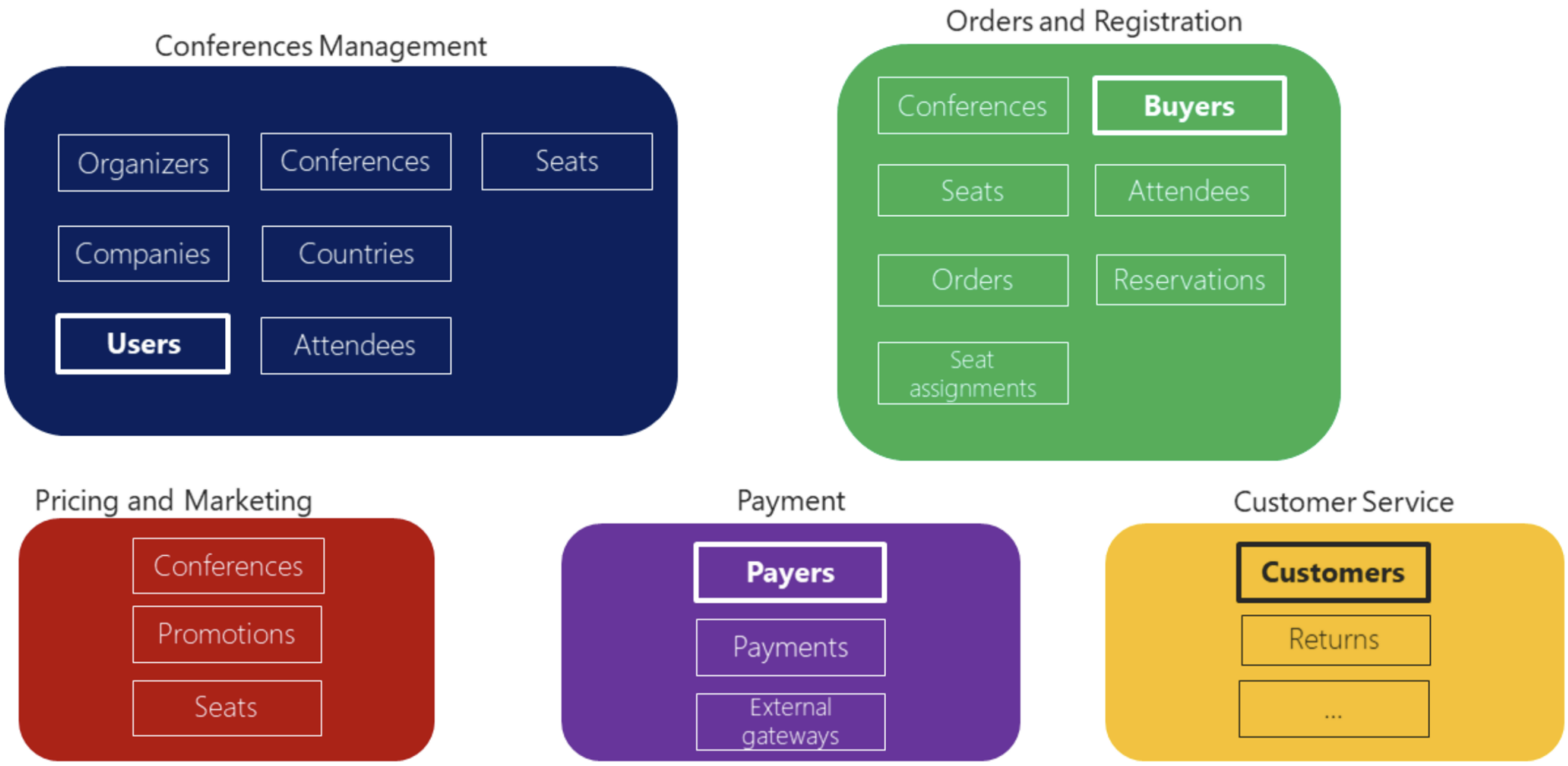
Microservices – The last mile

Bart Wullems

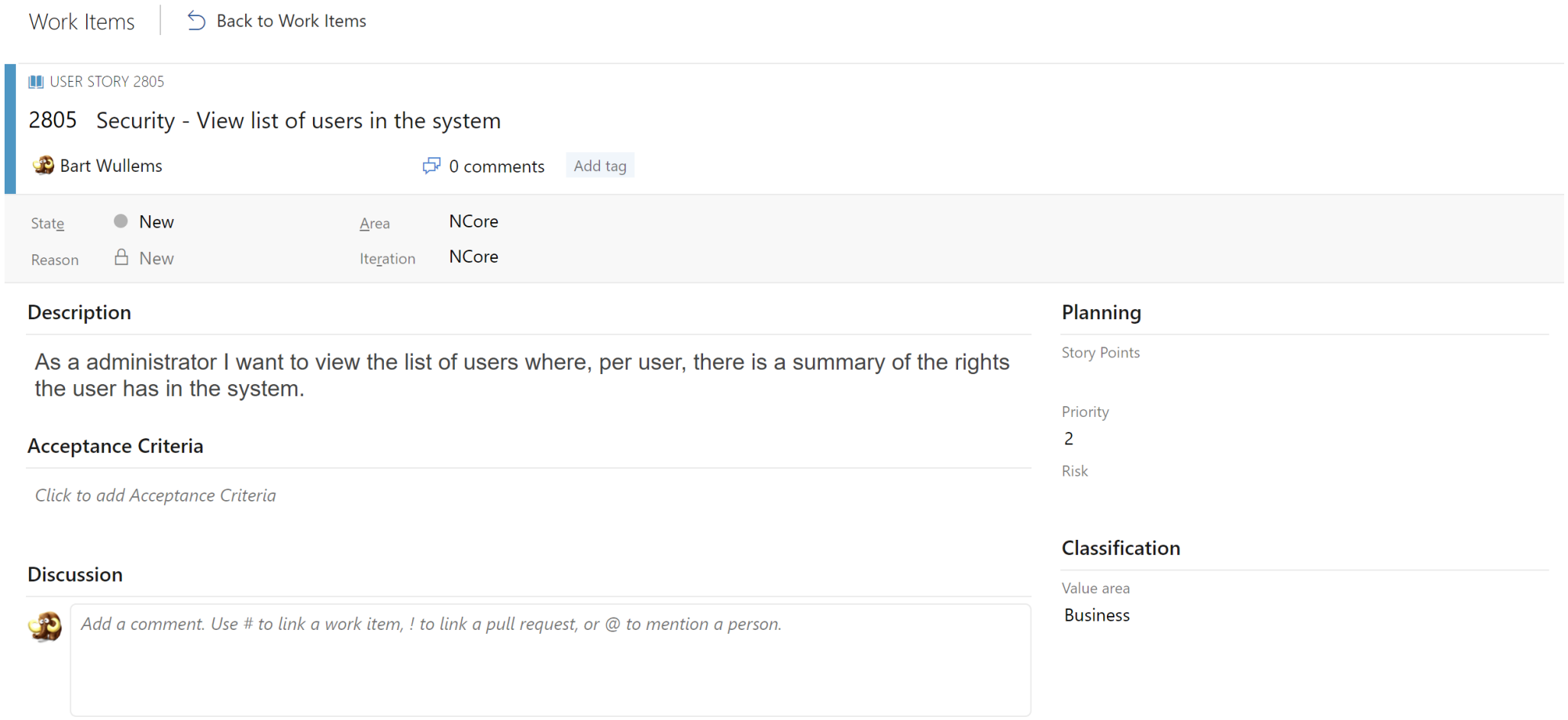
https://bartwullems.blogspot.com

bart.wullems@ordina.be

Introduction



Introduction



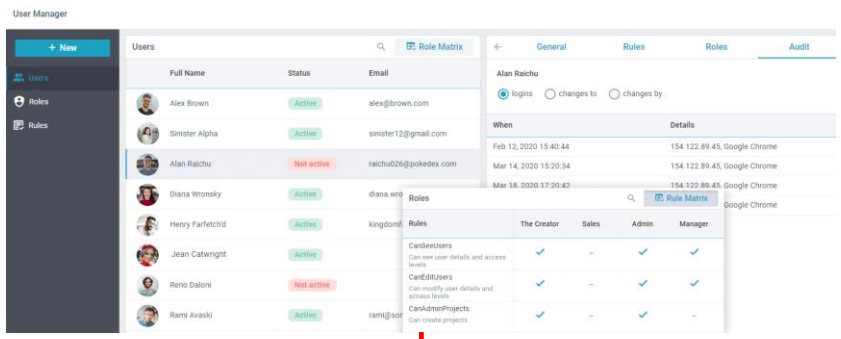
Introduction

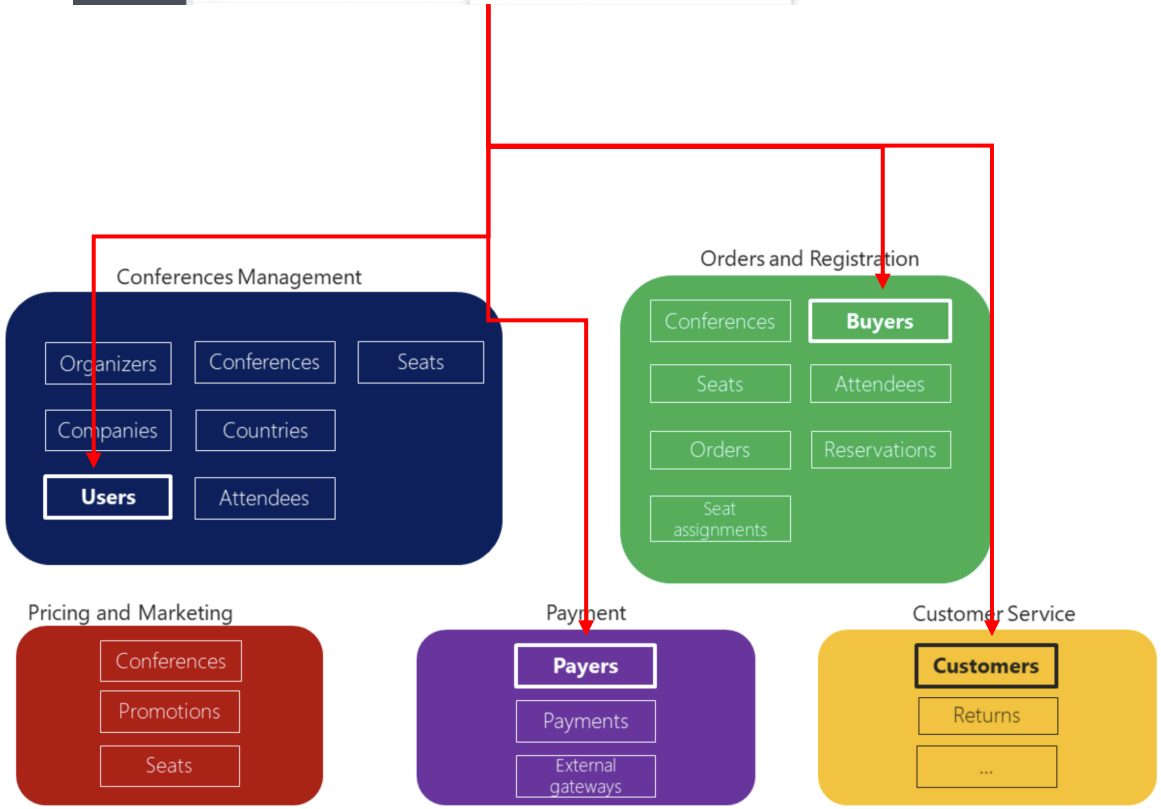
Frontend

Back in Monolith land

Backend

Our beautiful microservices





ORDINA Ahead of change

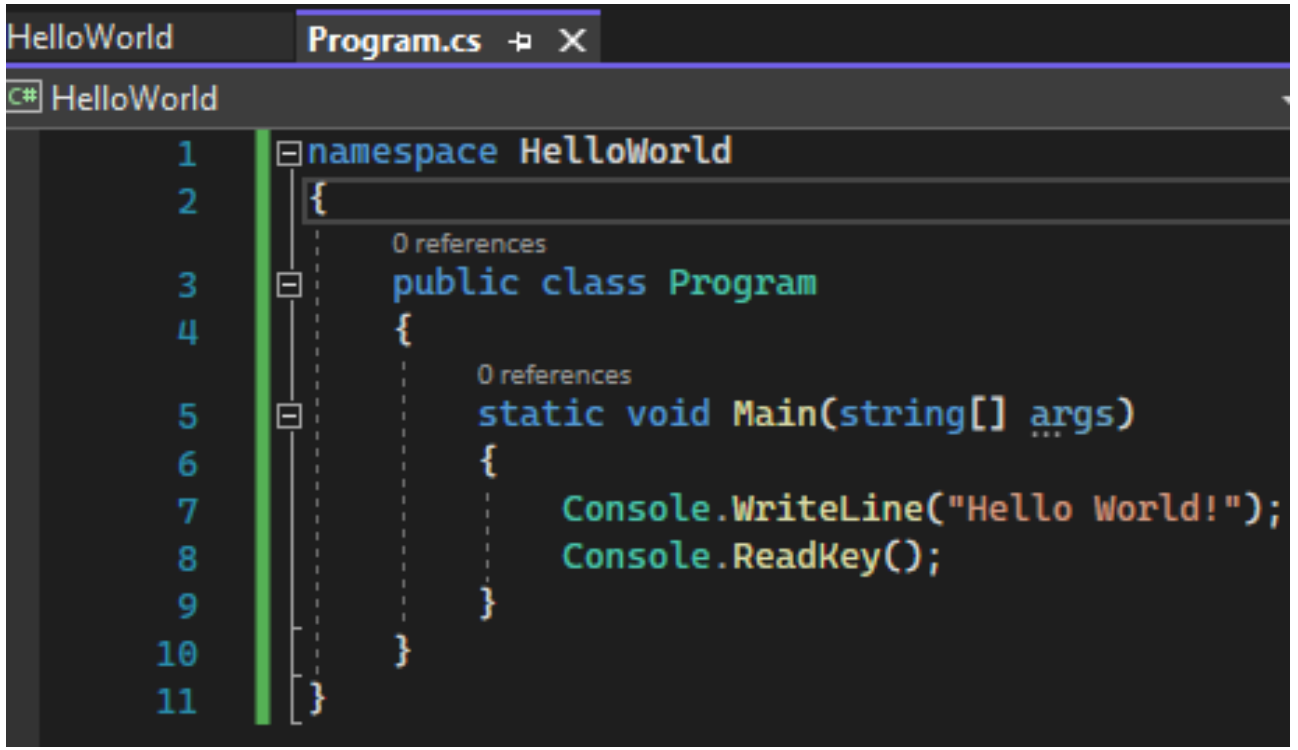
Microservices – The last mile

Bart Wullems

https://bartwullems.blogspot.com

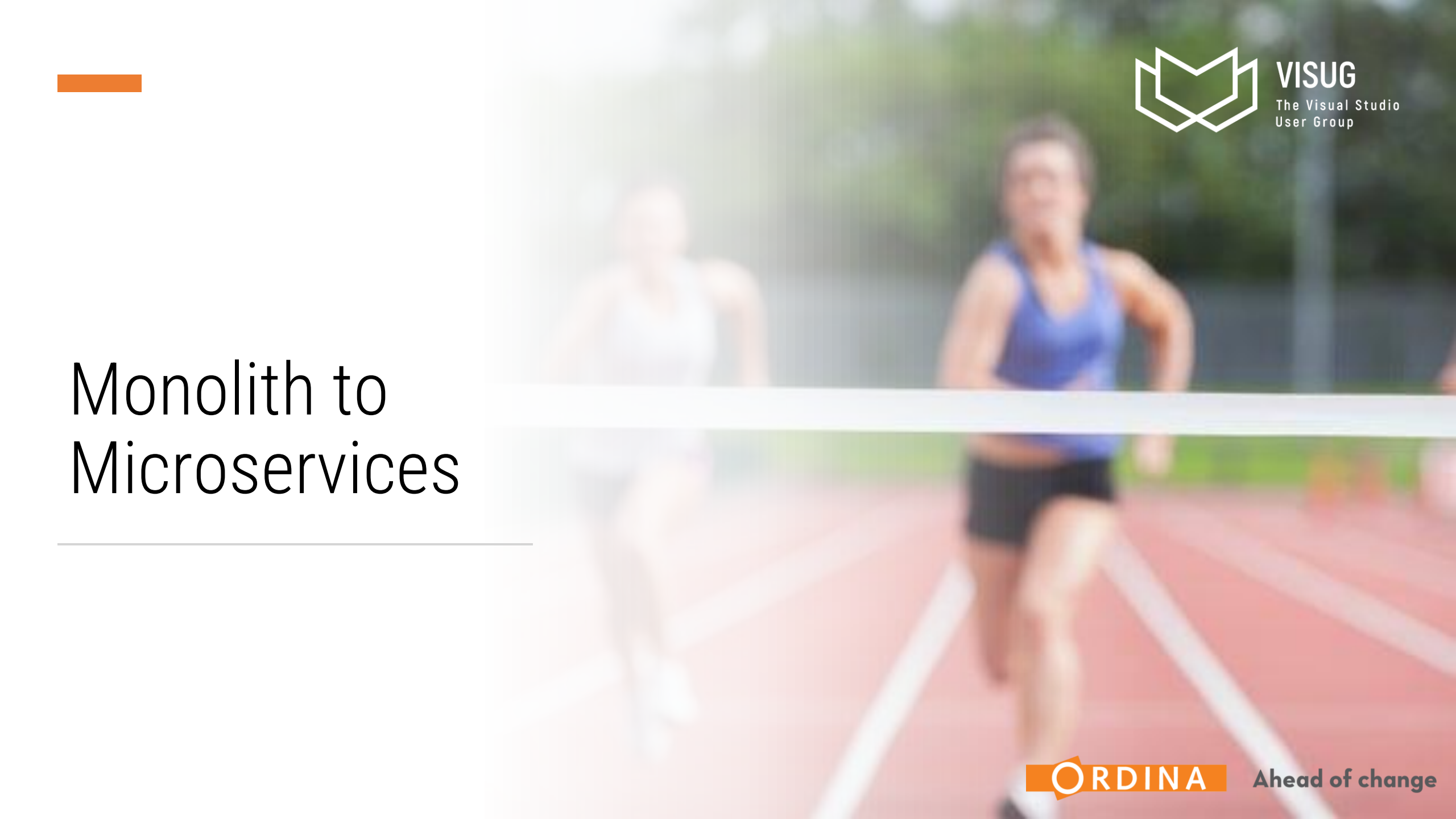
bart.wullems@ordina.be

WARNING



Agenda

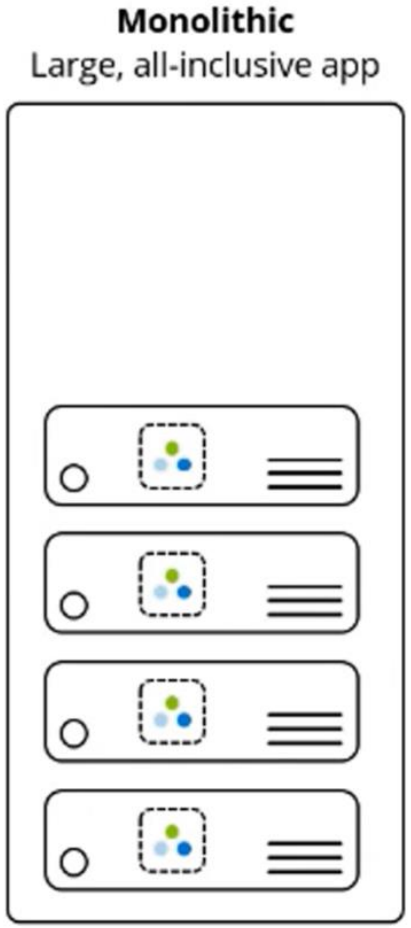
- From monolith to microservice
- Frontend monolith
- Frontend integration
- Conclusions(sort of)



Monolith to Microservices

