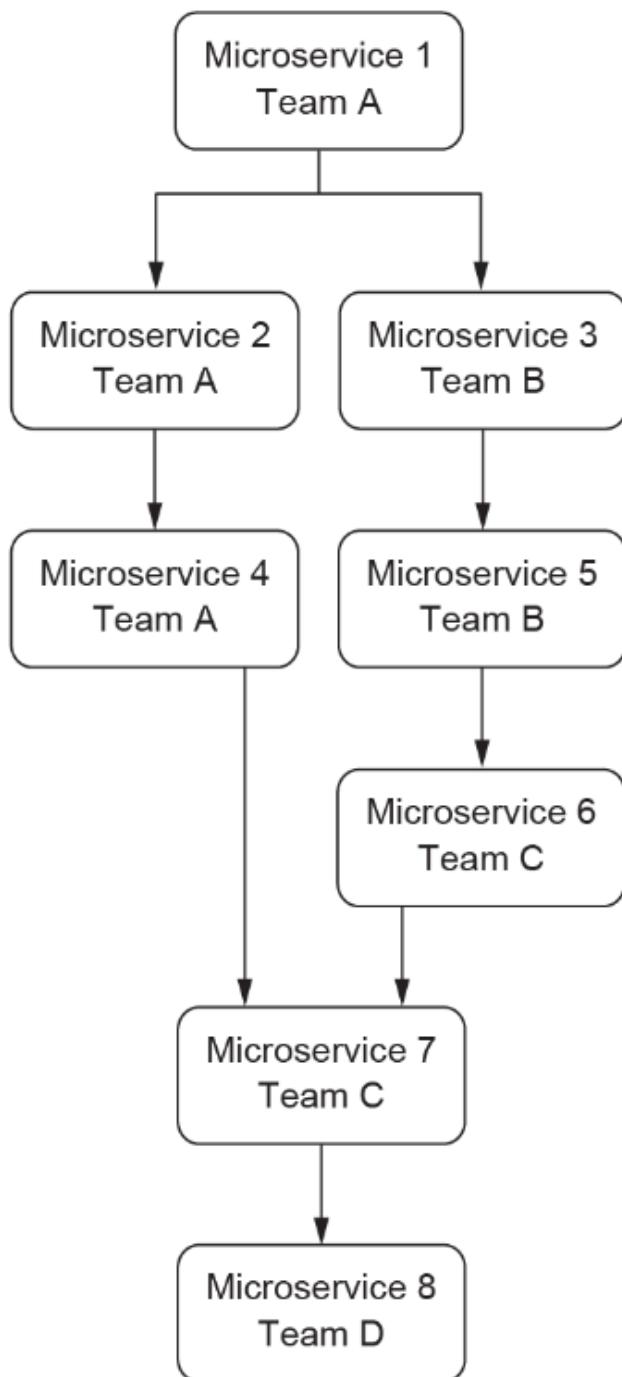
The question of whether a supporting technical capability should be implemented in its own microservice is about what will be easiest in the long run. You should ask these questions:

- If the supporting technical capability stays in a microservice scoped to a business capability, is there a risk that the microservice will no longer be replaceable with reasonable effort?
- Is the supporting technical capability implemented in several microservices scoped to business capabilities?
- Will a microservice implementing the supporting capability be individually deployable?
- Will all microservices scoped to business capabilities still be individually deployable if the supporting technical capability is implemented in a separate microservice?

The tertiary driver for scoping microservices: Supporting efficiency of work

- The third driver when we find the scope and boundaries of microservices is supporting efficiency of work.
- The microservices we design and create should be efficient to work with and should let teams that develop and maintain the microservices work efficiently.
- This, largely, is a matter of respecting and using Conway's law, which you may recall from earlier:

Any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization's communication structure.



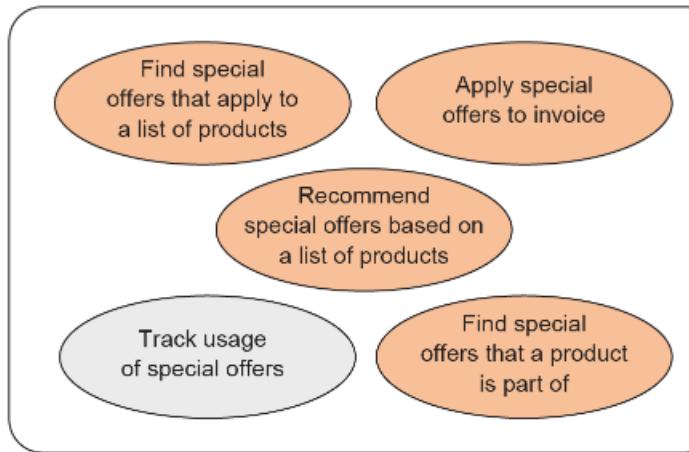
What to do when the correct scope isn't clear

Lack of clarity can have several causes, including the following:

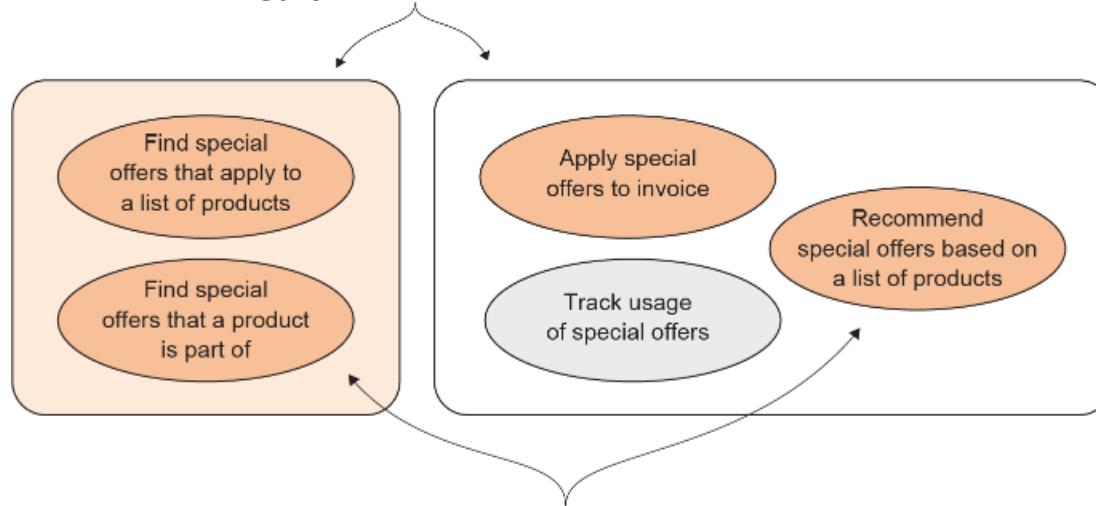
- Insufficient understanding of the business domain
- Confusion in the business domain
- Incomplete knowledge of the details of a technical capability
- Inability to estimate the complexity of a technical capability

Starting a bit bigger

The special offers business capability



The special offers business capability
wrongly split over two microservices

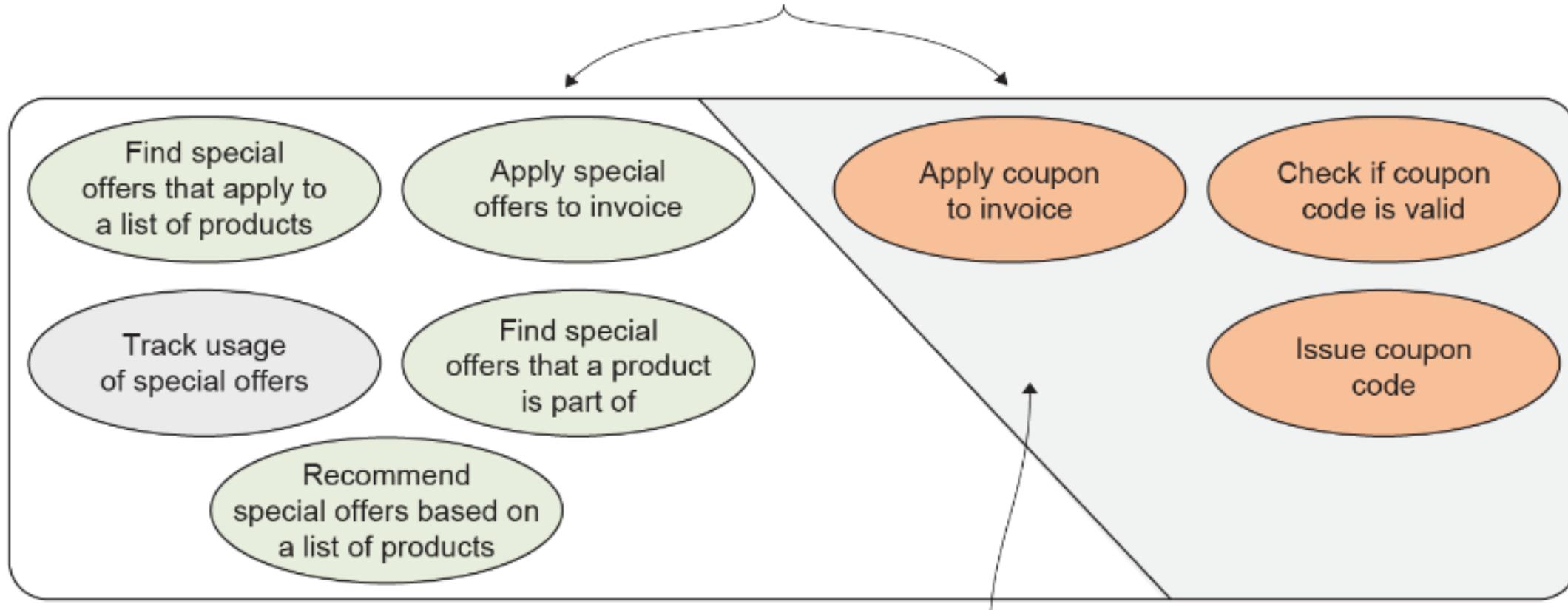


Unclear factoring of responsibility: two processes
with related functionality that need the same data
and the same search logic

If you base the scope of your microservices on only part of the special offers business capability, you'll incur some significant costs:

- Data and data model duplication between the two microservices
- Unclear factoring of responsibility
- Obstacles to refactoring the code for the business capability
- Difficulty deploying the two microservices independently

The special offers and coupons business capabilities both included in the special offers microservice

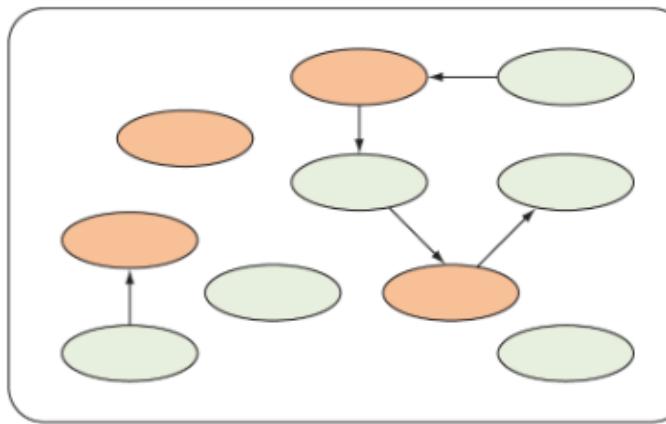


No data and no logic are shared across this line.

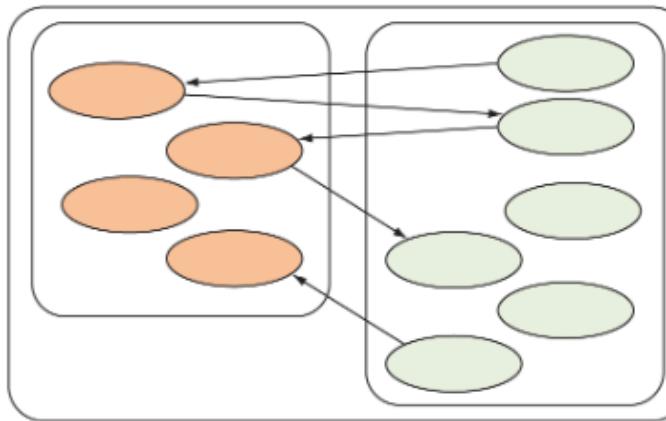
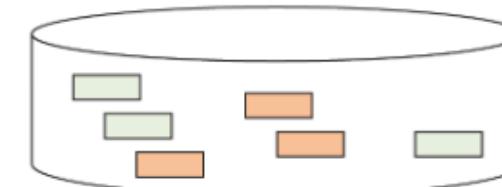
There are costs associated with including too much in the scope of a microservice:

- The code base becomes bigger and more complex, which can lead to changes being more expensive.
- The microservice is harder to replace.

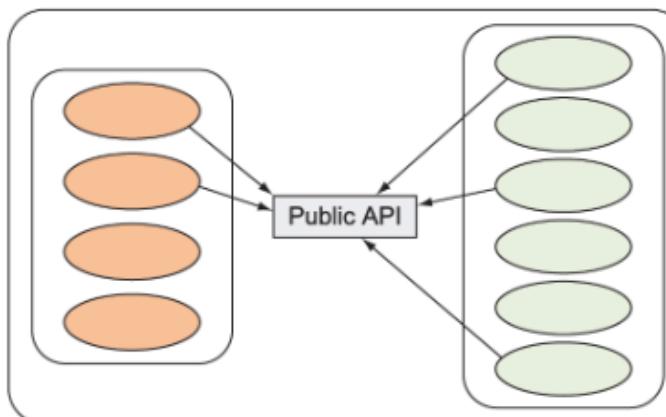
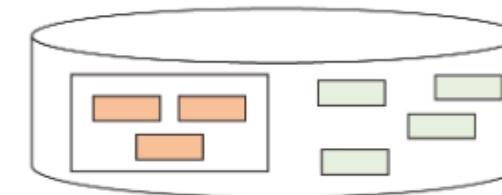
Carving out new microservices from existing microservices



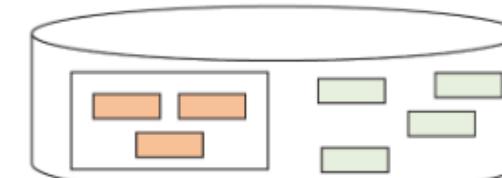
Step 0: Special offers microservice includes coupons capability.



Step 1: Special offers microservice still includes coupons capability, but coupons capability is refactored into a separate project.



Step 2: Special offers microservice still includes coupons capability, but it is refactored and all communication is going through a public API.



Well-scoped microservices adhere to the microservice characteristics

Characteristics of microservices:

- A microservice is responsible for a single capability.
- A microservice is individually deployable.
- A small team can maintain a handful of microservices.
- A microservice is replaceable.

Primary scoping to business capabilities leads to good microservices

The primary driver for scoping microservices is identifying business capabilities. Let's see how that makes for microservices that adhere to the microservice characteristics.

- Responsible for a single capability
- Individually deployable
- Replaceable and maintainable by a small team

Secondary scoping to support technical capabilities leads to good microservices

The secondary driver for scoping microservices is identifying supporting technical capabilities. Let's see how that makes for microservices that adhere to the microservice characteristics.

- Responsible for a single capability
- Individually deployable
- Replaceable and maintainable by a small team

Tertiary scoping to support efficiency of work

The third driver for scoping microservices is supporting the efficiency of work. Let's see how making sure teams can work efficiently aligns with the microservice characteristics.

- Individually deployable
- A small team can maintain a few handfuls of microservices

Summary

- Identifying supporting technical capabilities is an opportunistic form of design. You should only pull a supporting technical capability into a separate microservice if it will be an overall simplification.
- The tertiary driver in scoping microservices is efficiency of work. A team assigned to develop and maintain microservices should be able to work efficiently and autonomously.
- When you're in doubt about the scope of a microservice, lean toward making the scope slightly bigger rather than slightly smaller.
- Because scoping microservices well is difficult, you'll probably be in doubt sometimes. You're also likely to get some of the scopes wrong in your first iteration.
- You must expect to have to carve new microservices out of existing ones from time to time.
- You can use your doubt about scope to organize the code in your microservices so that they lend themselves to carving out new microservices at a later stage.

5. Microservice collaboration

Introduction

This Module covers

- Understanding how microservices collaborate through commands, queries, and events
- Comparing event-based collaboration with collaboration based on commands and queries
- Implementing an event feed
- Implementing command-, query-, and event-based collaboration
- Deploying collaborating microservices to Kubernetes

Types of collaboration: Commands, queries, and events



Types of collaboration: Commands, queries, and events

When two microservices collaborate, there are three main styles:

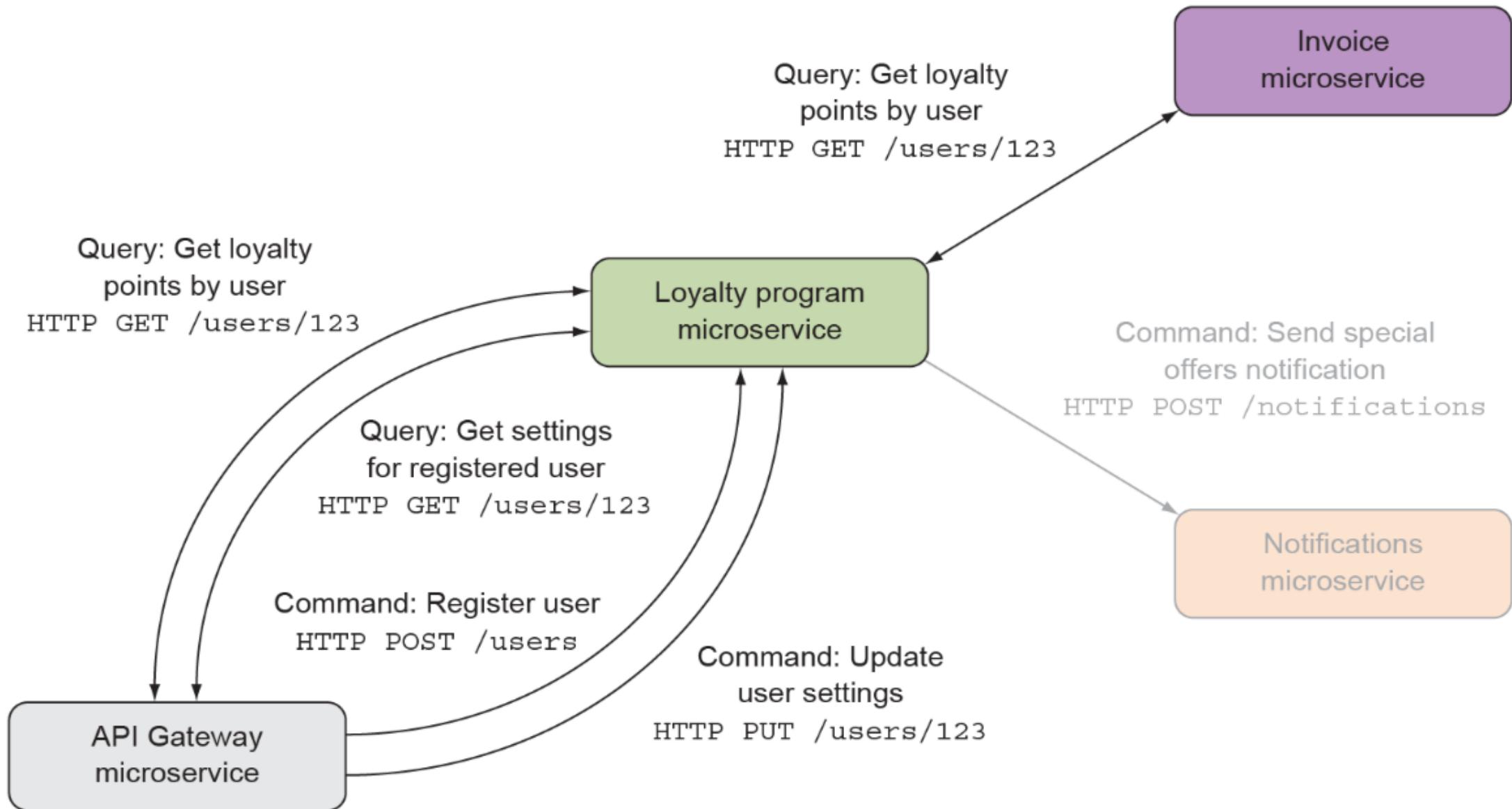
- Commands
- Queries
- Events



Commands and queries: Synchronous collaboration

The main benefits of using gRPC for microservice collaboration are as follows:

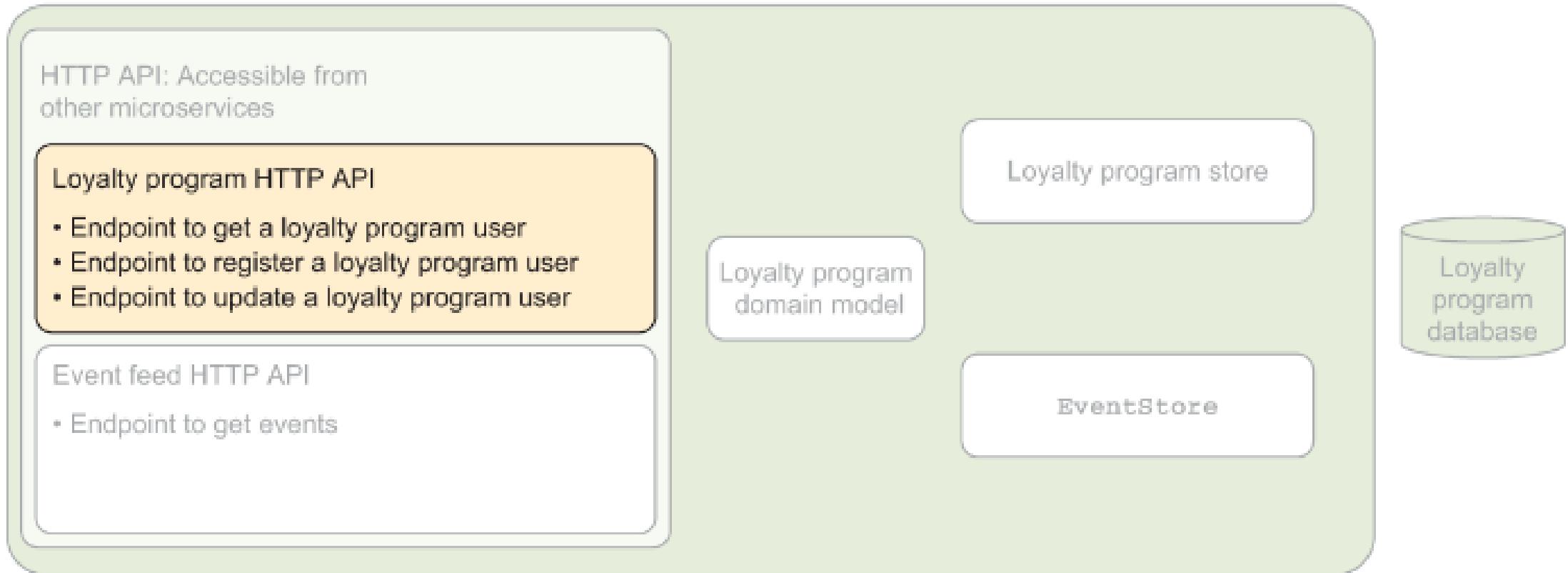
- Efficiency
- Explicit contracts
- Good .NET support



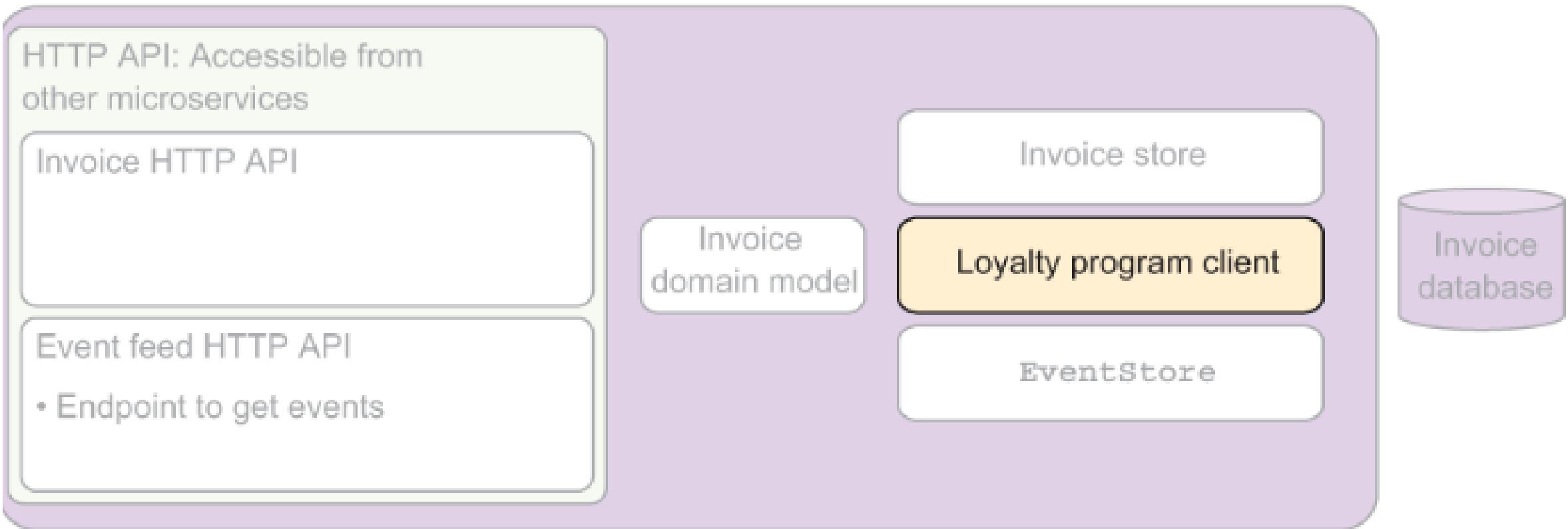
All in all, the loyalty program microservice needs to expose three endpoints:

- An HTTP GET endpoint at URLs of the form `/users/{userId}` that responds with a representation of the user.
- An HTTP POST endpoint at “`/users/`” that expects a representation of a user in the body of the request and then registers that user in the loyalty program.
- An HTTP PUT endpoint at URLs of the form “`/users/{userId}`” that expects a representation of a user in the body of the request and then updates an already registered user.

Loyalty program microservice



Invoice microservice

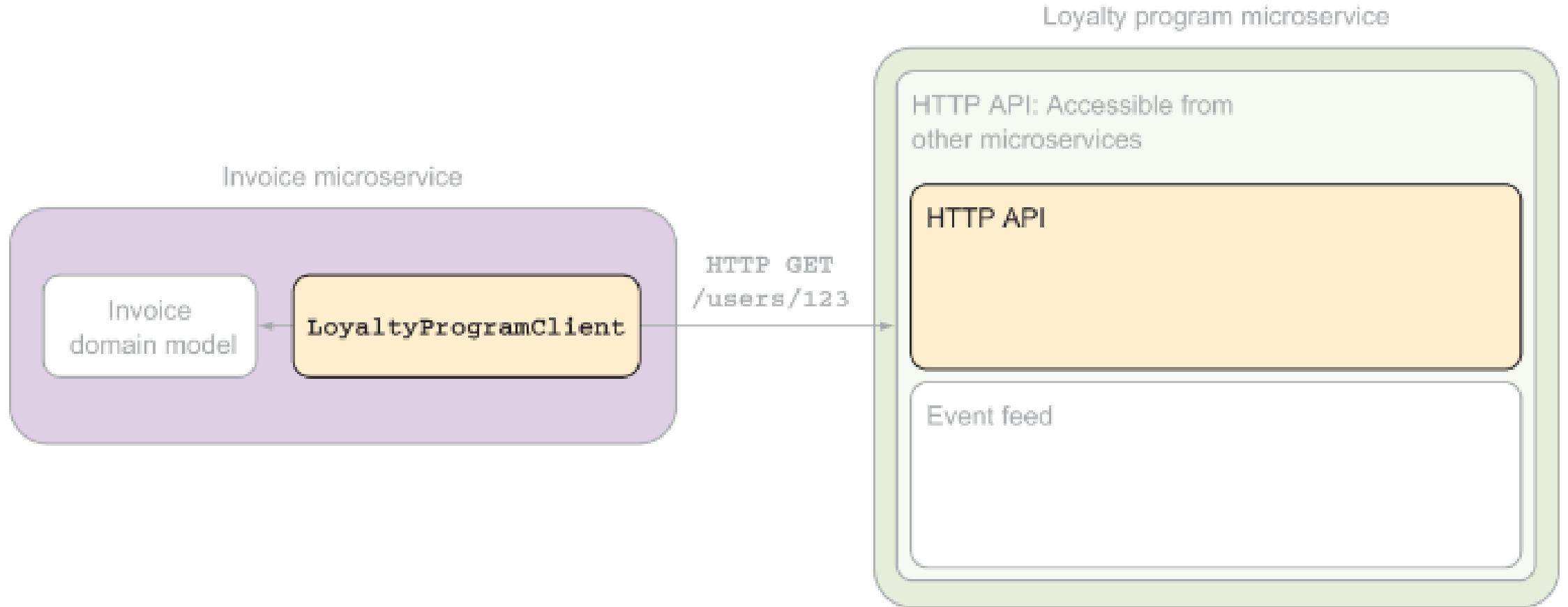


```
public record LoyaltyProgramUser(  
    int Id,  
    string Name,  
    int LoyaltyPoints,  
    LoyaltyProgramSettings Settings);
```

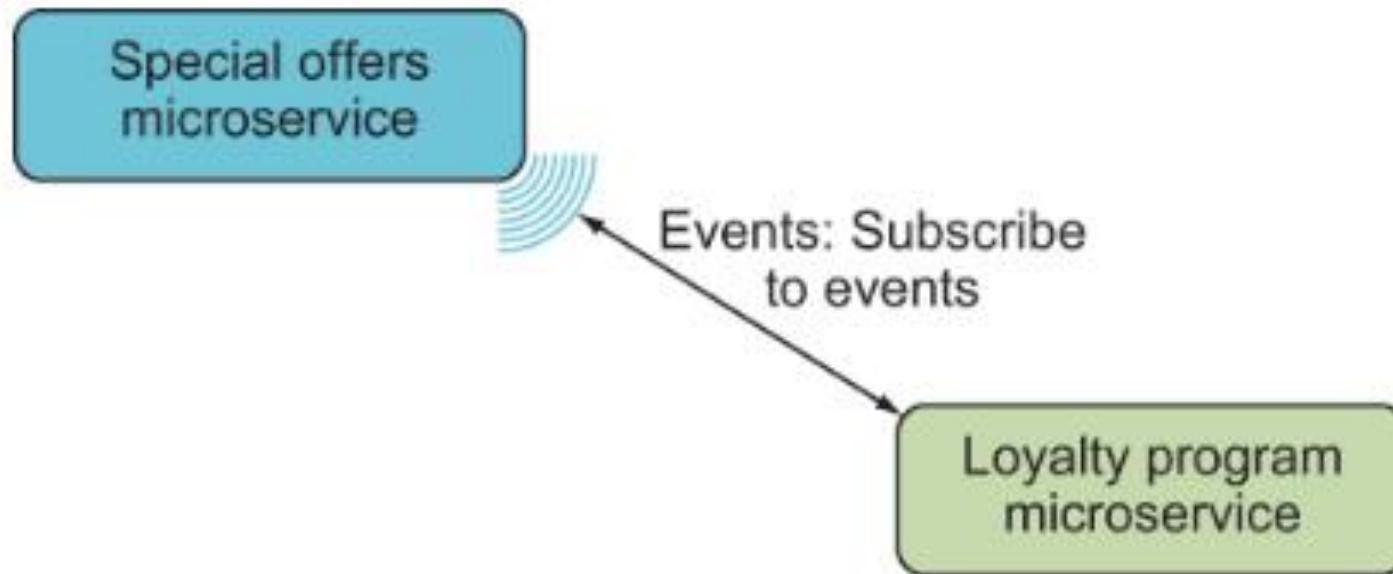
```
public record LoyaltyProgramSettings()  
{
```

```
    public LoyaltyProgramSettings(string[] interests) : this()  
{  
        this.Interests = interests;  
    }
```

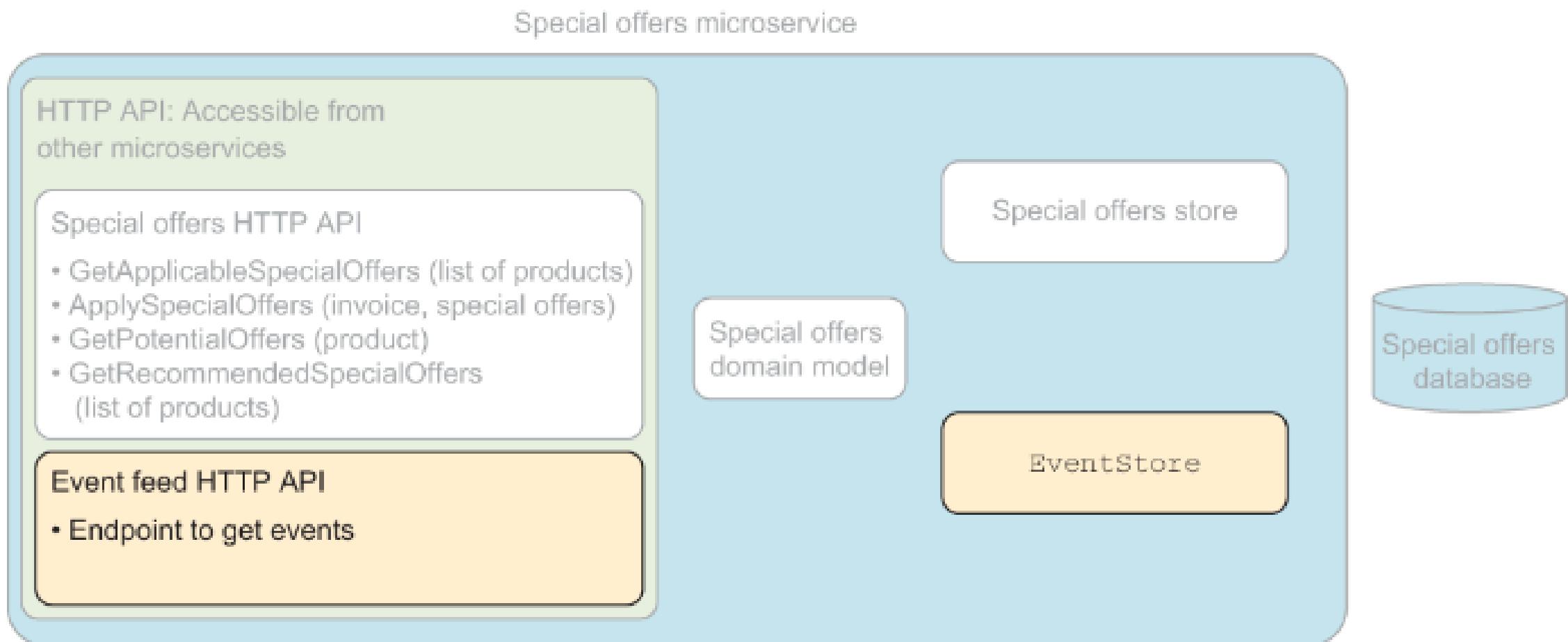
```
    public string[] Interests { get; init; } = Array.Empty<string>();  
}
```



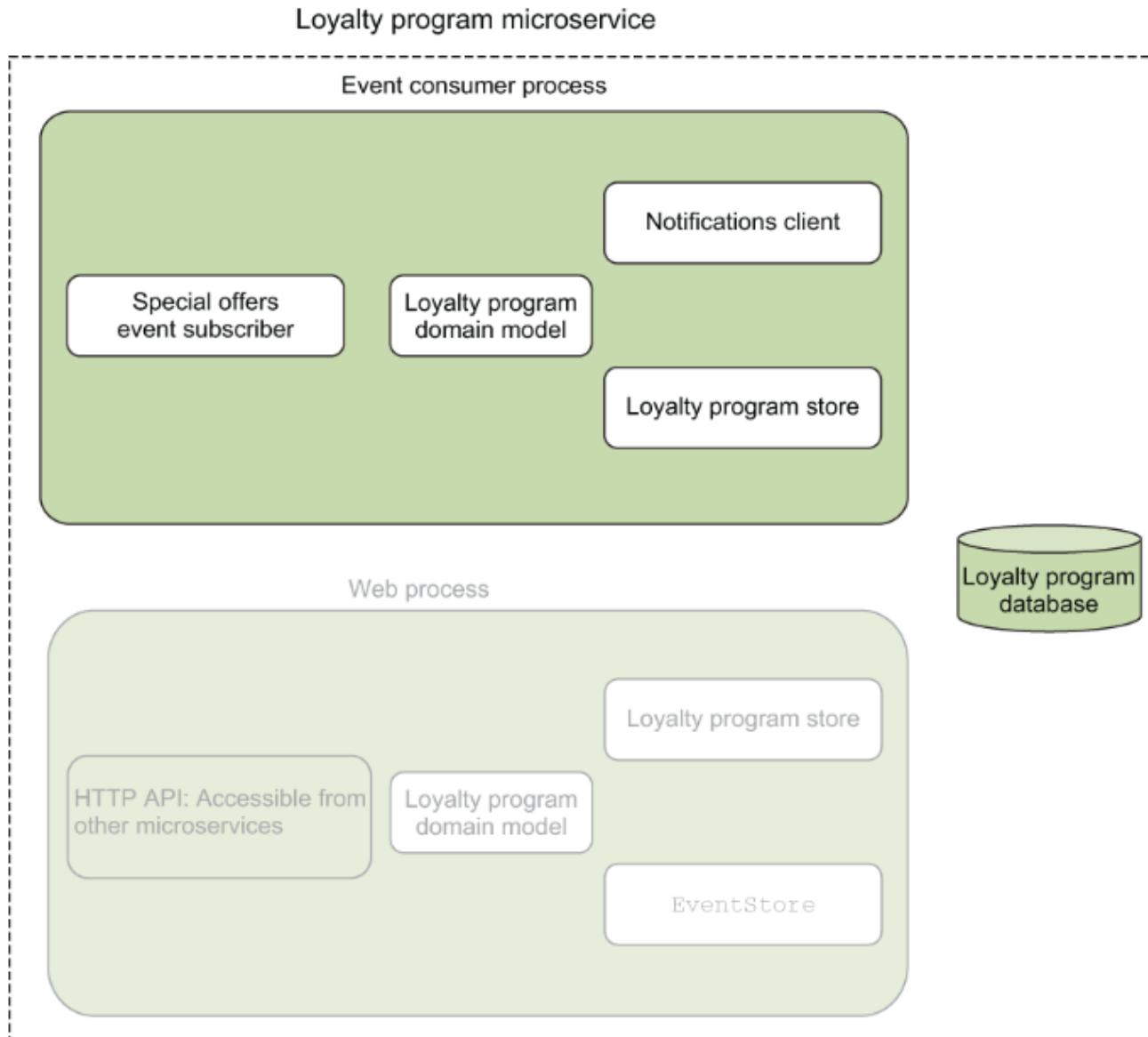
Events: Asynchronous collaboration



Exposing an event feed



Subscribing to events



Data formats

- If you need to exchange a lot of data, a more compact format may be needed. Text-based formats such as JSON and XML are a lot more verbose than binary formats like protocol buffers.
- If you need a more structured format than JSON that's still human readable, you may want to use YAML.
- If your company uses proprietary data formatting, you may need to support that format.

POST /users HTTP/1.1

Host: localhost:5000

Accept: application/yaml

Content-Type: application/yaml

Name: Christian

Settings:

Interests:

- whisky
- cycling
- "software design"

The response to this request also uses YAML:

HTTP/1.1 201 Created

Content-Type: application/yaml

Location: http://localhost:5000/users/1

Id: 1

Name: Christian

Settings:

Interests:

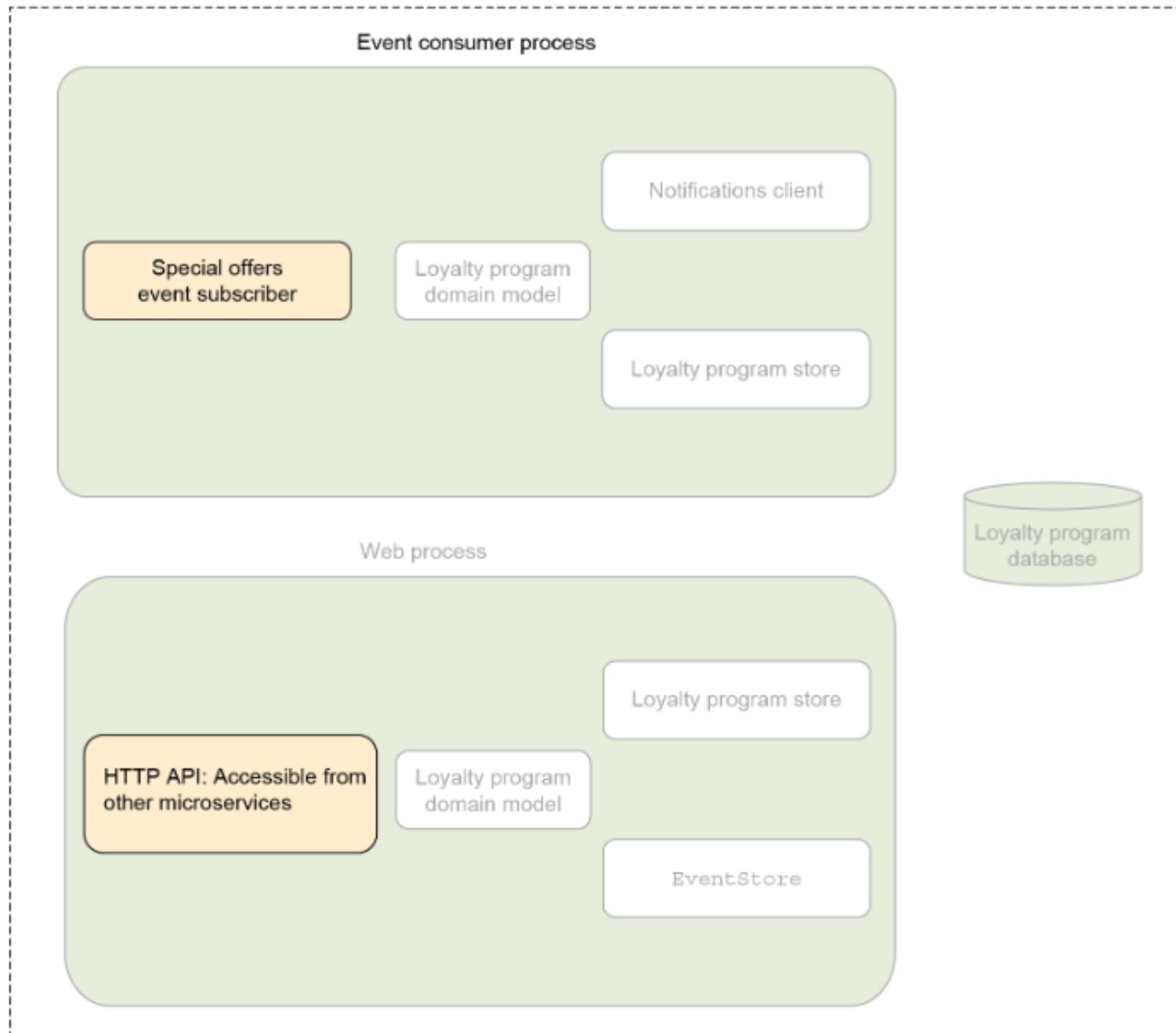
- whisky
- cycling
- "software design"

Implementing collaboration

Three steps are involved in implementing the collaboration:

- Set up a project for the loyalty program.
- Implement the command- and query-based collaborations.
- Implement the event-based collaboration.

Loyalty program microservice



Setting up a project for the loyalty program

```
namespace LoyaltyProgram
{
    using Microsoft.AspNetCore.Builder;
    using Microsoft.AspNetCore.Hosting;
    using Microsoft.Extensions.DependencyInjection;

    public class Startup
    {
        public void ConfigureServices(IServiceCollection services)
        {
            services.AddControllers();
        }

        public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
        {
            app.UseHttpsRedirection();
            app.UseRouting();
            app.UseEndpoints(endpoints => endpoints.MapControllers());
        }
    }
}
```

Implementing commands and queries

- An HTTP GET endpoint at URLs of the form “/users/{userId}” that responds with a representation of the user.
- An HTTP POST endpoint at “/users/” that expects a representation of a user in the body of the request and then registers that user in the loyalty program.
- An HTTP PUT endpoint at URLs of the form “/users/{userId}” that expects a representation of a user in the body of the request and then updates an already registered user.

Implementing commands with HTTP POST or PUT

POST /users HTTP/1.1

Host: localhost:5001

Content-Type: application/json

Accept: application/json

{

 "name": "Christian",

 "loyaltyPoints": 0,

 "settings": { "interests" : ["whisky", "cycling"] }

}

```
using System;
using System.Collections.Generic;
using Microsoft.AspNetCore.Mvc;

[Route("/users")]
public class UsersController : ControllerBase
{
    [HttpPost("")]
    public ActionResult<LoyaltyProgramUser> CreateUser(
        [FromBody] LoyaltyProgramUser user)
    {
        if (user == null)
            return BadRequest();
        var newUser = RegisterUser(user);
        return Created(
            new Uri($"/users/{newUser.Id}", UriKind.Relative),
            newUser);
    }

    private LoyaltyProgramUser RegisterUser(LoyaltyProgramUser user)
    {
        // store the new user to a data store
    }
}
```

- The response to the preceding request looks like this:

HTTP/1.1 201 Created

Content-Type: application/json; charset=utf-8

Location: /users/4

```
{  
  "id": 4,  
  "name": "Christian",  
  "loyaltyPoints": 0,  
  "settings": { "interests": ["whisky", "cycling"] }  
}
```

```
private static readonly IDictionary<int, LoyaltyProgramUser>  
    RegisteredUsers = new Dictionary<int, LoyaltyProgramUser>();
```

```
[HttpPut("{userId:int}")]  
public LoyaltyProgramUser UpdateUser(  
    int userId,  
    [FromBody] LoyaltyProgramUser user)  
=> RegisteredUsers[userId] = user;
```

```
using System.Net.Http;
using System.Text;
using System.Threading.Tasks;
using System.Text.Json;

public class LoyaltyProgramClient
{
    private readonly HttpClient httpClient;

    public LoyaltyProgramClient(HttpClient httpClient)
    {
        this.httpClient = httpClient;
    }

    public async Task<HttpResponseMessage> RegisterUser(string name)
    {
        var user = new {name, Settings = new { }};
        return await this.httpClient.PostAsync("/users/",
            CreateBody(user));
    }

    private static StringContent CreateBody(object user)
    {
        return new StringContent(
            JsonSerializer.Serialize(user),
            Encoding.UTF8,
            "application/json");
    }
}
```

The API Gateway microservice updating users

```
public async Task<HttpResponseMessage> UpdateUser(dynamic user) =>  
    await this.httpClient.PutAsync(  
        $"'/users/{user.id}'",  
        CreateBody(user));
```

Implementing queries with HTTP GET

```
namespace LoyaltyProgram.Users
{
    using System;
    using System.Collections.Generic;
    using Microsoft.AspNetCore.Mvc;

    [Route("/users")]
    public class UsersController : ControllerBase
    {
        private static readonly Dictionary<int, LoyaltyProgramUser>
            RegisteredUsers = new();

        [HttpGet("{userId:int}")]
        public ActionResult<LoyaltyProgramUser> GetUser(int userId) =>
            RegisteredUsers.ContainsKey(userId)
                ? (ActionResult<LoyaltyProgramUser>) Ok(RegisteredUsers[userId])
                : NotFound();

        ...
    }
}
```

- There's nothing about this code that you haven't already seen several times.
- Likewise, the code needed in the API Gateway microservice to query this endpoint shouldn't come as a surprise:

```
public class LoyaltyProgramClient
{
    ...
    public async Task<HttpResponseMessage> QueryUser(string arg) =>
        await this.httpClient.GetAsync($"/users/{int.Parse(arg)}");
}
```

Implementing an event-based collaboration



Implementing an event feed

GET /events?start=10&end=110 HTTP/1.1

Host: localhost:5002

Accept: application/json

HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8

```
[  
  {  
    "sequenceNumber": 1,  
    "occurredAt": "2020-06-16T20:13:53.6678934+00:00",  
    "name": "SpecialOfferCreated",  
    "content": {  
      "description": "Best deal ever!!!",  
      "id": 0  
    }  
  },  
  {  
    "sequenceNumber": 2,  
    "occurredAt": "2020-06-16T20:14:22.6229836+00:00",  
    "name": "SpecialOfferCreated",  
    "content": {  
      "description": "Special offer - just for you",  
      "id": 1  
    }  
  },  
  {  
    "sequenceNumber": 3,  
    "occurredAt": "2020-06-16T20:14:39.841415+00:00",  
    "name": "SpecialOfferCreated",  
    "content": {  
      "description": "Nice deal",  
      "id": 2  
    }  
  },  
  {  
    "sequenceNumber": 4,  
    "occurredAt": "2020-06-16T20:14:47.3420926+00:00",  
    "name": "SpecialOfferUpdated",  
    "content": {  
      "oldOffer": {  
        "description": "Nice deal",  
        "id": 2  
      },  
      "newOffer": {  
        "description": "Best deal ever - JUST GOT BETTER",  
        "id": 0  
      }  
    }  
  },  
  {  
    "sequenceNumber": 5,  
    "occurredAt": "2020-06-16T20:14:51.8986625+00:00",  
    "name": "SpecialOfferRemoved",  
    "content": {  
      "offer": {  
        "description": "Special offer - just for you",  
        "id": 1  
      }  
    }  
  }]  
]
```

Endpoint that reads and returns events

```
namespace SpecialOffers.Events
{
    using System.Linq;
    using Microsoft.AspNetCore.Mvc;

    [Route("/events")]
    public class EventFeedController : ControllerBase
    {
        private readonly IEventStore eventStore;

        public EventFeedController(IEventStore eventStore)
        {
            this.eventStore = eventStore;
        }

        [HttpGet("")]
        public ActionResult<EventFeedEvent[]> GetEvents([FromQuery] int start,
            [FromQuery] int end)
        {
            if (start < 0 || end < start)
                return BadRequest();

            return this.eventStore.GetEvents(start, end).ToArray();
        }
    }
}
```

EventFeedEvent record that represents events

```
public record EventFeedEvent(  
    long SequenceNumber,  
    DateTimeOffset OccuredAt,  
    string Name,  
    object Content);ant to hold the event data.
```

Creating an event-subscriber process

We will implement this periodic polling as two main parts:

- A simple console application that reads one batch of events
- A Kubernetes CronJob to run the console application at intervals

- The first step in implementing an event-subscriber process is to create a console application with the following dotnet command:

```
PS> dotnet new console -n EventConsumer
```

- And run it with dotnet too:

```
PS> dotnet run
```

Subscribing to an event feed

```
using System;
using System.IO;
using System.Net.Http;
using System.Net.Http.Headers;
using System.Text.Json;
using System.Threading.Tasks;

var start = await GetStartIdFromDatastore();
var end = 100;
var client = new HttpClient();
client.DefaultRequestHeaders
    .Accept
    .Add(new MediaTypeWithQualityHeaderValue("application/json"));
using var resp = await client.GetAsync(
    new Uri($"http://special-offers:5002/events?start={start}&end={end}"));
await ProcessEvents(await resp.Content.ReadAsStreamAsync());
await SaveStartIdToDataStore(start);

Task<long> GetStartIdFromDatastore(){...}
async Task ProcessEvents(Stream content){...}
Task SaveStartIdToDataStore(long startId){...}
```

Deserializing and then handling events

```
async Task ProcessEvents(Stream content)
{
    var events =
        await JsonSerializer.DeserializeAsync<SpecialOfferEvent[]>(content)
        ?? new SpecialOfferEvent[0];
    foreach (var @event in events)
    {
        Console.WriteLine(@event);
        start = Math.Max(start, @event.SequenceNumber + 1);
    }
}
```

- The SpecialOfferEvent type used here is simple and contains only the fields used in the loyalty program:

```
public record SpecialOfferEvent(  
    long SequenceNumber,  
    DateTimeOffset OccuredAt,  
    string Name,  
    object Content);
```

Deploying to Kubernetes

- We will build Docker containers for both microservices.
- We will deploy the special offers microservice similarly to how we deployed to Kubernetes in Module 3.
- We will deploy the loyalty program, which will consist of deploying the loyalty program container in two copies of different configurations: one running as the API and one running as the event consumer.

Deploying to Kubernetes

With those three steps done we will have three parts running in Kubernetes:

- A pod for the special offers microservice
- A pod for the API part of the loyalty program microservice
- A CronJob for the event consumer part of the loyalty program

Building a Docker container special offers microservice

```
FROM mcr.microsoft.com/dotnet/sdk:5.0 AS build
WORKDIR /src
COPY . .
RUN dotnet restore "SpecialOffers.csproj"
WORKDIR "/src"
RUN dotnet build "SpecialOffers.csproj" -c Release -o /api/build

FROM build AS publish
WORKDIR "/src"
RUN dotnet publish "SpecialOffers.csproj" -c Release -o /api/publish

FROM mcr.microsoft.com/dotnet/aspnet:5.0 AS final
WORKDIR /app
EXPOSE 80
COPY --from=publish /api/publish ./api
ENTRYPOINT dotnet api/SpecialOffers.dll
```

Building a Docker container for both parts of the loyalty program

```
FROM mcr.microsoft.com/dotnet/sdk:5.0 AS build
WORKDIR /src
COPY . .
RUN dotnet restore "LoyaltyProgram/LoyaltyProgram.csproj"
RUN dotnet restore "EventConsumer/EventConsumer.csproj"
WORKDIR "/src/LoyaltyProgram"
RUN dotnet build "LoyaltyProgram.csproj" -c Release -o /api/build
WORKDIR "/src/EventConsumer"
RUN dotnet build "EventConsumer.csproj" -c Release -o /consumer/build/
    consumer

FROM build AS publish
WORKDIR "/src/LoyaltyProgram"
RUN dotnet publish "LoyaltyProgram.csproj" -c Release -o /api/publish
WORKDIR "/src/EventConsumer"
RUN dotnet publish "EventConsumer.csproj" -c Release -o /consumer/
    publish

FROM mcr.microsoft.com/dotnet/aspnet:5.0 AS final
WORKDIR /app
EXPOSE 80
COPY --from=publish /api/publish ./api
COPY --from=publish /consumer/publish ./consumer
ENTRYPOINT dotnet $STARTUPDLL
```

- Let's see this Dockerfile in action. First we build a loyalty program Docker image:

```
> docker build . -t loyalty-program
```

- Now we run the loyalty program image as either the API part or the event consumer part. Let's first try to run it as the API by passing in the path to the LoyaltyProgram .dll in the \$STARTUPDLL:

```
> docker run --rm -p 5001:80 -e STARTUPDLL="api/LoyaltyProgram.dll"  
loyalty-program
```

- Let's run the same Docker image again, but start it up as the Event Consumer:

```
> docker run --rm -e STARTUPDLL="consumer/EventConsumer.dll" loyalty-program
```

- We created a new Docker network called “microservices”:

```
> docker network create --driver=bridge microservices
```

- We will take advantage of that when we run the special offers microservice by adding it to the microservices network and giving it the name special-offers. To do just that, run this command:

```
> docker run --rm -p 5002:80 --network=microservices --name=special-offers  
special-offers
```

- Now we can rerun the event consumer, this time on the microservices network, and it will succeed:

```
docker run --rm -e STARTUPDLL="consumer/EventConsumer.dll"  
--network=microservices loyalty-program
```

Deploying the loyalty program API and the special offers

```
kind: Deployment
apiVersion: apps/v1
metadata:
  name: loyalty-program
spec:
  replicas: 1
  selector:
    matchLabels:
      app: loyalty-program
  template:
    metadata:
      labels:
        app: loyalty-program
    spec:
      containers:
        - name: loyalty-program
          image: your_unique_registry_name.azurecr.io/
            loyalty-program:1.0.2
          imagePullPolicy: IfNotPresent
          ports:
            - containerPort: 80
          env:
            - name: STARTUPDLL
              value: "api/LoyaltyProgram.dll"
---
apiVersion: v1
kind: Service
metadata:
  name: loyalty-program
spec:
  type: LoadBalancer
  ports:
    - name: loyalty-program
      port: 5001
      targetPort: 80
  selector:
    app: loyalty-program
```

- With this file in place, we are ready deploy the loyalty program API to Kubernetes with this command:

```
> kubectl apply -f loyalty-program.yaml
```

- Once the loyalty program API and the special offers microservice are deployed, we should see both running in the Kubernetes dashboard

```
> kubectl get pods
```

| NAME | READY | STATUS | RESTARTS | AGE |
|----------------------------------|-------|---------|----------|-----|
| loyalty-program-5d87df4656-9x89c | 1/1 | Running | 0 | 55s |
| special-offers-67d6b78998-mtpp6 | 1/1 | Running | 0 | 43s |

- We can also find the IP addresses and ports where the two APIs are available:

```
> kubectl get services
```

| NAME | TYPE | CLUSTER-IP | EXTERNAL-IP | PORT(S) | AGE |
|-----------------|--------------|--------------|---------------|----------------|-------|
| kubernetes | ClusterIP | 10.0.0.1 | <none> | 443/TCP | 20d |
| loyalty-program | LoadBalancer | 10.0.137.255 | 40.127.231.56 | 5001:32553/TCP | 2m28s |
| special-offers | LoadBalancer | 10.0.76.165 | 52.142.115.22 | 80:32391/TCP | 2m16s |

Deploy EventConsumer

```
---
apiVersion: batch/v1beta1
kind: CronJob
metadata:
  name: loyalty-program-consumer
spec:
  schedule: "*/1 * * * *"
  jobTemplate:
    spec:
      template:
        spec:
          containers:
            - name: loyalty-program
              image: your_unique_registry_name.azurecr.io/
                loyalty-program:1.0.2
              imagePullPolicy: IfNotPresent
              env:
                - name: STARTUPDLL
                  value: "consumer/EventConsumer.dll"
  restartPolicy: Never
  concurrencyPolicy: Forbid
```

- To deploy the event consumer, we use `loyalty-program.yaml` again:

```
> kubectl apply -f loyalty-program.yaml
```

- Now that the event consumer has also been deployed, we should see a CronJob in Kubernetes too:

```
> kubectl get cronjob
```

| NAME | SCHEDULE | SUSPEND | ACTIVE | LAST SCHEDULE | AGE |
|--------------------------|-------------|---------|--------|---------------|-------|
| loyalty-program-consumer | */1 * * * * | False | 0 | 20s | 1m20s |

- We can also see a list of recent invocations of the loyalty program event consumer by looking at the Kubernetes jobs list:

```
> kubectl get jobs
```

| NAME | COMPLETIONS | DURATION | AGE |
|-------------------------------------|-------------|----------|-------|
| loyalty-program-consumer-1590755940 | 1/1 | 2s | 11s |
| loyalty-program-consumer-1590757080 | 1/1 | 2s | 1m11s |
| loyalty-program-consumer-1590757140 | 1/1 | 3s | 2m11s |

If we want to convince ourselves that the event consumer is indeed running, we can inspect the logs of one of the instantiations of the CronJob:

```
> kubectl logs job.batch/loyalty-program-consumer-1590755940
[{"sequenceNumber":1,"occurredAt":"2020-06-18T18:07:13.7973414+00:00","name":
"SpecialOfferCreated","content":{"description":"Nice deal","id":0}},
 {"sequenceNumber":2,"occurredAt":"2020-06-18T18:07:17.7957514+00:00",
 "name":"SpecialOfferCreated","content":{"description":"Nice deal","id":1}},
 {"sequenceNumber":3,"occurredAt":"2020-06-18T18:07:47.4246091+00:00","name":
"SpecialOfferUpdated","content":{"oldOffer":{"description":"Nice deal",
 "id":1}, "newOffer":{"description":"Best deal ever - JUST GOT BETTER","id":0}}}, {"sequenceNumber":4,"occurredAt":"2020-06-18T18:08:04.5908816+00:00",
 "name":"SpecialOfferRemoved","content":{"offer":{"description":"Nice deal",
 "id":0}}}]
```

Summary

- Event-based collaboration is more loosely coupled than command- and query-based collaboration.
- You can use HttpClient to send commands to other microservices and to query other microservices.
- You can use MVC controllers to expose the endpoints for receiving and handling commands and queries.
- An MVC controller can expose a simple event feed.

6. Data ownership and data storage

Introduction

This Module covers

- Which data microservices store
- Understanding how data ownership follows business capabilities
- Using data replication for speed and robustness
- Building read models from event feeds with event subscribers
- Implementing data storage in microservices

Each microservice has a data store

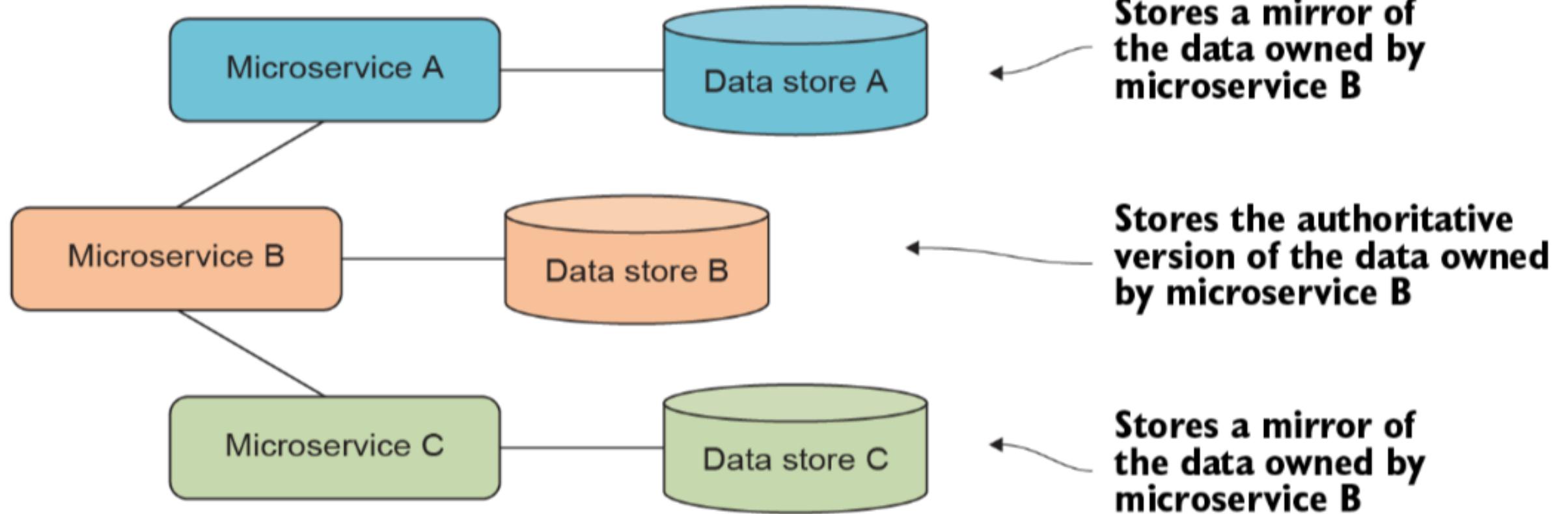
A microservice typically needs to store three types of data:

- Data belonging to the capability the microservice implements.
- Events raised by the microservice. During command processing, the microservice may need to raise events to inform the rest of the system about updates to the data the microservice is responsible for.
- Read models based on data in events from other microservices or occasionally on data from queries to other microservices.

Partitioning data between microservices

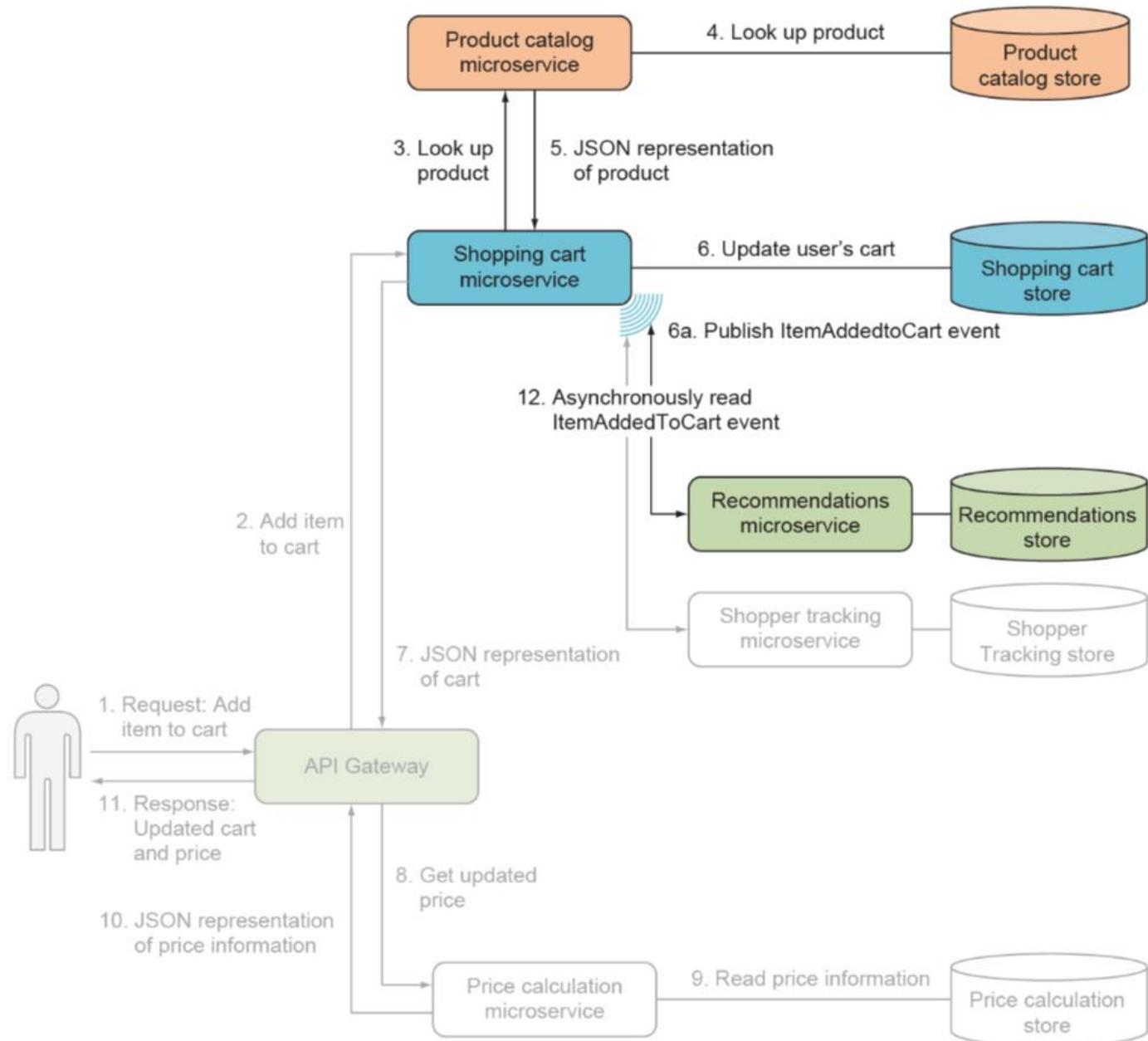
When you're deciding where to store data in a microservice system, competing forces are at play. The two main forces are data ownership and locality:

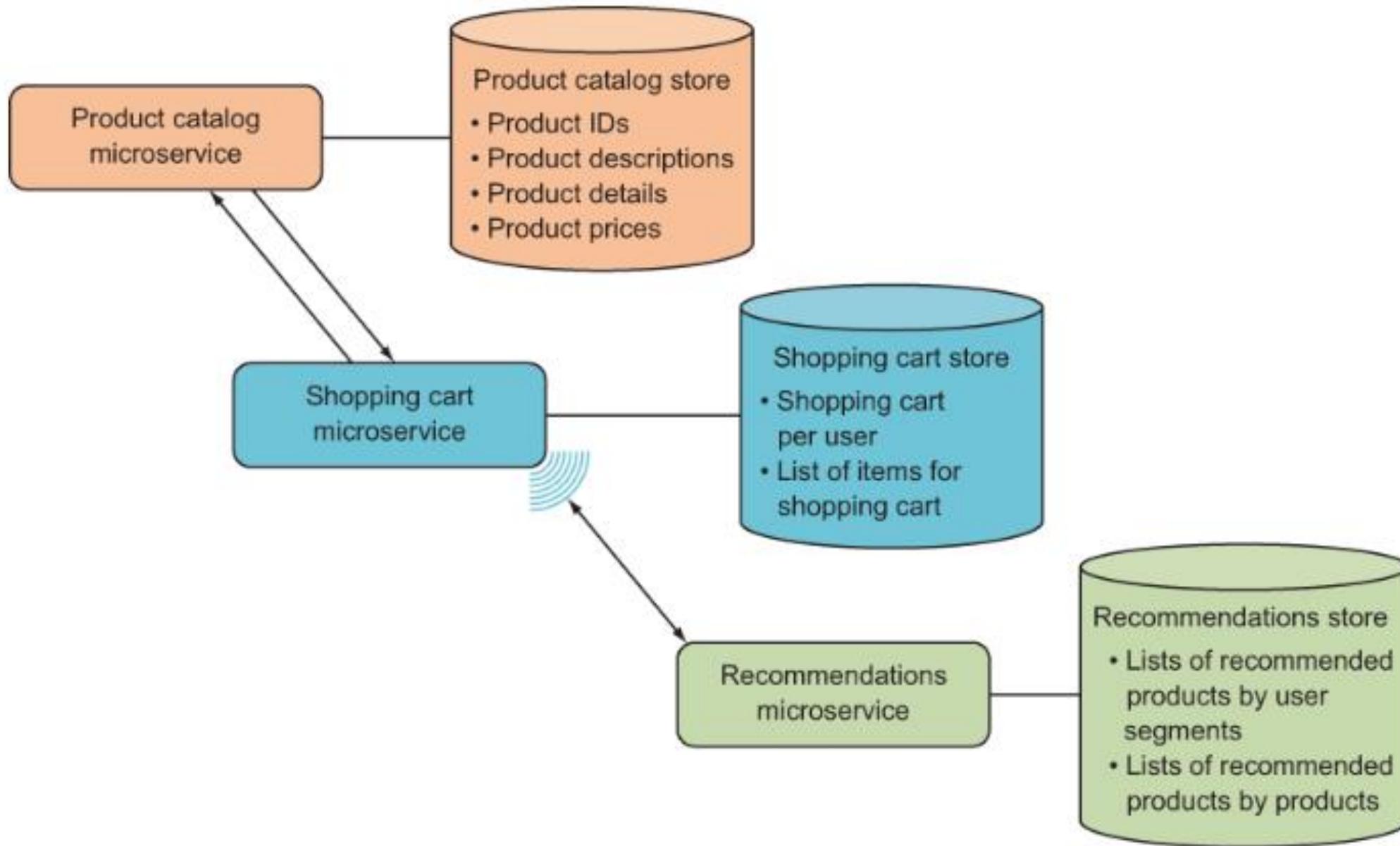
- *Ownership of data*
- *Locality of data*



Rule 1: Ownership of data follows business capabilities

- The first rule when deciding where a piece of data belongs in a microservices system is that ownership of data follows business capabilities.
- As discussed in Module 4, the primary driver in deciding on the responsibility of a microservice is that it should handle a business capability.
- The business capability defines the boundaries of the microservice—everything belonging to the capability should be implemented in the microservice.
- This includes storing the data that falls under the business capability.





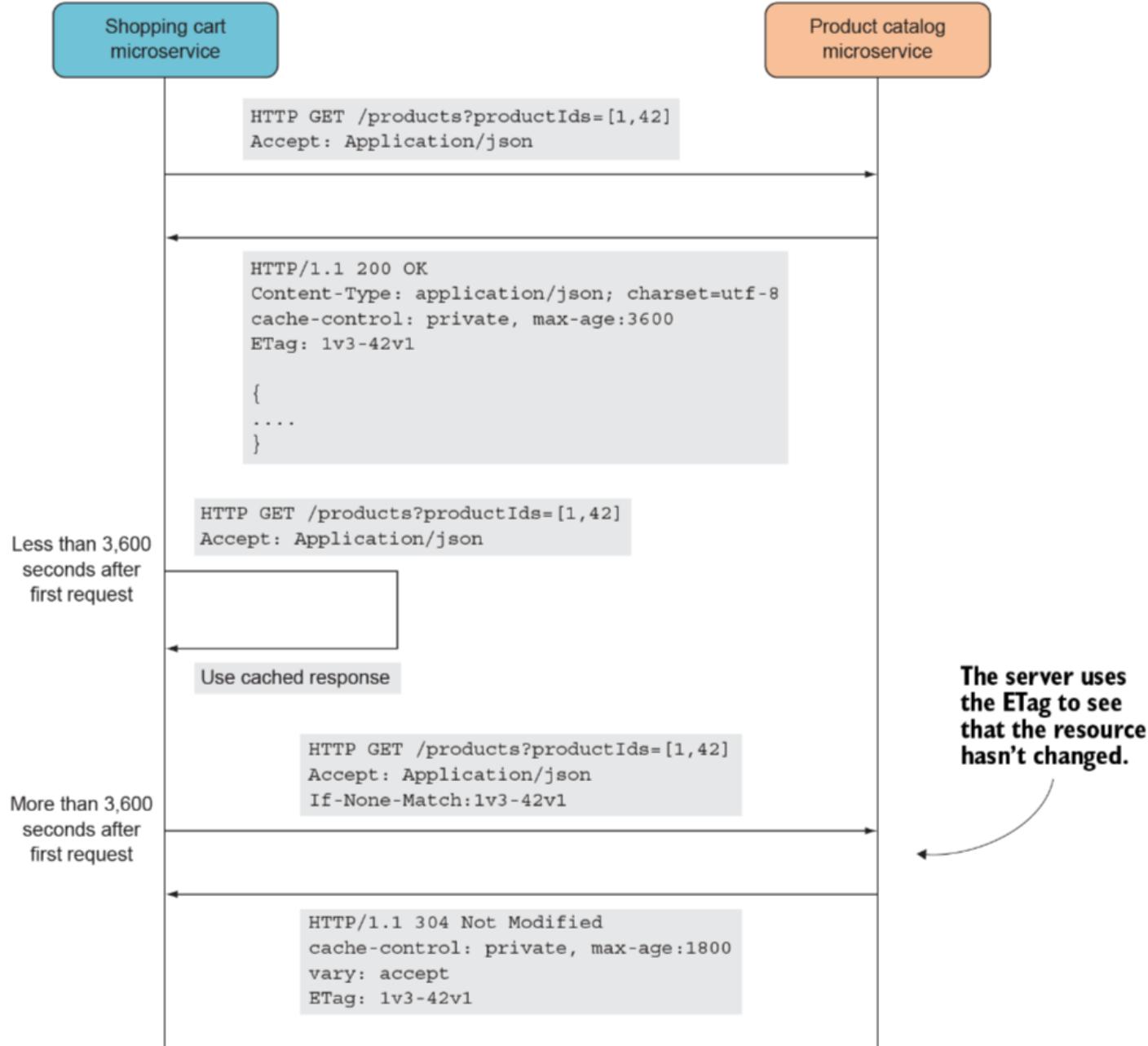
Rule 2: Replicate for speed and robustness

- The second force at play when deciding where a piece of data should be stored in a microservices system is locality.
- There's a big difference between a microservice querying its own database for data and a microservice querying another microservice for that same data.
- Querying its own database is generally both faster and more reliable than querying another microservice.

Using HTTP cache headers to control caching

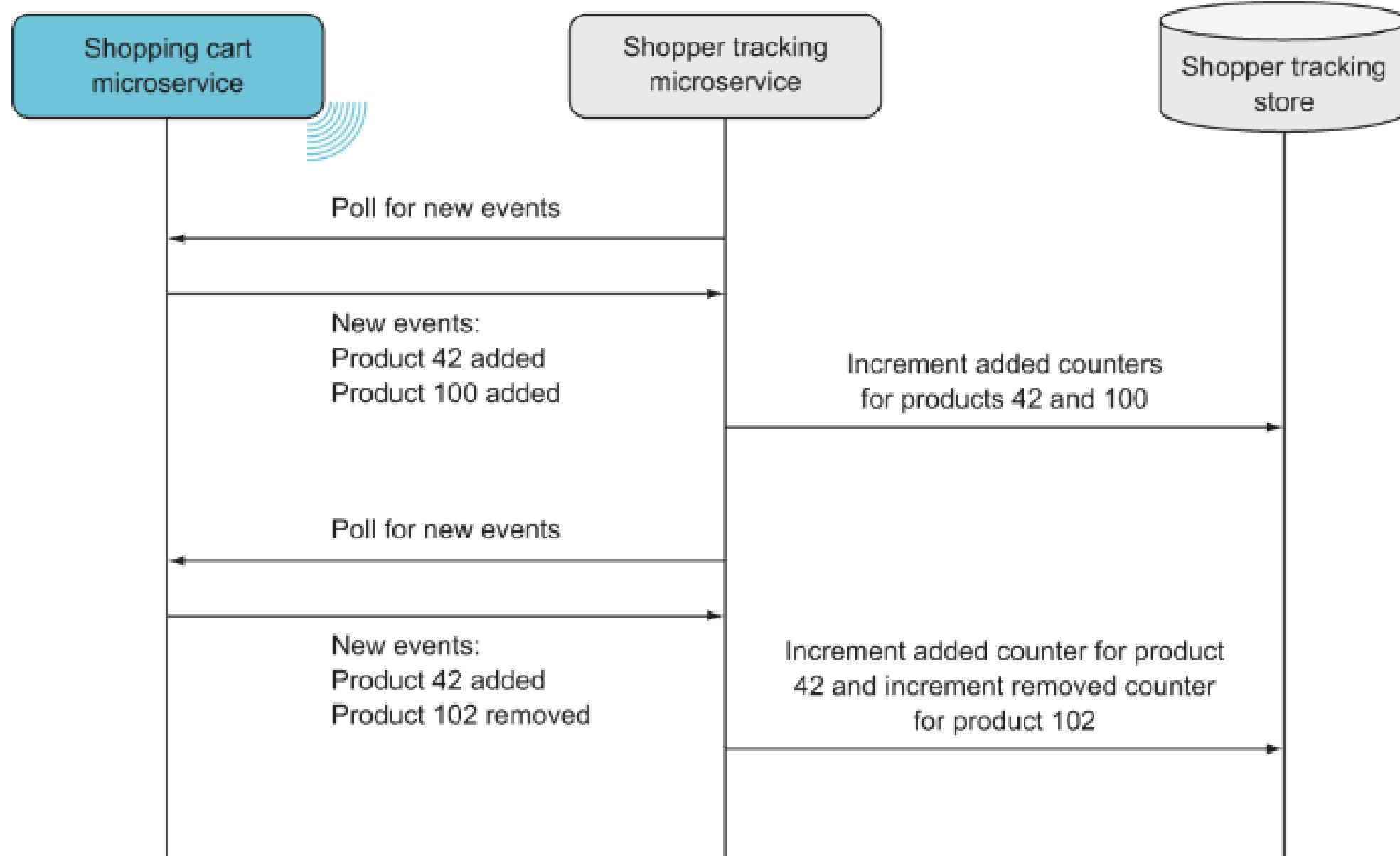
HTTP defines a number of headers that can be used to control how HTTP responses can be cached. The purpose of the HTTP caching mechanisms is twofold:

- To eliminate the need, in many cases, to request information the caller already has
- To eliminate the need, in some other situations, to send full HTTP responses



Using read models to mirror data owned by other microservices

- Read models are often based on events from other microservices.
- One microservice subscribes to events from another microservice and updates its own model of the event data as events arrive.
- Read models can also be built from responses to queries to other microservices.
- In this case, the lifetime of the data in the read model is decided by the cache headers on those responses, just as in a straight cache of the responses.