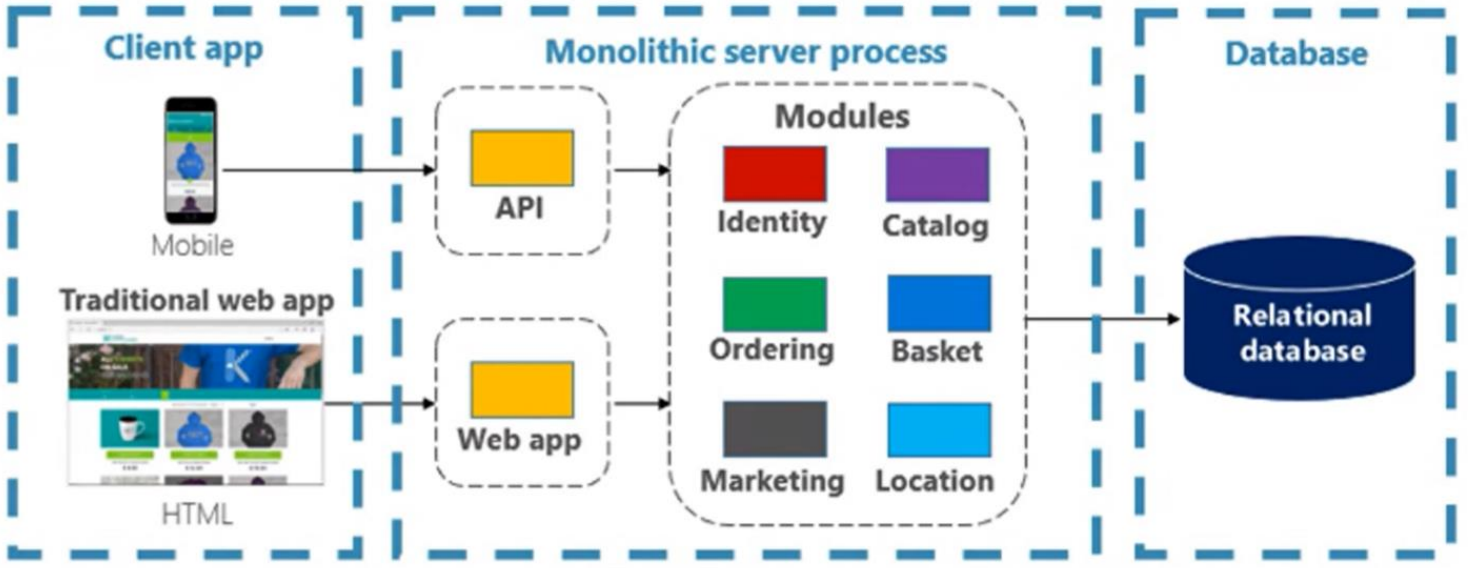
From Monolith to Microservices

- Independent deployments
- Improved scale and resource utilization per service
- Smaller, focused teams



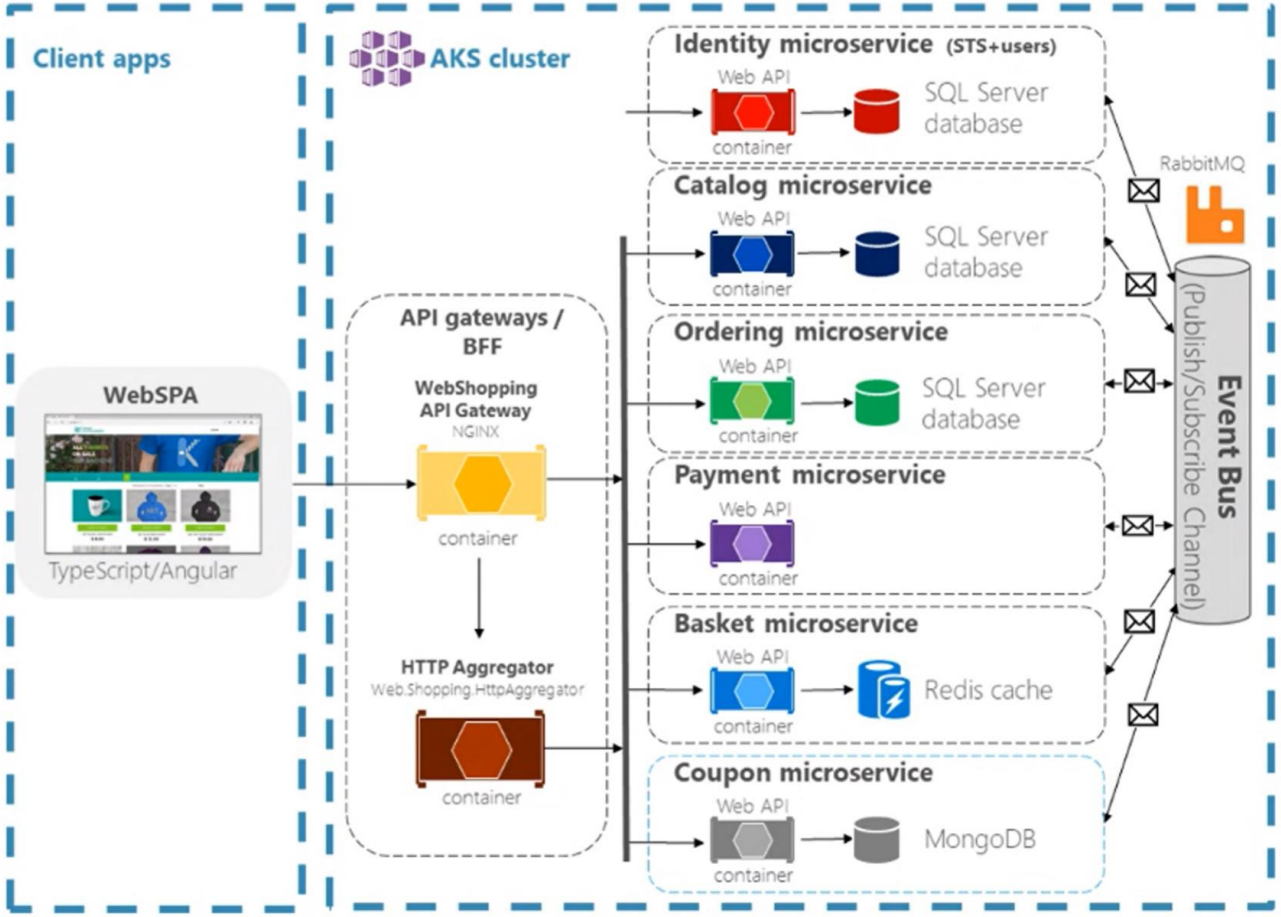
Is Monolith bad?

- Not all is bad. Monoliths offer some distinct advantages.
- Many successful apps that exist today were created as monoliths.
- You don't need to migrate all your apps to microservices.



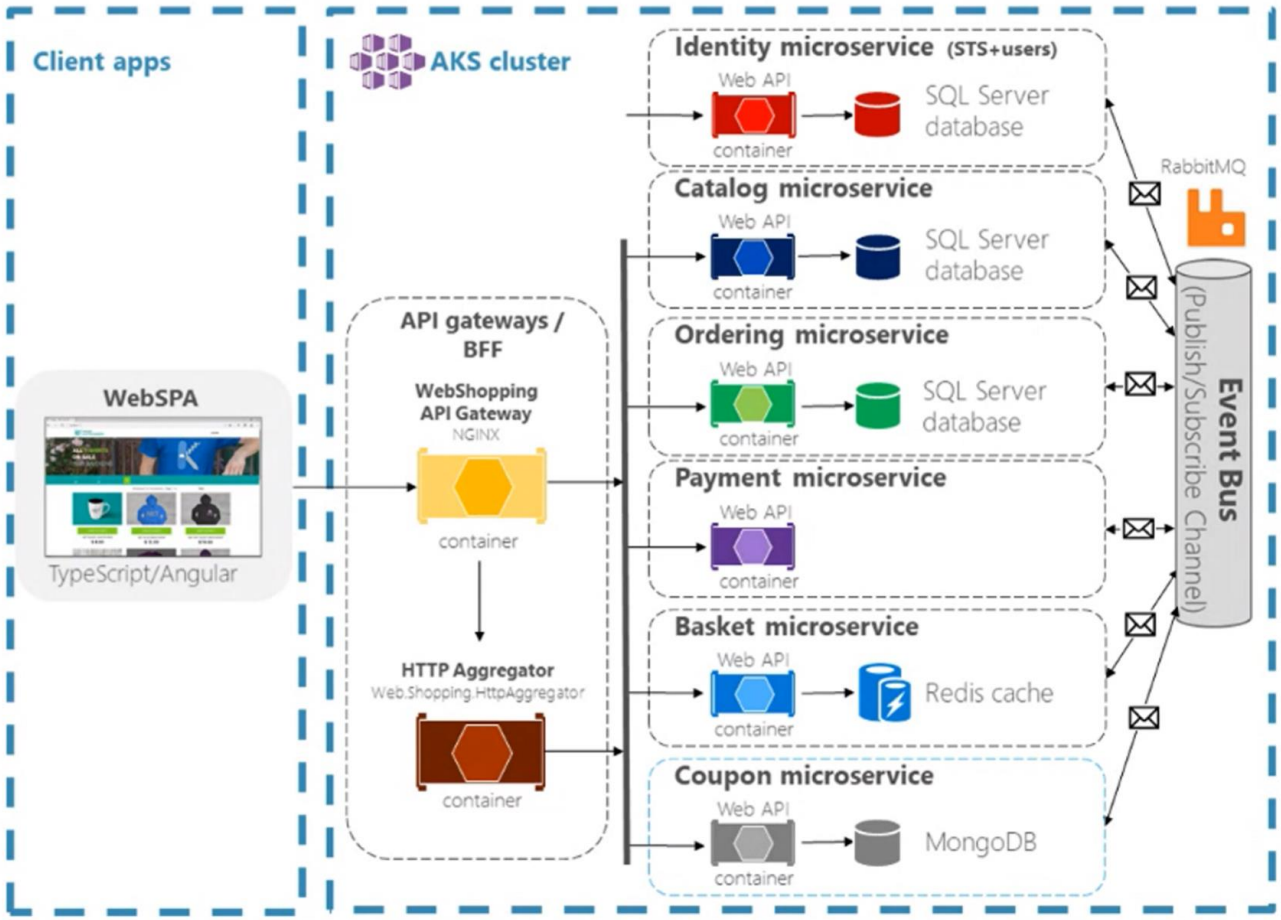
Microservices Benefits

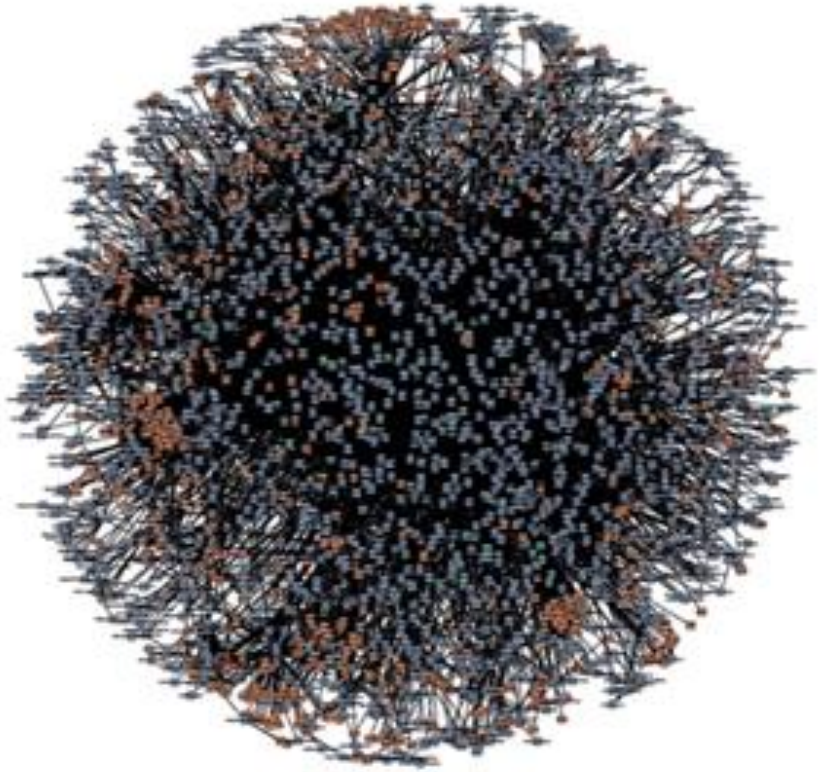
- Relatively small—easy to manage and evolve.
- Designed, developed, and deployed independently of other microservices
- Possible to scale out individual areas of the application.
- Divide the development work between multiple teams.
- You can use the latest technologies.



Microservices Challenges

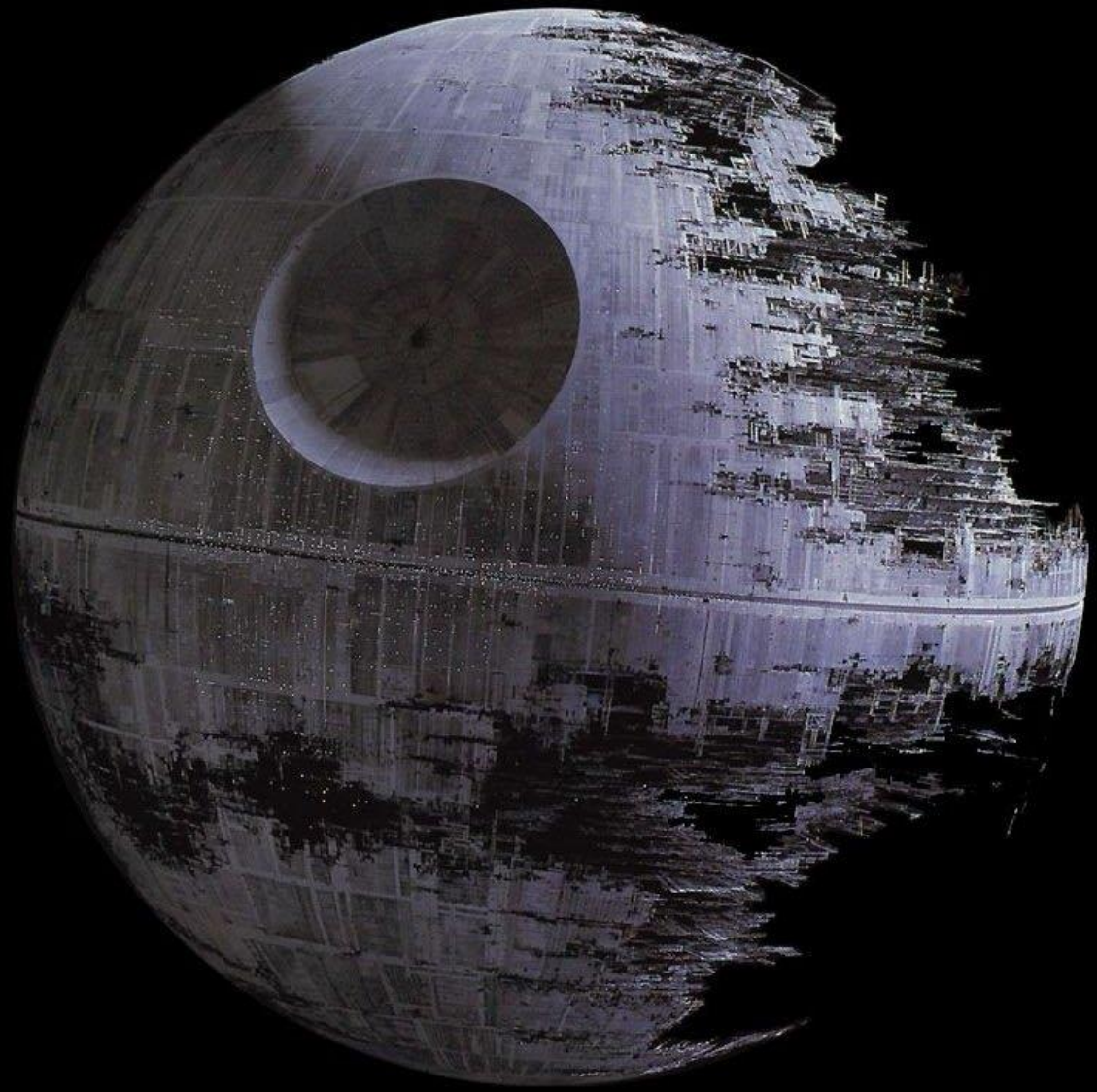
- Operational complexity
- Data consistency
- Partitioning
- Debugging issues
- Tracing & Monitoring
- And more!





amazon.com

NETFLIX



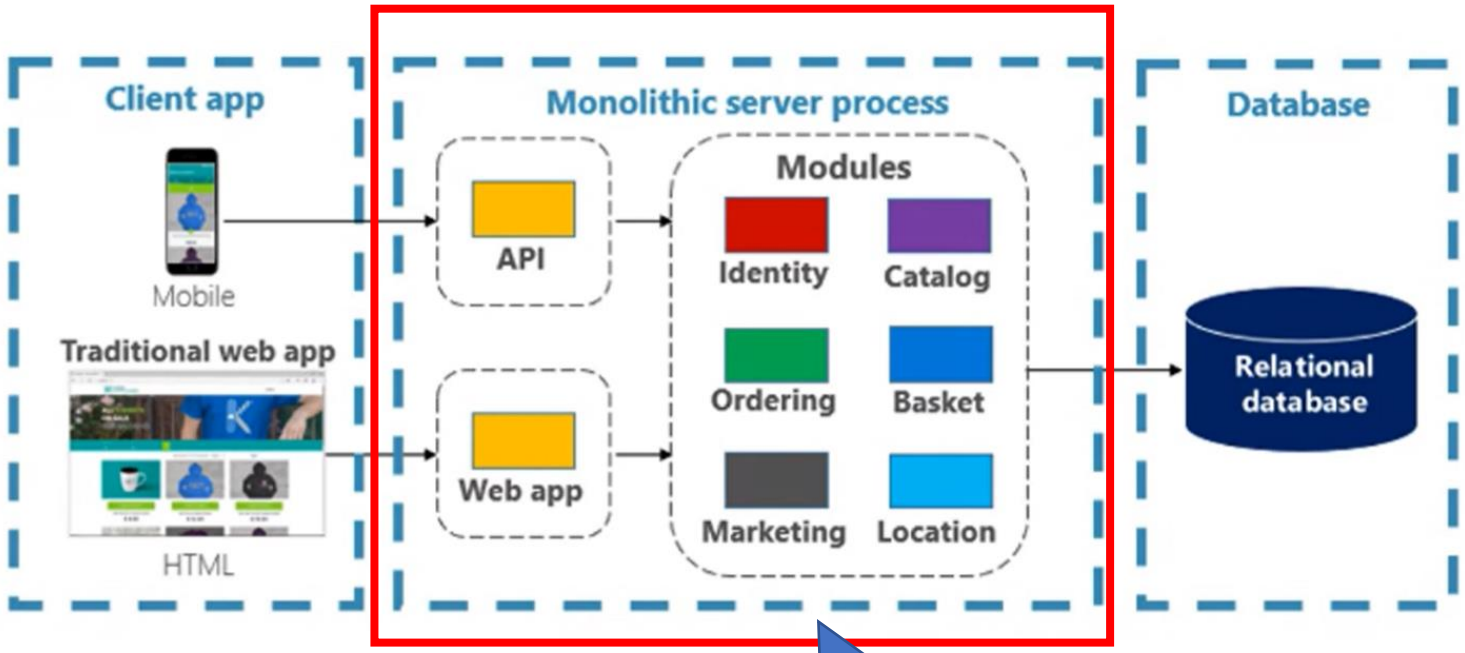
ORDINA Ahead of change

The last mile

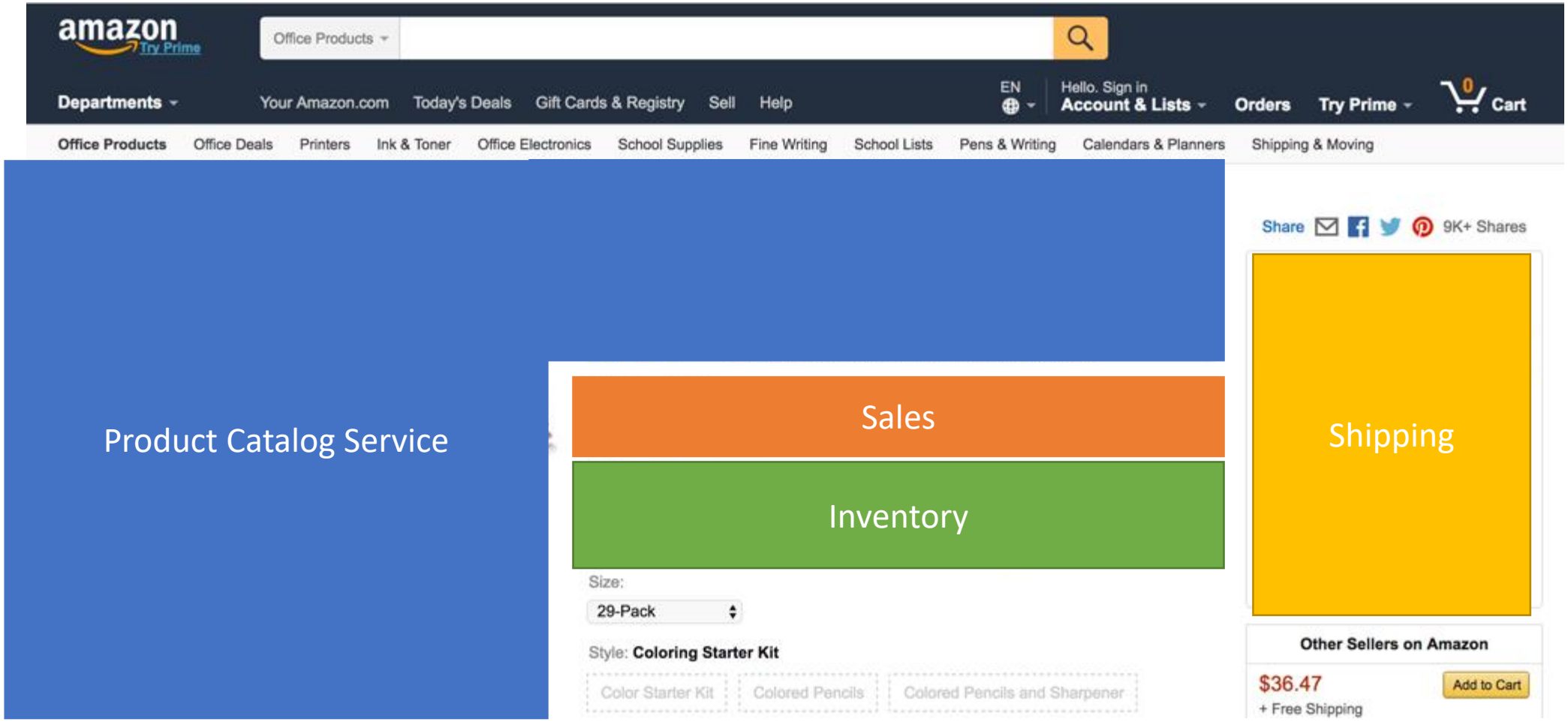
Which service owns this page?

ORDINA Ahead of change

The answer is simple in a monolithic app



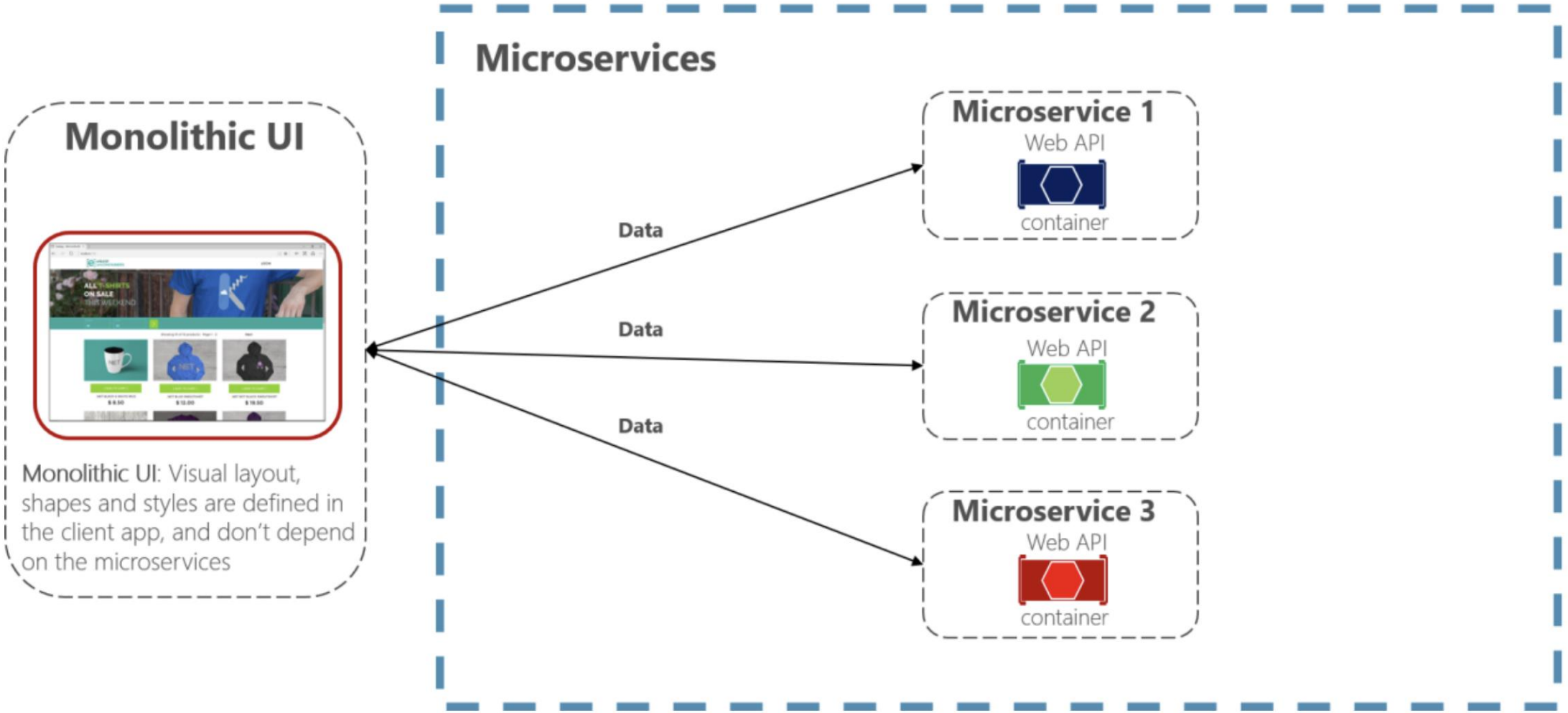
Which service owns this page?