Where does a microservice store its data?

The choice of database technology (or technologies) for a microservice can be influenced by many factors, including these:

- What shape is your data? Does it fit well into a relational model, a document model, or a key/value store, or is it a graph?
- What are the write scenarios? How much data is written? Do the writes come in bursts, or are they evenly distributed over time?
- What are the read scenarios? How much data is read at a time? How much is read altogether? Do the reads come in bursts?
- How much data is written compared to how much is read?

HOW MANY DATABASE TECHNOLOGIES IN THE SYSTEM?

On the side of standardizing are goals like these:

- Running the databases reliably in production and continuing to do so in the long run.
- Developers being able to get into and work effectively in the codebase of a microservice they haven't touched before.

Implementing data storage in a microservice

- We've discussed where data should go in a microservice system, including which data a microservice should own and which it should mirror.
- It's time to switch gears and look at the code required to store the data.

DAPPER: A LIGHTWEIGHT O/RM

- Dapper is a simple library for working with data in a SQL database.
- It's part of a family of libraries sometimes referred to as *micro ORMs*, which also includes Simple.Data and Massive.
- These libraries focus on being simple to use and fast, and they embrace SQL.

EVENTSTOREDB: A DEDICATED EVENT DATABASE

- EventStoreDB is an open source database server designed specifically for storing events.
- EventStoreDB stores events as JSON documents, but it differs from a document database by assuming that the JSON documents are part of a stream of events.
- Although EventStoreDB is a niche product because it's so narrowly focused on storing events, it's in widespread use and has proven itself in heavy-load production scenarios.

Preparing a development setup

- First, pull down the latest SQL Server docker image to your machine

```
docker pull mcr.microsoft.com/mssql/server
```

- and then run it:

```
docker run -e 'ACCEPT_EULA=Y' -e 'SA_PASSWORD=yourStrong(!)Password' -p  
1433:1433 -d mcr.microsoft.com/mssql/server
```

Preparing a development setup

- Last, confirm that SQL is indeed running by listing the locally running container and checking that SQL Server is in the list:

```
docker ps
```

| CONTAINER ID | IMAGE | COMMAND | CREATED | STATUS | PORTS | NAMES |
|--------------|--------------------------------|--------------------------|----------------|---------------|------------------------|---------------|
| 3388b710892f | mcr.microsoft.com/mssql/server | "/opt/mssql/bin/perm..." | 52 minutes ago | Up 52 minutes | 0.0.0.0:1433->1433/tcp | friendly_bell |

Storing data owned by a microservice

If you're familiar with storing data in SQL Server, the implementation should be no surprise, and that's the point. Storing the data owned by a microservice doesn't need to involve anything fancy. These are the steps for storing the shopping cart:

- Create a database.
- Use Dapper to implement the code to read, write, and update shopping carts.

A diagram illustrating a relationship between two database tables: **ShoppingCart** and **ShoppingCartItem**. An arrow points from the **ShoppingCartItem** table to the **ShoppingCart** table, indicating that **ShoppingCartItem** references **ShoppingCart**.

ShoppingCart

| ShoppingCart | |
|--------------|-----------|
| PK | <u>ID</u> |
| | UserId |

ShoppingCartItem

| ShoppingCartItem | |
|------------------|--------------------|
| PK | <u>ID</u> |
| FK | ShoppingCartId |
| | ProductCatalogId |
| | ProductName |
| | ProductDescription |
| | Amount |
| | Currency |

```
CREATE DATABASE ShoppingCart
GO

USE [ShoppingCart]
GO

CREATE TABLE [dbo].[ShoppingCart](
    [ID] int IDENTITY(1,1) PRIMARY KEY,
    [UserId] [bigint] NOT NULL,
    CONSTRAINT ShoppingCartUnique UNIQUE([ID], [UserID])
)
GO

CREATE INDEX ShoppingCart_UserId
ON [dbo].[ShoppingCart] (UserId)
GO

CREATE TABLE [dbo].[ShoppingCartItem](
    [ID] int IDENTITY(1,1) PRIMARY KEY,
    [ShoppingCartId] [int] NOT NULL,
    [ProductCatalogId] [bigint] NOT NULL,
    [ProductName] [nvarchar](100) NOT NULL,
    [ProductDescription] [nvarchar](500) NULL,
    [Amount] [int] NOT NULL,
    [Currency] [nvarchar](5) NOT NULL
)
GO

ALTER TABLE [dbo].[ShoppingCartItem] WITH CHECK ADD CONSTRAINT [
    FK_ShoppingCart] FOREIGN KEY([ShoppingCartId])
REFERENCES [dbo].[ShoppingCart] ([Id])
GO

ALTER TABLE [dbo].[ShoppingCartItem] CHECK CONSTRAINT [FK_ShoppingCart]
GO

CREATE INDEX ShoppingCartItem_ShoppingCartId
ON [dbo].[ShoppingCartItem] (ShoppingCartId)
GO
```

```
<ItemGroup>
  <PackageReference Include="Dapper" Version="2.0.35" />
  <PackageReference Include="Microsoft.Extensions.Http.Polly" Version="3.1.0" />
  <PackageReference Include="Polly" Version="7.2.0" />
  <PackageReference Include="Scrutor" Version="3.1.0" />
</ItemGroup>
```

- You'll change that interface slightly to allow the implementation of it to make asynchronous calls to the database. This is the modified interface:

```
public interface IShoppingCartStore
{
    Task<ShoppingCart> Get(int userId);
    Task Save(ShoppingCart shoppingCart);
}
```

```

namespace ShoppingCart.ShoppingCart
{
    using System.Data;
    using System.Data.SqlClient;
    using System.Linq;
    using System.Threading.Tasks;
    using Dapper;

    public interface IShoppingCartStore
    {
        Task<ShoppingCart> Get(int userId);
        Task Save(ShoppingCart shoppingCart);
    }

    public class ShoppingCartStore : IShoppingCartStore
    {
        private string connectionString =
            @"Data Source=localhost;Initial Catalog=ShoppingCart;
User Id=SA; Password=yourStrong(!)Password";

        private const string readItemsSql =
            @"
select ShoppingCart.ID, ProductCatalogId,
ProductName, ProductDescription, Currency, Amount
from ShoppingCart, ShoppingCartItem
where ShoppingCartItem.ShoppingCartId = ShoppingCart.ID
and ShoppingCart.UserId=@UserId";

        public async Task<ShoppingCart> Get(int userId)
        {
            await using var conn = new SqlConnection(this.connectionString);
            var items = (await
                conn.QueryAsync(
                    readItemsSql,
                    new { UserId = userId }))
                .ToList();
            return new ShoppingCart(
                items.FirstOrDefault()?.ID,
                userId,
                items.Select(x =>
                    new ShoppingCartItem(
                        (int) x.ProductCatalogId,
                        x.ProductName,
                        x.ProductDescription,
                        new Money(x.Currency, x.Amount))));

        }
    }
}

```

```
private const string insertShoppingCartSql =
@"insert into ShoppingCart (UserId) OUTPUT inserted.ID VALUES (@UserId);"

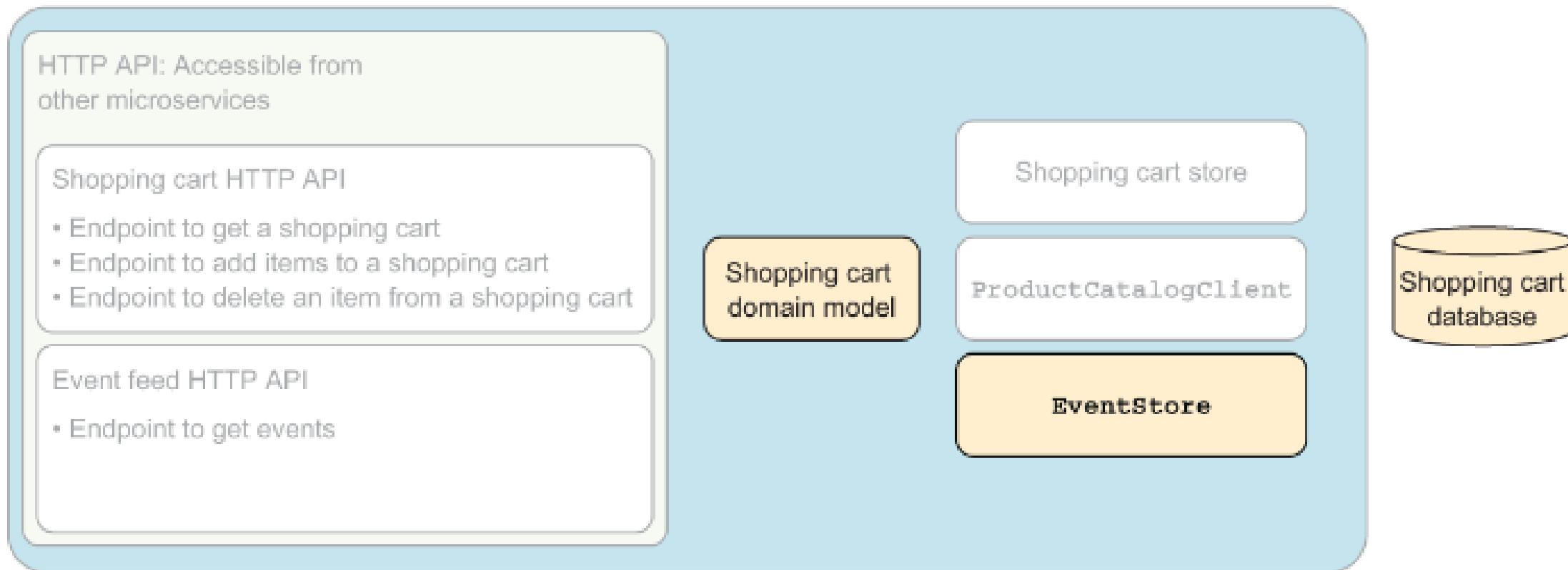
private const string deleteAllForShoppingCartSql =
@"delete item from ShoppingCartItem item
inner join ShoppingCart cart on item.ShoppingCartId = cart.ID
and cart.UserId=@UserId";

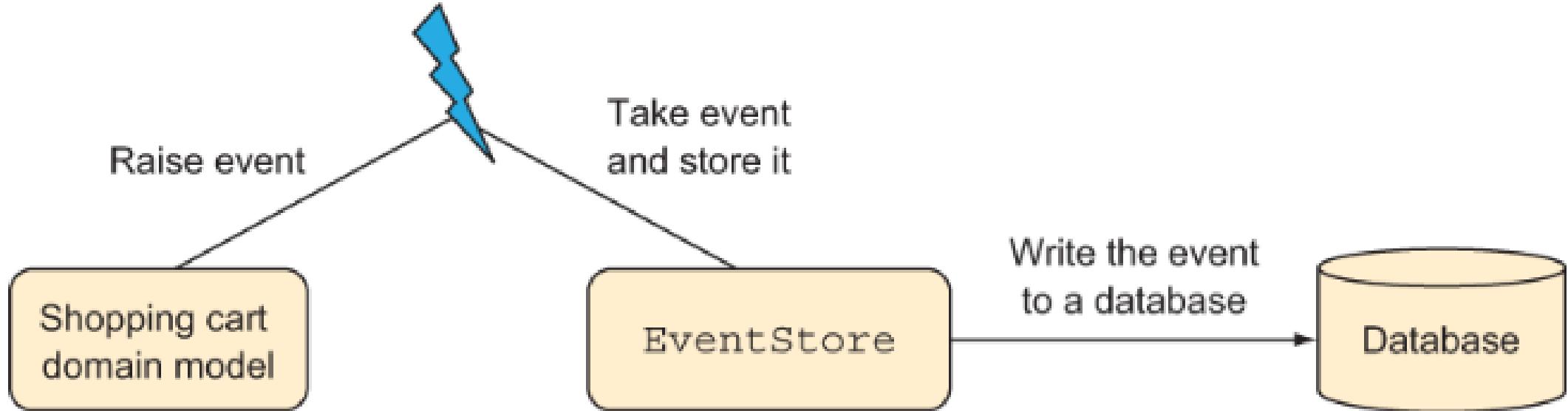
private const string addAllForShoppingCartSql =
@"insert into ShoppingCartItem
(ShoppingCartId, ProductCatalogId, ProductName,
ProductDescription, Amount, Currency)
values
(@ShoppingCartId, @ProductCatalogId, @ProductName,
@ProductDescription, @Amount, @Currency)";

public async Task Save(ShoppingCart shoppingCart)
{
    await using var conn = new SqlConnection(this.connectionString);
    await conn.OpenAsync();
    await using (var tx = conn.BeginTransaction())
    {
        var shoppingCartId =
            shoppingCart.Id ??
            await conn.QuerySingleAsync<int>(
                insertShoppingCartSql,
                new {shoppingCart.UserId}, tx);

        await conn.ExecuteAsync(
            deleteAllForShoppingCartSql,
            new {UserId = shoppingCart.UserId},
            tx);
        await conn.ExecuteAsync(
            addAllForShoppingCartSql,
            shoppingCart.Items.Select(x =>
            new
            {
                shoppingCartId,
                x.ProductCatalogId,
                Productdescription = x.Description,
                x.ProductName,
                x.Price.Amount,
                x.Price.Currency
            }),
            tx);
        await tx.CommitAsync();
    }
}
```

Storing events raised by a microservice





Storing events by hand

The following steps are involved in implementing the EventStore component:

- Add an EventStore table to the ShoppingCart database. This table will contain a row for every event raised by the domain model.
- Use Dapper to implement the writing part of the EventStore component.
- Use Dapper to implement the reading part of the EventStore component.

- Before we dive into implementing the EventStore component, here's a reminder of what the Event type in the shopping cart looks like.

```
public record Event(  
    long SequenceNumber,  
    DateTimeOffset OccuredAt,  
    string Name,  
    object Content);
```

| EventStore | |
|------------|-----------|
| PK | ID |
| | Name |
| | OccuredAt |
| | Content |

```
namespace ShoppingCart.EventFeed
{
    using System;
    using System.Collections.Generic;
    using System.Data.SqlClient;
    using System.Text.Json;
    using System.Threading.Tasks;
    using Dapper;

    public interface IEventStore
    {
        Task<IEnumerable<Event>> GetEvents(long firstEventSequenceNumber,
            long lastEventSequenceNumber);
        Task Raise(string eventName, object content);
    }

    public class EventStore : IEventStore
    {
        private string connectionString =
            @"Data Source=localhost;Initial Catalog=ShoppingCart;
User Id=SA; Password=yourStrong(!)Password";

        private const string writeEventSql =
@"insert into EventStore(Name, OccurredAt, Content)
values (@Name, @OccurredAt, @Content)";

        public async Task Raise(string eventName, object content)
        {
            var jsonContent = JsonSerializer.Serialize(content);
            await using var conn = new SqlConnection(this.connectionString);
            await conn.ExecuteAsync(
                writeEventSql,
                new
                {
                    Name = eventName,
                    OccurredAt = DateTimeOffset.Now,
                    Content = jsonContent
                });
        }
    }
}
```

EventStore method for reading events

```
private const string readEventsSql =  
    @"select * from EventStore where ID >= @Start and ID <= @End";  
  
public async Task<IEnumerable<Event>> GetEvents(  
    long firstEventSequenceNumber,  
    long lastEventSequenceNumber)  
{  
    await using var conn = new SqlConnection(this.connectionString);  
    return await conn.QueryAsync<Event>(  
        readEventsSql,  
        new  
        {  
            Start = firstEventSequenceNumber,  
            End = lastEventSequenceNumber  
        });  
}
```

Storing events using the EventStoreDB system

You'll implement this version with the following steps. When you're finished, you'll have a fully working implementation of the EventStore component in the shopping cart microservice based on the EventStoreDB:

- Run EventStoreDB in a Docker container.
- Write events to EventStoreDB via the EventStore component.
- Read events from EventStoreDB via the EventStore component.

- You can pull the EventStoreDB Docker image down with this command:

```
docker pull eventstore/eventstore
```

- Once pulled down you can run the EventStoreDB container like this:

```
docker run --name eventstore-node -it -p 2113:2113 -p 1113:1113 --rm  
eventstore/eventstore:latest --run-projections All --enable-external-tcp  
--enable-atom-pub-over-http
```

Storing events to the EventStoreDB

```
namespace ShoppingCart.EventFeed
{
    using System;
    using System.Collections.Generic;
    using System.Linq;
    using System.Text;
    using System.Text.Json;
    using System.Threading.Tasks;
    using EventStore.ClientAPI;

    public class EsEventStore : IEventStore
    {
        private const string ConnectionString =
            "tcp://admin:changeit@localhost:1113";

        public async Task Raise(string eventName, object content)
        {
            using var connection =
                EventStoreConnection.Create(
                    ConnectionSettings.Create().DisableTls().Build(),
                    new Uri(ConnectionString));
            await connection.ConnectAsync();
            await connection.AppendToStreamAsync(
                "ShoppingCart",
                ExpectedVersion.Any,
                new EventData(
                    Guid.NewGuid(),
                    "ShoppingCartEvent",
                    isJson: true,
                    data: Encoding.UTF8.GetBytes(
                        JsonSerializer.Serialize(content)),
                    metadata: Encoding.UTF8.GetBytes(
                        JsonSerializer.Serialize(new EventMetadata
                        (
                            OccuredAt = DateTimeOffset.UtcNow,
                            EventName = eventName
                        )))));
        }

        public record EventMetadata(
            DateTimeOffset OccuredAt,
            string EventName);
    }
}
```

Reading events from the EventStoreDB

```
public async Task<IEnumerable<Event>>
GetEvents(long firstEventSequenceNumber, long lastEventSequenceNumber)
{
    using var connection =
        EventStoreConnection.Create(
            ConnectionSettings.Create().DisableTls().Build(),
            new Uri(ConnectionString));
    await connection.ConnectAsync();
    var result = await connection.ReadStreamEventsForwardAsync(
        "ShoppingCart",
        start: firstEventSequenceNumber,
        count: (int) (lastEventSequenceNumber - firstEventSequenceNumber),
        resolveLinkTos: false);
    return result.Events
        .Select(e =>
        {
            new
            {
                Content = Encoding.UTF8.GetString(e.Event.Data),
                Metadata = JsonSerializer.Deserialize<EventMetadata>(
                    Encoding.UTF8.GetString(e.Event.Metadata),
                    new JsonSerializerOptions
                    {
                        PropertyNameCaseInsensitive = true
                    })
            }
        })
        .Select((e, i) =>
        new Event(
            i + firstEventSequenceNumber,
            e.Metadata.OccuredAt,
            e.Metadata.EventName,
            e.Content));
}
```

Setting cache headers in HTTP responses

```
namespace ProductCatalog
{
    using System.Collections.Generic;
    using System.Linq;
    using Microsoft.AspNetCore.Mvc;

    [Route("/products")]
    public class ProductCatalogController : ControllerBase
    {
        private readonly IProductStore productStore;

        public ProductCatalogController(IProductStore productStore) =>
            this.productStore = productStore;

        [HttpGet("")]
        [ResponseCache(Duration = 86400)]
        public IEnumerable<ProductCatalogProduct> Get(
            [FromQuery] string productIds)
        {
            var products = this.productStore.GetProductsByIds(
                ParseProductIdsFromQueryString(productIds));
            return products;
        }

        private static IEnumerable<int>
            ParseProductIdsFromQueryString(string productIdsString) => ...
    }
}
```

This implementation adds a cache-control header to the response that looks like this:

cache-control: public,max-age:86400

Reading and using cache headers

```
private async Task<HttpResponseMessage>
    RequestProductFromProductCatalog(int[] productCatalogIds)
{
    var productsResource =
        string.Format(getProductPathTemplate,
            string.Join(",", productCatalogIds));
    return await
        this.client.GetAsync(productsResource);
}
```

```
private readonly HttpClient client;  
private readonly ICache cache;  
private static string productCatalogueBaseUrl = @"https://git.io/JeHiE";  
private static string getProductPathTemplate = "?productIds=[{0}]";  
  
public ProductCatalogClient(HttpClient client, ICache cache)  
{  
    client.BaseAddress = new Uri(productCatalogueBaseUrl);  
    client.DefaultRequestHeaders.Accept.Add(  
        new MediaTypeWithQualityHeaderValue("application/json"));  
    this.client = client;  
    this.cache = cache;  
}
```

In the code download, you can find a simple static cache implementing the interface. The interface is straightforward and has two methods:

```
public interface ICache  
{  
    void Add(string key, object value, TimeSpan ttl);  
    object Get(string key);  
}
```

```
private async Task<HttpResponseMessage>
    RequestProductFromProductCatalog(int[] productCatalogIds)
{
    var productsResource = string.Format(
        getProductPathTemplate,
        string.Join(",", productCatalogIds));
    var response =
        this.cache.Get(productsResource) as HttpResponseMessage;
    if (response is null)
    {
        response = await this.client.GetAsync(productsResource);
        AddToCache(productsResource, response);
    }
    return response;
}

private void AddToCache(string resource, HttpResponseMessage response)
{
    var cacheHeader = response
        .Headers
        .FirstOrDefault(h => h.Key == "cache-control");
    if (!string.IsNullOrEmpty(cacheHeader.Key))
        && CacheControlHeaderValue.TryParse(
            cacheHeader.Value.ToString(), out var cacheControl)
        && cacheControl.MaxAge.HasValue)
        this.cache.Add(resource, response, cacheControl.MaxAge.Value);
}
```

Summary

- A microservice stores and owns all the data that belongs to the capability the microservice implements.
- A microservice is the authoritative source for the data it owns.
- A microservice stores its data in its own dedicated database.
- A simple version of an event store involves storing events to a table in a SQL database.
- Storing events is essentially a matter of storing a serialized event to a database.

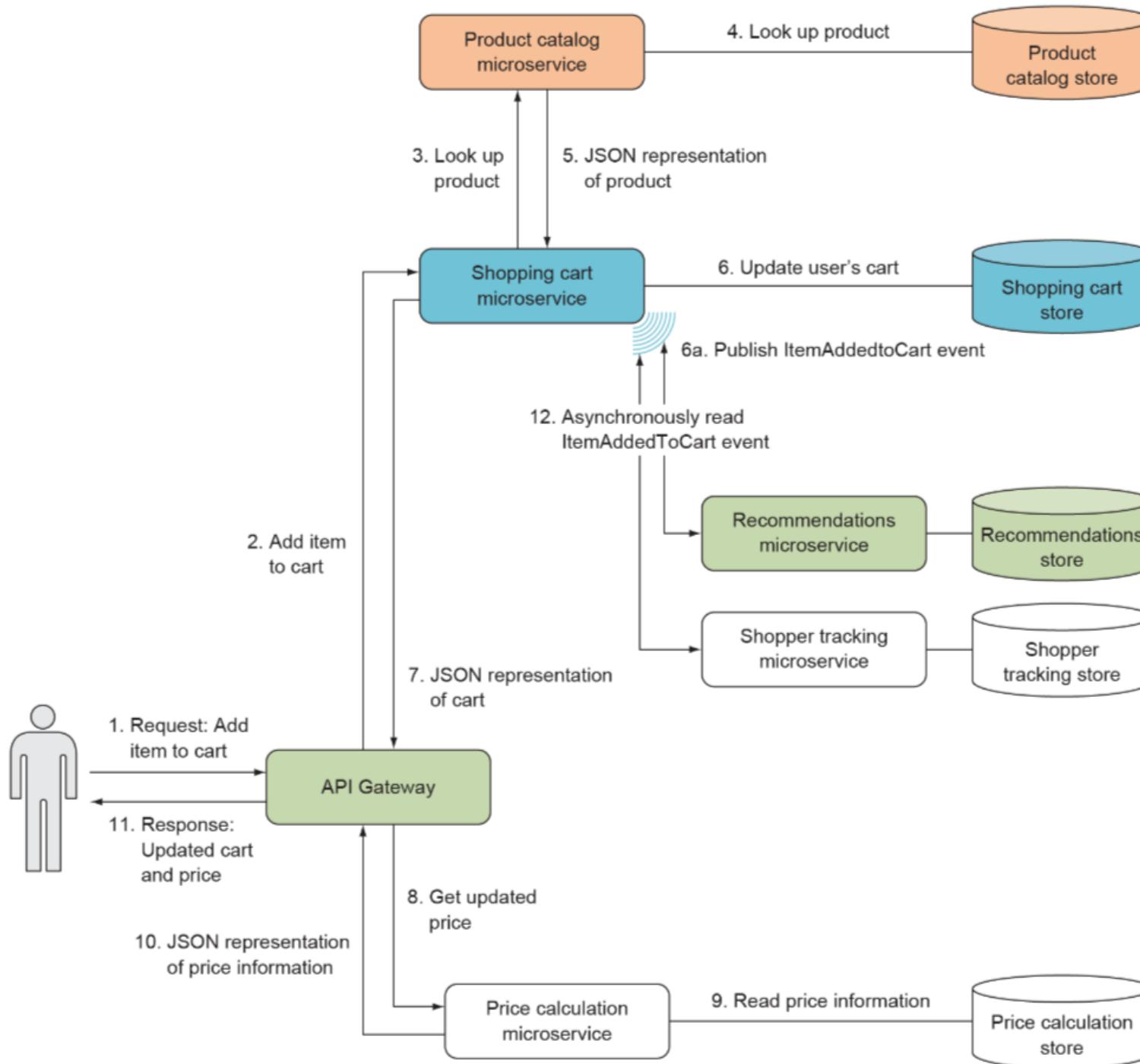
7. Designing for robustness

Introduction

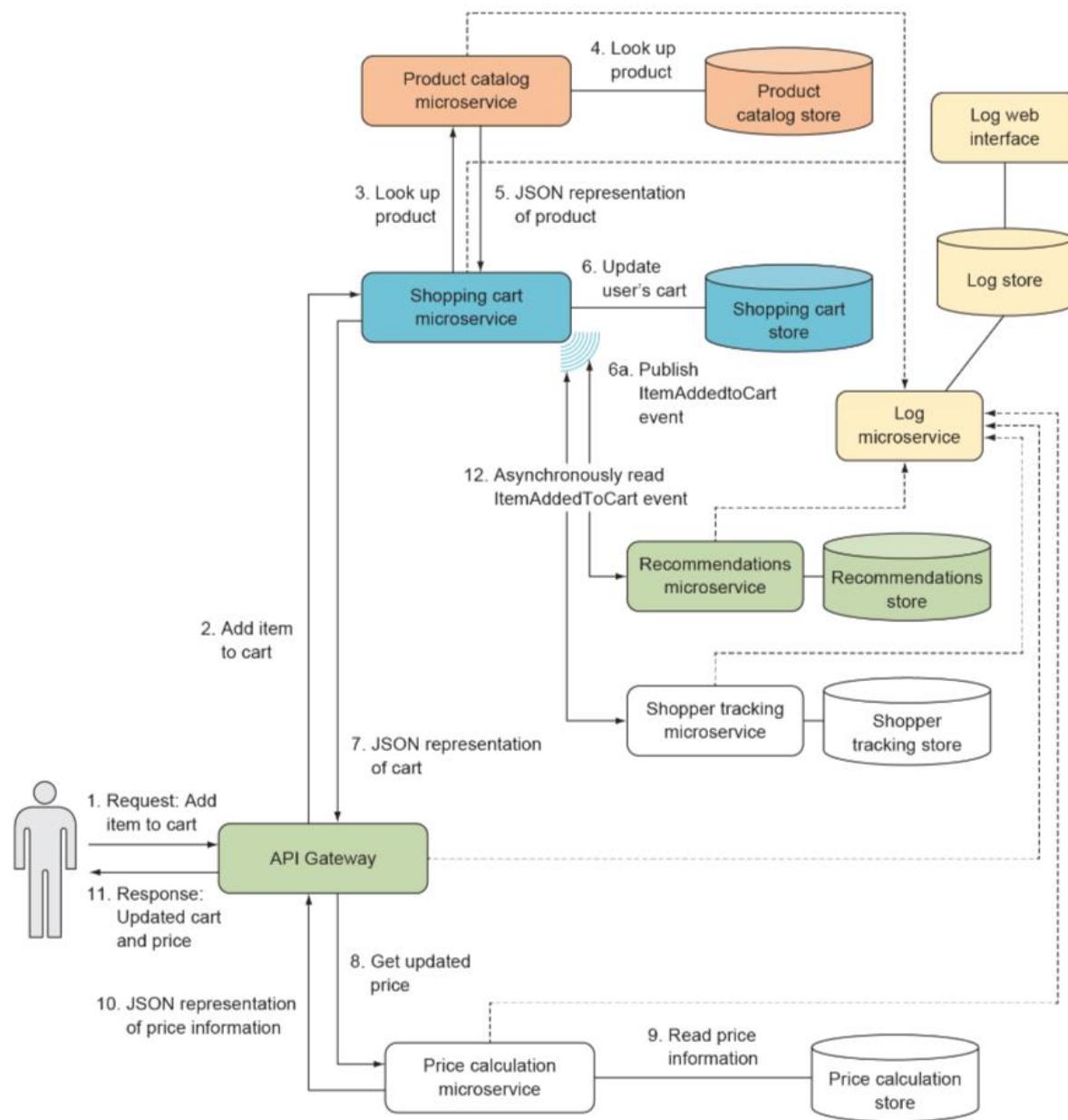
This Module covers

- Communicating robustly between microservices
- Letting the calling side take responsibility for robustness in the face of failure
- Rolling back versus rolling forward
- Implementing robust communication

Expect failures



Keeping good logs



Using trace IDs

- To find all log messages related to a particular action in the system, you can use *trace IDs*.
- A trace ID is an identifier attached, for example, to a request from an end user when it comes into the system.
- The trace ID is passed along from microservice to microservice in any communication that stems from that end user request.

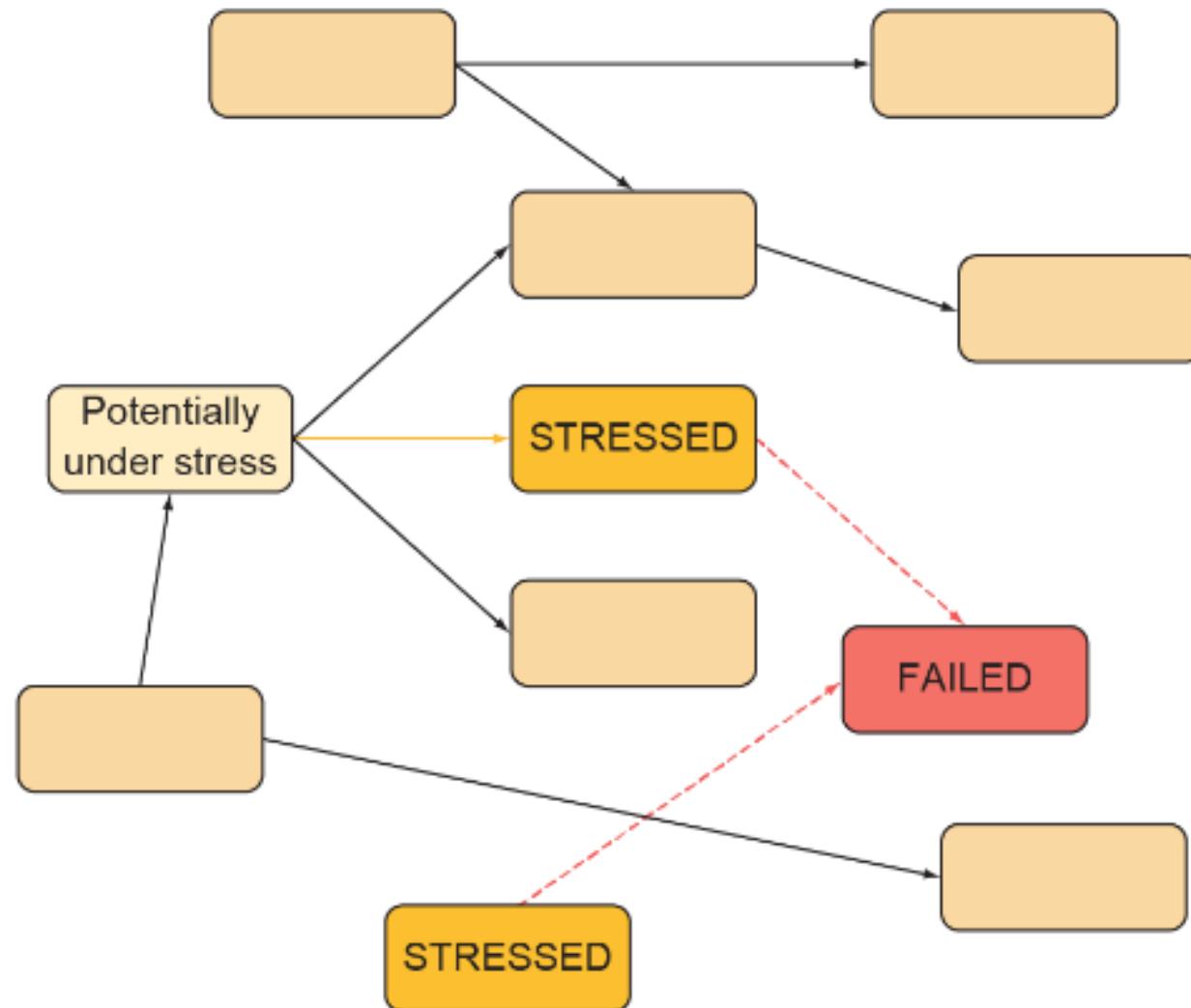
Rolling forward vs. rolling backward

- When errors happen in production, you're faced with the question of how to fix them.
- In many traditional systems, if errors begin to occur shortly after deployment, the default response is to roll back to the previous version of the system.
- In a microservice system, the default can be different. As discussed in Module 1, microservices lend themselves to continuous delivery.
- With continuous delivery, microservices are deployed frequently, and each deployment should be both fast and easy to perform.

Don't propagate failures

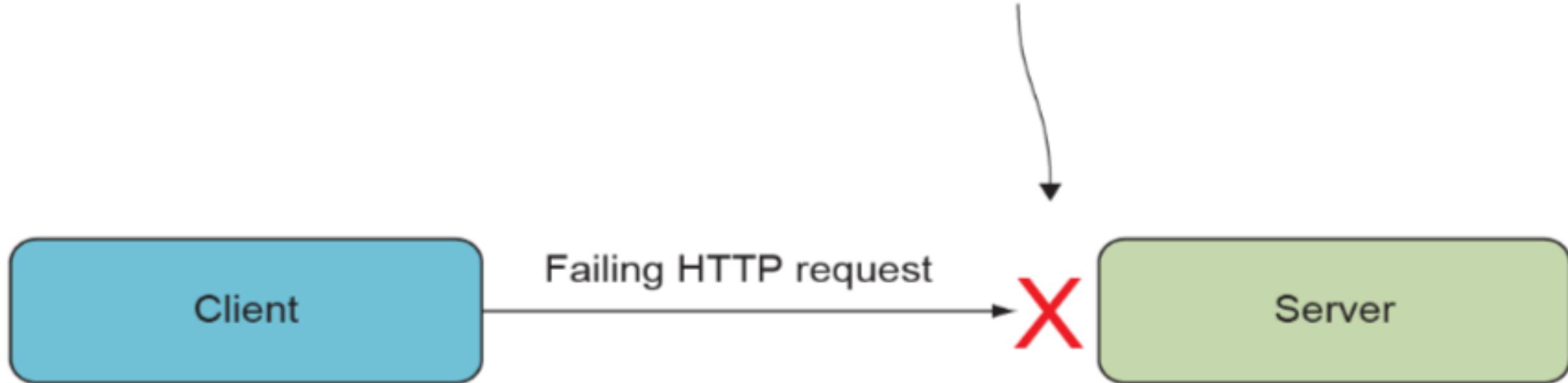
Sometimes, things happen around a microservice that may disturb its normal operation. We say that the microservice is *under stress* in such situations. There are many sources of stress, including the following:

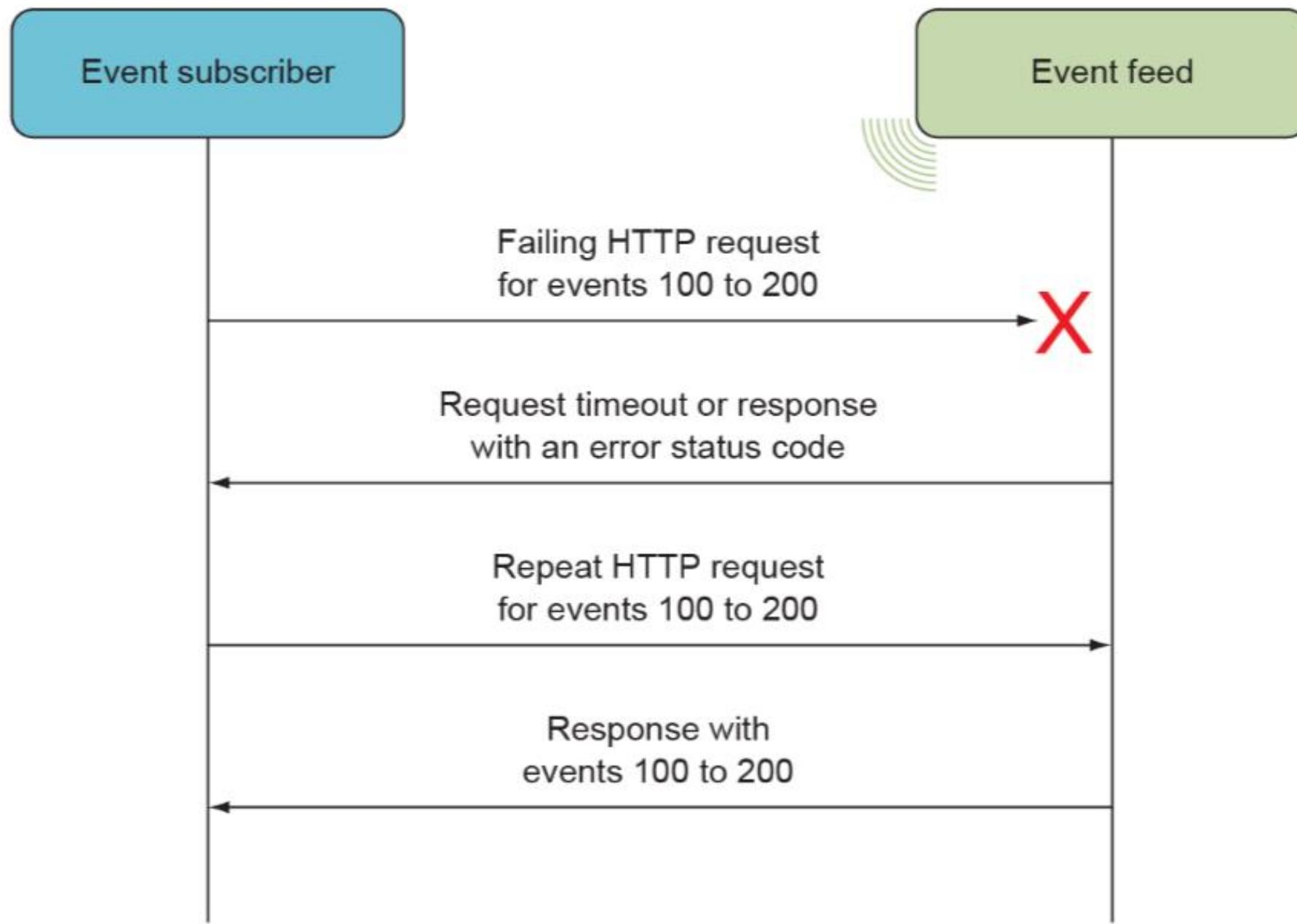
- One of the machines in the cluster on which the microservice's data store runs has crashed.
- The microservice has lost network connectivity to one of its collaborators.
- The microservice is receiving unusually high amounts of traffic.
- One of its collaborators is down.



The client side's responsibility for robustness

**Server cannot respond
to a failed request.**

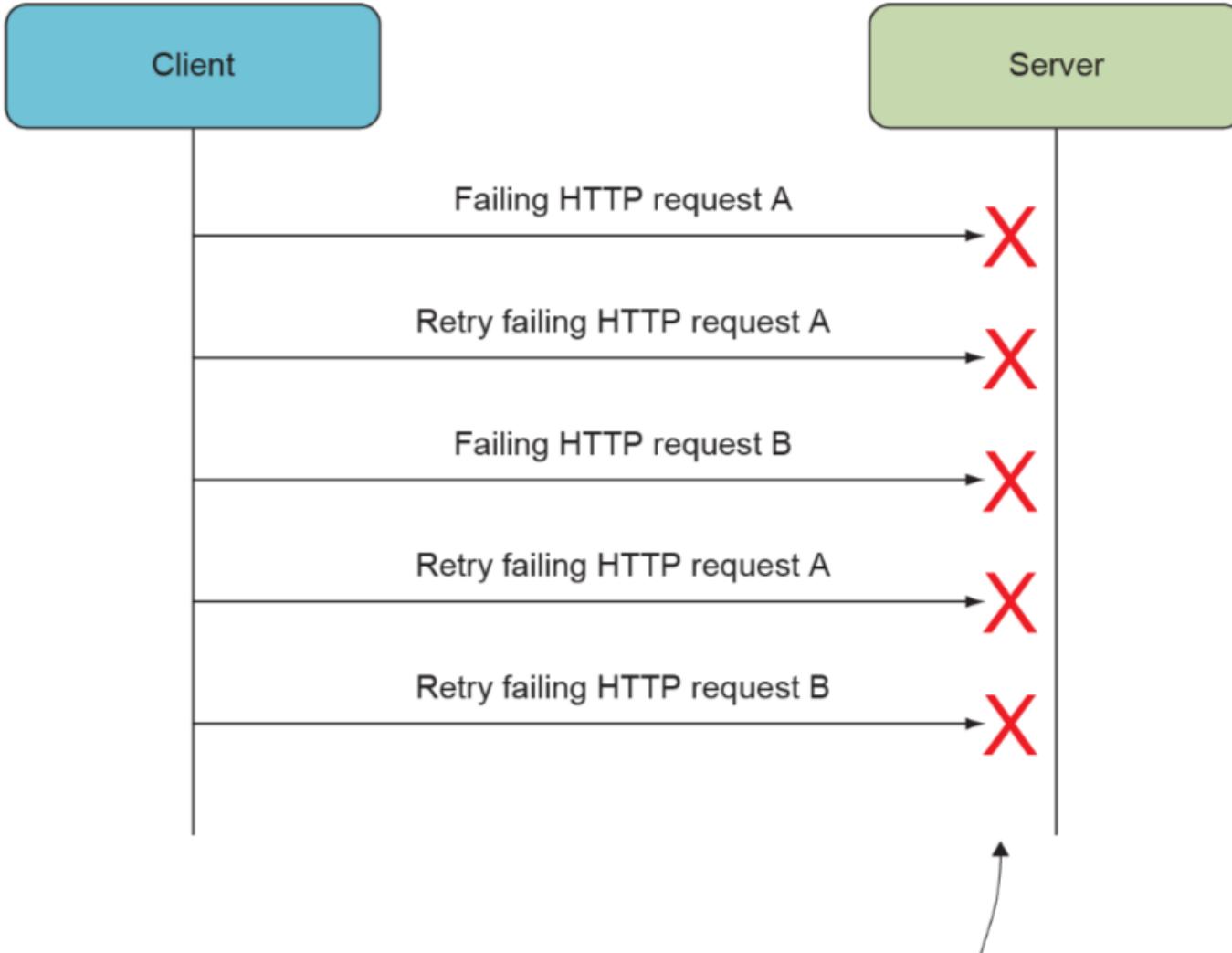




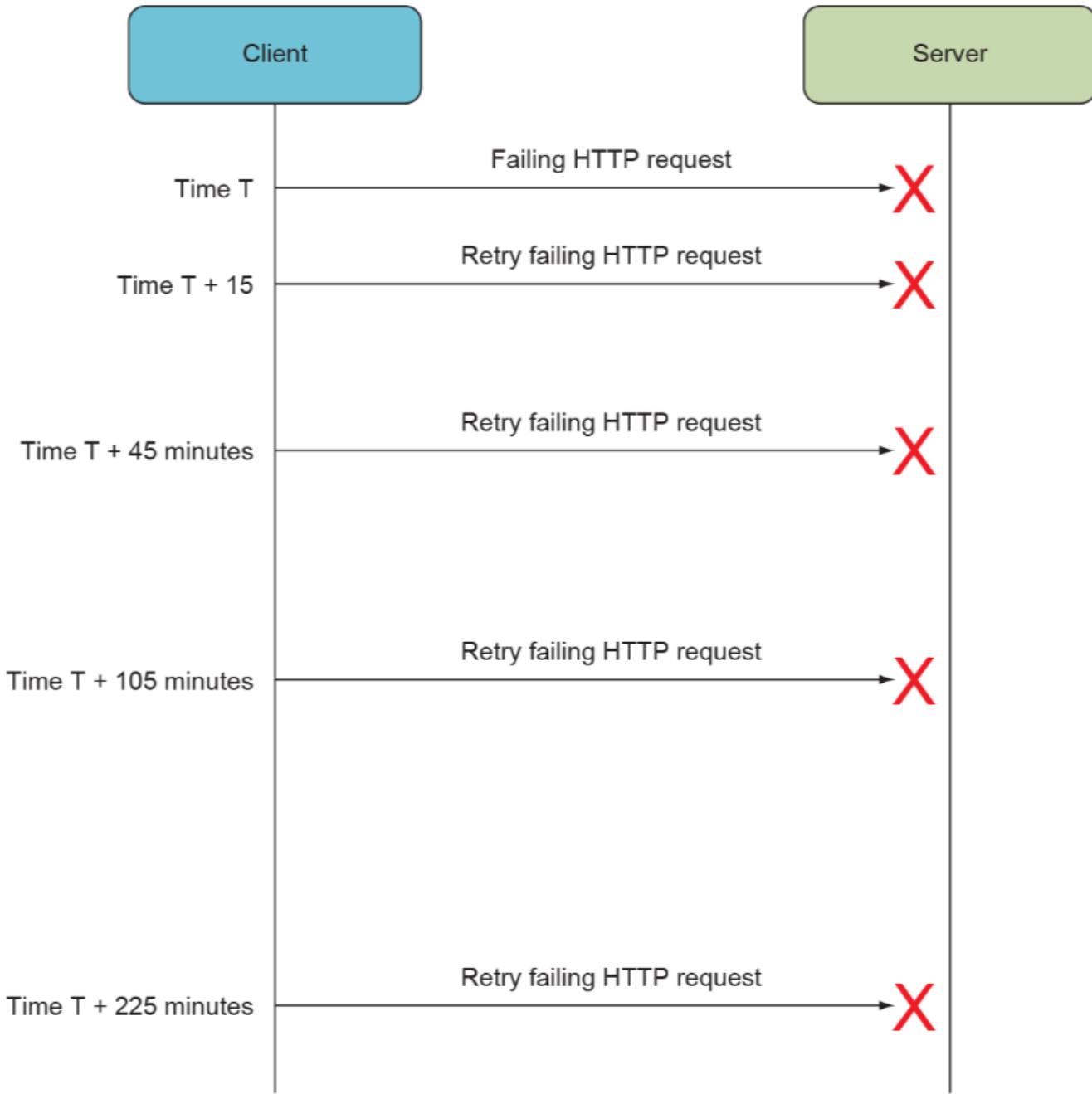
Robustness pattern: Retry

Transient failures are common, and the reasons for them include the following:

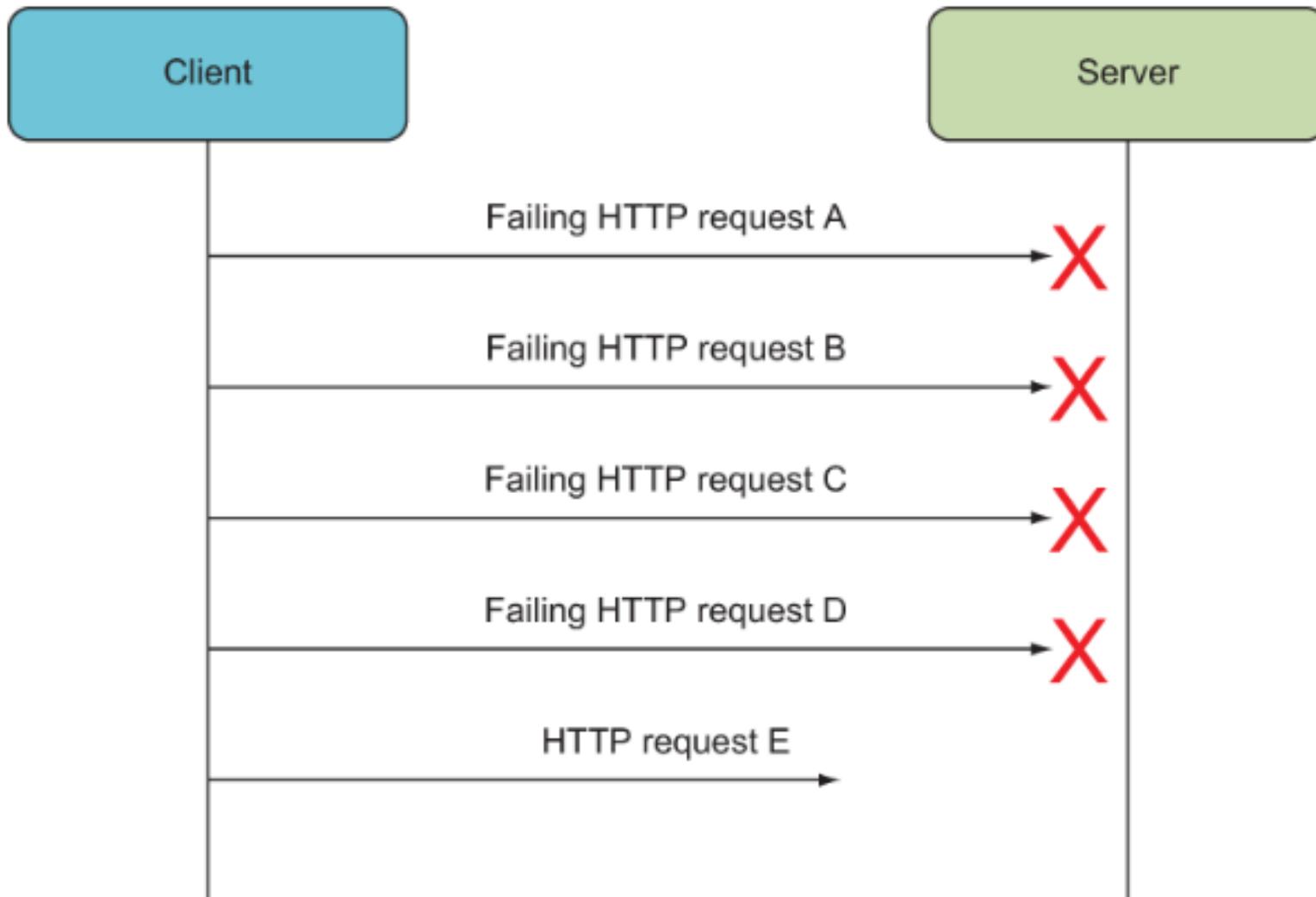
- Network congestion.
- The server microservice being deployed.
- Depending on how the microservice is deployed, there may be a short window when the microservice is unavailable or slow—for example, while a load balancer is switched over to a new version. Even if the server is slow only during deployment, requests may fail due to timeouts.

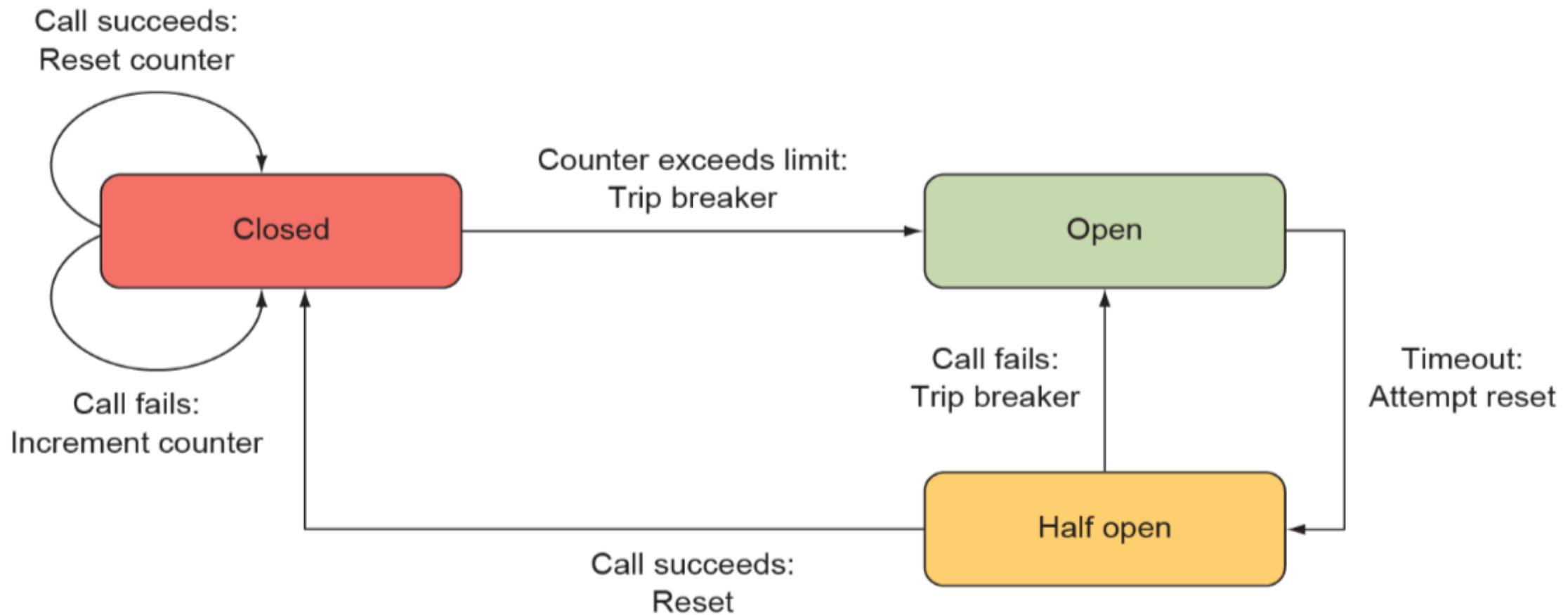


**Server may fail
due to stress.**

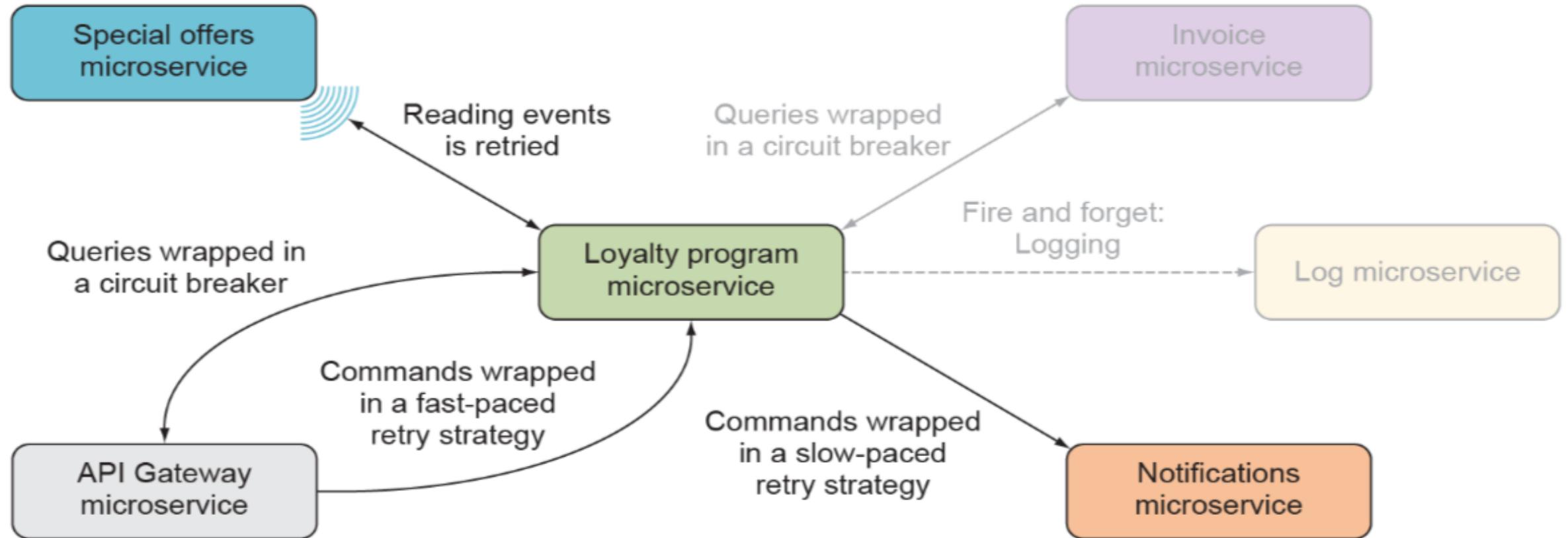


Robustness pattern: Circuit breaker





Implementing robustness patterns



POLLY

The three basic steps to using Polly are as follows:

- Decide which exceptions to handle, such as `HttpException`.
- Decide which policy to use, such as a retry policy.
- Apply the policy to a function.

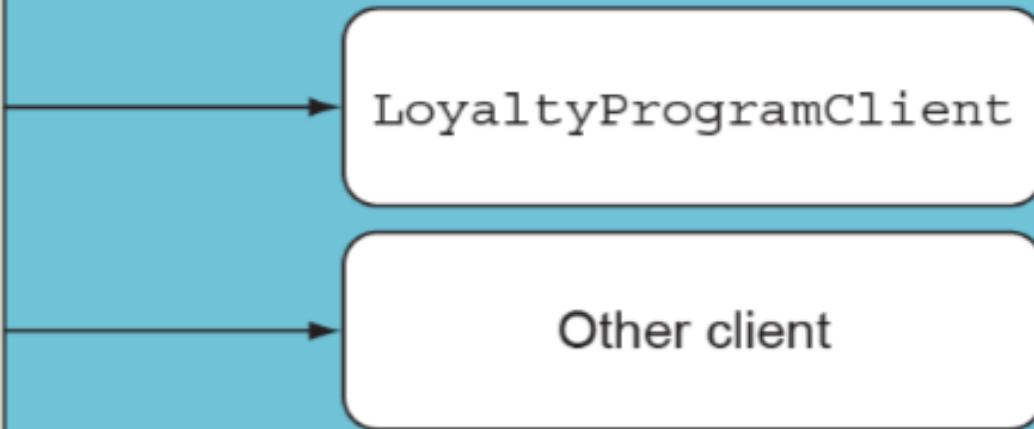
POLLY

The entry point to working with Polly is the Policy class:

```
var retryStrategy =  
    Policy  
        .Handle<HttpException>()  
        .Retry();  
  
retryStrategy.Execute(() => DoHttpRequest());
```

API Gateway microservice

HTTP API: Used by clients



Implementing a fast-paced retry strategy with Polly

```
using System;
```

```
using Polly;
```

```
public class LoyaltyProgramClient
{
    private static readonly IAsyncPolicy<HttpResponseMessage>
        ExponentialRetryPolicy =
            Policy<HttpResponseMessage>
                .Handle<HttpRequestException>()
                .OrTransientHttpStatusCode()
                .WaitAndRetryAsync(
                    3,
                    attempt =>
                        TimeSpan.FromMilliseconds(100 * Math.Pow(2, attempt))
                );
}
```

Using a Polly policy around an HTTP request

```
public async Task<HttpResponseMessage>
RegisterUser(string name)
{
    var user = new {name, Settings = new { }};
    return await ExponentialRetryPolicy
        .ExecuteAsync(() =>
            this.httpClient.PostAsync("/users/", CreateBody(user)));
}

private static StringContent CreateBody(object user) => ...
```

```
public class Startup
{
    private static readonly IAsyncPolicy<HttpResponseMessage>
        ExponentialRetryPolicy =
            Policy<HttpResponseMessage>
                .Handle<HttpRequestException>()
                .OrTransientHttpStatusCode()
                .WaitAndRetryAsync(
                    3,
                    attempt =>
                        TimeSpan.FromMilliseconds(100 * Math.Pow(2, attempt)));
}

public void ConfigureServices(IServiceCollection services)
{
    services.AddHttpClient<LoyaltyProgramClient>()
        .AddPolicyHandler(_ => ExponentialRetryPolicy)
        .ConfigureHttpClient(c => c.BaseAddress = new Uri(host));
    ...
}

...
}
```

Implementing a circuit breaker with Polly

```
private static readonly IAsyncPolicy<HttpResponseMessage>
CircuitBreakerPolicy =
    Policy<HttpResponseMessage>
        .Handle<HttpRequestException }()
        .OrTransientHttpStatusCode()
        .CircuitBreakerAsync(5, TimeSpan.FromMinutes(1));
```

Wrapping a query in a circuit breaker

```
services.AddHttpClient<LoyaltyProgramClient>()
    .AddPolicyHandler(request =>
        request.Method == HttpMethod.Get
            ? CircuitBreakerPolicy
            : ExponentialRetryPolicy)
    .ConfigureHttpClient(c => c.BaseAddress = new Uri(host));
```

Implementing a slow-paced retry strategy

```
using System;
using System.IO;
using System.Net.Http;
using System.Net.Http.Headers;
using System.Text.Json;
using System.Threading.Tasks;

var start = await GetStartIdFromDatastore();
var end = 100;
var client = new HttpClient();
client
    .DefaultRequestHeaders
    .Accept
    .Add(new MediaTypeWithQualityHeaderValue("application/json"));
using var resp = await client
    .GetAsync(
        new Uri($"http://special-offers:5002/events?start={start}&end={end}"));
await ProcessEvents(await resp.Content.ReadAsStreamAsync());
await SaveStartIdToDatastore(start);

static int GetStartIdFromDatastore() { ... }
static void SaveStartIdToDatastore(int start) { ... }
static void ProcessEvents(string rawEvents) { ... }

public record SpecialOfferEvent(
    long SequenceNumber,
    DateTimeOffset OccurredAt,
    string Name,
    object Content);
```

Handling a batch of events

```
private async Task ProcessEvents(string rawEvents)
{
    var events = await
        JsonSerializer.DeserializeAsync<SpecialOfferEvent[]>(content)
        ?? new SpecialOfferEvent[0];
    foreach (var ev in events)
    {
        dynamic eventData = ev.Content;
        if (ShouldSendNotification(eventData))
            await SendNotification(eventData);
        this.start = ev.SequenceNumber + 1;
    }
}

private bool ShouldSendNotification(dynamic eventData)
{
    // decide if notification should be sent based on business rules
}

private Task SendNotification(dynamic eventData)
{
    // use HttpClient to send command to notification microservice
}
```

Kubernetes manifest for the LoyaltyProgram event consumer

```
---
apiVersion: batch/v1beta1
kind: CronJob
metadata:
  name: loyalty-program-consumer
spec:
  schedule: "*/1 * * * *"
  jobTemplate:
    spec:
      template:
        spec:
          containers:
            - name: loyalty-program
              image: microservicesindotnetregistry1.azurecr.io/loyalty-
                    program:1.0.2
              imagePullPolicy: IfNotPresent
              env:
                - name: STARTUPDLL
                  value: "consumer/EventConsumer.dll"
  restartPolicy: Never
  concurrencyPolicy: Forbid
```

Logging all unhandled exceptions

- To see this in action, you can add a new endpoint to the UsersController that simply throws a NotImplementedException:

```
[HttpGet("fail")]
public IActionResult Fail() => throw new NotImplementedException();
```

Errors are logged to standard out

```
dotnet run
info: Microsoft.Hosting.Lifetime[0]
      Now listening on: https://localhost:5001
info: Microsoft.Hosting.Lifetime[0]
      Now listening on: http://localhost:5000
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
info: Microsoft.Hosting.Lifetime[0]
      Content root path: C:\Users\chors\Documents\horsdal3\code\Chapter07\
          LoyaltyProgram\LoyaltyProgram
fail: Microsoft.AspNetCore.Server.Kestrel[13]
      Connection id "0HM2KQ4LI56GR", Request id "0HM2KQ4LI56GR:00000001":
          An unhandled exception was thrown by the application.
System.NotImplementedException: The method or operation is not implemented.
   at LoyaltyProgram.Users UsersController.Fail() in C:\Users\chors\
       Documents\horsdal3\code\Chapter07\LoyaltyProgram\LoyaltyProgram\Users\
       UsersController.cs:line 13
   at lambda_method(Closure , Object , Object[] )
   at Microsoft.Extensions.Internal.ObjectMethodExecutor.Execute(
       Object target, Object[] parameters)
   at Microsoft.AspNetCore.Mvc.Infrastructure.ActionMethodExecutor.
       SyncActionResultExecutor.Execute(IActionResultTypeMapper mapper,
       ObjectMethodExecutor executor, Object controller, Object[] arguments)
   at Microsoft.AspNetCore.Mvc.Infrastructure.ControllerActionInvoker.
       InvokeActionMethodAsync()
   at Microsoft.AspNetCore.Mvc.Infrastructure.ControllerActionInvoker.
       Next(State& next, Scope& scope, Object& state, Boolean& isCompleted)
   at Microsoft.AspNetCore.Mvc.Infrastructure.ControllerActionInvoker.
       InvokeNextActionFilterAsync()
--- End of stack trace from previous location where exception was thrown ---
   at Microsoft.AspNetCore.Mvc.Infrastructure.ControllerActionInvoker.
       Rethrow(ActionExecutedContextSealed context)
   at Microsoft.AspNetCore.Mvc.Infrastructure.ControllerActionInvoker.
       Next(State& next, Scope& scope, Object& state, Boolean& isCompleted)
   at Microsoft.AspNetCore.Mvc.Infrastructure.ControllerActionInvoker.
       InvokeInnerFilterAsync()
--- End of stack trace from previous location where exception was thrown ---
   at Microsoft.AspNetCore.Mvc.Infrastructure.ResourceInvoker.<
       InvokeFilterPipelineAsync>g__Awaited|19_0(ResourceInvoker invoker, Task
       lastTask, State next, Scope scope, Object state, Boolean isCompleted)
   at Microsoft.AspNetCore.Mvc.Infrastructure.ResourceInvoker.<InvokeAsync>
       g__Awaited|17_0(ResourceInvoker invoker, Task task, IDisposable scope)
   at Microsoft.AspNetCore.Routing.EndpointMiddleware.<Invoke>
       g__AwaitRequestTask|6_0(Endpoint endpoint, Task requestTask,
       ILogger logger)
   at Microsoft.AspNetCore.Server.Kestrel.Core.Internal.Http.HttpProtocol.
       ProcessRequests[TContext](IHttpApplication`1 application)
```

Deploying to Kubernetes

- We have implemented robustness patterns in all the collaborations around the loyalty program microservice, but we haven't changed the collaboration styles.
- Queries are still queries, commands are still commands, and events are still events.
- Moreover, the dependencies between the microservices are still the same: the loyalty program microservice still subscribes to events from the special offers microservice and still sends commands to the notifications microservice.
- This means that the deployment to Kubernetes does not have to change.

Summary

- Due to the amount of communication between microservices, you must expect some communication to fail.
- You should design robustness into your microservices so that failures don't propagate through the system and eventually become errors.
- The client side of a collaboration is responsible for making communication robust in the face of failures.
- Polly makes it easy to set up and use retry policies as well as circuit breakers.

8. Writing tests for microservices

Introduction

This Module covers:

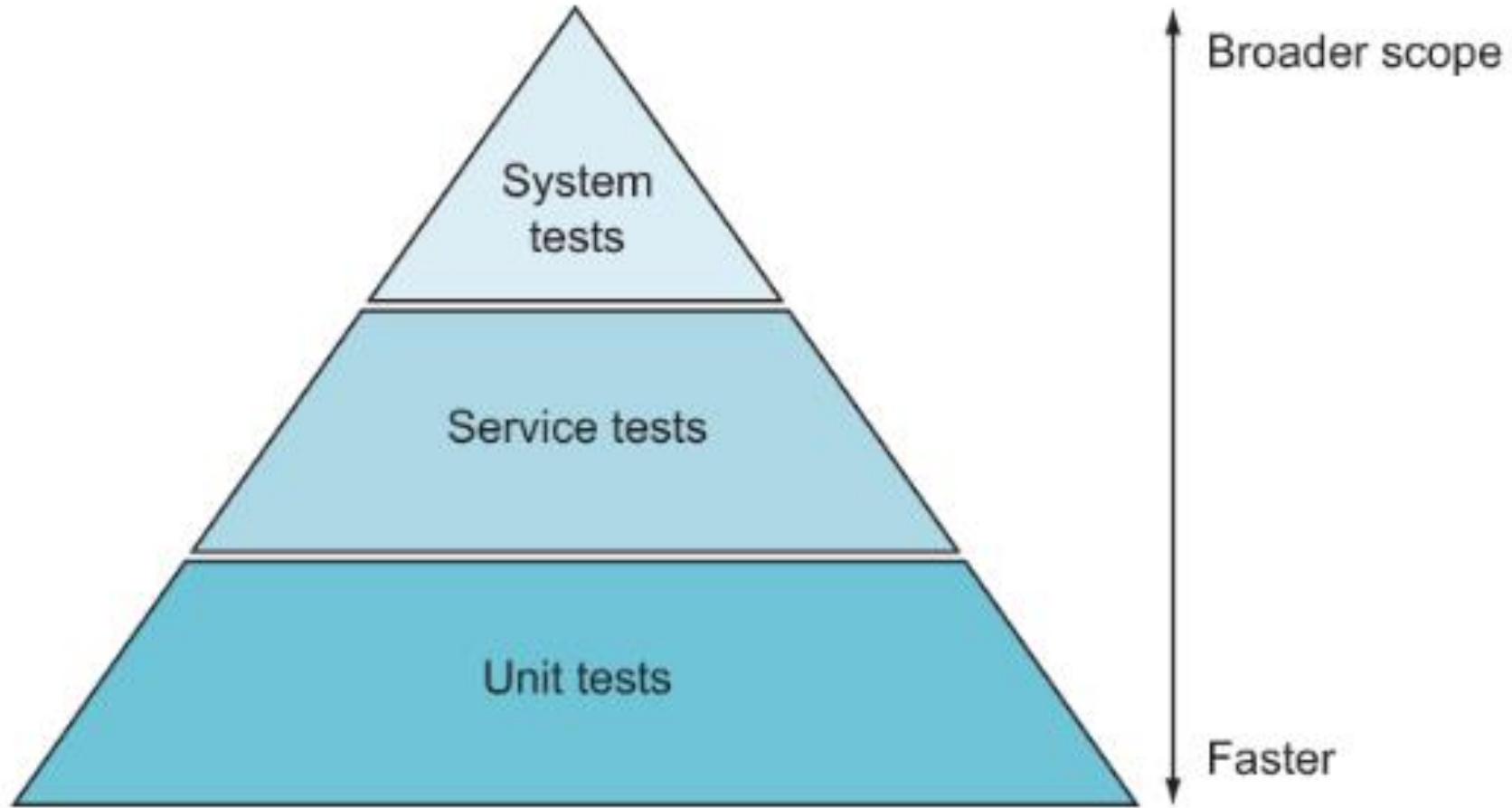
- Writing good automated tests
- Understanding the test pyramid and how it applies to microservices
- Testing microservices from the outside
- Writing fast, in-process tests for endpoints
- Using `Microsoft.AspNetCore.TestHost` for integration and unit tests

What and how to test

In Module 1, you saw three characteristics of a microservice that make it good for continuous delivery:

- Individually deployable
- Replaceable
- Maintainable by a small team

The test pyramid: What to test in a microservices system



The test pyramid

The version of the test pyramid that I use here has three levels:

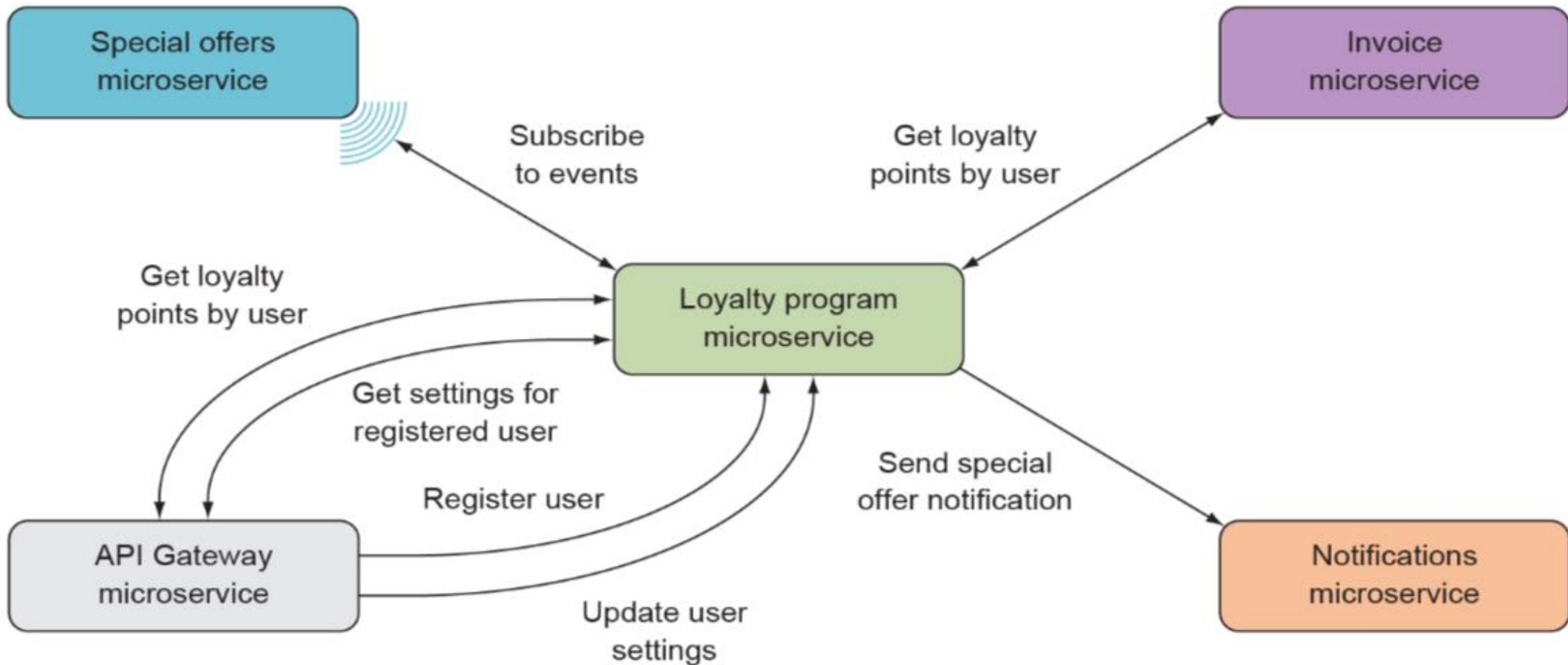
- *System tests (top level)*
- *Service tests (middle level)*
- *Unit tests (bottom level)*

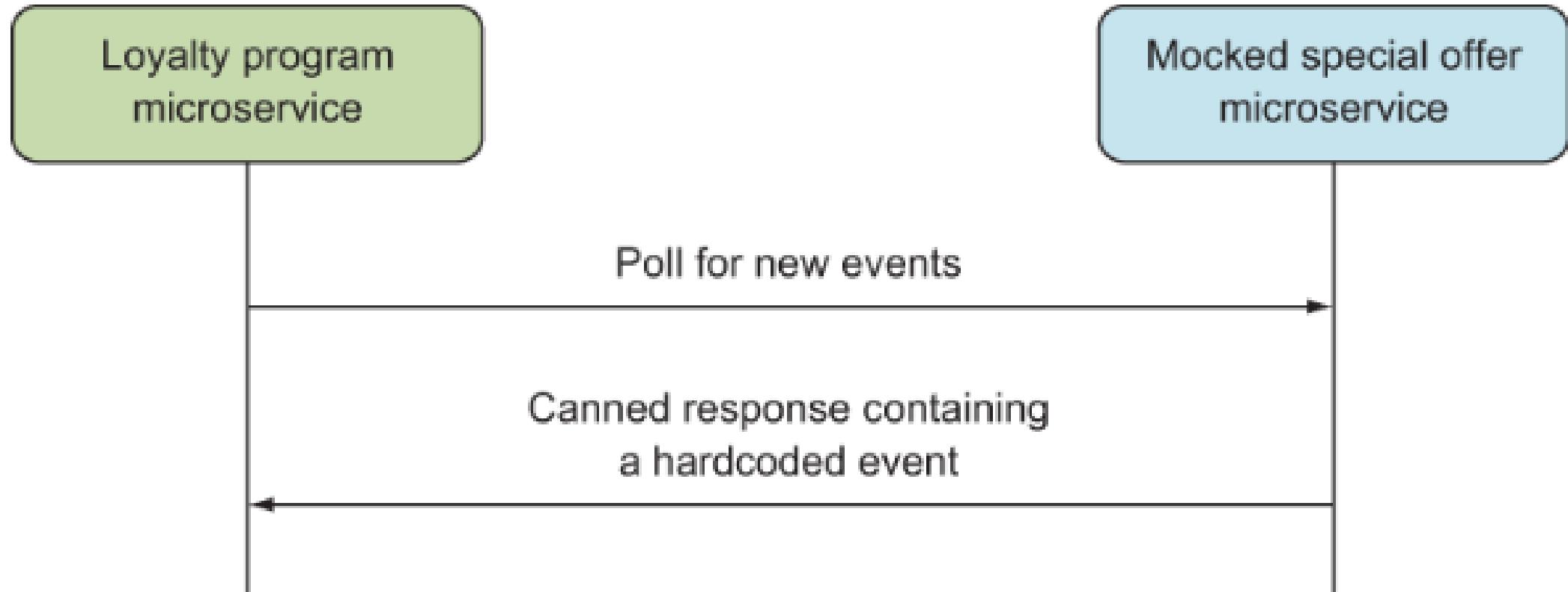
System-level tests: Testing a complete microservice system end to end

- The tests at the top of the pyramid have a very broad scope and therefore cover a lot of code with just a few tests. Because they have such a broad scope, they're also imprecise.
- The second downside to system-level tests is that they tend to be slow.
- This again is the flip side of them involving the complete system: real HTTP requests are made, things are written to real data stores, and real event feeds are polled.

Service-level tests: Testing a microservice from outside its process

- The tests in the middle level of the test pyramid interact with one microservice as a whole and in isolation.
- The collaborators of the microservice under test are replaced with microservice mocks.
- Like system tests, these tests interact with the microservice under test from the outside.
- But unlike system-level tests, they interact directly with the public API of the microservice and make assertions about responses to the microservice.