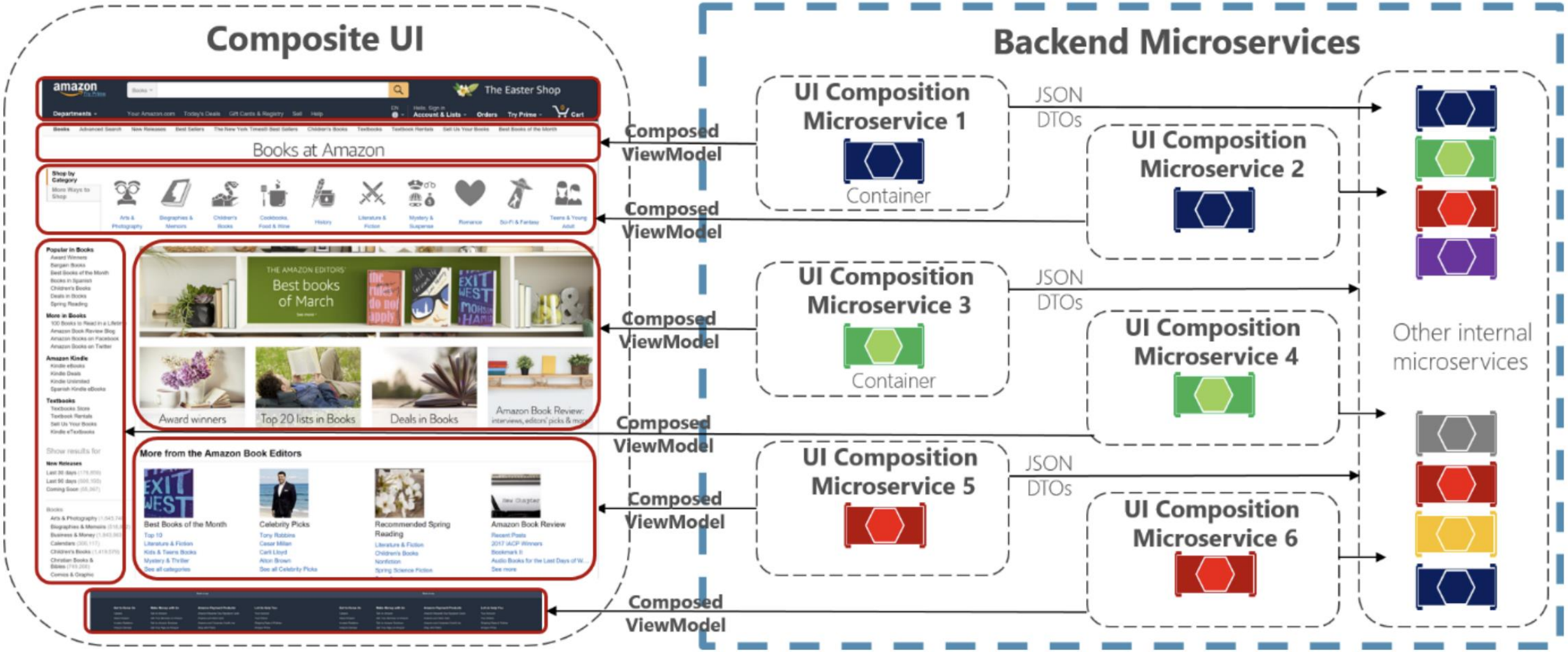
ORDINA Ahead of change

2 approaches

Frontend Monolith

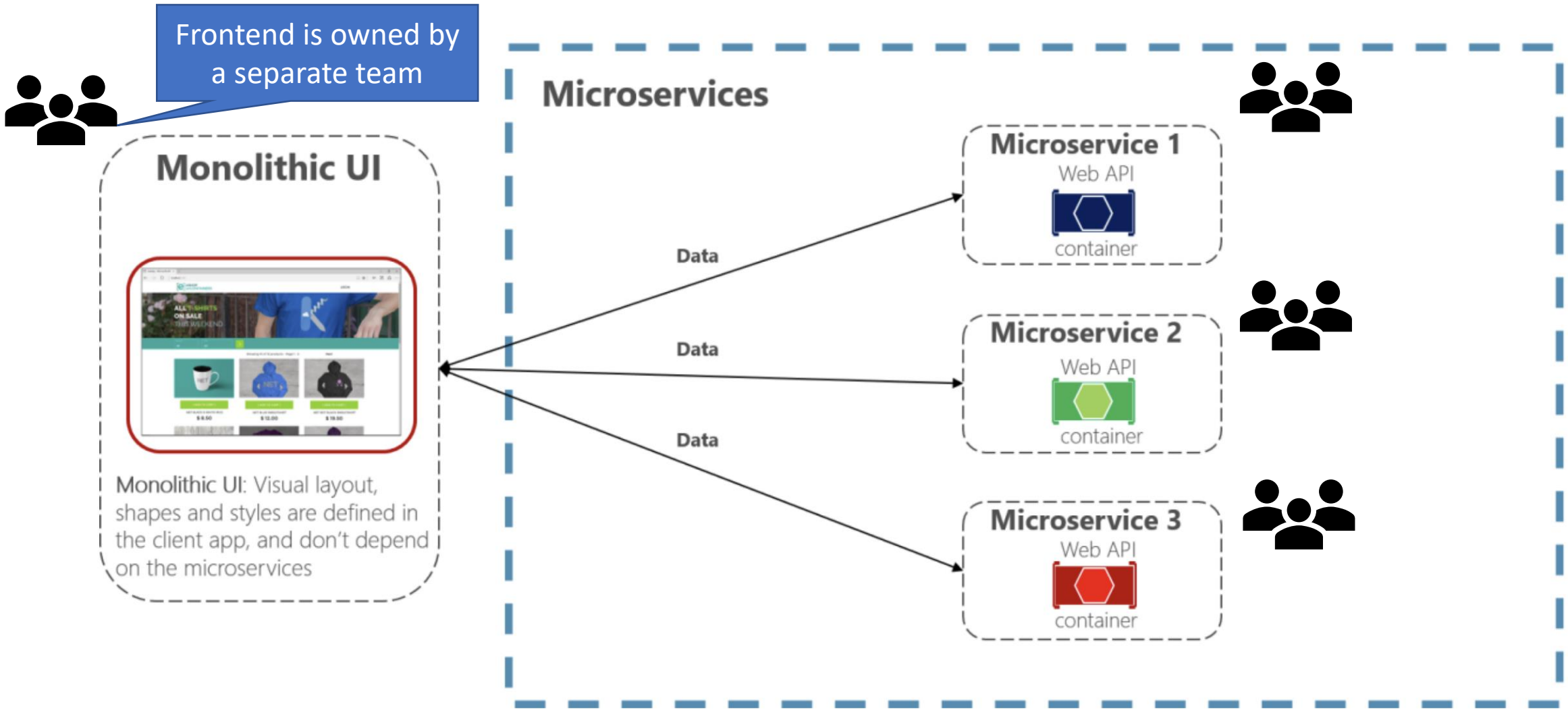


Frontend Integration



1. Frontend Monolith

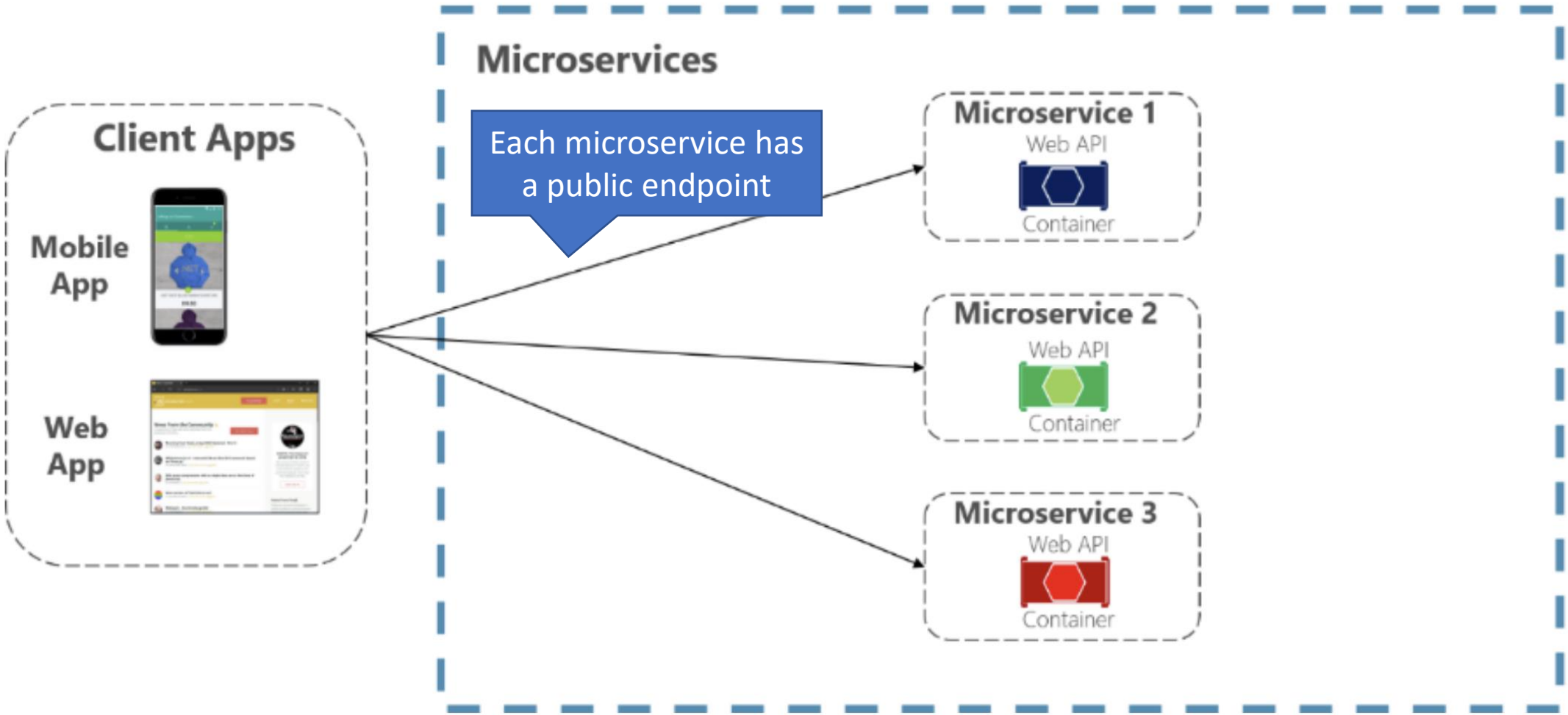
1. Frontend monolith



How do we bring the data from multiple microservices together?

1.1 Direct Client-To-Microservice communication

1.1 Direct Client-To-Microservice communication



1.1 Direct Client-To-Microservice communication

- Pros
  - Simple ☺
  - Good enough for small microservices application
  - Could still use load balancer(L4-L7), e.g. Azure Application Gateway, as a transparant tier that does load balancing, SSL termination, …
- Cons
  - Increased latency and complexity on the UI side

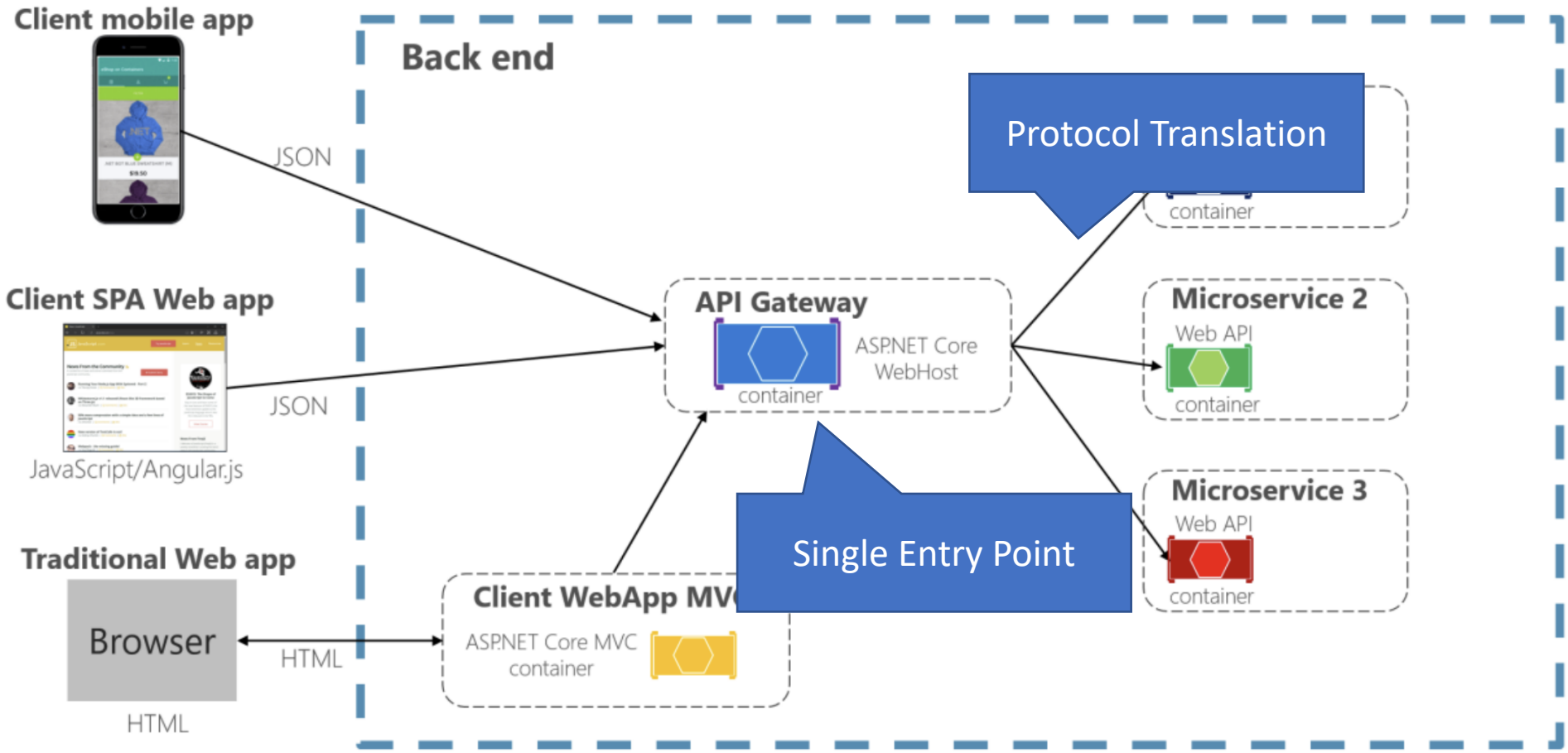
1.1 Challenges

- What happens when the application evolves and new microservices are introduced or existing microservices are updated?
- How can client apps minimize the number of requests to the back end and reduce chatty communication to multiple microservices?
- How can you handle cross-cutting concerns such as authorization, data transformations, and dynamic request dispatching?
- How can client apps communicate with services that use non-Internet-friendly protocols?
- How can you shape a facade especially made for mobile apps?

All problems in computer science can be solved by another level of indirection.” [David Wheeler]

1.2 API Gateway

1.2 API Gateway



1.2 API Gateway - Main features

- Gateway routing(Reverse proxy)
- Gateway aggregation
- Gateway offloading

1.2 API Gateway - Main features

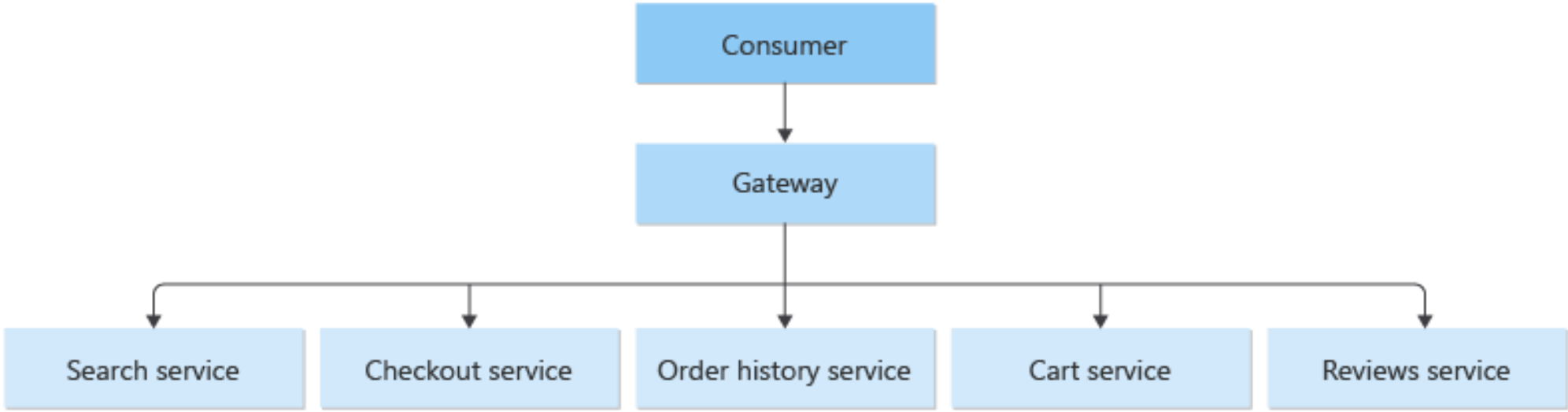
- Gateway routing(Reverse proxy)
- Gateway aggregation
- Gateway offloading

Gateway routing

- Route requests to multiple services or multiple service instances using a single endpoint.
- Uses application Layer 7 routing to route the request to the appropriate instances.
- With this pattern, the client application only needs to know about a single endpoint and communicate with a single endpoint.

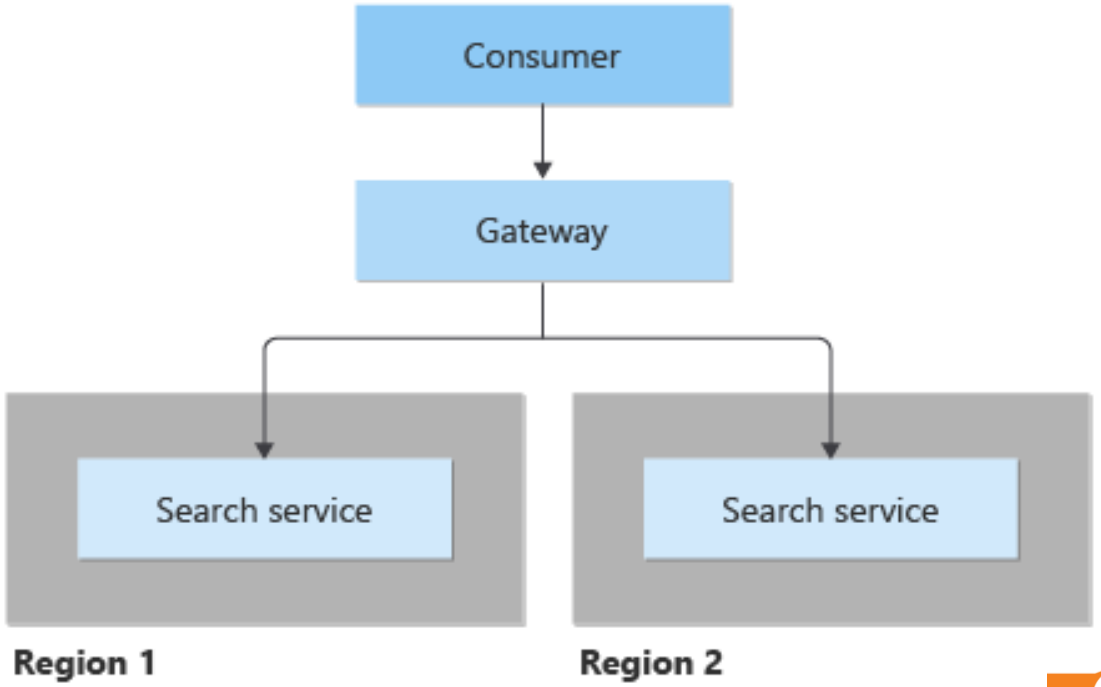
Gateway routing

- The pattern is useful when you want to:
  - Expose multiple services on a single endpoint and route to the appropriate service based on the request



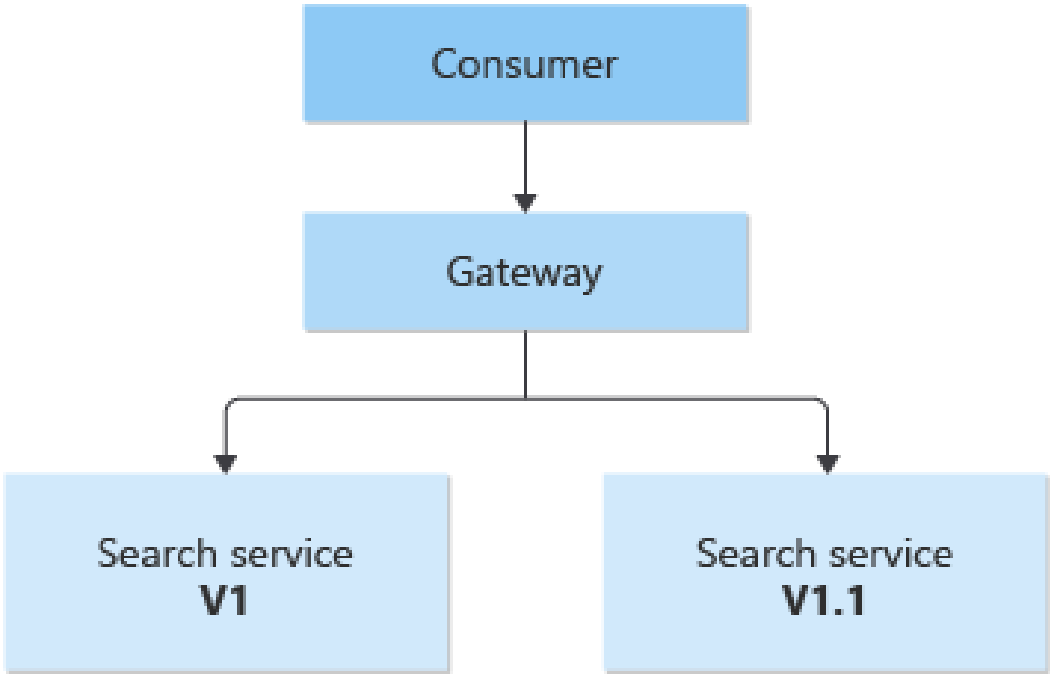
Gateway routing

- The pattern is useful when you want to:
  - Expose multiple instances of the same service on a single endpoint for load balancing or availability purposes



Gateway routing

- The pattern is useful when you want to:
  - Expose differing versions of the same service on a single endpoint and route traffic across the different versions



Gateway routing - Considerations

- Introduces a single point of failure. Think about resiliency and fault tolerance.
- Introduces a bottleneck. Ensure it can handle the load and easily scale.
- Gateway services can be global(e.g. Azure Front Door) or regional(Azure Application Gateway).
- Consider limiting public network access to the backend services, by making the services only accessible via the gateway or via a private virtual network.

API Gateway Comparison

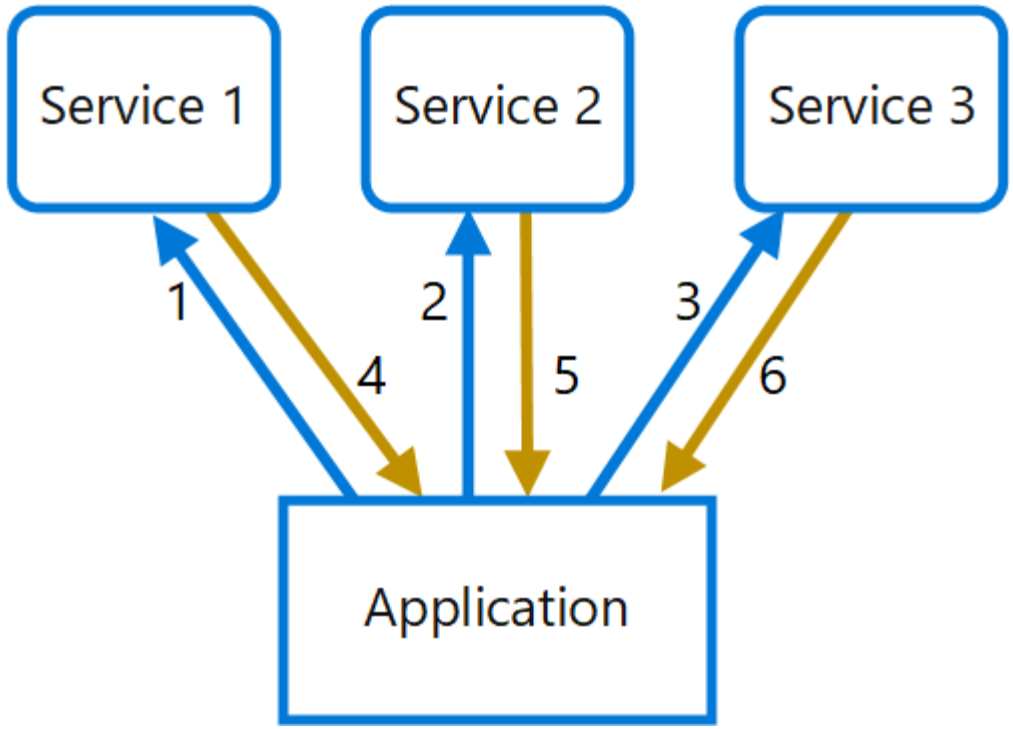
	Bring your own	YARP	Ocelot	Azure API Management	Any other (commercial) API Management solution
Gateway routing	Yes	Yes	Yes	Yes	Yes
Gateway Aggregation					
Gateway offloading					

Main features

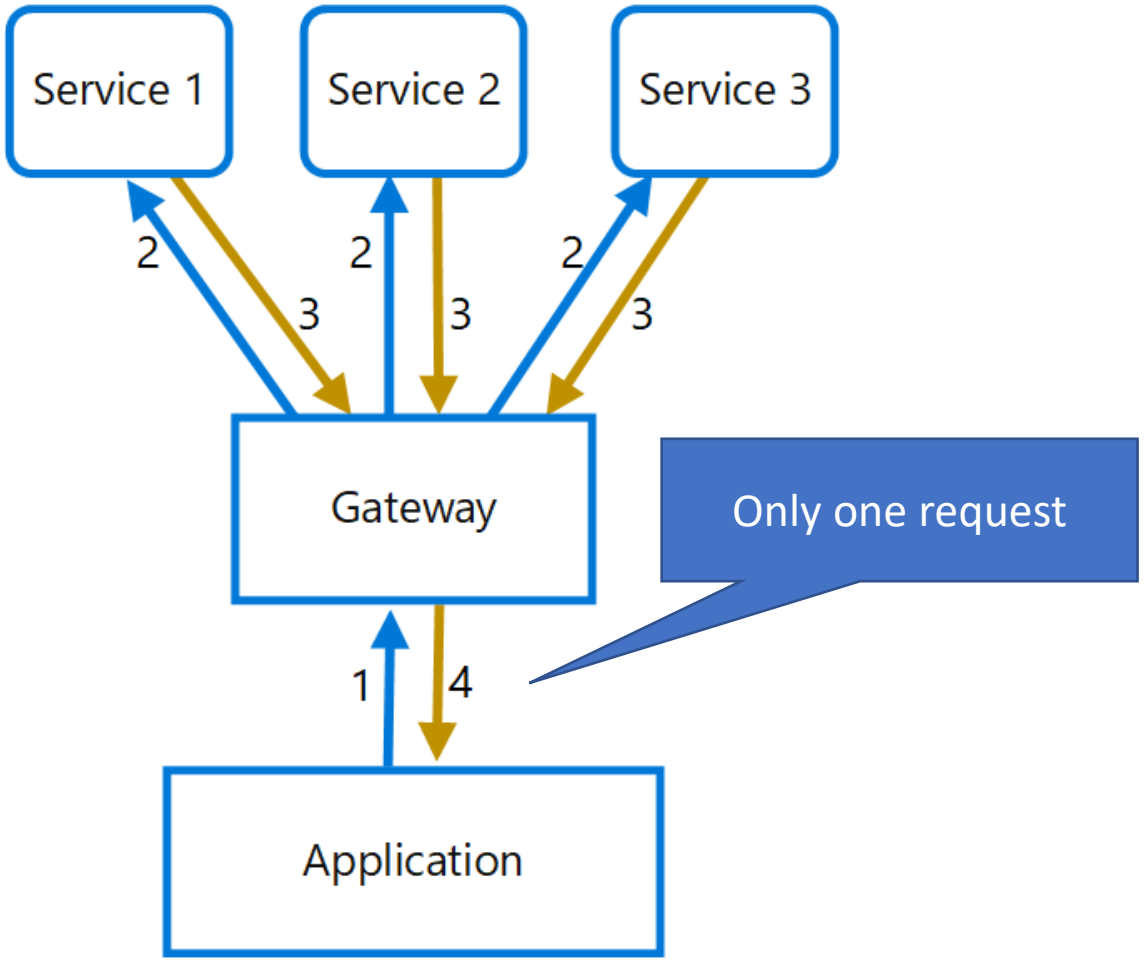
- Gateway routing(Reverse proxy)
- **Gateway aggregation**
- Gateway offloading

Gateway Aggregation - Context

- Application Chattiness



Gateway Aggregation



Gateway aggregation - Considerations

- The gateway should not introduce service coupling across the backend services.
- The gateway should be located near the backend services to reduce latency as much as possible.
- Implement a resilient design, using techniques such as bulkheads, circuit breaking, retry, and timeouts.
- If one or more service calls takes too long, it may be acceptable to timeout and return a partial set of data. Consider how your application will handle this scenario.
- Use asynchronous I/O to ensure that a delay at the backend doesn't cause performance issues in the application.
- Instead of building aggregation into the gateway, consider placing an aggregation service behind the gateway. Request aggregation will likely have different resource requirements than other services in the gateway and may impact the gateway's routing and offloading functionality.

Some ways to implement this

- Ocelot Request Aggregation
- Build your own
- ViewModel Composition
- GraphQL Federation

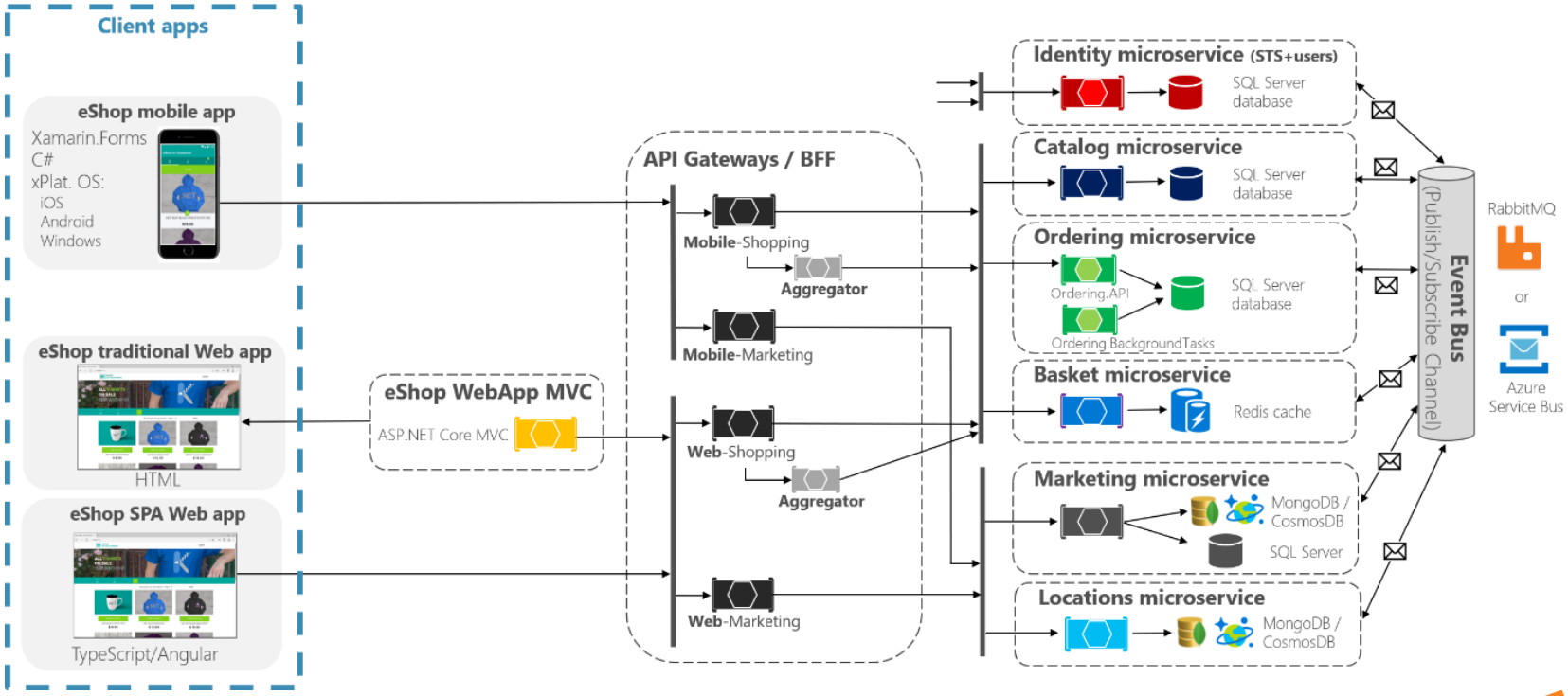
Ocelot Request Aggregation

- https://ocelot.readthedocs.io/en/latest/features/requestaggregation.html#request-aggregation


```
{
    "Routes": [
        {
            "DownstreamPathTemplate": "/",
            "UpstreamPathTemplate": "/laura",
            "UpstreamHttpMethod": [
                "Get"
            ],
            "DownstreamScheme": "http",
            "DownstreamHostAndPorts": [
                {
                    "Host": "localhost",
                    "Port": 51881
                }
            ],
            "Key": "Laura"
        },
        {
            "DownstreamPathTemplate": "/",
            "UpstreamPathTemplate": "/tom",
            "UpstreamHttpMethod": [
                "Get"
            ],
            "DownstreamScheme": "http",
            "DownstreamHostAndPorts": [
                {
                    "Host": "localhost",
                    "Port": 51882
                }
            ],
            "Key": "Tom"
        }
    ],
    "Aggregates": [
        {
            "RouteKeys": [
                "Tom",
                "Laura"
            ],
            "UpstreamPathTemplate": "/",
            "Aggregator": "FakeDefinedAggregator"
        }
    ]
}
```