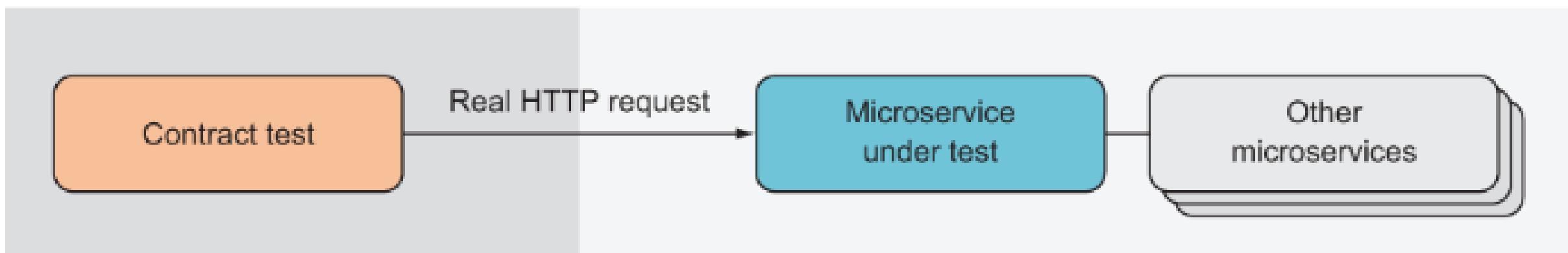
Service-level test

A service-level test for the loyalty program microservice could do the following:

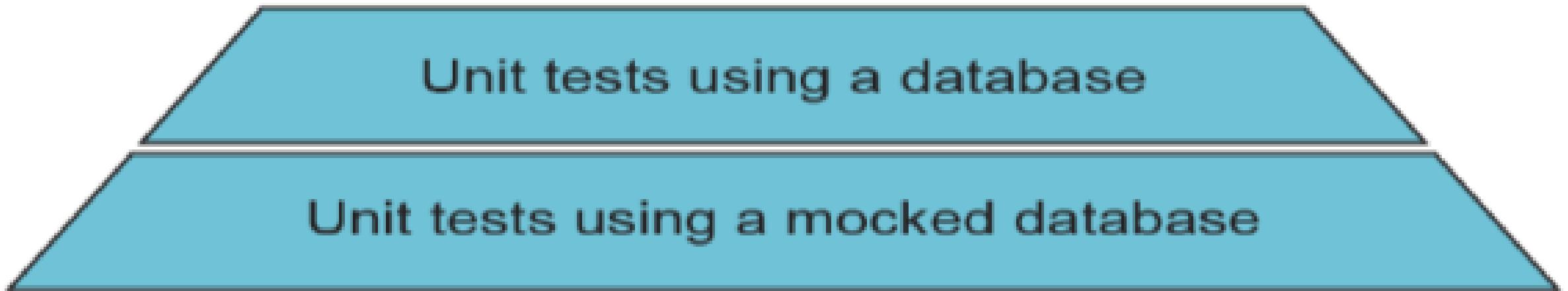
- Send a command to create a user.
- Wait for the loyalty program microservice to query a mock special offer microservice for events and then receive a hardcoded event about a new special offer.
- Record any commands sent to the notifications microservice and assert that a command for a notification to the new user about the new special offer was sent.

CI server environment

Complete system



Unit-level tests: Testing endpoints from within the process



Testing libraries: Microsoft.AspNetCore.TestHost and xUnit

In this Module, we'll use two new libraries:

- Microsoft.AspNetCore.TestHost
- xUnit

Meet Microsoft.AspNetCore.TestHost

We can do all the usual configuration including, but not limited to

- Overriding registration in the dependency injection container, for instance, to provide mock objects in place of real ones.
- Adding special hooks to the ASP.NET pipeline.
- Configuring logging, for instance, to allow tests to inspect the contents of log messages.

Meet xUnit

- xUnit is a unit-test tool for .NET.
- It has a library part that allows you to write automated tests and a runner part that can run those tests.
- To write a test with xUnit, you create a method with a Fact attribute over it and put the code to perform the test there.
- The xUnit runner scans for methods with a Fact attribute and executes all of them.
- In addition, xUnit has an API for making assertions in tests.

xUnit and Microsoft.AspNetCore.TestHost working together

```
namespace LoyaltyProgramUnitTests
{
    using System;
    using System.Net;
    using System.Net.Http;
    using System.Threading.Tasks;
    using Microsoft.AspNetCore.Builder;
    using Microsoft.AspNetCore.Hosting;
    using Microsoft.AspNetCore.Mvc;
    using Microsoft.AspNetCore.TestHost;
    using Microsoft.Extensions.Hosting;
    using Xunit;

    public class TestController_should : IDisposable
    {
        private readonly IHost host;
        private readonly HttpClient sut;

        public class TestController : ControllerBase
        {
            [HttpGet("/")]
            public OkResult Get() => Ok();
        }

        public TestController_should()
        {
            this.host = new HostBuilder()
                .ConfigureWebHost(host =>
                    host
                        .ConfigureServices(x =>
                            x.AddControllersByType(typeof(TestController)))
                        .Configure(x =>
                            x.UseRouting()
                                .UseEndpoints(opt => opt.MapControllers())
                                .UseTestServer())
                        .Start());
            this.sut = this.host.GetTestClient();
        }

        [Fact]
        public async Task respond_ok_to_request_to_root()
        {
            var actual = await this.sut.GetAsync("/");
            Assert.Equal(HttpStatusCode.OK, actual.StatusCode);
        }

        public void Dispose()
        {
            this.host?.Dispose();
            this.sut?.Dispose();
        }
    }
}
```

NAMING CONVENTIONS

Tests follow these naming conventions:

- My tests work on an object called sut for system under test.
- In the previous test, sut is a HttpClient object that I use to make a call to an endpoint.
- I name my test classes after the thing they test—TestController in this example test—followed by _should.
- I name the Fact method after the scenario being tested and the expected result.

Extension method to control which controllers are part of the test

```
namespace LoyaltyProgramUnitTests
{
    using System;
    using System.Linq;
    using System.Reflection;
    using Microsoft.AspNetCore.Mvc.Controllers;
    using Microsoft.Extensions.DependencyInjection;

    public class FixedControllerProvider : ControllerFeatureProvider
    {
        private readonly Type[] controllerTypes;

        public FixedControllerProvider(params Type[] controllerTypes)
        {
            this.controllerTypes = controllerTypes;
        }

        protected override bool IsController(TypeInfo typeInfo)
        {
            return this.controllerTypes.Contains(typeInfo);
        }
    }

    public static class MvcBuilderExtensions
    {
        public static IMvcBuilder AddControllersByType(
            this IServiceCollection services,
            params Type[] controllerTypes)
            =>
        services
            .AddControllers()
            .ConfigureApplicationPartManager(mgr =>
        {
            mgr.FeatureProviders.Remove(
                mgr.FeatureProviders.First(f =>
                    f is ControllerFeatureProvider));
            mgr.FeatureProviders.Add(
                new FixedControllerProvider(controllerTypes));
        });
    }
}
```

Writing unit tests using Microsoft.AspNetCore.TestHost

In Module 5, you saw that the loyalty program has three command and query endpoints:

- An HTTP GET endpoint at URLs of the form “/users/{userId}” that responds with a representation of the user
- An HTTP POST endpoint to “/users/” that expects a representation of a user in the body of the request and then registers that user in the loyalty program
- An HTTP PUT endpoint at URLs of the form /“users/{userId}” that expects a representation of a user in the body of the request and then updates an already registered user

Setting up a unit-test project

```
C:\.
  Ch08.sln
  Dockerfile
  loyalty-program.yaml

  --EventConsumer
    EventConsumer.csproj
    Program.cs

  --LoyaltyProgram
    appsettings.Development.json
    appsettings.json
    LoyaltyProgram.csproj
    Program.cs
    Startup.cs

  --EventFeed
    EventFeedController.cs
    EventStore.cs
  --Users
    LoyaltyProgramSettings.cs
    LoyaltyProgramUser.cs
    UsersController.cs

  --LoyaltyProgramUnitTests
    LoyaltyProgramUnitTests.csproj
    UnitTest1.cs
```

- You can now run the tiny test in UnitTest1.cs with dotnet like this:

PS> dotnet test

- The dependencies in the LoyaltyProgramUnitTests.csproj now look like this:

```
<ItemGroup>
  <PackageReference Include="Microsoft.AspNetCore.TestHost" Version="3.1.8"/>
  <PackageReference Include="Microsoft.NET.Test.Sdk" Version="16.5.0" />
  <PackageReference Include="xunit" Version="2.4.0" />
  <PackageReference Include="xunit.runner.visualstudio" Version="2.4.0" />
  <PackageReference Include="coverlet.collector" Version="1.2.0" />
</ItemGroup>

<ItemGroup>
  <ProjectReference Include=".\\LoyaltyProgram\\LoyaltyProgram.csproj" />
</ItemGroup>
```

Using the TestServer and HttpClient to unit-test endpoints

```
namespace LoyaltyProgramUnitTests
{
    using System;
    using System.Net;
    using System.Net.Http;
    using System.Threading.Tasks;
    using LoyaltyProgram;
    using Microsoft.AspNetCore.Hosting;
    using Microsoft.AspNetCore.TestHost;
    using Microsoft.Extensions.Hosting;
    using Xunit;

    public class UsersEndpoints_should : IDisposable
    {
        private readonly IHost host;
        private readonly HttpClient sut;

        public UsersEndpoints_should()
        {
            this.host = new HostBuilder()
                .ConfigureWebHost(x => x
                    .UseStartup<Startup>()
                    .UseTestServer())
                .Start();
            this.sut = this.host.GetTestClient();
        }

        [Fact]
        public async Task respond_not_fount_when_queried_for_unregistered_user()
        {
            var actual = await this.sut.GetAsync("/users/1000");      1((C03-5))
            Assert.Equal(HttpStatusCode.NotFound, actual.StatusCode);
        }

        public void Dispose()
        {
            this.host?.Dispose();
            this.sut?.Dispose();
        }
    }
}
```

Test for registering a user through the users endpoint

```
[Fact]
public async Task allow_to_register_new_user()
{
    var expected = new LoyaltyProgramUser(
        0,
        "Christian",
        0,
        new LoyaltyProgramSettings());

    var registrationResponse =
        await this.sut.PostAsync(
            "/users",
            new StringContent(
                JsonSerializer.Serialize(expected),
                Encoding.UTF8,
                "application/json"));
    var newUser =
        await JsonSerializer.DeserializeAsync<LoyaltyProgramUser>(
            await registrationResponse.Content.ReadAsStreamAsync(),
            new JsonSerializerOptions
            {
                PropertyNameCaseInsensitive = true
            });

    var actual = await this.sut.GetAsync($"/users/{newUser?.Id}");
    var actualUser =
        JsonSerializer.Deserialize<LoyaltyProgramUser>(
            await actual.Content.ReadAsStringAsync(),
            new JsonSerializerOptions
            {
                PropertyNameCaseInsensitive = true
           });

    Assert.Equal(HttpStatusCode.OK, actual.StatusCode);
    Assert.Equal(expected.Name, actualUser?.Name);
}
```

Test for modifying users through the users endpoint

```
[Fact]
public async Task allow_modifying_users()
{
    var expected = "jane";
    var user = new LoyaltyProgramUser(
        0, "Christian", 0, new LoyaltyProgramSettings());

    var registrationResponse = await this.sut.PostAsync(
        "/users",
        new StringContent(
            JsonSerializer.Serialize(user),
            Encoding.UTF8,
            "application/json"));
    var newUser = await
        JsonSerializer.DeserializeAsync<LoyaltyProgramUser>(
            await registrationResponse.Content.ReadAsStreamAsync(),
            new JsonSerializerOptions
            {
                PropertyNameCaseInsensitive = true
            });
}

var updatedUser = newUser! with {Name = expected};
var actual = await this.sut.PutAsync($""/users/{newUser.Id}""",
    new StringContent(
        JsonSerializer.Serialize(updatedUser),
        Encoding.UTF8,
        "application/json"));
var actualUser = await
    JsonSerializer.DeserializeAsync<LoyaltyProgramUser>(
        await actual.Content.ReadAsStreamAsync(),
        new JsonSerializerOptions
        {
            PropertyNameCaseInsensitive = true
        });

Assert.Equal(HttpStatusCode.OK, actual.StatusCode);
Assert.Equal(expected, actualUser?.Name);
}
```

Injecting mocks into endpoints

```
public interface IEventStore
{
    Task RaiseEvent(string name, object content);
    Task<IEnumerable<EventFeedEvent>>
        GetEvents(int start, int end);
}
```

Event feed

```
namespace LoyaltyProgram.EventFeed
{
    using System.Linq;
    using System.Threading.Tasks;
    using Microsoft.AspNetCore.Mvc;

    [Route("/events")]
    public class EventFeedController : ControllerBase
    {
        private readonly IEventStore eventStore;

        public EventFeedController(IEventStore eventStore)
        {
            this.eventStore = eventStore;
        }

        [HttpGet("")]
        public async Task<ActionResult<EventFeedEvent[]>>
            GetEvents([FromQuery] int start, [FromQuery] int end)
        {
            if (start < 0 || end < start)
                return BadRequest();

            return
                (await this.eventStore.GetEvents(start, end))
                .ToArray();
        }
    }
}
```

Fake IEvent Store to use in tests

```
public class FakeEventStore : IEventStore
{
    public Task RaiseEvent(string name, object content) =>
        throw new System.NotImplementedException();

    public Task<IEnumerable<EventFeedEvent>>
        GetEvents(int start, int end)
    {
        if (start > 100)
            return Task.FromResult(
                Enumerable.Empty<EventFeedEvent>());
    }

    return Task.FromResult(Enumerable
        .Range(start, end - start)
        .Select(i =>
            new EventFeedEvent(
                i,
                DateTimeOffset.UtcNow,
                "some event",
                new object())));
    }
}
```

Using the fake event store while testing

```
public EventFeed_should()
{
    this.host = new HostBuilder()
        .ConfigureWebHost(host =>
            host
                .ConfigureServices(x => x
                    .AddScoped<IEventStore, FakeEventStore>()
                    .AddControllersByType(typeof(EventFeedController))
                    .AddApplicationPart(typeof(EventFeedController).Assembly))
                .Configure(x =>
                    x.UseRouting().UseEndpoints(opt => opt.MapControllers()))
                    .UseTestServer())
            .Start();
    this.sut = this.host.GetTestClient();
}
```

Tests for the event feed, using the fake event store

```
[Fact]
```

```
public async Task return_events_when_from_event_store()
{
    var actual = await this.sut.GetAsync("/events?start=0&end=100");

    Assert.Equal(HttpStatusCode.OK, actual.StatusCode);
    var eventFeedEvents = await
        JsonSerializer.DeserializeAsync<IEnumerable<EventFeedEvent>>(
            await actual.Content.ReadAsStreamAsync())
        ?? Enumerable.Empty<EventFeedEvent>();
    Assert.Equal(100, eventFeedEvents.Count());
}
```

```
[Fact]
```

```
public async Task return_empty_response_when_there_are_no_more_events()
{
    var actual = await this.sut.GetAsync("/events?start=200&end=300");

    var eventFeedEvents = await
        JsonSerializer.DeserializeAsync<IEnumerable<EventFeedEvent>>(
            await actual.Content.ReadAsStreamAsync());
    Assert.Empty(eventFeedEvents);
}
```

```
> dotnet test
```

```
Test run for .\LoyaltyProgramUnitTests\bin\Debug\netcoreapp3.1\  
LoyaltyProgramUnitTests.dll(.NETCoreApp,Version=v3.1)
```

```
Microsoft (R) Test Execution Command Line Tool Version 16.6.0
```

```
Copyright (c) Microsoft Corporation. All rights reserved.
```

Starting test execution, please wait...

A total of 1 test files matched the specified pattern.

Test Run Successful.

Total tests: 6

Passed: 6

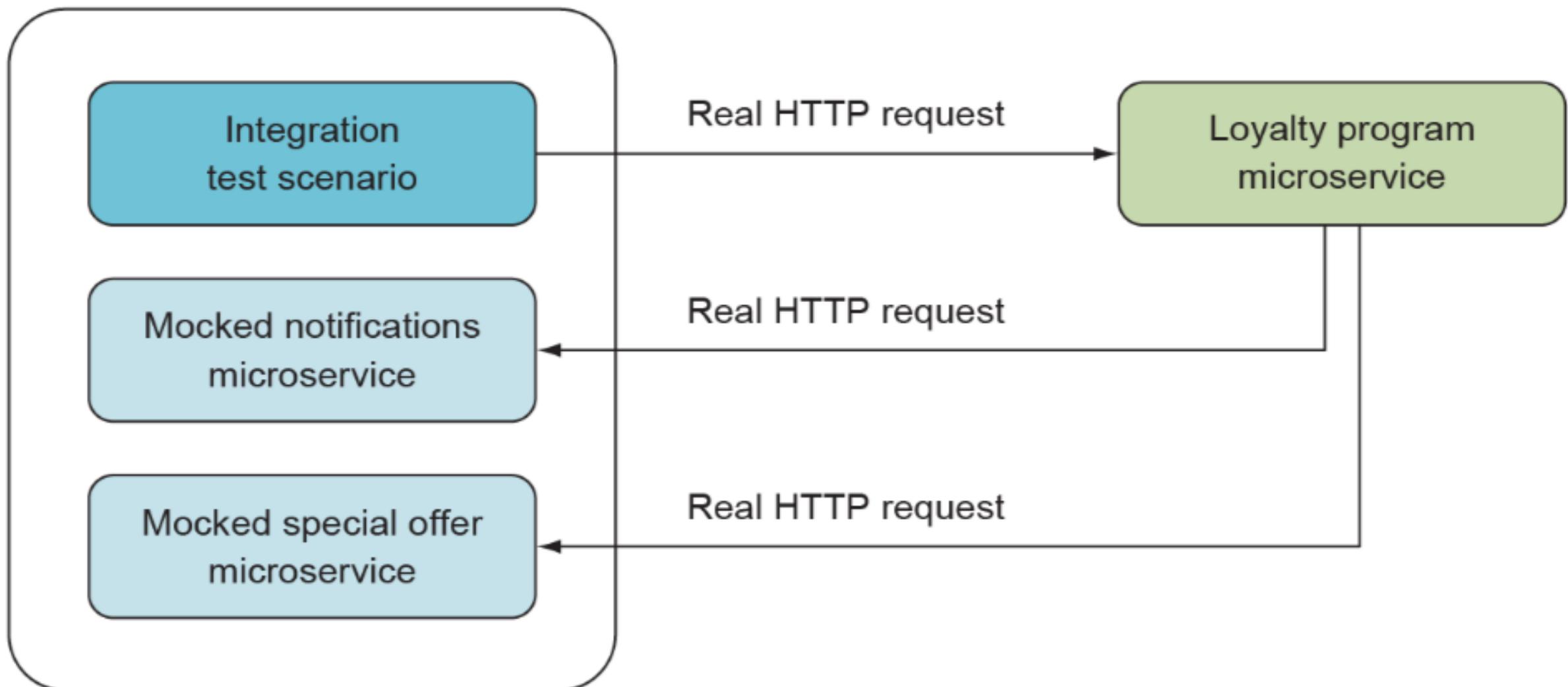
Total time: 1,9937 Seconds

Writing service-level tests

The service-level tests for the loyalty program go through these steps:

- Set up two endpoints in the same process as the test.
- Start the loyalty program microservice and configure it to use the mocked endpoints in place of the real collaborators.
- Execute a scenario against the loyalty program as a sequence of requests to the loyalty program endpoints.
- Record any calls to the mocked endpoints.
- Make assertions on the responses from the loyalty program and on the requests made to the mocked endpoints.

Service-level test process



Creating a service-level test project

We have four folders in the loyalty program:

- LoyaltyProgram
- EventConsumer
- LoyaltyProgramServiceTests
- LoyaltyProgramUnitTests

Creating mocked endpoints

```
namespace LoyaltyProgramServiceTests.Mocks
{
    using Microsoft.AspNetCore.Mvc;

    public class SpecialOffersMock : ControllerBase
    {
        [HttpGet("/specialoffers/events")]
        public ActionResult<object[]> GetEvents(
            [FromQuery] int start,
            [FromQuery] int end)
        {
            return new[]
            {
                new
                {
                    SequenceNumber = 1,
                    Name = "baz",
                    Content = new
                    {
                        OfferName = "foo",
                        Description = "bar",
                        Item = new {ProductName = "name"}
                    }
                }
            };
        }
    }
}
```

Mock endpoint that records when it was called

```
namespace LoyaltyProgramServiceTests.Mocks
{
    using Microsoft.AspNetCore.Mvc;

    public class NotificationsMock : ControllerBase
    {
        public static bool ReceivedNotification = false;

        [HttpPost("/notify")]
        public OkResult Notify()
        {
            ReceivedNotification = true;
            return Ok();
        }
    }
}
```

Service-level tests project file

```
<Project Sdk="Microsoft.NET.Sdk.Web">

    <PropertyGroup>
        <TargetFramework>net5.0</TargetFramework>
        <LangVersion>9</LangVersion>
        <WarningsAsErrors>true</WarningsAsErrors>
        <TreatWarningsAsErrors>true</TreatWarningsAsErrors>
        <Nullable>Enable</Nullable>
        <IsPackable>false</IsPackable>
    </PropertyGroup>

    <ItemGroup>
        <PackageReference Include="Microsoft.AspNetCore.TestHost"
            Version="3.1.8" />
        <PackageReference Include="Microsoft.NET.Test.Sdk"
            Version="16.5.0" />
        <PackageReference Include="xunit"
            Version="2.4.0" />
        <PackageReference Include="xunit.runner.visualstudio"
            Version="2.4.0" />
        <PackageReference Include="coverlet.collector"
            Version="1.2.0" />
    </ItemGroup>

    <ItemGroup>
        <ProjectReference
            Include=".\\LoyaltyProgram\\LoyaltyProgram.csproj" />
        <ProjectReference
            Include=".\\EventConsumer\\EventConsumer.csproj" />
    </ItemGroup>

</Project>
```

Starting ASP.NET inside the test process

```
namespace LoyaltyProgramServiceTests.Mocks
{
    using System;
    using System.Threading;
    using Microsoft.AspNetCore.Builder;
    using Microsoft.AspNetCore.Hosting;
    using Microsoft.Extensions.DependencyInjection;
    using Microsoft.Extensions.Hosting;

    public class MocksHost : IDisposable
    {
        private readonly IHost hostForMocks;

        public MocksHost(int port)
        {
            this.hostForMocks =
                Host.CreateDefaultBuilder()
                    .ConfigureWebHostDefaults(x => x
                        .ConfigureServices(services => services.AddControllers())
                        .Configure(app =>
                            app.UseRouting().UseEndpoints(opt => opt.MapControllers()))
                        .UseUrls($"http://localhost:{port}"))
                    .Build();

            new Thread(() => this.hostForMocks.Run()).Start();
        }

        public void Dispose()
        {
            this.hostForMocks.Dispose();
        }
    }
}
```

Executing the test scenario against the microservice under test

We're now ready to write the test. It has three steps:

- Make an HTTP request through the test host to register a user.
- Run the event consumer.
- Assert that a request to the notifications endpoint was made.

Service-level test using an outside loyalty program

```
1 namespace LoyaltyProgramServiceTests.Scenarios
2 {
3     using System;
4     using System.Net;
5     using System.Net.Http;
6     using System.Text;
7     using System.Threading.Tasks;
8     using LoyaltyProgram;
9     using LoyaltyProgram.Users;
10    using Microsoft.AspNetCore.Hosting;
11    using Microsoft.AspNetCore.TestHost;
12    using Microsoft.Extensions.Hosting;
13    using Mocks;
14    using Newtonsoft.Json;
15    using Xunit;
16
17    public class RegisterUserAndGetNotification : IDisposable
18    {
19        private static int mocksPort = 5050;
20        private readonly MocksHost serviceMock;
21        private readonly IHost loyaltyProgramHost;
22        private readonly HttpClient sut;
23
24        public RegisterUserAndGetNotification()
25        {
26            this.serviceMock = new MocksHost(mocksPort);
27            this.loyaltyProgramHost = new HostBuilder()
28                .ConfigureWebHost(x => x
29                    .UseStartup<Startup>()
30                    .UseTestServer())
31                .Start();
32            this.sut = this.loyaltyProgramHost.GetTestClient();
33        }
34
35        [Fact]
36        public async Task Scenario()
37        {
38            await RegisterNewUser();
39            await RunConsumer();
```

Service-level test using an outside loyalty program

```
40     AssertNotificationWasSent();
41 }
42
43 private async Task RegisterNewUser()
44 {
45     var actual = await this.sut.PostAsync(
46         "/users",
47         new StringContent(
48             JsonSerializer.Serialize(
49                 new LoyaltyProgramUser(
50                     0, "Chr", 0, new LoyaltyProgramSettings())),
51                 Encoding.UTF8,
52                 "application/json"));
53
54     Assert.Equal(HttpStatusCode.Created, actual.StatusCode);
55 }
56
57 private Task RunConsumer() =>
58     EventConsumer.ConsumeBatch(
59         0,
60         100,
61         $"http://localhost:{mocksPort}/specialoffers",
62         $"http://localhost:{mocksPort}"
63     );
64
65 private void AssertNotificationWasSent() =>
66     Assert.True(NotificationMock.ReceivedNotification);
67
68 public void Dispose()
69 {
70     this.serviceMock?.Dispose();
71     this.sut?.Dispose();
72     this.loyaltyProgramHost?.Dispose();
73 }
74 }
75 }
```

Summary

- The test pyramid tells you to have few system-level tests that test the complete system, several service-level tests for each microservice, and many unit tests for each microservice.
- System-level tests are likely to be slow and are very imprecise.
- You should write system-level tests for important success scenarios to provide some test coverage for most of the system.
- Service-level tests are likely to be somewhat slow, but they're faster and more precise than system-level tests.
- You can test endpoints in a test host with both real and mocked data stores.

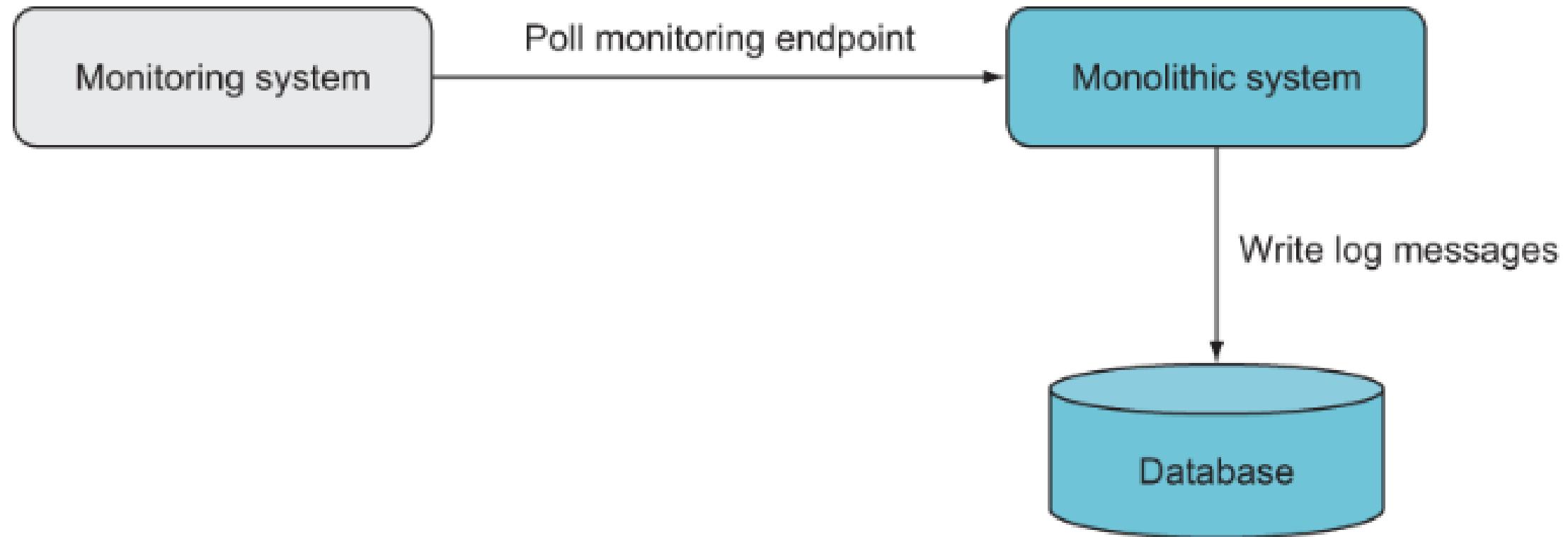
9. Cross-cutting concerns: Monitoring and logging

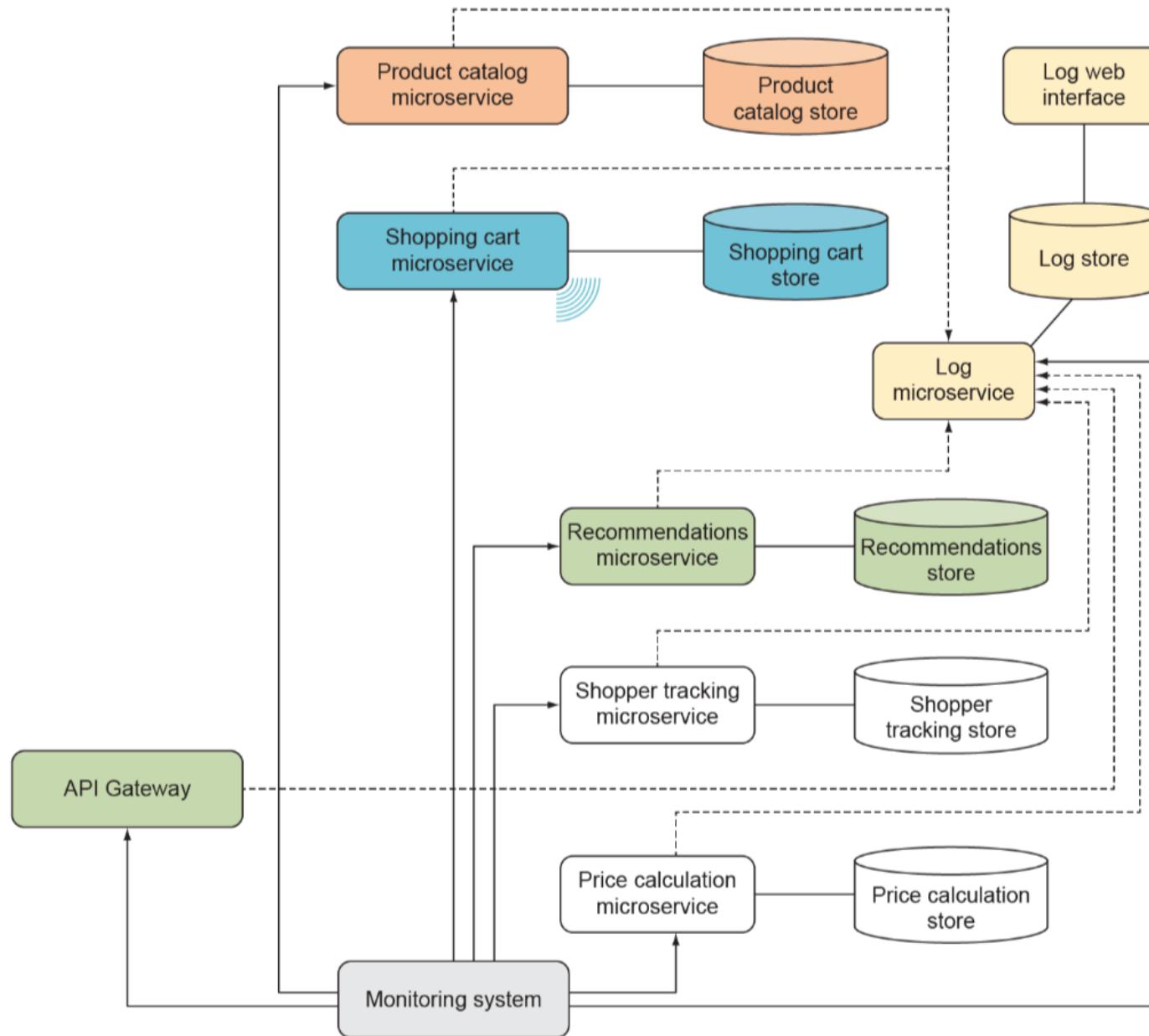
Introduction

This Module covers

- Monitoring in a microservice system
- Exploring structured logging
- Distributed tracing across microservices
- Logging unhandled exceptions

Monitoring needs in microservices





HTTP API: Accessible from other microservices

Command and query HTTP API

Event feed HTTP API

Monitoring endpoints

- Health/startup
- Health/live

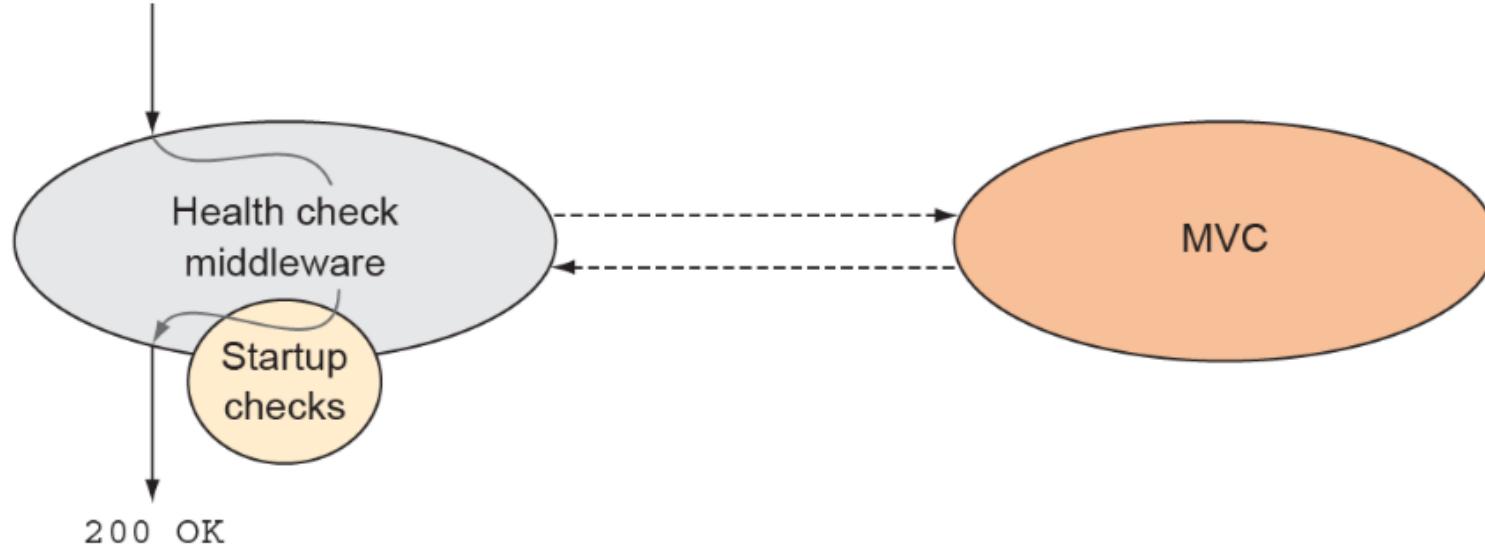
Domain model

Data store

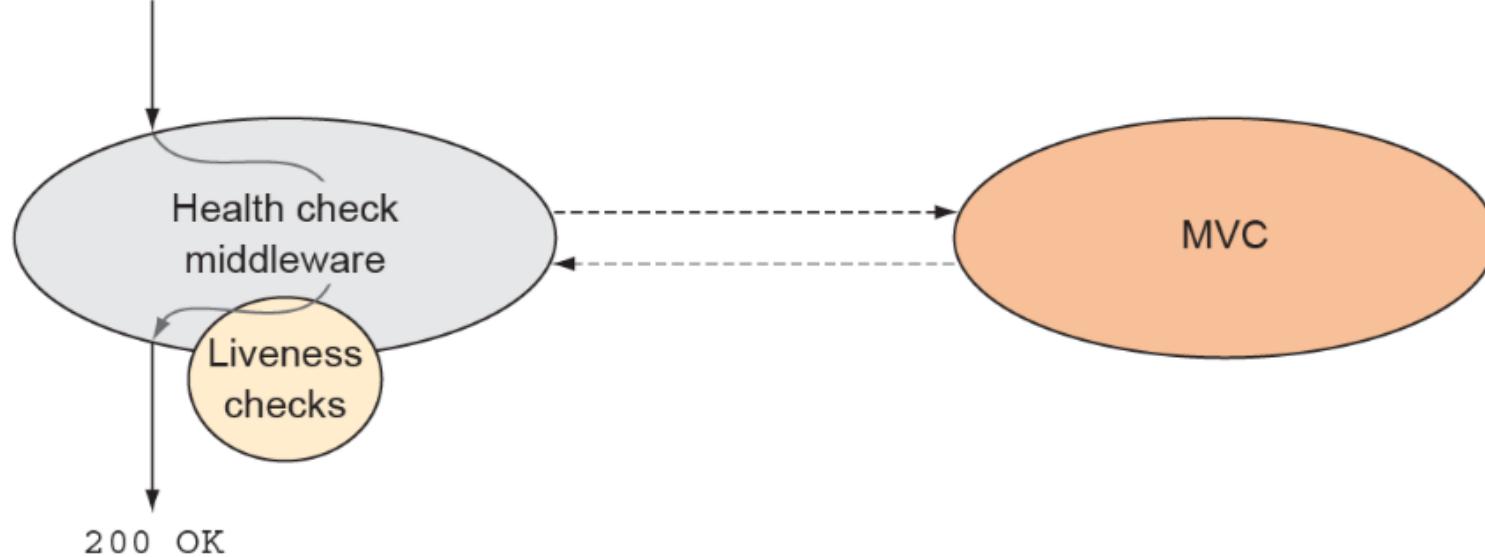
EventStore

Database

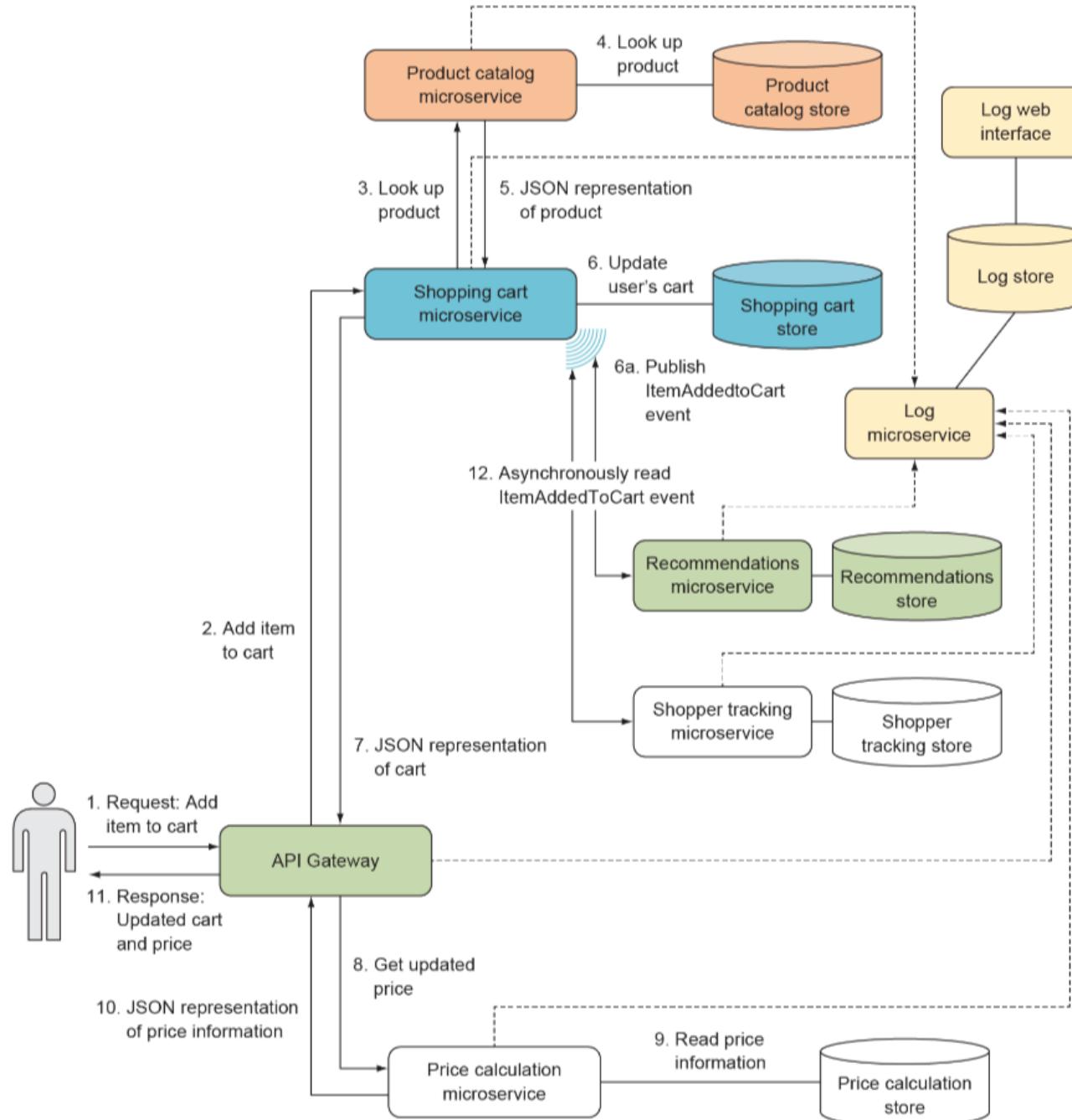
HTTP GET /health/startup



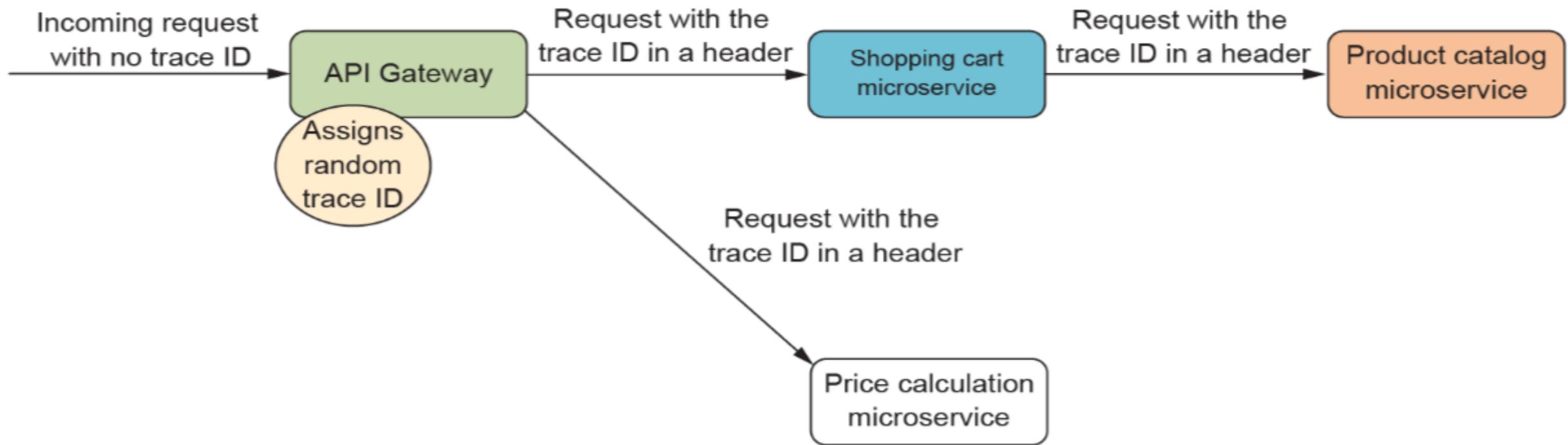
HTTP GET /health/live



Logging needs in microservices



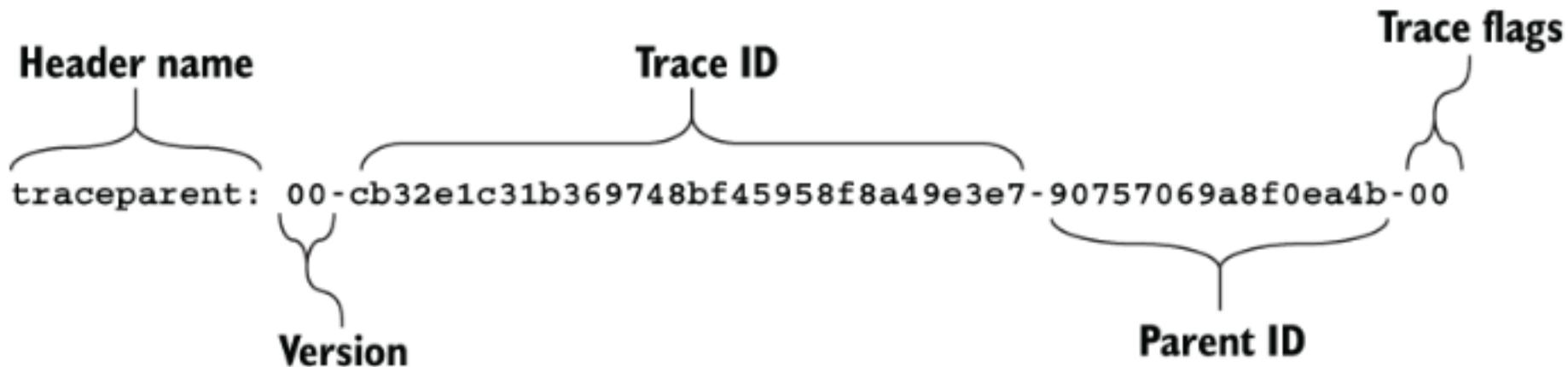
Tracing requests across microservices



- The format of the traceparent header is:

"version"- "trace id"- "parent id"- "trace flags"

- For instance, if the trace ID is cb32e1c31b369748bf45958f8a49e3e7 and the parent ID is 90757069a8f0ea4b, the trace parent header looks like this:



- This means a request from the shopping cart microservice to the product catalog microservice could look like this:

HTTP GET /products?productIds=[1,2] HTTP/1.1

Accept: application/json

traceparent: 00-cb32e1c31b369748bf45958f8a49e3e7-90757069a8f0ea4b-00

Structured logging with Serilog

- For instance, you can use Serilog in conjunction with .NET's own logger to send a log message like this—assuming that `log` is an instance the .NET type `Ilogger` configured to use Serilog:

```
var simplifiedRequest = new
{
    Path = "/foo",
    Method = "GET",
    Protocol = "HTTP",
};

var requestTime = 200;

log.LogInformation(
    "Processed request: {@Request} in {RequestTime:000}
ms.",
    simplifiedRequest,
    requestTime);
```

- `SimplifiedRequest` and `requestTime` are objects that are part of the message.
- Like other logging frameworks, Serilog lets you configure different sinks to which log messages are written.
- The sink you'll use here is the console sink.
- When the previous log message is written to the console, `simplifiedRequest` and `requestTime` are JSON serialized and inserted into the log message string.
- Serilog also adds some metadata, so the message written to the console looks like this:

2020-12-03 09:14:33 INF Processed request {"Path":"/foo", "Method":"GET", "Protocol":"HTTP"} in 200 ms.

Implementing the monitoring endpoints

The plan to implement the monitoring endpoints is as follows:

- Add the /health/live endpoint to the ASP.NET pipeline.
- Create the health check for the shopping cart database.
- Add the /health/startup endpoint to the pipeline.

Implementing the /health/live monitoring endpoint

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddHealthChecks()
        .AddCheck(
            "LivenessHealthCheck",
            () => HealthCheckResult.Healthy(),
            tags: new []{ "liveness" });
    services.AddControllers();
}
```

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    app.UseHttpsRedirection();
    app.UseHealthChecks("/health/live",
        new HealthCheckOptions
    {
        Predicate = x => x.Tags.Contains("liveness")
    });
    app.UseRouting();
    app.UseEndpoints(endpoints => endpoints.MapControllers());
}
```

Implementing the /health/startup monitoring endpoint

```
namespace ShoppingCart
{
    using System.Data.SqlClient;
    using System.Threading;
    using System.Threading.Tasks;
    using Dapper;
    using Microsoft.Extensions.Diagnostics.HealthChecks;

    public class DbHealthCheck : IHealthCheck
    {
        public async Task<HealthCheckResult> CheckHealthAsync(
            HealthCheckContext context,
            CancellationToken cancellationToken)
        {
            await using var conn =
                new SqlConnection(@"Data Source=localhost;
Initial Catalog=ShoppingCart;
User Id=SA; Password=yourStrong(!)Password");

            var result = await conn.QuerySingleAsync<int>("SELECT 1");
            return result == 1
                ? HealthCheckResult.Healthy()
                : HealthCheckResult.Degraded();
        }
    }
}
```

```
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddHealthChecks()
            .AddCheck<DbHealthCheck>(
                nameof(DbHealthCheck),
                tags: new []{ "startup" })
            .AddCheck(
                "LivenessHealthCheck",
                () => HealthCheckResult.Healthy(),
                tags: new []{ "liveness" });
        services.AddControllers();
    }

    public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
    {
        app.UseHttpsRedirection();
        app.UseHealthChecks(
            "/health/startup",
            new HealthCheckOptions
            {
                Predicate = x => x.Tags.Contains("startup"))
            });
        app.UseHealthChecks(
            "/health/live",
            new HealthCheckOptions
            {
                Predicate = x => x.Tags.Contains("liveness"))
            });
        app.UseRouting();
        app.UseEndpoints(endpoints => endpoints.MapControllers());
    }
}
```

- With this in place, you can run the shopping cart microservice with dotnet and test the monitoring endpoints.
- A request to the /health/live endpoint looks like this:

GET <https://localhost:5001/health/live>

- A successful response looks as follows:

HTTP/1.1 200 OK

Content-Type: text/plain

Healthy

- Here's a request to the /health/startup endpoint:

GET <https://localhost:5001/health/startup>

- And this is a successful response:

HTTP/1.1 200 OK

Content-Type: text/plain

Healthy

- Finally, the following is a failure response:

HTTP/1.1 503 Service Unavailable

Content-Type: text/plain

Unhealthy

Implementing structured logging

There are three small steps to making the logs structured:

- Adding the Serilog NuGet package and a helper package to the shopping cart microservice
- Configuring the .NET logger to use Serilog behind the scenes
- Changing the log output to a JSON format, which is more machine-friendly

Using Serilog as the backend for the .NET logger

```
namespace ShoppingCart
{
    using Microsoft.AspNetCore.Hosting;
    using Microsoft.Extensions.Hosting;
    using Serilog;
    using Serilog.Enrichers.Span;
    using Serilog.Formatting.Json;

    public class Program
    {
        public static void Main(string[] args)
        {
            CreateHostBuilder(args).Build().Run();
        }

        private static IHostBuilder CreateHostBuilder(string[] args) =>
            Host.CreateDefaultBuilder(args)
                .UseSerilog((context, logger) =>
                {
                    logger
                        .Enrich.FromLogContext()
                        .WriteTo.ColoredConsole();
                })
                .ConfigureWebHostDefaults(x => x.UseStartup<Startup>());
    }
}
```

```
PS > dotnet run
2020-12-03 10:24:34 INF Now listening on: "https://localhost:5001"
2020-12-03 10:24:34 INF Now listening on: "http://localhost:5000"
2020-12-03 10:24:34 INF Application started. Press Ctrl+C to shut down.
2020-12-03 10:24:34 INF Hosting environment: "Development"
2020-12-03 10:24:34 INF Content root path: "C:\\Users\\chors\\Documents\\
horsdal3\\code\\Module10\\ShoppingCart"
```

Taking a dependency on ILogger to be able to log from controllers

```
public ShoppingCartController(  
    IShoppingCartStore shoppingCartStore,  
    IProductCatalogClient productCatalog,  
    IEventStore eventStore,  
    ILogger<ShoppingCartController> logger)  
{  
    this.shoppingCartStore = shoppingCartStore;  
    this.productCatalog = productCatalog;  
    this.eventStore = eventStore;  
    this.logger = logger;  
}
```

Including structure in logs from controllers

```
[HttpPost("{userId:int}/items")]
public async Task<ShoppingCart> Post(int userId, [FromBody] int[]
    productIds)
{
    var shoppingCart = await shoppingCartStore.Get(userId);
    var shoppingCartItems = await
        this.productCatalog.GetShoppingCartItems(productIds);
    await shoppingCart.AddItems(shoppingCartItems, this.eventStore);
    await this.shoppingCartStore.Save(shoppingCart);

    this.logger.LogInformation(
        "Successfully added products to shopping cart {@productIds},
        {@shoppingCart}", productIds, shoppingCart);

    return shoppingCart;
}
```

Structured log message including a shopping cart

If we make an HTTP post to add items to the cart now, one of the log messages we will see in the console is as follows.

```
2020-12-03 11:37:08 INF Successfully added products to shopping cart [1, 2],  
{"userId":123,"items":[{"productCatalogueId":1,"productName":  
"Basic t-shirt","description":"a quiet t-shirt","price":{"currency":  
"eur","amount":40}},{"productCatalogueId":2,"productName":"Fancy shirt",  
"description":"a loud t-shirt","price":{"currency":"eur","amount":50}}]}
```

Using JSON logs in production

```
private static IHostBuilder CreateHostBuilder(string[] args) =>
    Host.CreateDefaultBuilder(args)
        .UseSerilog((context, logger) =>
    {
        logger
            .Enrich.FromLogContext();
        if (context.HostingEnvironment.IsDevelopment())
            logger.WriteTo.ColoredConsole();
        else
            logger.WriteTo.Console(new JsonFormatter());
    })
    .ConfigureWebHostDefaults(x => x.UseStartup<Startup>());
```

The JSON-formatted log messages created when we start the shopping cart microservice in production look like this:

```
{"Timestamp":"2020-12-03T12:00:41.5723302+01:00","Level":"Information",
 "MessageTemplate":"Now listening on: {address}","Properties": {"address":
 "https://localhost:5001","SourceContext":"Microsoft.Hosting.Lifetime"}}
 {"Timestamp":"2020-12-03T12:00:41.5803748+01:00","Level":"Information",
 "MessageTemplate":"Now listening on: {address}","Properties": {"address":
 "http://localhost:5000","SourceContext":"Microsoft.Hosting.Lifetime"}}
 {"Timestamp":"2020-12-03T12:00:41.5809434+01:00","Level":"Information",
 "MessageTemplate":"Application started. Press Ctrl+C to shut down.",
 "Properties": {"SourceContext":"Microsoft.Hosting.Lifetime"}}
 {"Timestamp":"2020-12-03T12:00:41.5812286+01:00","Level":"Information",
 "MessageTemplate":"Hosting environment: {envName}","Properties": {
 "envName":"Production","SourceContext":"Microsoft.Hosting.Lifetime"}}
 {"Timestamp":"2020-12-03T12:00:41.5815246+01:00","Level":"Information",
 "MessageTemplate":"Content root path: {contentRoot}","Properties": {
 "contentRoot":"C:\\\\Users\\\\chors\\\\Documents\\\\horsdal3\\\\code\\\\Module10\\\\
 ShoppingCart","SourceContext":"Microsoft.Hosting.Lifetime"}}
```

Adding a trace ID to all log messages

```
private static IHostBuilder CreateHostBuilder(string[] args) =>
    Host.CreateDefaultBuilder(args)
        .UseSerilog((context, logger) =>
    {
        logger
            .Enrich.FromLogContext()
            .Enrich.WithSpan();
        if (context.HostingEnvironment.IsDevelopment())
            logger.WriteTo.ColoredConsole(
                outputTemplate: @"""{Timestamp:yyyy-MM-dd HH:mm:ss}
{TraceId} {Level:u3} {Message}{NewLine}{Exception}"""
            );
        else
            logger.WriteTo.Console(new JsonFormatter());
    })
    .ConfigureWebHostDefaults(x => x.UseStartup<Startup>());
}
```

```
2020-12-07 11:10:39 f2e604d90da6114a8d4a77daa57a7890 INF Request starting
    HTTP/1.1 POST https://localhost:5001/users/0/items application/json 9
2020-12-07 11:10:39 f2e604d90da6114a8d4a77daa57a7890 INF Executing endpoint
    '"ShoppingCart.Users UsersController.Post (ShoppingCart)"'
2020-12-07 11:10:39 f2e604d90da6114a8d4a77daa57a7890 INF Route matched with
    "{action = \"Post\", controller = \"Users\"}". Executing controller action
    with signature "Task<Shoppingcart> Post(Int32, Int32[])" on controller
    "Shoppingcart.Users UsersController" ("ShoppingCart").
2020-12-07 11:10:39 f2e604d90da6114a8d4a77daa57a7890 INF Successfully added
    products to shopping cart [1, 2], {"userId":123,"items":[{"productCatalogueId":1,"productName":"Basic t-shirt","description":
        "a quiet t-shirt","price":{"currency":"eur","amount":40}}, {"productCatalogueId":2,"productName":"Fancy shirt","description":
        "a loud t-shirt","price":{"currency":"eur","amount":50}}]}
2020-12-07 11:10:39 f2e604d90da6114a8d4a77daa57a7890 INF Executed action
    "ShoppingCart.Users UsersController.Post (ShoppingCart)" in 45.891ms
2020-12-07 11:10:39 f2e604d90da6114a8d4a77daa57a7890 INF Executed endpoint
    '"ShoppingCart.Users UsersController.Post (ShoppingCart)"'
2020-12-07 11:10:39 f2e604d90da6114a8d4a77daa57a7890 INF Request finished
    HTTP/1.1 POST https://localhost:5001/users/0/items application/json 9
    - 200 0 - 193.4576ms
```

```

1 "Protocol":"HTTP/1.1","Method":"POST","ContentType":"application/json",
2 "ContentLength":9,"Scheme":"https","Host":"localhost:5001","PathBase":"",
3 "Path":"/users/0/items","QueryString":"","HostingRequestStartingLog":
4 "Request starting HTTP/1.1 POST https://localhost:5001/users/0/items
5 application/json 9","EventId":{"Id":1,"Name":"RequestStarting"},
6 "SourceContext":"Microsoft.AspNetCore.Hosting.Diagnostics","RequestId":
7 "0HM4QJ9RN6M7Q:00000002","RequestPath":"/users/0/items","ConnectionId":
8 "0HM4QJ9RN6M7Q","SpanId":"469950e2392d4a4b","TraceId":
9 "7e413f37415aea4d8c4ff04924292ad2","ParentId":"0000000000000000"},
10 "Renderings": {"HostingRequestStartingLog": [{"Format": "l", "Rendering": "
11 "Request starting HTTP/1.1 POST https://localhost:5001/users/0/items
12 application/json 9"}]}}
13 {"Timestamp":"2020-12-07T11:25:49.4047774+01:00","Level":"Information",
14 "MessageTemplate":"Executing endpoint '{EndpointName}'","Properties"
15 :{"EndpointName":"ShoppingCart.Users.UsersController.Post (ShoppingCart)",
16 "EventId":{"Name":"ExecutingEndpoint"},"SourceContext":
17 "Microsoft.AspNetCore.Routing.EndpointMiddleware","RequestId":
18 "0HM4QJ9RN6M7Q:00000002","RequestPath":"/users/0/items","ConnectionId":
19 "0HM4QJ9RN6M7Q","SpanId":"469950e2392d4a4b","TraceId":
20 "7e413f37415aea4d8c4ff04924292ad2","ParentId":"0000000000000000"}}
21 {"Timestamp":"2020-12-07T11:25:49.4527811+01:00","Level":"Information",
22 "MessageTemplate":"Route matched with {RouteData}. Executing controller
23 action with signature {MethodInfo} on controller {Controller} ({
24     AssemblyName}),"Properties": {"RouteData": {"action": "\\"Post\"",
25     controller: "\\"Users\\""}, "MethodInfo": "Void Post(Int32, Int32[])",
26     "Controller": "ShoppingCart.Users.UsersController", "AssemblyName":
27     "ShoppingCart", "EventId": {"Id": 3, "Name": "ControllerActionExecuting"},
28     "SourceContext": "Microsoft.AspNetCore.Mvc.Infrastructure.
29     ControllerActionInvoker", "ActionId": "68aa2aa9-04a9-449d-ab84-
30     cb196a437710", "ActionName": "ShoppingCart.Users.UsersController.Post
31     (ShoppingCart)", "RequestId": "0HM4QJ9RN6M7Q:00000002", "RequestPath":
32     "/users/0/items", "ConnectionId": "0HM4QJ9RN6M7Q", "SpanId":
33     "469950e2392d4a4b", "TraceId": "7e413f37415aea4d8c4ff04924292ad2",
34     "ParentId": "0000000000000000"}}
35 {"Timestamp":"2020-12-07T11:25:49.4982026+01:00","Level":"Information",
36 "MessageTemplate":"Successfully added products to shopping cart
37 {@productIds}, {@shoppingCart}","Properties": {"productIds": [1, 2, 3],
38     "shoppingCart": {"userId": 123, "items": [{"productCatalogueId": 1,
39     "productName": "Basic t-shirt", "description": "a quiet t-shirt",
40     "price": {"currency": "eur", "amount": 40}}], "productCatalogueId": 2,

```

```
41 "productName":"Fancy shirt","description":"a loud t-shirt",
42 "price":{ "currency":"eur","amount":50}}]}]} "SourceContext":
43 "ShoppingCart.Users.UsersController","ActionId":
44 "68aa2aa9-04a9-449d-ab84-cb196a437710","ActionName":
45 "ShoppingCart.Users.UsersController.Post (ShoppingCart)","RequestId":
46 "0HM4QJ9RN6M7Q:00000002","RequestPath":"/users/0/items","ConnectionId":
47 "0HM4QJ9RN6M7Q","SpanId": "469950e2392d4a4b","TraceId":
48 "7e413f37415aea4d8c4ff04924292ad2","ParentId": "0000000000000000"}}
49 {"Timestamp": "2020-12-07T11:25:49.5054475+01:00", "Level": "Information",
50 "MessageTemplate": "Executed action {ActionName} in {
51     ElapsedMilliseconds}ms", "Properties": {"ActionName":
52     "ShoppingCart.Users.UsersController.Post (ShoppingCart)",
53     "ElapsedMilliseconds": 45.7401, "EventId": {"Id": 2, "Name":
54     "ActionExecuted"}, "SourceContext": "Microsoft.AspNetCore.Mvc.
55     Infrastructure.ControllerActionInvoker", "ActionId": "68aa2aa9-04a9-
56     449d-ab84-cb196a437710", "RequestId": "0HM4QJ9RN6M7Q:00000002",
57     "RequestPath": "/users/0/items", "ConnectionId": "0HM4QJ9RN6M7Q",
58     "SpanId": "469950e2392d4a4b", "TraceId":
59     "7e413f37415aea4d8c4ff04924292ad2", "ParentId": "0000000000000000"}}
60 {"Timestamp": "2020-12-07T11:25:49.5069173+01:00", "Level": "Information",
61 "MessageTemplate": "Executed endpoint '{EndpointName}'", "Properties":
62 {"EndpointName": "ShoppingCart.Users.UsersController.Post (ShoppingCart)",
63 "EventId": {"Id": 1, "Name": "ExecutedEndpoint"}, "SourceContext": "Microsoft.
64     AspNetCore.Routing.EndpointMiddleware", "RequestId":
65     "0HM4QJ9RN6M7Q:00000002", "RequestPath": "/users/0/items", "ConnectionId":
66     "0HM4QJ9RN6M7Q", "SpanId": "469950e2392d4a4b", "TraceId":
67     "7e413f37415aea4d8c4ff04924292ad2", "ParentId": "0000000000000000"}}
68 {"Timestamp": "2020-12-07T11:25:49.5124936+01:00", "Level": "Information",
69 "MessageTemplate": "{HostingRequestFinishedLog:1}", "Properties":
70 {"ElapsedMilliseconds": 165.5149, "StatusCode": 200, "ContentType": null,
71 "ContentLength": 0, "Protocol": "HTTP/1.1", "Method": "POST", "Scheme": "https",
72 "Host": "localhost:5001", "PathBase": "", "Path": "/users/0/items", "QueryString":
73 "", "HostingRequestFinishedLog": "Request finished HTTP/1.1 POST
74 https://localhost:5001/users/0/items application/json 9 - 200 0
75 - 165.5149ms", "EventId": {"Id": 2, "Name": "RequestFinished"}, "SourceContext":
76 "Microsoft.AspNetCore.Hosting.Diagnostics", "RequestId":
77 "0HM4QJ9RN6M7Q:00000002", "RequestPath": "/users/0/items", "ConnectionId":
78 "0HM4QJ9RN6M7Q", "SpanId": "469950e2392d4a4b", "TraceId":
79 "7e413f37415aea4d8c4ff04924292ad2", "ParentId": "0000000000000000"}, "Renderings": [{"HostingRequestFinishedLog": [{"Format": "l", "Rendering": "Request finished HTTP/1.1 POST https://localhost:5001/users/0/items
81 application/json 9 - 200 0 - 165.5149ms"}]}]}
```

Trace ID is included in outgoing HTTP requests

- The requests the shopping cart microservice makes to the product catalog microservice already include traceparent headers.
- The trace IDs are sent along to the product catalog microservice, meaning that we can trace requests across microservices in the logs.

Logging unhandled exceptions

- MVC will catch any otherwise unhandled exception from controllers and log the exception message.
- Like any other log message, these logs will include a trace ID.
- This means we can trace what led to the unhandled exception.
- We can find the original user request and all other log messages made along the way up to the point the exception was thrown.

Implementing monitoring and logging in Kubernetes

- There are two parts we want to use with Kubernetes:
 - *Monitoring*
 - *Logging*

Configure monitoring in Kubernetes

Recall that we created two monitoring endpoints:

- Startup
- Liveness

Kubernetes manifest with health probes

```
kind: Deployment
apiVersion: apps/v1
metadata:
  name: shopping-cart
spec:
  replicas: 1
  selector:
    matchLabels:
      app: shopping-cart
  template:
    metadata:
      labels:
        app: shopping-cart
    spec:
      containers:
        - name: shopping-cart
          image: shopping-cart
          imagePullPolicy: IfNotPresent
          ports:
            - containerPort: 80
          env:
            - name: STARTUPDLL
              value: "api/ShoppingCart.dll"
          livenessProbe:
            httpGet:
              path: /health/live
              port: 80
              periodSeconds: 30
          startupProbe:
            httpGet:
              path: /health/startup
              port: 80
              initialDelaySeconds: 10
              periodSeconds: 10
---
apiVersion: v1
kind: Service
metadata:
  name: shopping-cart
spec:
  type: LoadBalancer
  ports:
    - name: shopping-cart
      port: 5001
      targetPort: 80
  selector:
    app: shopping-cart
```

We can test if the container is running as expected with `kubectl get pods`, which should show that the shopping cart pod is running and has not been restarted with a line like this one:

| NAME | READY | STATUS | RESTARTS | AGE |
|--------------------------------|-------|---------|----------|------|
| shopping-cart-54ff7c4846-rvkcc | 1/1 | Running | 0 | 8m3s |

- To get all the logs for the shopping cart deployment, we can run this command to print the logs from the shopping cart deployment:

```
kubectl logs -f --tail=10 deployment.apps/shopping-cart
```

```
{"Timestamp":"2020-12-07T13:15:21.1589211+00:00", "Level": "Information",  
 "MessageTemplate": "{HostingRequestStartingLog:1}", "Properties":  
 {"Protocol": "HTTP/1.1", "Method": "GET", "ContentType": null, "ContentLength":  
 null, "Scheme": "http", "Host": "10.1.0.22:80", "PathBase": "", "Path":  
 "/health/live", "QueryString": "", "HostingRequestStartingLog":  
 "Request starting HTTP/1.1 GET http://10.1.0.22:80/health/live - -",  
 "EventId": {"Id": 1, "Name": "RequestStarting"}, "SourceContext":  
 "Microsoft.AspNetCore.Hosting.Diagnostics", "RequestId":  
 "0HM4QLU8CA00I:00000002", "RequestPath": "/health/live", "ConnectionId":  
 "0HM4QLU8CA00I", "SpanId": "a2c71b41ba089b44", "TraceId":  
 "ca68d09e29e61e4d92d363cb3bfa4827", "ParentId": "0000000000000000"},  
 "Renderings": {"HostingRequestStartingLog": [{"Format": "l", "Rendering":  
 "Request starting HTTP/1.1 GET http://10.1.0.22:80/health/live - -"}]}}  
 {"Timestamp": "2020-12-07T13:15:21.1591769+00:00", "Level": "Information",  
 "MessageTemplate": "{HostingRequestFinishedLog:1}", "Properties":  
 {"ElapsedMilliseconds": 0.3009, "StatusCode": 200, "ContentType": "text/plain",  
 "ContentLength": null, "Protocol": "HTTP/1.1", "Method": "GET", "Scheme": "http",  
 "Host": "10.1.0.22:80", "PathBase": "", "Path": "/health/live", "QueryString":  
 "", "HostingRequestFinishedLog": "Request finished HTTP/1.1  
 GET http://10.1.0.22:80/health/live - - 200 - text/plain 0.3009ms",  
 "EventId": {"Id": 2, "Name": "RequestFinished"}, "SourceContext":  
 "Microsoft.AspNetCore.Hosting.Diagnostics", "RequestId":  
 "0HM4QLU8CA00I:00000002", "RequestPath": "/health/live", "ConnectionId":  
 "0HM4QLU8CA00I", "SpanId": "a2c71b41ba089b44", "TraceId":  
 "ca68d09e29e61e4d92d363cb3bfa4827", "ParentId": "0000000000000000"},  
 "Renderings": {"HostingRequestFinishedLog": [{"Format": "l", "Rendering":  
 "Request finished HTTP/1.1 GET http://10.1.0.22:80/health/live - -  
 - 200 - text/plain 0.3009ms"}]}}
```

Summary

- Microservice systems need monitoring and logging just like any other server-side systems.
- Every microservice should be monitored and sending log messages.
- Serilog is a library for doing structured logging.
- ASP.NET stores trace information in `Activity.Current`.
- Kubernetes supports health probes that fit well with our monitoring endpoints.
- Kubernetes allows us to easily access the logs from our containers.

10. Securing microservice-to-microservice communication

Introduction

This Module covers

- Where to perform user authentication and authorization in a microservice system
- Deciding on the level of trust in your microservice system
- Limiting microservice-to-microservice requests
- Using an API Gateway to authenticate users
- Using Kubernetes network policies to limit microservice-to-microservice requests

Microservice security concerns

We'll concentrate on two areas of security that are relevant to developers of microservice systems:

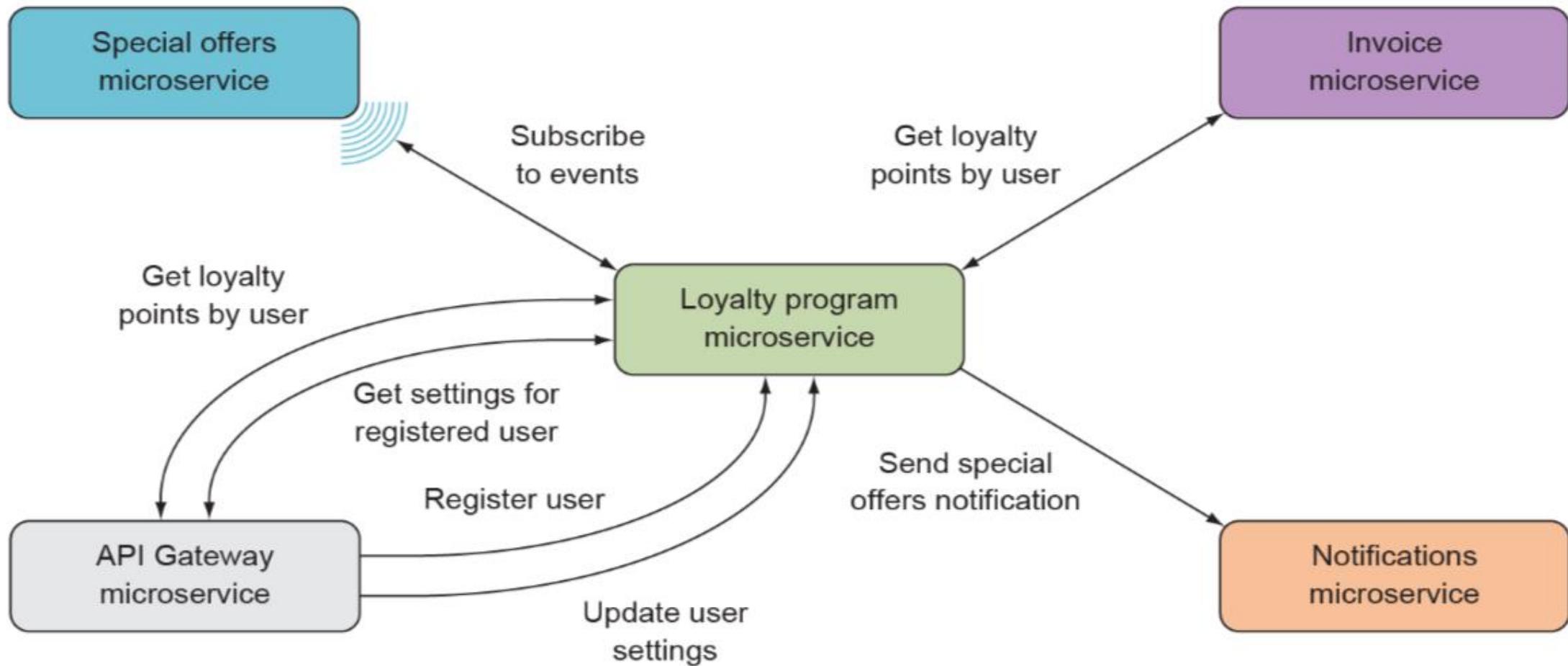
- Authentication and authorization
- How to limit communication between microservices

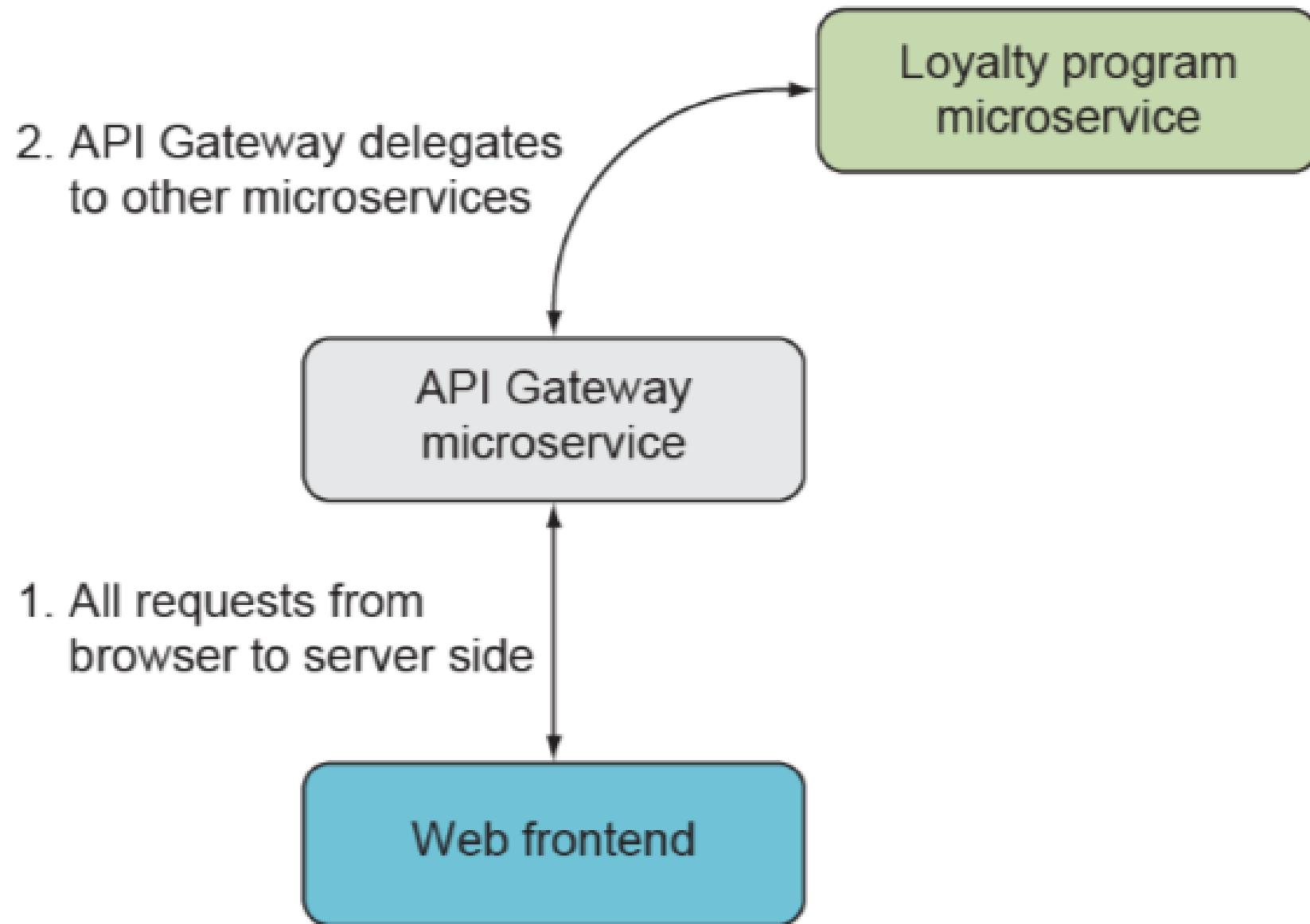
DATA IN MOTION VS. DATA AT REST

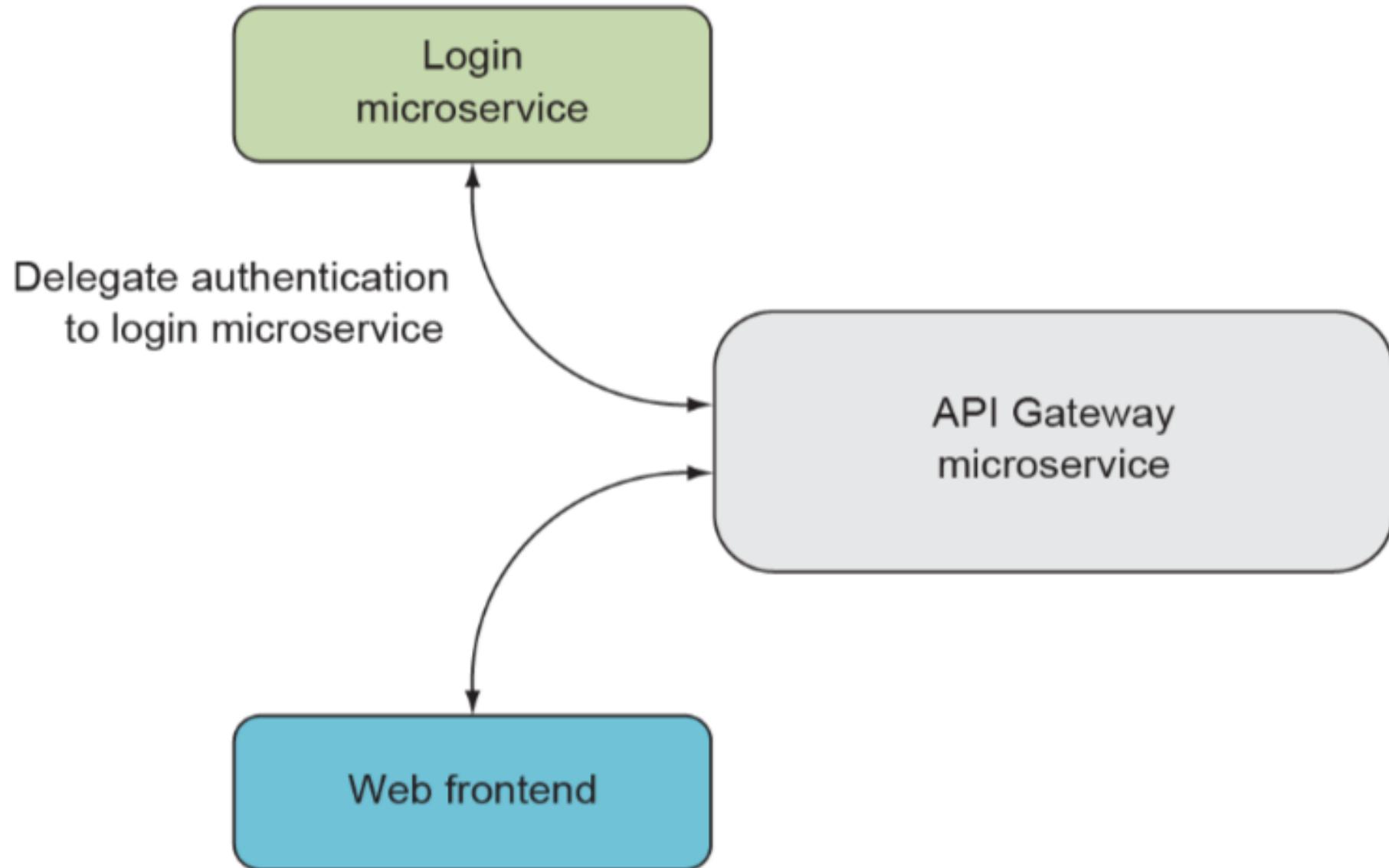
The data your systems handle is important and needs to be handled safely. Broadly speaking, we can place data handling in one of two categories:

- *Data in motion*
- *Data at rest*

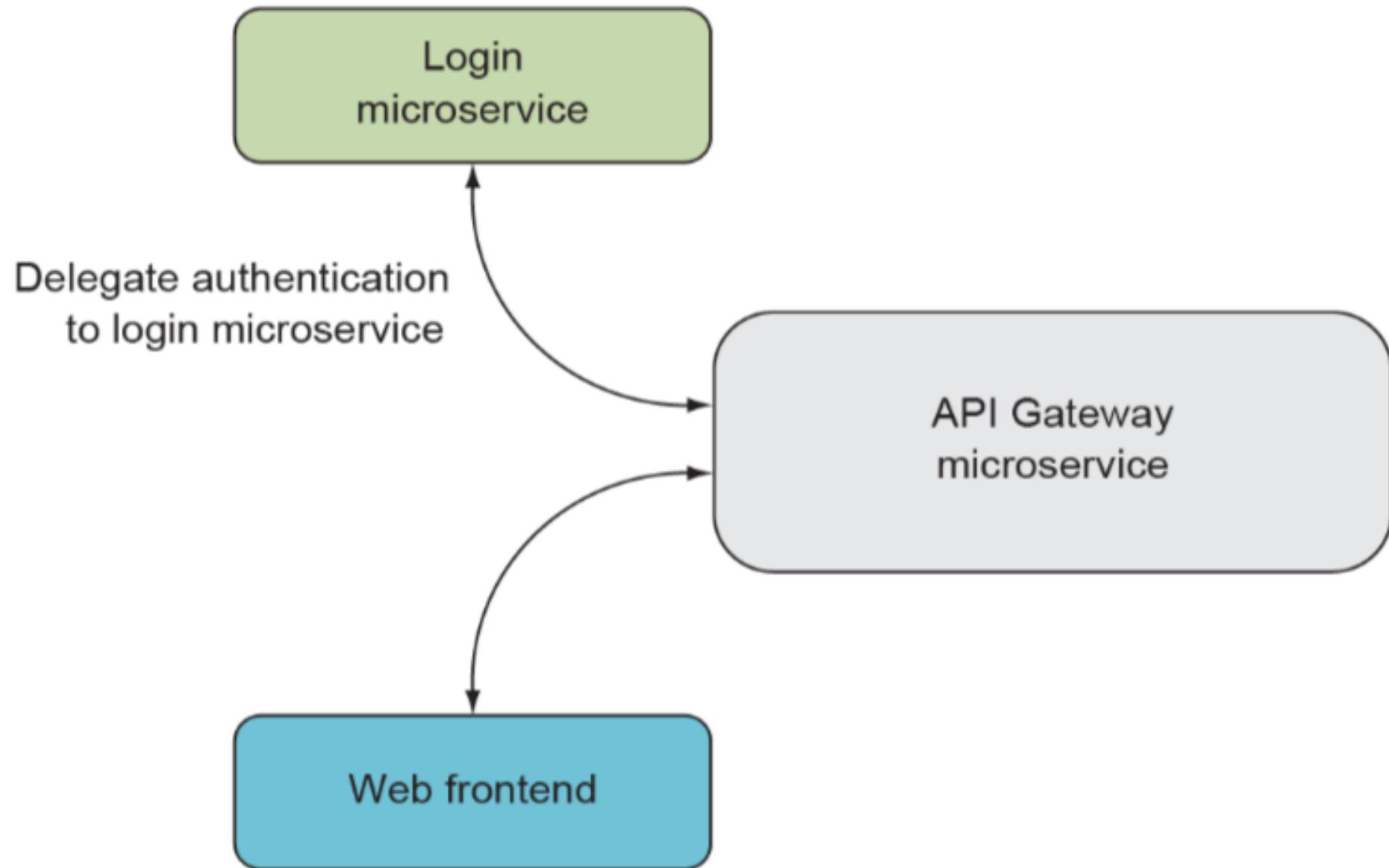
Authenticating users at the edge







Authorizing users in microservices



How much should microservices trust each other?