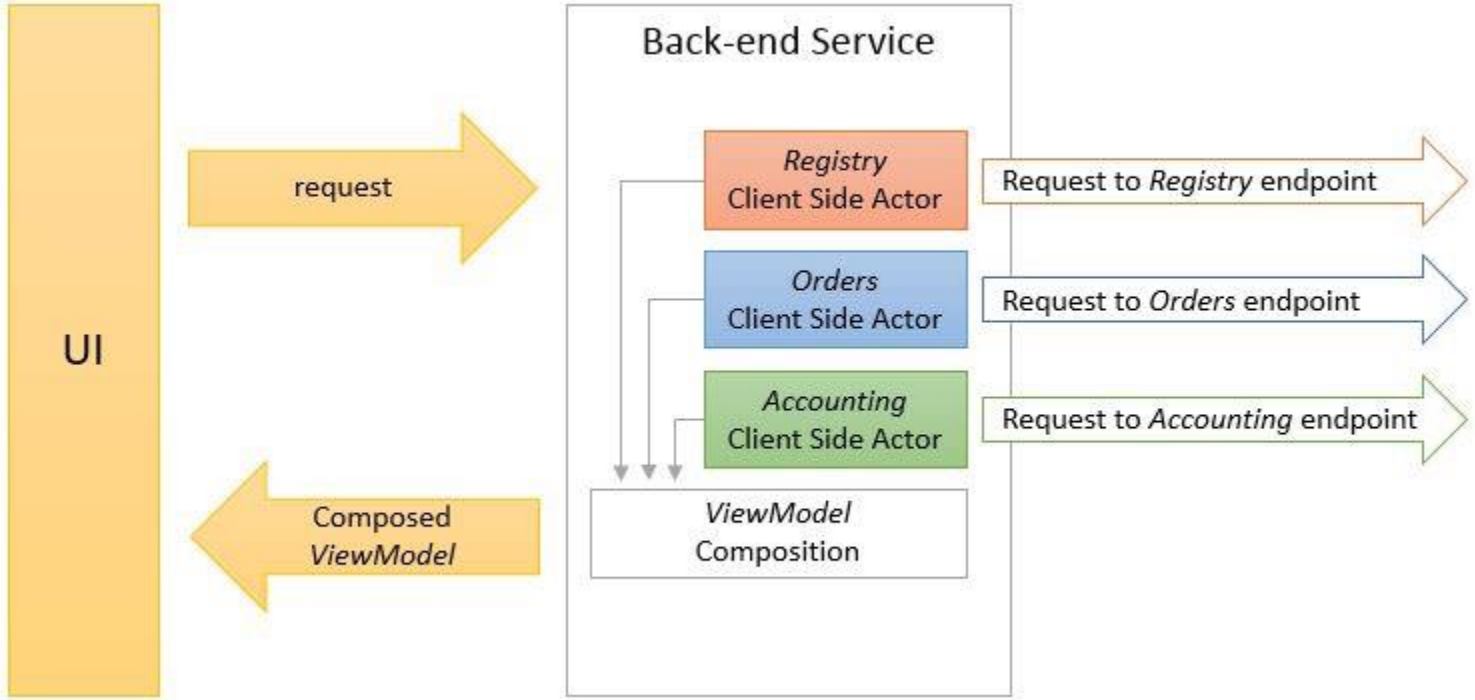
Build your own

- https://github.com/dotnet-architecture/eShopOnContainers/tree/dev/src/ApiGateways/Web.Bff.Shopping/aggregator



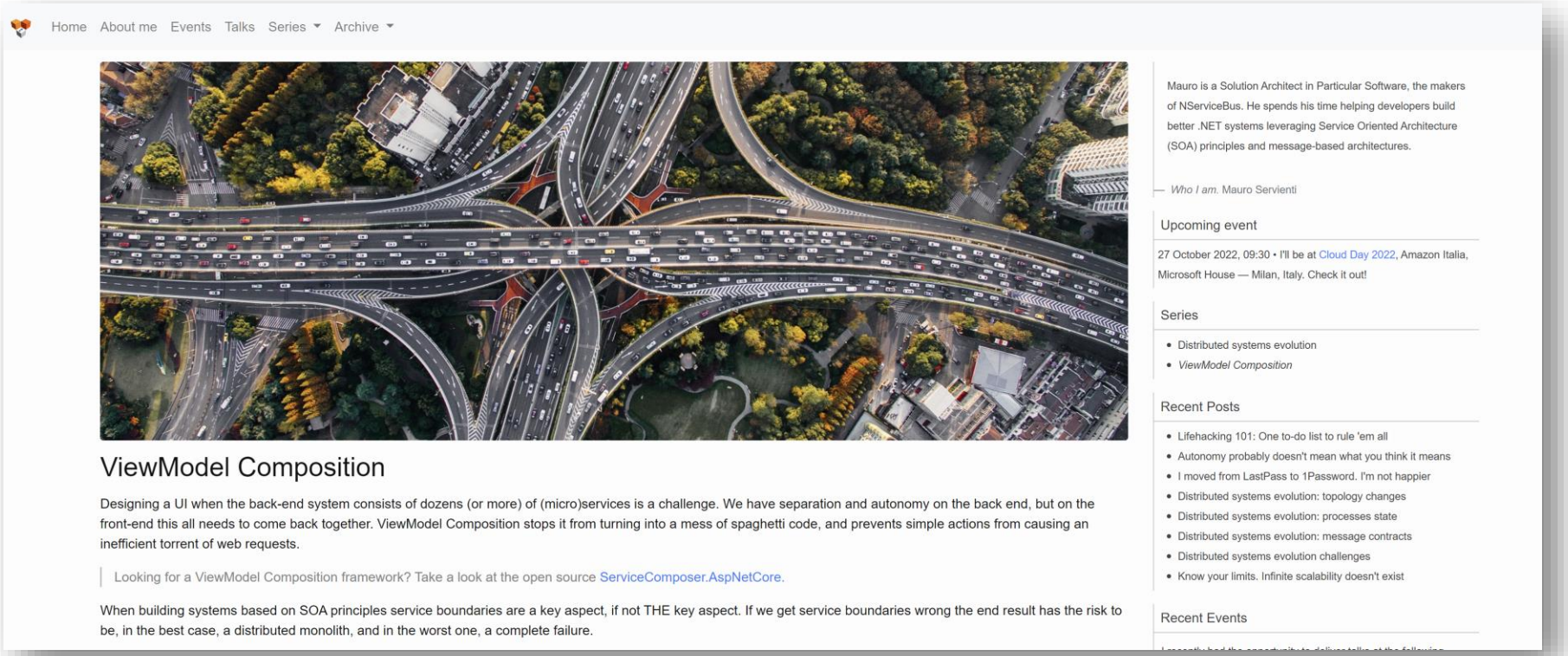
ViewModel Composition

- https://github.com/Particular/Workshop/tree/master/demos/asp-net-core

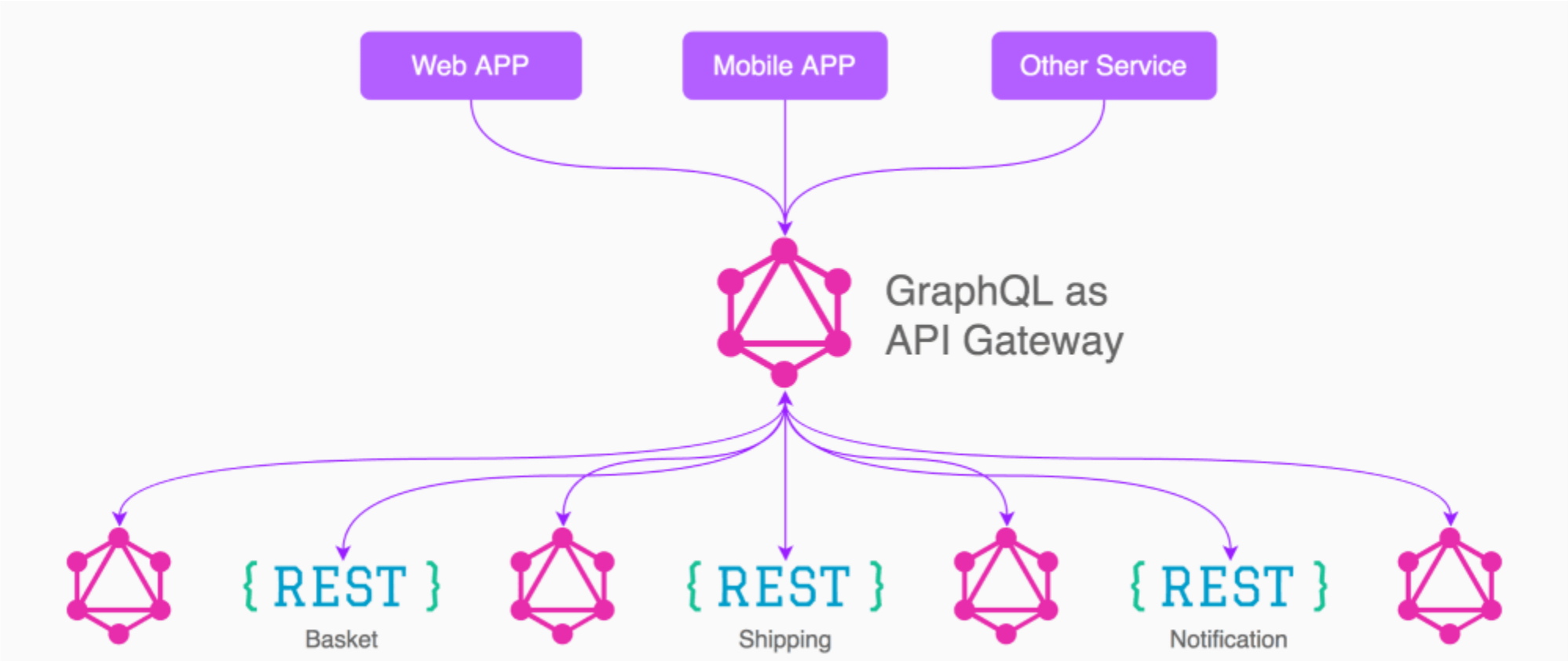


ViewModel Composition

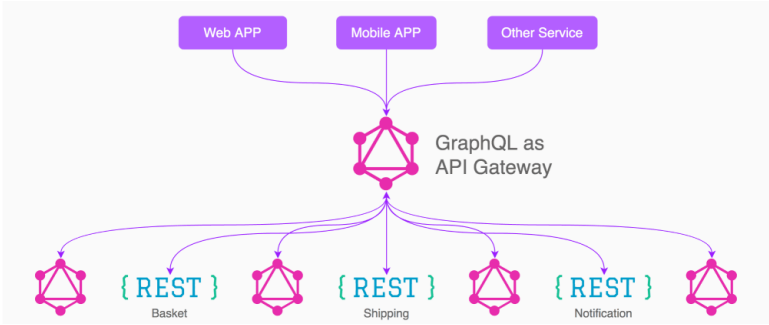
- https://milestone.topics.it/series/view-model-composition.html



GraphQL Federation



GraphQL Federation



- GraphQL federation allows you to set up a single GraphQL API, or a gateway, that fetches from all your other APIs.
- With GraphQL federation, you tell the gateway where it needs to look for the different objects and what URLs they live at. The subgraphs provide metadata that the gateway uses to automatically stitch everything together.
- https://netflixtechblog.com/how-netflix-scales-its-api-with-graphql-federation-part-1-ae3557c187e2

API Gateway Comparison

	Bring your own	YARP	Ocelot	Azure API Management	Any other (commercial) API Management solution
Gateway routing	Yes	Yes	Yes	Yes	Yes
Gateway Aggregation	Yes	No	Yes	Yes(Synthetic GraphQL,…)	Some(*)
Gateway offloading					

(*): It is mainly considered a controversial feature that can lead to unwanted coupling

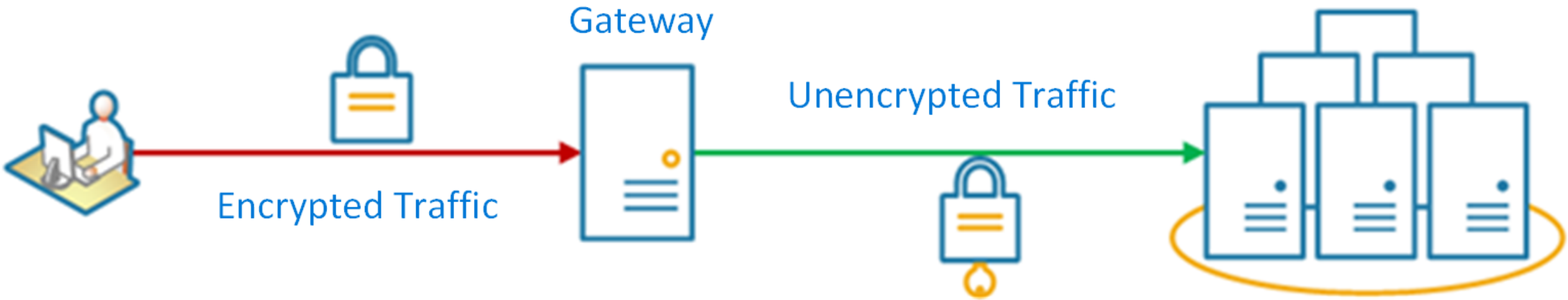
1.2 API Gateway - Main features

- Gateway routing(Reverse proxy)
- Gateway aggregation
- Gateway offloading

Gateway offloading - context

- Some features are commonly used across multiple services, and these features require configuration, management, and maintenance
- Some common services such as authentication, authorization, logging, monitoring, or throttling can be difficult to implement and manage across a large number of deployments.

Gateway offloading



Gateway offloading

- Benefits of this pattern include:
  - Simplify the development of services by removing the need to distribute and maintain supporting resources, such as web server certificates and configuration for secure websites.
  - Allow dedicated teams to implement features that require specialized expertise, such as security.
  - Provide consistency for request and response logging and monitoring.