- At the highest level of trust between microservices, all microservices completely trust every request and every response from any other microservice.
- The microservices you've built so far in this book have, implicitly, had this high level of trust.

DEFENSE IN DEPTH

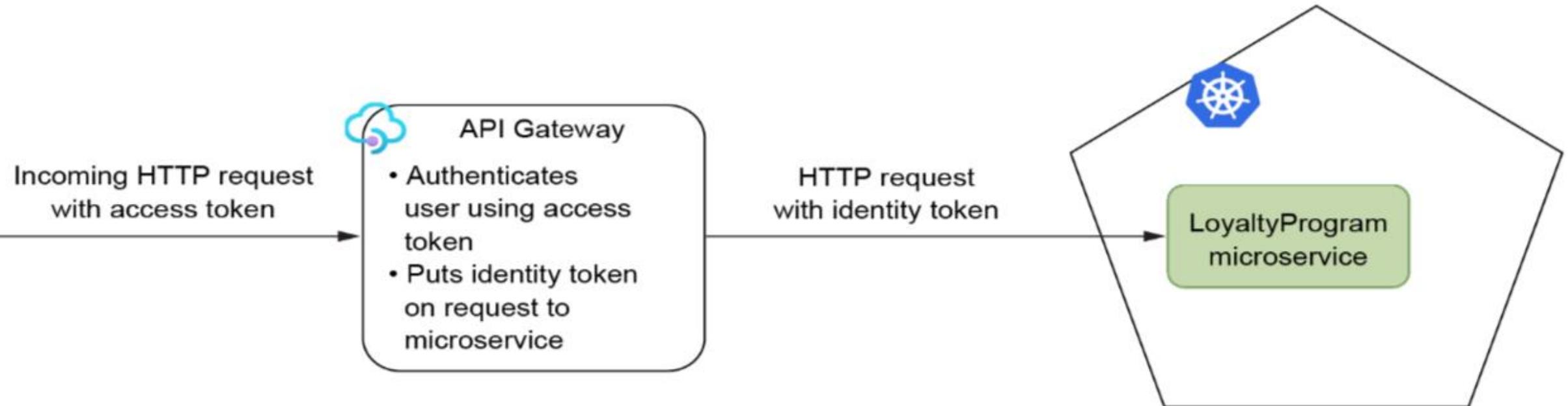
- Defense in depth is an approach to security that uses several defense mechanisms in combination.
- The idea is to employ a layering strategy: if an attacker is able to get past the first line of defense, they meet the next line of defense.
- Even if an attacker circumvents the authorization and tricks a microservice into executing uploaded code, the attacker still doesn't have full control over the server.

Implementing secure microservice-to-microservice communication

The security requirements are as follows:

- Authenticate users in the API Gateway.
- Allow microservices to access the user identity.
- Limit which microservices may collaborate.

Accessing the user identity in the loyalty program



Accessing the user identity in the loyalty program

In the following sections, you'll learn how to use the identity token in the loyalty program by

- Validating the token using ASP.NET authentication middleware
- Authorizing in MVC controllers

Configuration to use JWT bearer tokens by default

- In the loyalty program, we can change the ConfigureServices method in Startup to the following.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
        .AddJwtBearer(x => x.TokenValidationParameters =
            new TokenValidationParameters
            {
                ValidAudience = ...,
                ValidIssuer = ...,
                IssuerSigningKey = ...,
            });
    services.AddControllers();
}
```

Using authorization middleware to make identities available to controllers

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    app.UseHttpsRedirection();
    app.UseRouting();
    app.UseAuthorization();
    app.UseEndpoints(endpoints => endpoints.MapControllers());
}
```

- To protect the PUT endpoint used to update existing users, we can add the Authorize attribute to the UpdateUser method in the UsersController:

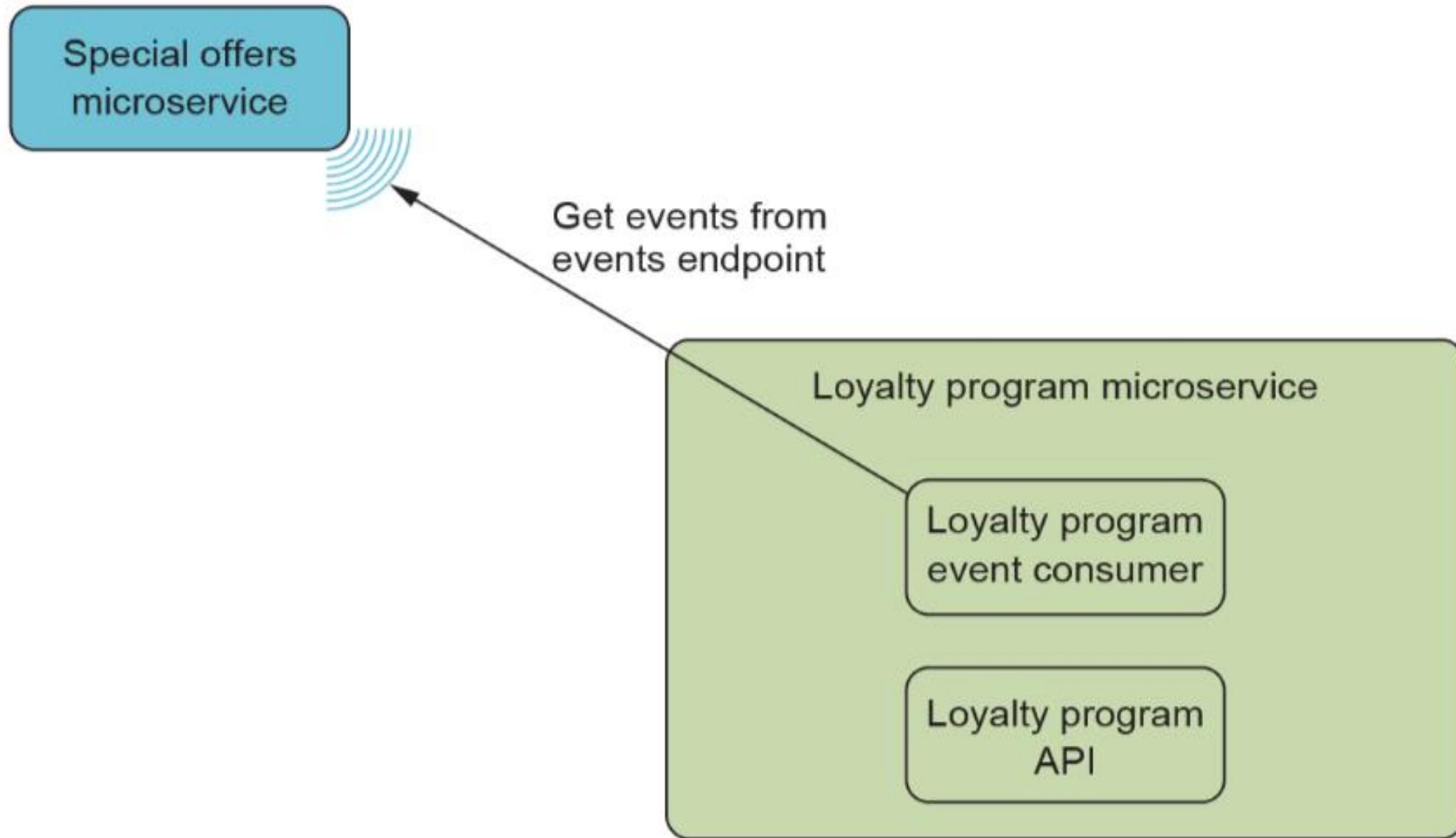
```
[HttpPut("{userId:int}")]
[Authorize(AuthenticationSchemes =
    JwtBearerDefaults.AuthenticationScheme)]
public LoyaltyProgramUser UpdateUser(
    int userId,
    [FromBody] LoyaltyProgramUser user) =>
    return RegisteredUsers[userId] = user;
```

Inspecting the claims on the user

```
[HttpPut("{userId:int}")]
[Authorize(AuthenticationSchemes = JwtBearerDefaults.AuthenticationScheme)]
public ActionResult<LoyaltyProgramUser> UpdateUser(
    int userId,
    [FromBody] LoyaltyProgramUser user)
{
    var hasUserId = int.TryParse(
        this.User.Claims.FirstOrDefault(c => c.Type == "userid")?.Value,
        out var userIdFromToken);
    if (!hasUserId || userId != userIdFromToken)
        return Unauthorized();

    return RegisteredUsers[userId] = user;
}
```

Limiting which microservices can communicate



Special offers microservice Kubernetes manifest

```
kind: Deployment
apiVersion: apps/v1
metadata:
  name: special-offers
spec:
  replicas: 1
  selector:
    matchLabels:
      app: special-offers
  template:
    metadata:
      labels:
        app: special-offers
    spec:
      containers:
        - name: special-offers
          image: your_unique_registry_name.azurecr.io/
              special-offers:1.0.0
          imagePullPolicy: IfNotPresent
          ports:
            - containerPort: 80
...
apiVersion: v1
kind: Service
metadata:
  name: special-offers
spec:
  type: LoadBalancer
  ports:
    - name: special-offers
      port: 5002
      targetPort: 80
  selector:
    app: special-offers
```

Modified manifest for special offers with a network policy

```
kind: Deployment
apiVersion: apps/v1
metadata:
  name: special-offers
spec:
  replicas: 1
  selector:
    matchLabels:
      app: special-offers-pod
  template:
    metadata:
      labels:
        app: special-offers-pod
    spec:
      containers:
        - name: special-offers
          image: your_unique_registry_name.azurecr.io/
            special-offers:1.0.0
          imagePullPolicy: IfNotPresent
          ports:
            - containerPort: 80
---
apiVersion: v1
kind: Service
metadata:
  name: special-offers
spec:
  selector:
    app: special-offers-pod
  ports:
    - port: 5002
      targetPort: 80
---
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: special-offers-network-policy
spec:
  podSelector:
    matchLabels:
      app: special-offers-pod
  policyTypes:
    - Ingress
  ingress:
    - from:
        - podSelector:
            matchLabels:
              app: loyalty-program-consumer
```

Summary

- Users should be authenticated at the edge of the system. That is, the microservice that first receives a user request should initiate authentication.
- You can set up the login microservice using a number of different off-the-shelf products.
- You can use an identity token to pass the user identity around in a more secure way than including a user ID as plain text.
- You can read the identity token from incoming requests and use the claims in it to perform authorization in controller actions.

11. Building a reusable microservice platform

Introduction

This Module covers

- Creating microservices more quickly with a reusable platform
- Components of a reusable platform
- Handling concerns that cut across several microservices
- Understanding middleware and the ASP.NET middleware pipelines
- Packaging reusable code with NuGet
- Building a reusable platform from several NuGet packages

Creating a new microservice should be quick and easy

- Another characteristic of microservices from Module 1 is as follows: *a microservice is replaceable.*
- The point is that a microservice can be completely rewritten quickly if its implementation becomes unsuitable.
- The code may get out of hand, or the design and technology choices you make early on may not be suitable for a growing load on the microservice.
- You'll sometimes need to replace an existing microservice with a new one.

Handling cross-cutting concerns

HTTP API: Accessible from other microservices

Shopping cart HTTP API

- Endpoint to get a shopping cart
- Endpoint to add items to shopping cart
- Endpoint to delete items from shopping cart

Event feed HTTP API

- Endpoint to get events

Shopping cart domain model

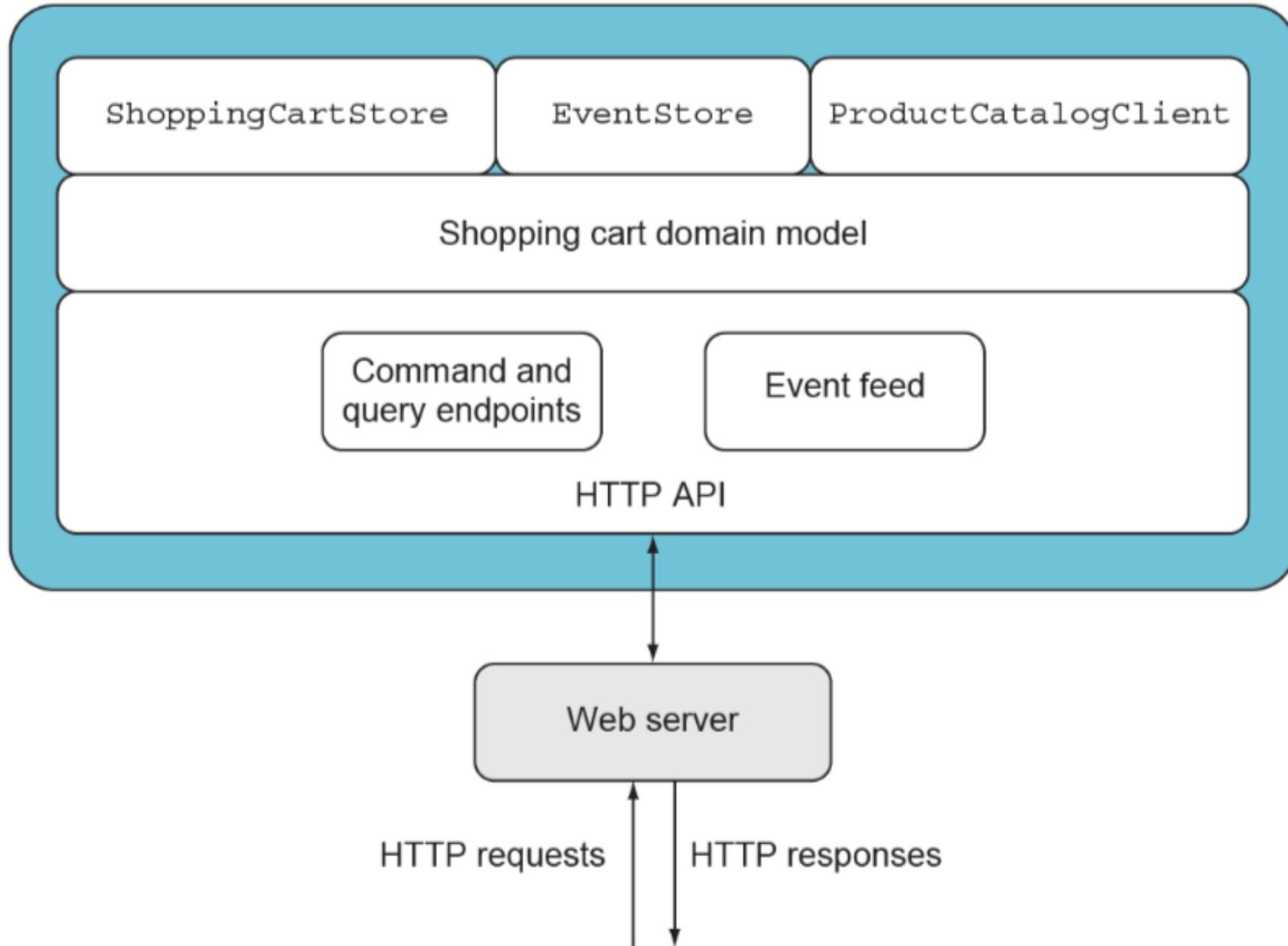
Shopping cart store

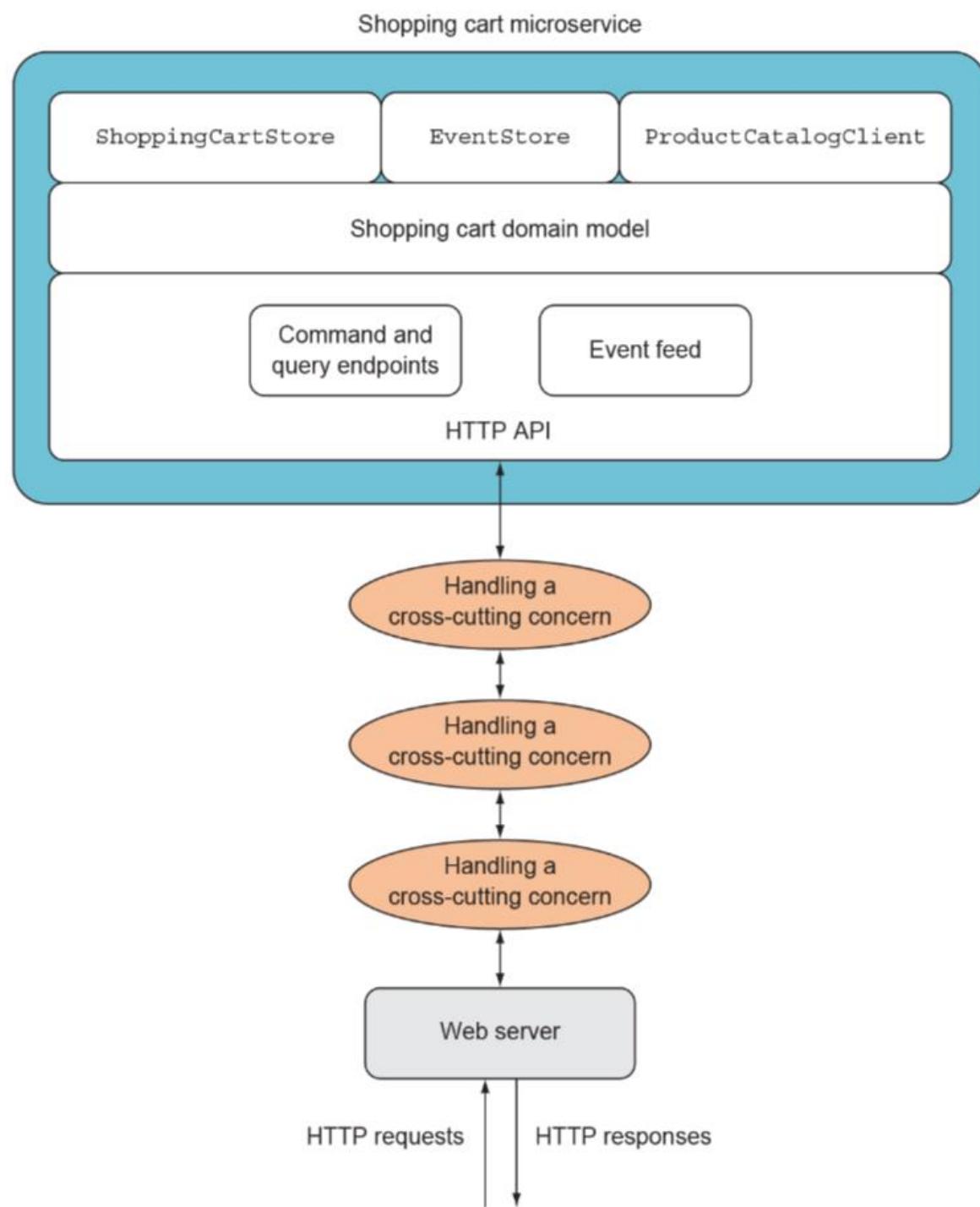
ProductCatalogClient

EventStore

Shopping cart database

Shopping cart microservice

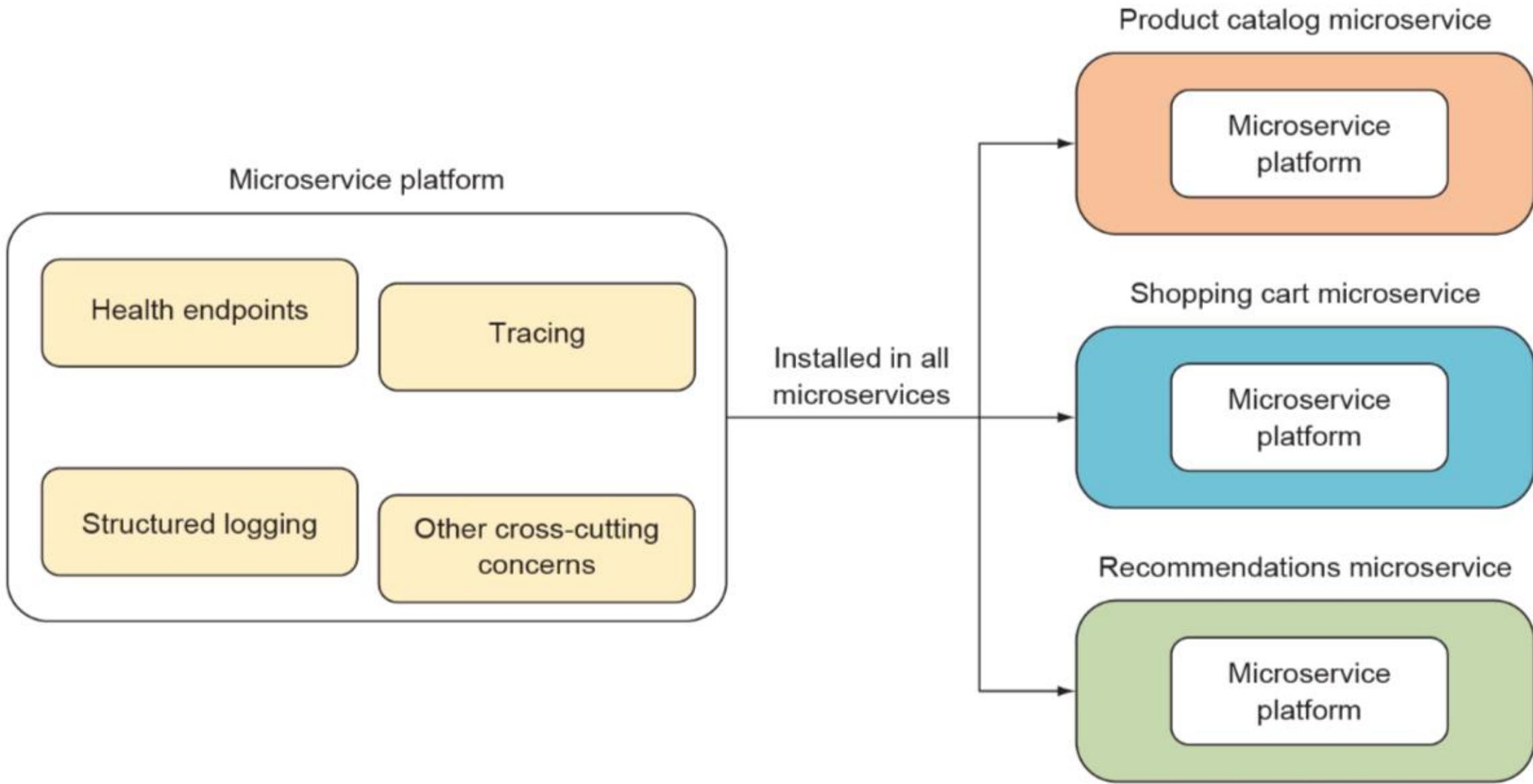




Creating a reusable microservice platform

You'll create a first iteration of such a platform, including the following:

- Monitoring endpoints that fit into Kubernetes
- Logging configuration that streamlines structured logging and tracing



Packaging and sharing cross-cutting code with NuGet

You'll build the platform from the following pieces:

- The health checks from Module 9
- The tracing from Module 9
- The structured logging from Module 9

Microservice platform

Logging package:

- Configure structure logging
- Configure log sink
- Add tracing to logs

Monitoring package:

- Basic health checks
- Additional checks
- Monitoring endpoints

Creating a logging package

In this section, you'll create the MicroserviceNET.Logging NuGet package by doing the following:

- Extract the structured logging and tracing configuration created in Module 9 from the shopping cart microservice to a class library called MicroserviceNET.Logging.
- Add a convenience method that makes it easy to add the monitoring and logging middleware to an ASP.NET pipeline.
- Create a NuGet package from the MicroserviceNET.Logging library.

Logger configuration

```
namespace MicroserviceNET.Logging
{
    using Microsoft.Extensions.Hosting;
    using Serilog;
    using Serilog.Enrichers.Span;
    using Serilog.Formatting.Json;

    public static class HostBuilderExtensions
    {
        public static IHostBuilder UseLogging(this IHostBuilder builder) =>
            builder.UseSerilog((context, logger) =>
            {
                logger
                    .Enrich.FromLogContext()
                    .Enrich.WithSpan();
                if (context.HostingEnvironment.IsDevelopment())
                    logger.WriteLineTo.ColoredConsole(
                        outputTemplate: "{Timestamp:yyyy-MM-dd HH:mm:ss} {TraceId} {
                            Level:u3} {Message}{NewLine}{Exception}");
                else
                    logger.WriteLineTo.Console(new JsonFormatter());
            });
    }
}
```

Using UseLogging in a Program.cs

```
using Microsoft.AspNetCore.Hosting;
```

```
using Microsoft.Extensions.Hosting;
```

```
using MicroserviceNET.Logging
```

```
CreateHostBuilder(args).Build().Run();
```

```
static IHostBuilder CreateHostBuilder(string[] args) =>
```

```
    Host.CreateDefaultBuilder(args)
```

```
        .UseLogging()
```

```
        .ConfigureWebHostDefaults(
```

```
            webBuilder => { webBuilder.UseStartup<Startup>(); });
```

- To create a NuGet package from the MicroserviceNET.Logging library, you can use dotnet. Go to the project folder—the one where the MicroserviceNET.Logging .csproj file is located—and run this command:

```
PS> dotnet pack --configuration Release
```

Creating a package with monitoring endpoints

There are three steps to creating the new package:

- Create a new class library called MicroserviceNET.Monitoring.
- Add the health check code from Module 9.
- Build the package with dotnet pack.

Convenient extension methods for adding health checks

```
namespace MicroserviceNET.Monitoring
{
    using Microsoft.Extensions.DependencyInjection;
    using Microsoft.Extensions.Diagnostics.HealthChecks;

    public static class ServiceCollectionExtensions
    {
        private const string Liveness = "liveness";
        private const string Startup = "startup";

        public static IServiceCollection AddBasicHealthChecks(
            this IServiceCollection services)
        {
            services.AddHealthChecks()
                .AddCheck("BasicStartupHealthCheck",
                    () => HealthCheckResult.Healthy(), tags: new[] {Startup})
                .AddCheck("BasicLivenessHealthCheck",
                    () => HealthCheckResult.Healthy(), tags: new[] {Liveness});

            return services;
        }

        public static IServiceCollection AddAdditionalStartupHealthChecks<T>(
            this IServiceCollection services) where T : class, IHealthCheck
        {
            services.AddHealthChecks().AddCheck<T>(nameof(T), tags: new[] {
                Startup});
            return services;
        }

        public static IServiceCollection AddAdditionalLivenessHealthChecks<T>(
            this IServiceCollection services) where T : class, IHealthCheck
        {
            services.AddHealthChecks().AddCheck<T>(nameof(T), tags: new[] {
                Liveness});
            return services;
        }
    }
}
```

Convenient extension method for adding health endpoints

```
namespace MicroserviceNET.Monitoring
{
    using Microsoft.AspNetCore.Builder;
    using Microsoft.AspNetCore.Diagnostics.HealthChecks;

    public static class ApplicationBuilderExtensions
    {
        public static IApplicationBuilder
            UseKubernetesHealthChecks(this IApplicationBuilder app) =>
        app
            .UseHealthChecks("/health/startup",
                new HealthCheckOptions
                {
                    Predicate = x => x.Tags.Contains("startup")
                })
            .UseHealthChecks("/health/live",
                new HealthCheckOptions
                {
                    Predicate = x => x.Tags.Contains("liveness")
                });
    }
}
```

Startup health check that uses an SQL Server database

```
namespace MicroserviceNET.Monitoring
{
    using System.Data.SqlClient;
    using System.Net.Http;
    using System.Threading;
    using System.Threading.Tasks;
    using Dapper;
    using Microsoft.Extensions.Diagnostics.Healthchecks;
    using Microsoft.Extensions.Logging;

    public class MsSqlStartupHealthCheck : IHealthCheck
    {
        private readonly HttpClient httpClient;

        public MsSqlStartupHealthCheck	ILoggerFactory logger)
        {
            this.httpClient = new HttpClient();
        }

        public async Task<HealthCheckResult> CheckHealthAsync(
            HealthCheckContext context,
            CancellationToken cancellationToken)
        {
            await using var conn = new SqlConnection(...);
            var result = await conn.QuerySingleAsync<int>("SELECT 1");
            return result == 1
                ? HealthCheckResult.Healthy()
                : HealthCheckResult.Degraded();
        }
    }
}
```

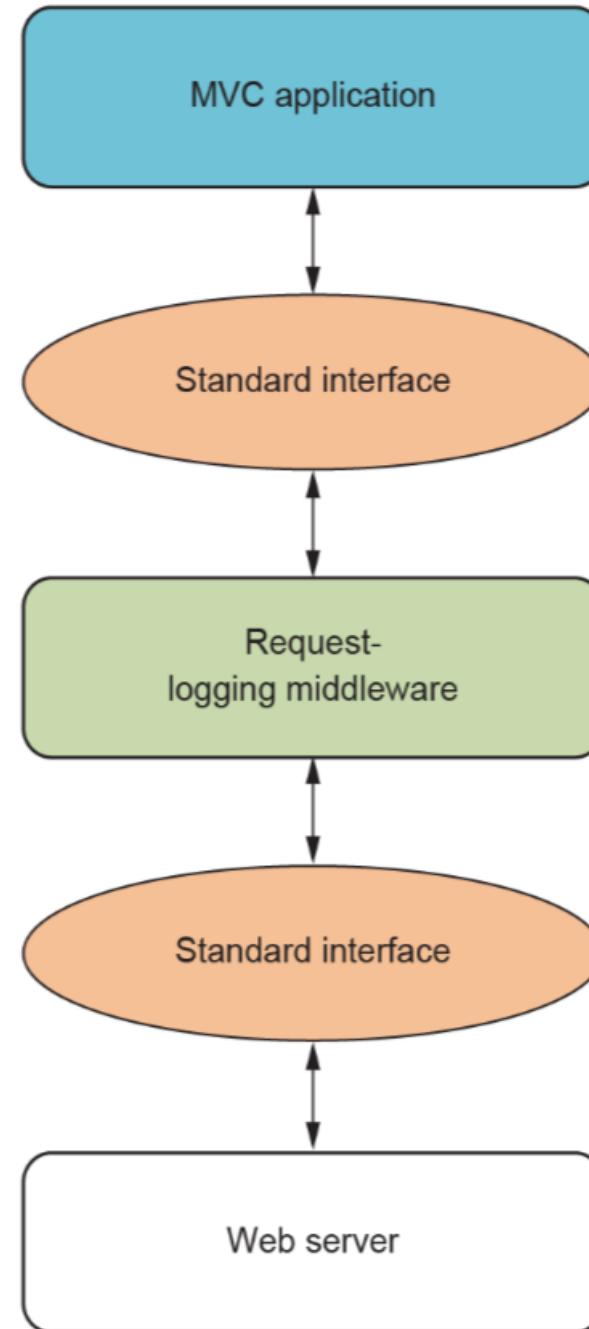
Registering health checks using the microservice platform

```
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        services
            .AddBasicHealthChecks()
            .AddAdditionalStartupHealthChecks<MsSqlStartupHealthCheck>();
    }
}
```

Adding the standardized monitoring endpoints

```
public void Configure(IApplicationBuilder app)  
{  
    app.UseRouting();  
    app.UseKubernetesHealthChecks();  
    app.UseEndpoints(endpoints => endpoints.MapControllers());  
}
```

The ASP.NET pipeline



- When the request and the response pass through the middleware, the middleware can read them and even change them.
- The interface used between the pieces of middleware is the RequestDelegate defined by ASP.NET:

```
public delegate Task RequestDelegate(HttpContext context);
```

- We solve that with a small layer of indirection: we create a lambda that takes the next piece of middleware and returns a lambda that implements RequestDelegate.
- Let's look at that in code:

```
next =>
ctx =>
{
    System.Console.WriteLine("Got request");
    return next(ctx);
}
```

- Let's break down this code snippet. First, the type of `next` is `RequestDelegate` and is expected to be the next piece of middleware in the pipeline.
- Second, the type of `ctx` is `HttpContext`.
- That means that the inner lambda takes an `HttpContext` as argument and returns a `Task`:

```
ctx =>
{
    System.Console.WriteLine("Got request");
    return next(ctx);
}
```

```
RequestDelegate secondMiddleware = ...  
var fisrtMiddleware = next =>  
    ctx =>  
    {  
        System.Console.WriteLine("Got request");  
        return next(ctx);  
    }
```

```
RequestDelegate pipeline = firstMiddleware(secondMiddleware);
```

What belongs in middleware?

To decide what to implement in middleware and what to implement in MVC filters, you can follow a few simple rules of thumb:

- Code that addresses a cross-cutting concern and is meant to be used across many microservices belongs in middleware.
- Code that addresses a domain or business rule of a single microservice belongs in the application code behind an MVC controller.
- Code that handles HTTP requests and responses in a way that's specific to a particular endpoint belongs in an MVC controller.
- Code that addresses a concern that cuts across all endpoints in a microservice.

Writing middleware

We'll look at two ways to write middleware and hook them into the ASP.NET pipeline:

- *As a lambda*—You've seen this style earlier in this Module, and we will build on that in the next section.
- *As a class that has a method that implements the middleware signature*— You'll see this style right after we've taken another look at the lambda style.