API Gateway Comparison

	Bring your own	YARP	Ocelot	Azure API Management	Any other (commercial) API Management solution
Gateway routing	Yes	Yes	Yes	Yes	Yes
Gateway Aggregation	Yes	No	Yes	Yes(Synthetic GraphQL,…)	Some(*)
Gateway offloading	Yes	Yes	Yes	Yes	Yes

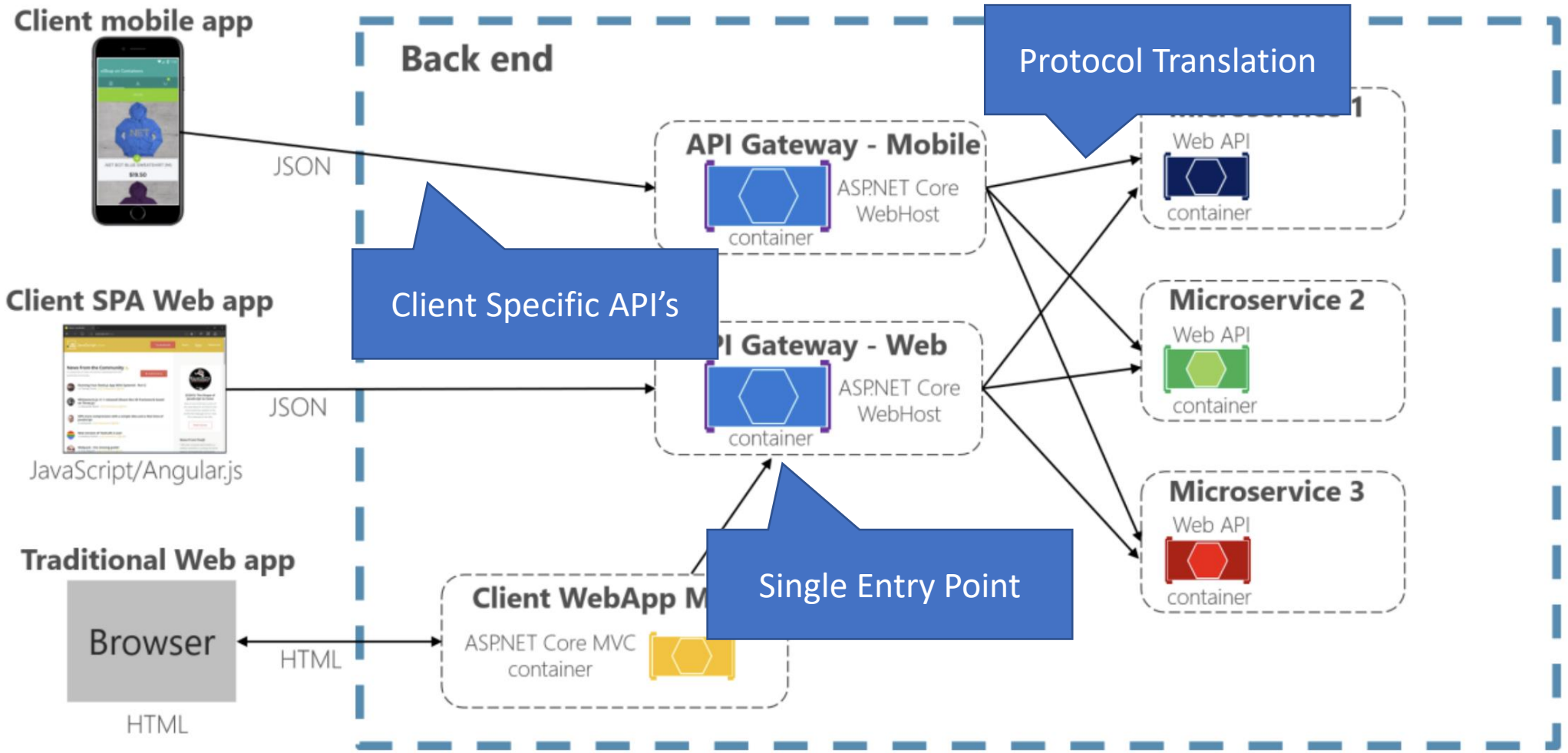
(*): It is mainly considered a controversial feature that can lead to unwanted coupling

1.2 API Gateway

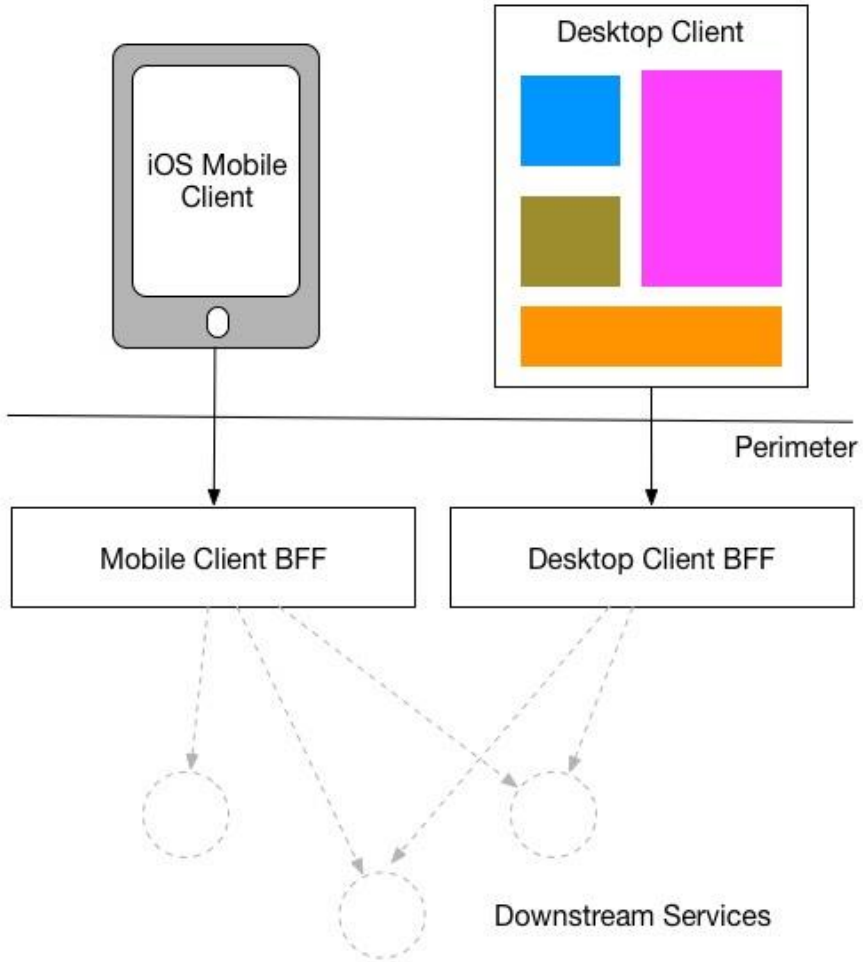
- Pros
  - Single Entry point
  - Allows gateway aggregation, offloading and routing abstracting away the internal microservices
  - Can also provide other cross-cutting features such as authentication, SSL termination, and cache
- Cons
  - Single point of failure
  - Increased latency(extra hop)
  - Can introduce extra coupling
  - Gateway can become bloated(*) because of different application needs and become a monolithic aggregator or orchestrator violating microservices autonomy

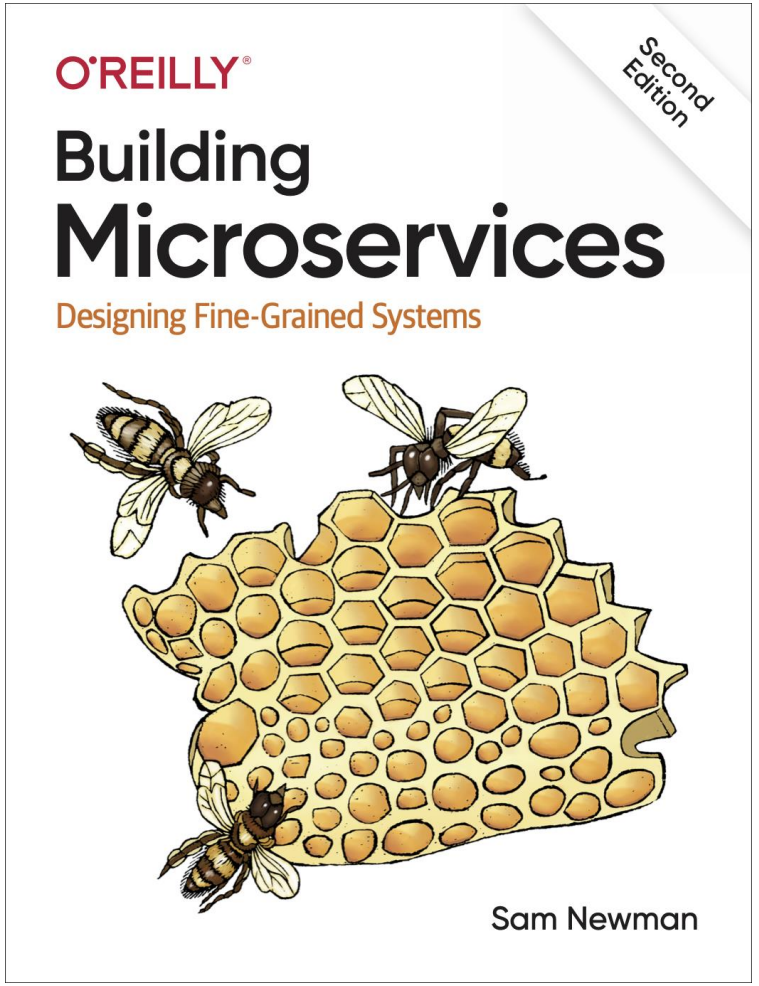
(*): GraphQL Federation helps to avoid this

1.3 Backend for Frontend(BFF)



1.3 Backend for Frontend(BFF)

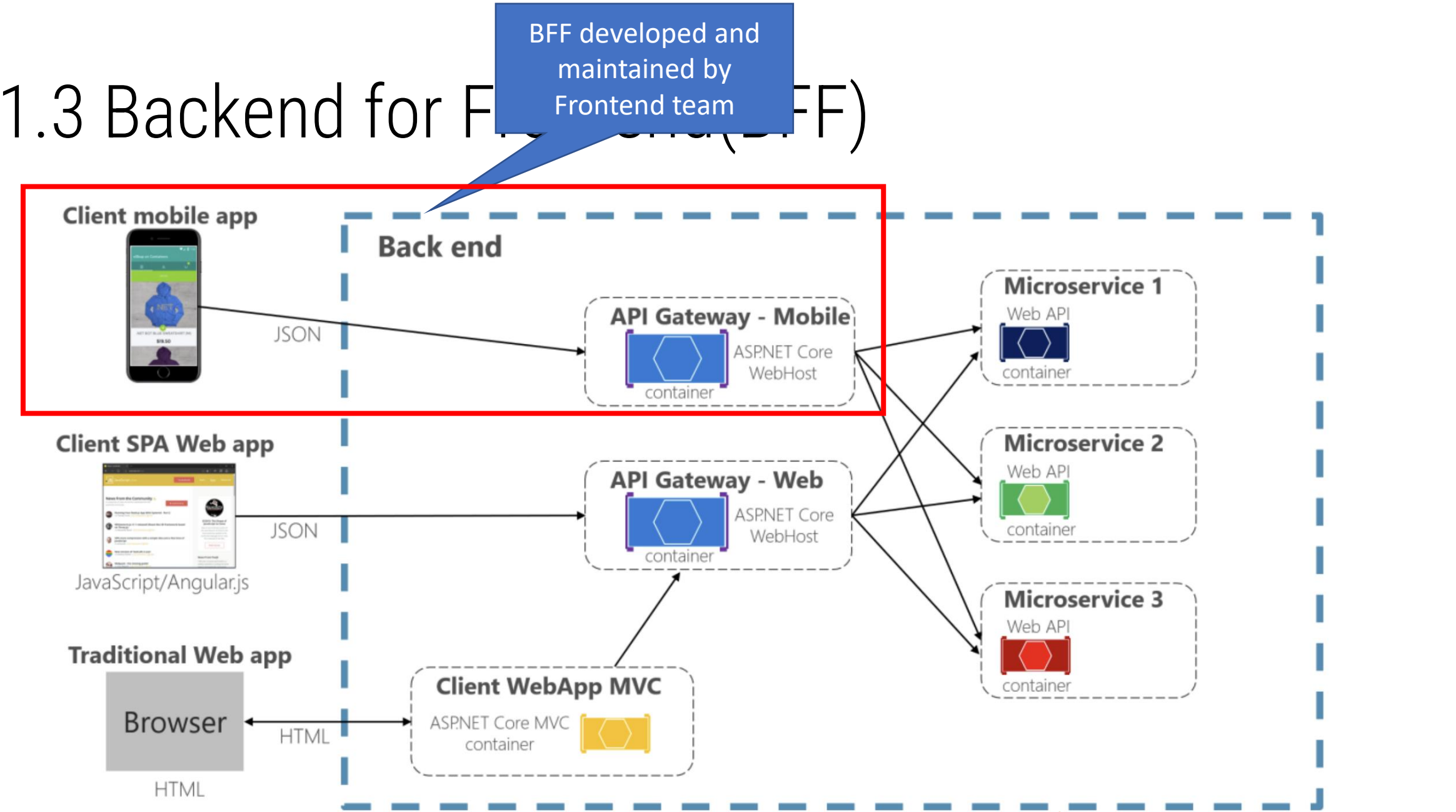




1.3 Backend for Frontend(BFF)

- Use this pattern when:
  - A shared or general purpose backend service must be maintained with significant development overhead.
  - You want to optimize the backend for the requirements of specific client interfaces.
  - Customizations are made to a general-purpose backend to accommodate multiple interfaces.
- This pattern may not be suitable:
  - When interfaces make the same or similar requests to the backend.
  - When only one interface is used to interact with the backend.

1.3 Backend for F…nd(BFF)



1.3 Backend for Frontend(BFF)

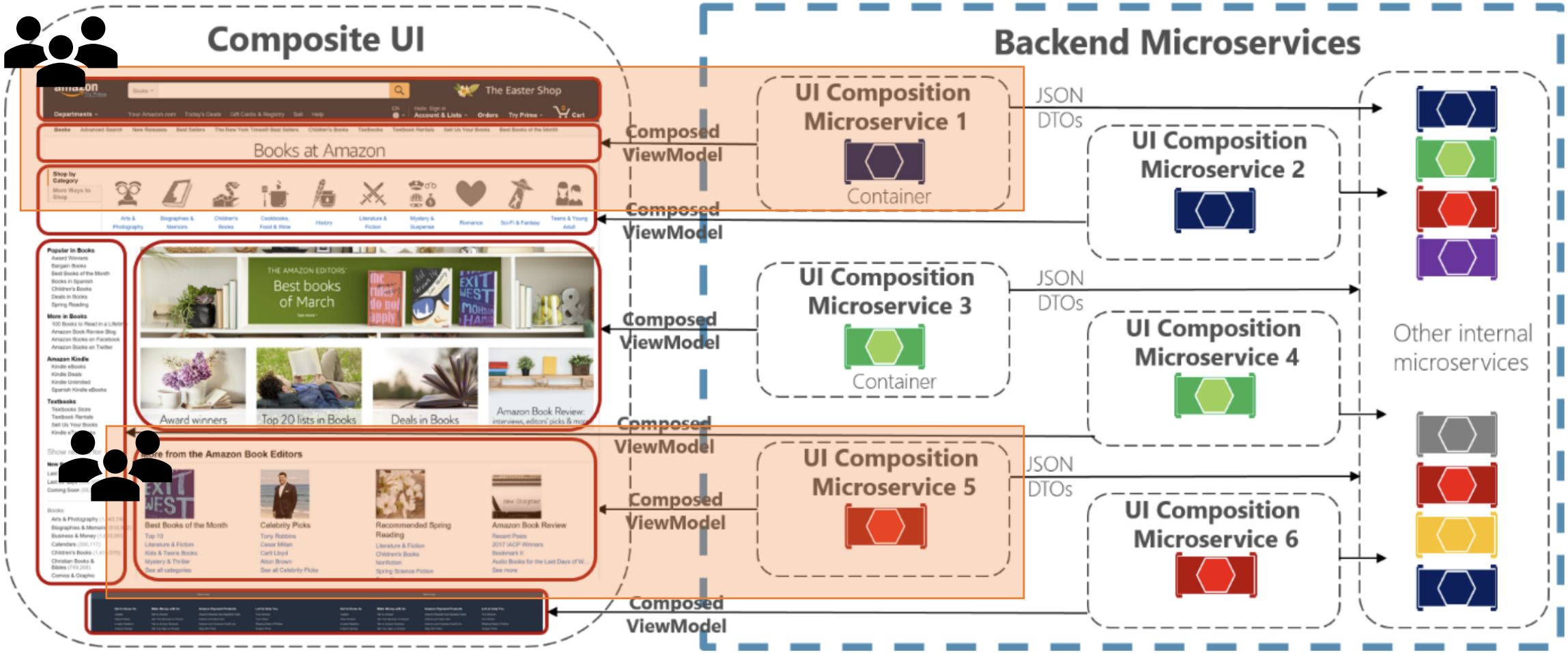
- Pros
  - Same advantages as the API gateway(as it is one ☺)
  - Optimized for specific experience(Mobile, Desktop, …)
- Cons
  - More moving parts
  - Possible duplication