Middleware as lambdas

- This is the lambda middleware you've seen a few times before:

```
next =>
    ctx =>
    {
        System.Console.WriteLine("Got request");
        return next(ctx);
    }
```

```
public class Startup
{
    public void Configure(IApplicationBuilder app)
    {
        app.Use(next => ctx =>
        {
            Console.WriteLine("Got request in lambda middleware");
            return next(ctx);
        })
        ...
    }
}
```

Middleware classes

This is the lambda middleware you've seen a few times before:

```
next =>
  ctx =>
  {
    System.Console.WriteLine("Got request");
    return next(ctx);
}
```

- In Startup.cs, we can add middleware to the pipeline by calling the Use method on the application builder:

```
public class Startup
{
    public void Configure(IApplicationBuilder app)
    {
        app.Use(next => ctx =>
        {
            Console.WriteLine("Got request in lambda middleware");
            return next(ctx);
        })
        ...
    }
}
```

Middleware classes

- To use a class to implement middleware, you can create a class that has a method whose signature is a RequestDelegate.

```
public class ConsoleMiddleware
{
    private readonly RequestDelegate next;

    public ConsoleMiddleware(RequestDelegate next)
    {
        this.next = next;
    }

    public Task Invoke(HttpContext ctx)
    {
        Console.WriteLine("Got request in class middleware");
        return this.next(ctx);
    }
}
```

- To add this kind of middleware to the ASP.NET pipeline, you instantiate it and then use the `Invoke` method as a delegate:

```
app.Use(next => new ConsoleMiddleware(next).Invoke);
```

- Or, we can use the more convenient extension method `UseMiddleware`, which essentially does the same thing by using reflection to find the `Invoke` method:

```
app.UseMiddleware<ConsoleMiddleware>()
```

Testing middleware and pipelines

- There's one small issue to overcome: in order to call middleware, you first need to provide it with a value for next.
- You get around that by passing in a RequestDelegate that does nothing:

```
RequestDelegate noOp = _ => Task.CompletedTask;
```

An example piece of middleware

```
namespace Middleware
{
    public class RedirectingMiddleware
    {
        private readonly RequestDelegate next;

        public RedirectingMiddleware(RequestDelegate next)
        {
            this.next = next;
        }

        public Task Invoke(HttpContext ctx)
        {
            if (ctx.Request.Path.Value.TrimEnd('/') == "/oldpath")
            {
                ctx.Response.Redirect("/newpath", permanent: true);
                return Task.CompletedTask;
            }

            return this.next(ctx);
        }
    }
}
```

Test that invokes middleware directly

```
namespace MiddlewareTests
{
    using System.Threading.Tasks;
    using Microsoft.AspNetCore.Http;
    using Middleware;
    using Xunit;

    public class RedirectingMiddleware_should
    {
        private readonly RedirectingMiddleware sut;

        public RedirectingMiddleware_should()
        {
            this.sut = new RedirectingMiddleware(_ => Task.CompletedTask);
        }

        [Fact]
        public async Task redirect_oldpath_to_newpath()
        {
            var ctx = new DefaultHttpContext
            {
                Request = {Path = "/oldpath"}
            };

            await this.sut.Invoke(ctx);

            Assert.Equal(StatusCodes.Status301MovedPermanently,
                ctx.Response.StatusCode);
            Assert.Equal("/newpath", ctx.Response.Headers["Location"]);
        }
    }
}
```

- You can use the same pattern to create more tests—for other paths than `oldpath`. See the following example:

```
[Fact]
public async Task not_redirect_other_paths()
{
    var ctx = new DefaultHttpContext
    {
        Request = {Path = "/otherpath"}
    };

    await this.sut.Invoke(ctx);

    Assert.NotEqual(
        StatusCodes.Status301MovedPermanently,
        ctx.Response.StatusCode);
    Assert.DoesNotContain("Location", ctx.Response.Headers);
}
```

Summary

- NuGet is a good format for distributing a microservice platform.
- You use the dotnet pack command to create a NuGet package.
- To meet cross-cutting requirements for monitoring, logging, and security, there are a number of things that all microservices in a system need to do.
- You can build NuGet packages from libraries.
- You can use your own custom NuGet packages in your microservices.
- You can leverage ASP.NET middleware for cross-cutting concerns.
- You can easily write tests for ASP.NET middleware.

12. Creating applications over microservices

Introduction

This Module covers

- Building an end user application on top of a microservice system
- Understanding the composite application, API Gateway, and backend for frontend design patterns
- Using server-side and client-side rendering in web applications

End user applications for microservice systems: One or many applications?



General-purpose applications

Users include the following:

- Salespeople who call and solicit potential customers.
- Actuaries who set policy prices and evaluate business risks based on estimated future claims and income.
- Appraisers who evaluate goods that customers want to insure and about which customers make claims.
- Claims adjusters who investigate and settle customer claims.
- IT staff who oversee users and permissions.

Specialized applications

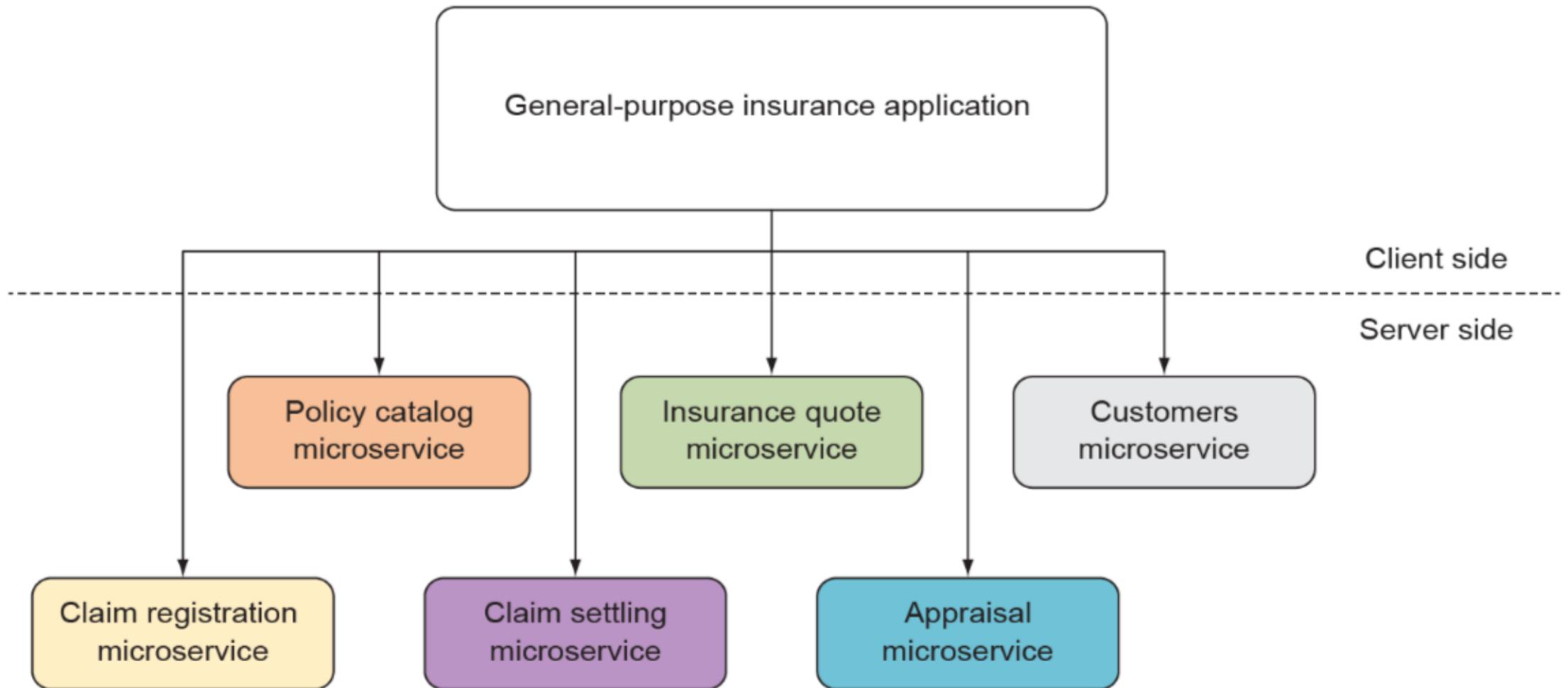
Continuing with the example of a line-of-business application for an insurance company, you may have separate specialized applications for the following:

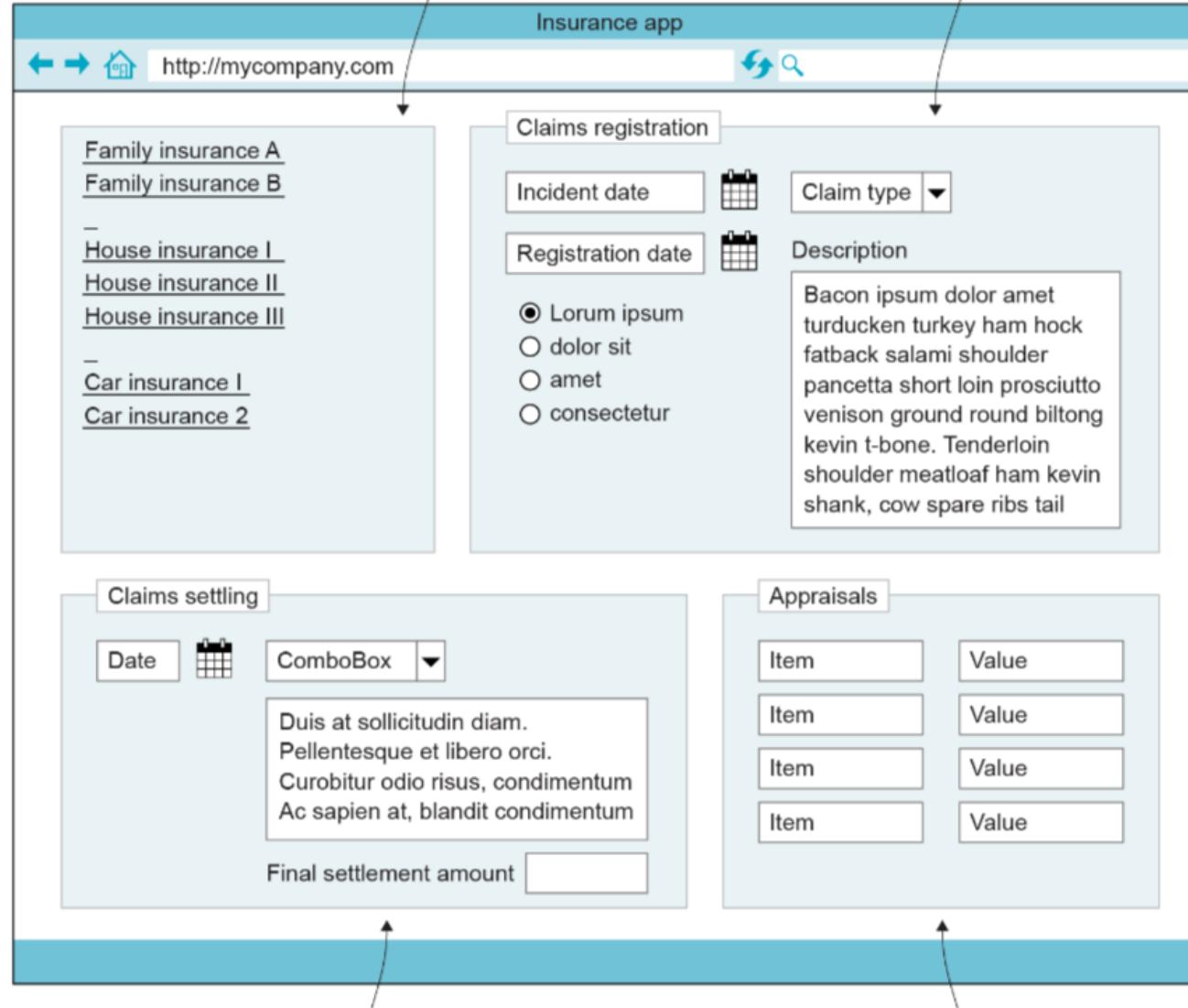
- Browsing the catalog of policies offered by the company
- Creating offers for potential customers
- Creating an insurance policy for a customer
- Creating reports about currently active insurance policies
- Creating forecasts about the cost of future claims
- Registering and investigating claims
- Appraising insured goods
- Settling claims



Patterns for building applications over microservices

Composite applications: Integrating at the frontend





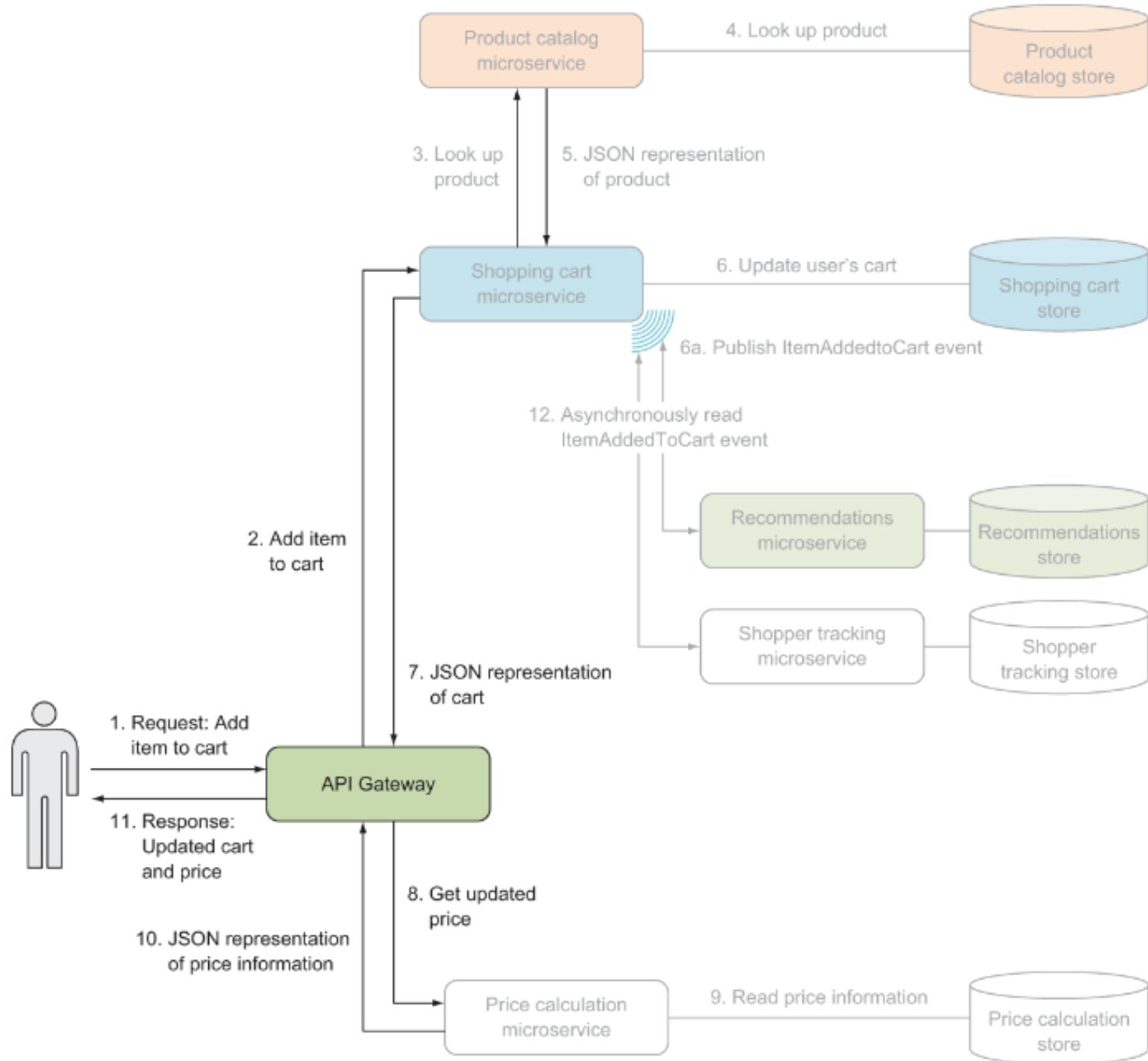
The list of insurance policies comes from the policy catalog microservice.

The claims registration form comes from the claims registration microservice.

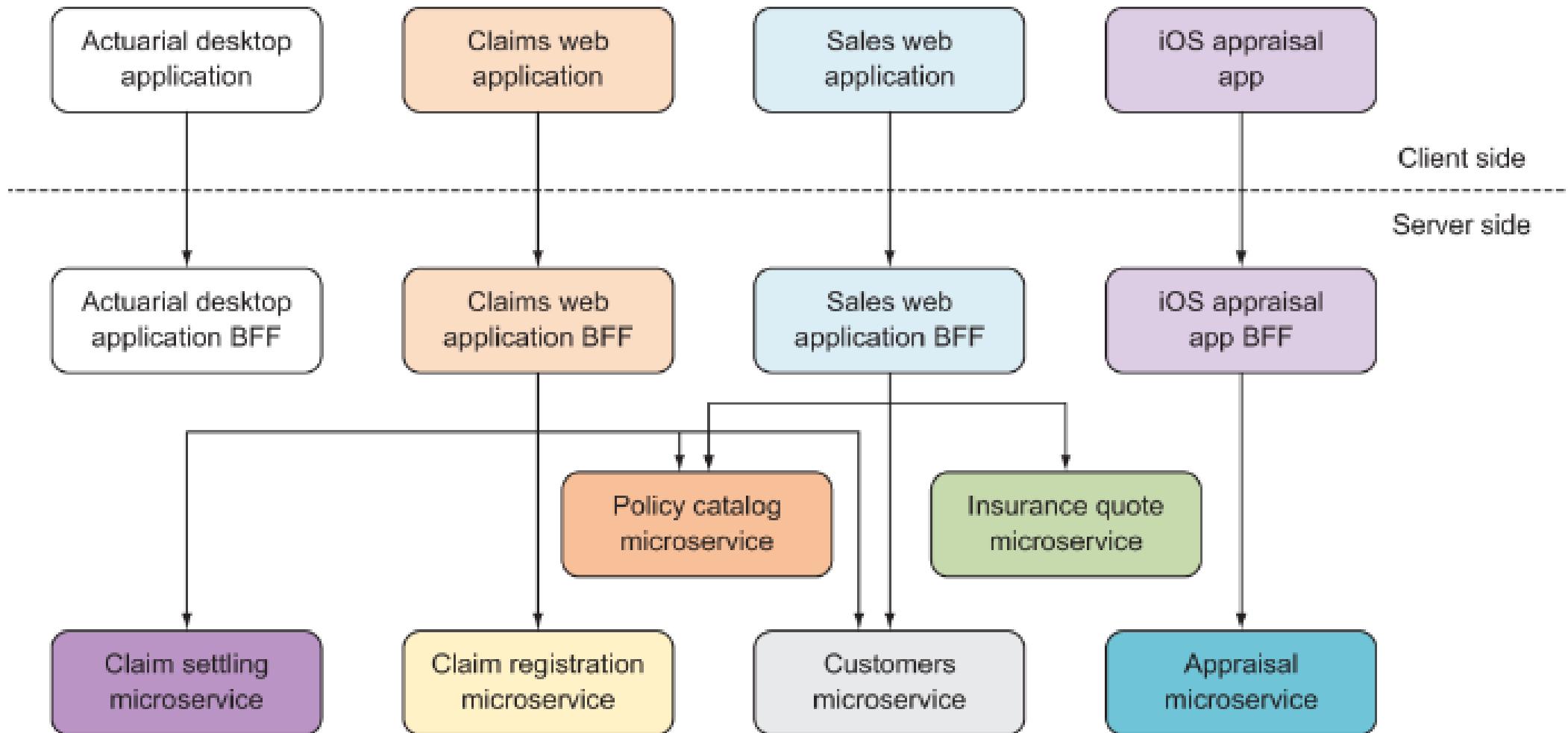
The claim settling form comes from the claim settling microservice.

The appraisals form comes from the appraisal microservice.

API Gateway



Backend for frontend (BFF) pattern

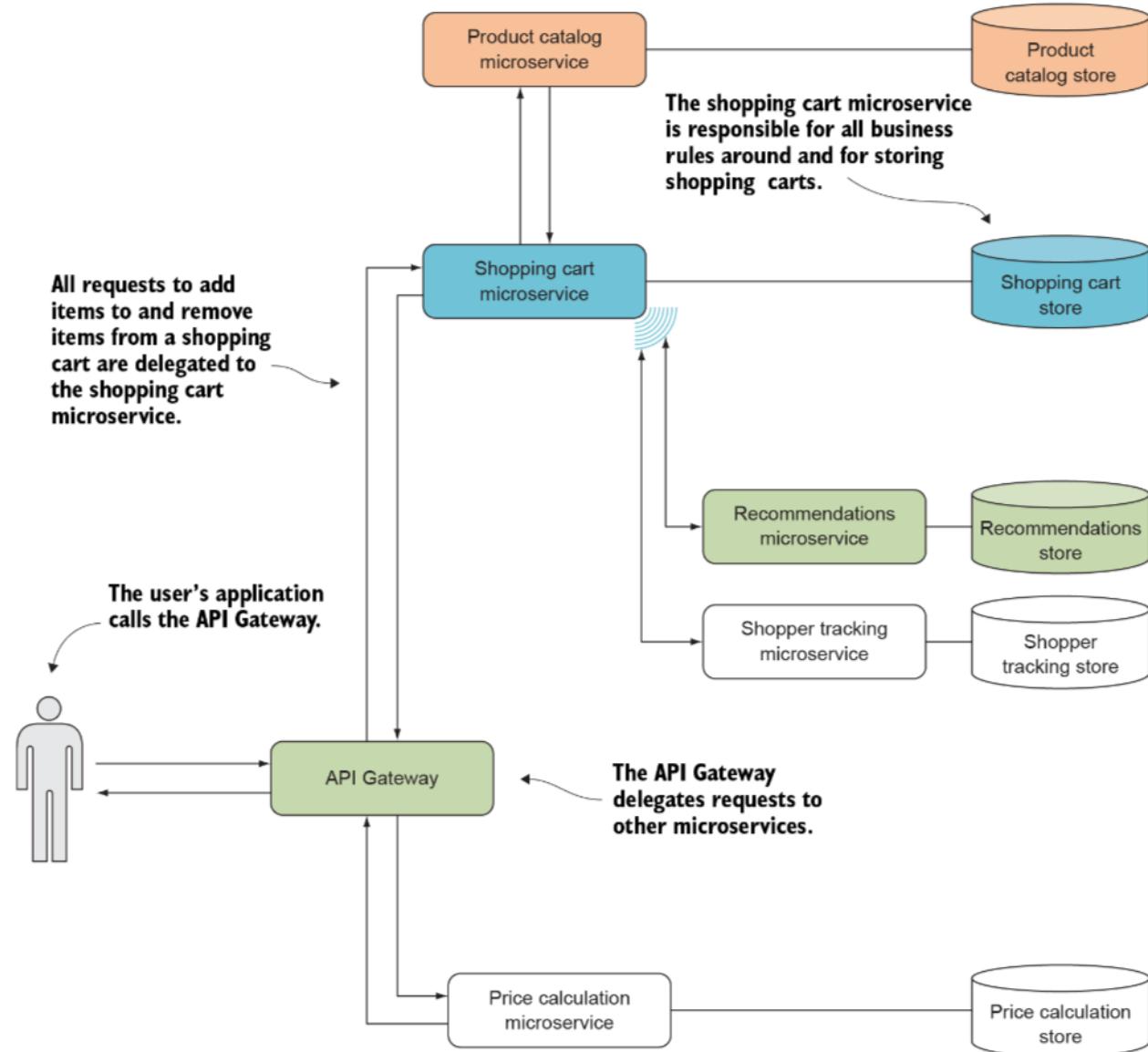


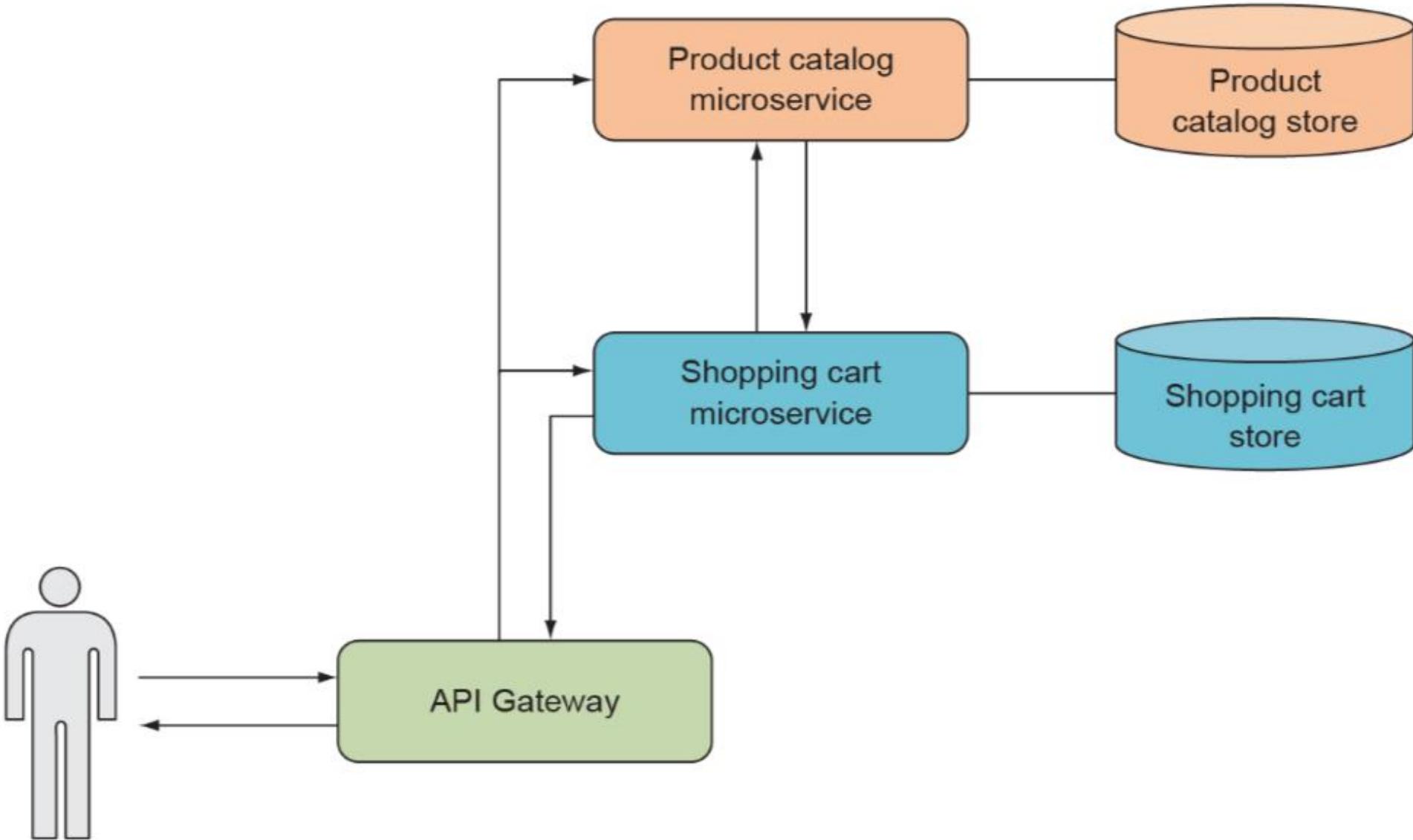
When to use each pattern

When you're about to build an application, you must make a choice. I base that choice on the following questions:

1. *How much intelligence do you want to put into the application?*
2. *Is there more than one application? If so, how different are the applications?*
3. *How big is the system?*

Example: A shopping cart and a product list





MicroCommerce.NET

https://localhost:5101/productlist?userId=42

MicroCommerce - Product List

Product 1
lorum ipsum [BUY!](#)

Product 2
lorum ipsum [BUY!](#)

Product 3
lorum ipsum [BUY!](#)

Product 4
lorum ipsum [BUY!](#)

Shopping Cart

Product 1 [X](#)

Product 4 [X](#)

Creating an API Gateway

```
<Project Sdk="Microsoft.NET.Sdk.Web">

<PropertyGroup>
  <TargetFramework>net5.0</TargetFramework>
</PropertyGroup>

<ItemGroup>
  <PackageReference Include="MicroserviceNET.Logging" Version="1.0.0" />
  <PackageReference Include="MicroserviceNET.Monitoring" Version="1.0.0" />
</ItemGroup>

</Project>
```

Using the logging from the microservice platform

```
using ApiGateway;
using MicroserviceNET.Logging;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Hosting;

CreateHostBuilder(args).Build().Run();

static IHostBuilder CreateHostBuilder(string[] args) =>
    Host.CreateDefaultBuilder(args)
        .UseLogging()
        .ConfigureWebHostDefaults(webBuilder =>
    {
        webBuilder.UseStartup<Startup>();
    });
}
```

Initializing Nancy and the microservice platform

```
namespace ApiGateway
{
    using MicroserviceNET.Monitoring;
    using Microsoft.AspNetCore.Builder;
    using Microsoft.AspNetCore.Hosting;
    using Microsoft.Extensions.DependencyInjection;

    public class Startup
    {
        public void ConfigureServices(IServiceCollection services)
        {
            services.AddBasicHealthChecks();
            services.AddControllers();
        }

        public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
        {
            app.UseRouting();
            app.UseKubernetesHealthChecks();
            app.UseEndpoints(endpoints => endpoints.MapControllers());
        }
    }
}
```

Creating the product list GUI

```
namespace ApiGateway.ProductList
{
    using Microsoft.AspNetCore.Mvc;
    using System.Threading.Tasks;

    public class ProductListController : Controller
    {
        [HttpGet("/productlist")]
        public async Task<IActionResult> ProductList([FromQuery] int userId)
            => StatusCode(501);
    }
}
```

Adding the types necessary to use Razor views

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddBasicHealthChecks();
    services.AddControllersWithViews();
}
```

Hardcoded endpoint implementation in the API Gateway

```
namespace ApiGateway.ProductList
{
    using Microsoft.AspNetCore.Mvc;
    using System.Threading.Tasks;

    public class ProductListController : Controller
    {
        [HttpGet("/productlist")]
        public async Task<IActionResult> ProductList([FromQuery] int userId)
        {
            var products = new[]
            {
                new Product(1, "T-shirt", "Really nice t-shirt"),
                new Product(2, "Hoodie", "The coolest hoodie ever"),
                new Product(3, "Jeans", "Perfect jeans"),
            };
            return View(new ProductListViewModel(products));
        }
    }

    public record Product(int ProductId,
        string ProductName, string Description);
    public record ProductListViewModel(Product[] Products);
}
```

Simple product list view

```
@model ApiGateway.ProductList.ProductListViewModel

<!DOCTYPE html>
<html>

    <head>
        <link rel="stylesheet" href="https://cdn.jsdelivr.net/npm/
            bootstrap@4.6.0/dist/css/bootstrap.min.css" integrity="
            sha384-B0vP5xMAtw1+K9KRQjQERJvTumQW0nPEzvF6L/
            Z6nronJ3oUOFUFpCjEUQouq2+l" crossorigin="anonymous">
        <title>MicroCommerce.NET</title>
    </head>
    <body>
        <div class="container">
            <div class="page-header">
                <h1>MicroCommerce <small>- Product List</small></h1>
            </div>
            <div class="row">
                <div class="col-md-8">
                    @foreach (var product in Model.Products)
                    {
                        <div class="row" style="border-bottom-style: solid">
                            <div class="col-md-8">
                                <h4>@product.ProductName</h4>
                                <p>@product.Description</p>
                            </div>
                            <div class="col-md-4">
                                <p></p>
                                <button class="btn btn-primary" type="button">
                                    BUY!
                                </button>
                            </div>
                        </div>
                    }
                </div>
            </div>
        </div>
    </body>
</html>
```

Configuring ASP.NET to find views in the same folders as controller

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddBasicHealthChecks();
    services.AddControllersWithViews();
    services.Configure<RazorViewEngineOptions>(x =>
        x.ViewLocationFormats.Add("{1}/{0}.cshtml"));
}
```

Configuring HttpClient for calling the product catalog

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddBasicHealthChecks();
    services.AddControllersWithViews();
    services.Configure<RazorViewEngineOptions>(x =>
        x.ViewLocationFormats.Add("{1}/{0}.cshtml"));
    services.AddHttpClient(
        "ProductCatalogClient",
        client => client.BaseAddress = new Uri("https://localhost:5001"))
        .AddTransientHttpErrorPolicy(p =>
            p.WaitAndRetryAsync(
                3,
                attempt => TimeSpan.FromMilliseconds(100 * Math.Pow(2, attempt))));
```

Finished endpoint implementation in the API Gateway

```
namespace ApiGateway.ProductList
{
    using System.Net.Http;
    using System.Net.Http.Json;
    using System.Text.Json;
    using System.Threading.Tasks;
    using Microsoft.AspNetCore.Mvc;

    public class ProductListController : Controller
    {
        private readonly HttpClient productCatalogClient;

        public ProductListController(IHttpClientFactory httpClientFactory)
        {
            this.productCatalogClient =
                httpClientFactory.CreateClient("ProductCatalogClient");
        }

        [HttpGet("/productlist")]
        public async Task<IActionResult> ProductList([FromQuery] int userId)
        {
            var products = await GetProductsFromCatalog();
            return View(new ProductListViewModel(products));
        }

        private async Task<Product[]> GetProductsFromCatalog()
        {
            var response = await
                this.productCatalogClient.GetAsync(
                    "/products?productIds=1,2,3,4");
            response.EnsureSuccessStatusCode();
            var content = await response.Content.ReadAsStreamAsync();
            var products =
                await JsonSerializer.DeserializeAsync<Product[]>(
                    content,
                    new JsonSerializerOptions {PropertyNameCaseInsensitive = true});
            return products;
        }
    ...
}
```



← → ↻ ⌄



https://localhost:5101/productlist?userId=42

MicroCommerce - Product List

Product 1

lorum ipsum

[BUY!](#)

Product 2

lorum ipsum

[BUY!](#)

Product 3

lorum ipsum

[BUY!](#)

Product 4

lorum ipsum

[BUY!](#)

Creating the shopping cart GUI

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddBasicHealthChecks();
    services.AddControllersWithViews();
    services.Configure<RazorViewEngineOptions>(x =>
        x.ViewLocationFormats.Add("{1}/{0}.cshtml"));
    services.AddHttpClient(
        "ProductCatalogClient",
        client => client.BaseAddress = new Uri("https://localhost:5001"))
        .AddTransientHttpErrorPolicy(p =>
            p.WaitAndRetryAsync(
                3,
                attempt => TimeSpan.FromMilliseconds(100 * Math.Pow(2, attempt))));
    services.AddHttpClient(
        "ShoppingCartClient",
        client => client.BaseAddress = new Uri("https://localhost:5201")));
}
```

```
1 namespace ApiGateway.ProductList
2 {
3     using System.Net.Http;
4     using System.Net.Http.Json;
5     using System.Text.Json;
6     using System.Threading.Tasks;
7     using Microsoft.AspNetCore.Mvc;
8
9     public class ProductListController : Controller
10    {
11        private readonly HttpClient productCatalogClient;
12        private readonly HttpClient shoppingCartClient;
13
14        public ProductListController(IHttpClientFactory httpClientFactory)
15        {
16            this.productCatalogClient =
17                httpClientFactory.CreateClient("ProductCatalogClient");
18            this.shoppingCartClient =
19                httpClientFactory.CreateClient("ShoppingCartClient");
20        }
21
22        [HttpGet("/productlist")]
23        public async Task<IActionResult> ProductList([FromQuery] int userId)
24        {
25            var products = await GetProductsFromCatalog();
26            var cartProducts = await GetProductsFromCart(userId);
27            return View(new ProductListViewModel(
28                products,
29                cartProducts
30            ));
31        }
32
33        private async Task<Product[]> GetProductsFromCart(int userId)
34        {
35            var response = await
```

```
36     this.shoppingCartClient.GetAsync($"/shoppingcart/{userId}");  
37     response.EnsureSuccessStatusCode();  
38     var content = await response.Content.ReadAsStreamAsync();  
39     var cart =  
40         await JsonSerializer.DeserializeAsync<ShoppingCart>(  
41             content,  
42             new JsonSerializerOptions {PropertyNameCaseInsensitive = true});  
43     return cart.Items;  
44 }  
45  
46 private async Task<Product[]> GetProductsFromCatalog()  
47 {  
48     var response = await  
49         this.productCatalogClient.GetAsync("/products?productIds=1,2,3,4");  
50     response.EnsureSuccessStatusCode();  
51     var content = await response.Content.ReadAsStreamAsync();  
52     var products =  
53         await JsonSerializer.DeserializeAsync<Product[]>(  
54             content,  
55             new JsonSerializerOptions {PropertyNameCaseInsensitive = true});  
56     return products;  
57 }  
58 }  
59  
60 public record Product(int ProductId, string ProductName,  
61     string Description);  
62 public record ShoppingCart(int UserId, Product[] Items);  
63 public record ProductListViewModel(  
64     Product[] Products,  
65     Product[] CartProducts);  
66  
67 }
```

```
@model ApiGateway.ProductList.ProductListViewModel

<!DOCTYPE html>
<html>

    <head>
        <link rel="stylesheet" href="https://cdn.jsdelivr.net/npm/
            bootstrap@4.6.0/dist/css/bootstrap.min.css" integrity="
            sha384-B0VP5xMAtw1+K9KRQjQERJVTumQW0nPEzVF6L/
            Z6nronJ3oUOFUFpCjEUQouq2+l" crossorigin="anonymous">
        <title>MicroCommerce.NET</title>
    </head>
    <body>
        <div class="container">
            <div class="page-header">
                <h1>MicroCommerce <small>- Product List</small></h1>
            </div>
            <div class="row">
                <div class="col-md-8">
                    @foreach (var product in Model.Products)
                    {
                        <div class="row" style="border-bottom-style: solid">
                            <div class="col-md-8">
                                <h4>@product.ProductName</h4>
                                <p>@product.Description</p>
                            </div>
                            <div class="col-md-4">
                                <p></p>
                                <button class="btn btn-primary" type="button">
                                    BUY!
                                </button>
                            </div>
                        </div>
                    }
                </div>
                <div class="col-md-4">
                    <div class="card bg-warning">
                        <div class="card-body">
                            <div class="card-title">Shopping Cart</div>
                            @foreach(var product in Model.CartProducts)
                            {
                                <div class="card-text">
                                    @product.ProductName
                                    <button class="btn btn-link">X</button>
                                </div>
                            }
                        </div>
                    </div>
                </div>
            </div>
        </div>
    </body>
</html>
```

MicroCommerce.NET

https://localhost:5101/productlist?userId=42

MicroCommerce - Product List

Product 1
lorum ipsum [BUY!](#)

Product 2
lorum ipsum [BUY!](#)

Product 3
lorum ipsum [BUY!](#)

Product 4
lorum ipsum [BUY!](#)

Shopping Cart

Product 1 X

Product 4 X

Endpoint to add a product to the shopping cart

```
[HttpPost("/shoppingcart/{userId}")]  
public async Task<OkResult> AddToCart(  
    int userId,  
    [FromBody] int productId)  
{  
    var response =  
        await this.shoppingCartClient.PostAsJsonAsync(  
            $""/shoppingcart/{userId}/items",  
            new[] {productId});  
    response.EnsureSuccessStatusCode();  
    return Ok();  
}
```

Calling the endpoint to add a product to the shopping cart

```
async function buy(productId){  
  const params = new URLSearchParams(location.search);  
  const rawResponse =  
    await fetch('/shoppingcart/' + params.get('userId'), {  
      method: 'POST',  
      headers: {  
        'Accept': 'application/json',  
        'Content-Type': 'application/json'  
      },  
      body: JSON.stringify(productId)  
    });  
  window.location.reload(true);  
}
```

```
<html>
```

```
...
```

```
    <button class="btn btn-primary" type="button"  
          onclick="buy(@Current.ProductId);">  
        BUY!  
    </button>
```

```
...
```

```
</html>
```

Letting users remove products from the shopping cart

```
[HttpDelete("/shoppingcart/{userId}")]  
public async Task<OkResult> RemoveFromCart(  
    int userId,  
    [FromBody] int productId)  
{  
    var request = new HttpRequestMessage(  
        HttpMethod.Delete,  
        $""/shoppingcart/{userId}/items");  
    request.Content =  
        new StringContent(JsonSerializer.Serialize(new[] {productId}));  
    var response = await this.shoppingCartClient.SendAsync(request);  
    response.EnsureSuccessStatusCode();  
    return Ok();  
}
```

Calling the endpoint to remove a product from the shopping cart

```
async function removeFromCart(productId){  
  const params = new URLSearchParams(location.search);  
  const rawResponse =  
    await fetch('/shoppingcart/' + params.get('userId'), {  
      method: 'DELETE',  
      headers: {  
        'Accept': 'application/json',  
        'Content-Type': 'application/json'  
      },  
      body: JSON.stringify(productId)  
    });  
  window.location.reload(true);  
}
```

- Next, use this JavaScript function from the X button in the shopping cart part of the view:

```
<html>
```

```
...
```

```
    <button class="btn btn-link" onclick="removeFromBasket">X</button>
```

```
...
```

```
</html>
```

Summary

- BFFs are less prone to growing bigger than API Gateways are.
- BFFs are tailored to the single application using them and should therefore make that application as simple to implement as possible.
- When you build web applications over microservices, you're free to use server-side rendering, client-side rendering, or a mix.
- All three patterns—composite application, API Gateway, and BFF—support server-side rendering, client-side rendering, and mixes of the two.
- The implementation of an API Gateway is very thin: all the endpoints you added to the example API Gateway delegate to other microservices.