1. Frontend Monolith Summary

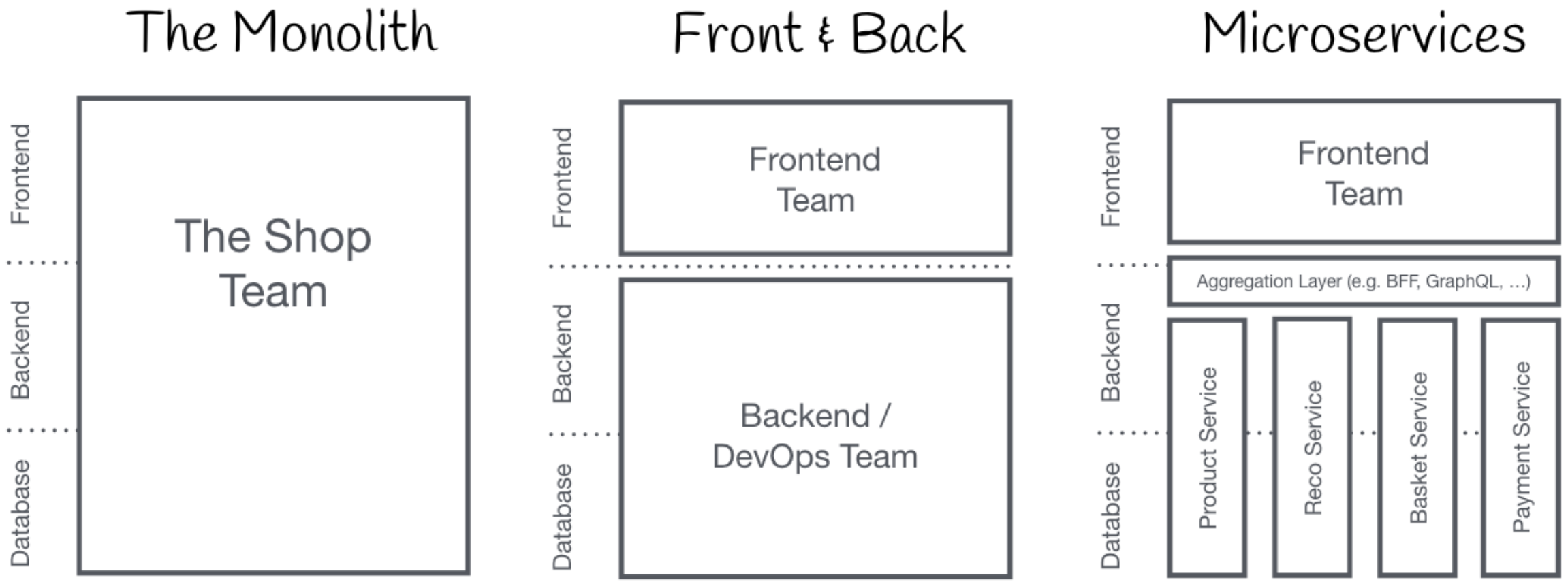
- 3 ways to bring data from multiple microservices together
  - Direct Client-to-Microservice communication
  - API Gateway
    - Routing
    - Request Aggregation
    - Offloading
  - Backend for Frontend

2. Frontend Integration

Frontend Integration

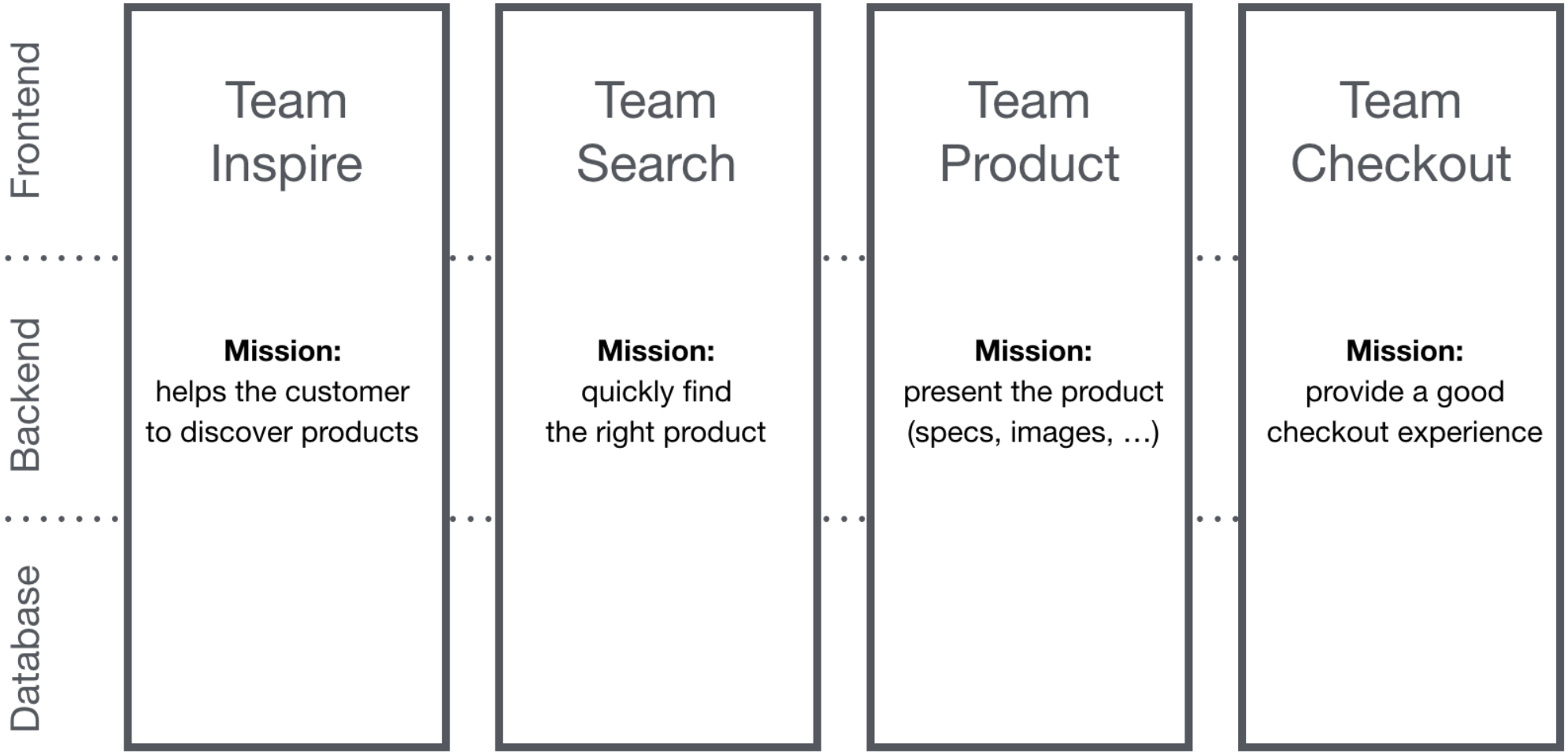


Frontend Integration



https://micro-frontends.org/

End-to-end teams with Micro-Frontends



https://micro-frontends.org/

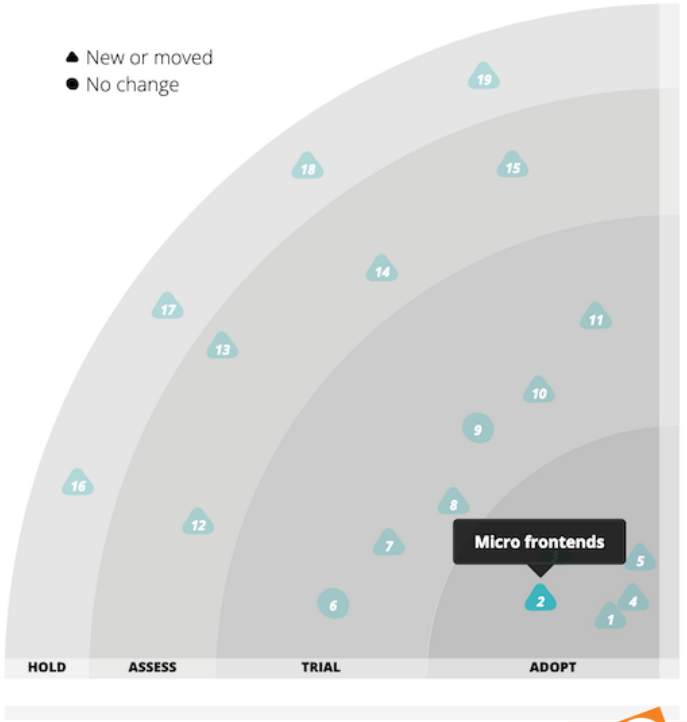
Micro-Frontends

- "*An architectural style where independently deliverable frontend applications are composed into a greater whole*"

ADOPT

1. Four key metrics
2. **Micro frontends**

We've seen significant benefits from introducing microservices, which have allowed teams to scale the delivery of independently deployed and maintained services. Unfortunately, we've also seen many teams create a frontend monolith — a large, entangled browser application that sits on top of the backend services — largely neutralizing the benefits of microservices. Since we first described **micro frontends** as a technique to address this issue, we've had almost universally positive experiences with the approach and have found a number of patterns to use micro frontends even as more and more code shifts from the server to the web browser. So far, web components have been elusive in this field, though.



Introduced by Thoughtworks in 2016(!)

ORDINA Ahead of change

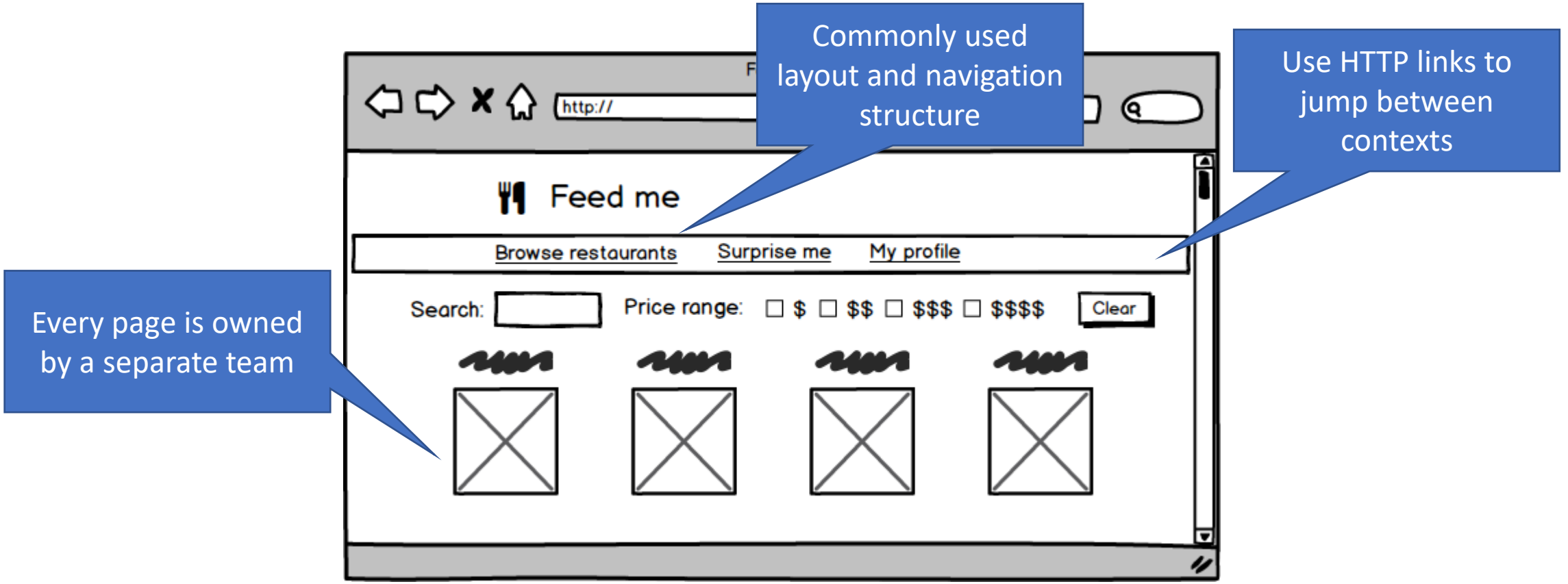
Micro-Frontends

- Pros
  - Smaller, more cohesive and maintainable codebases
  - More scalable organisations with decoupled, autonomous teams
  - The ability to upgrade, update, or even rewrite parts of the frontend in a more incremental fashion
- Cons:
  - Can lead to duplication of dependencies, increasing the number of bytes our users must download.

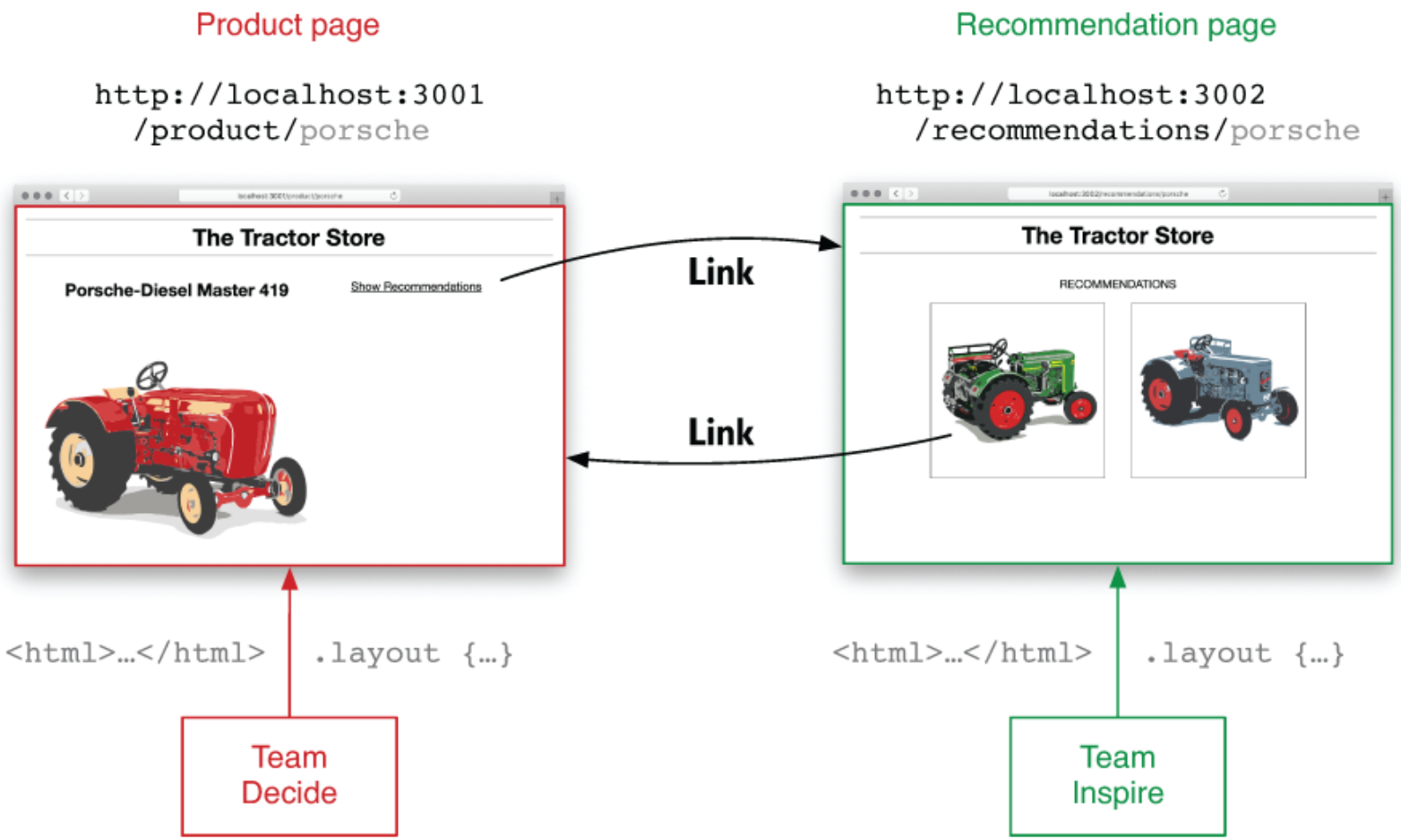
How can we compose our micro-frontend?

2.1 Microfrontend per page

2.1 Micro-frontend per page



2.1 Micro-frontend per page

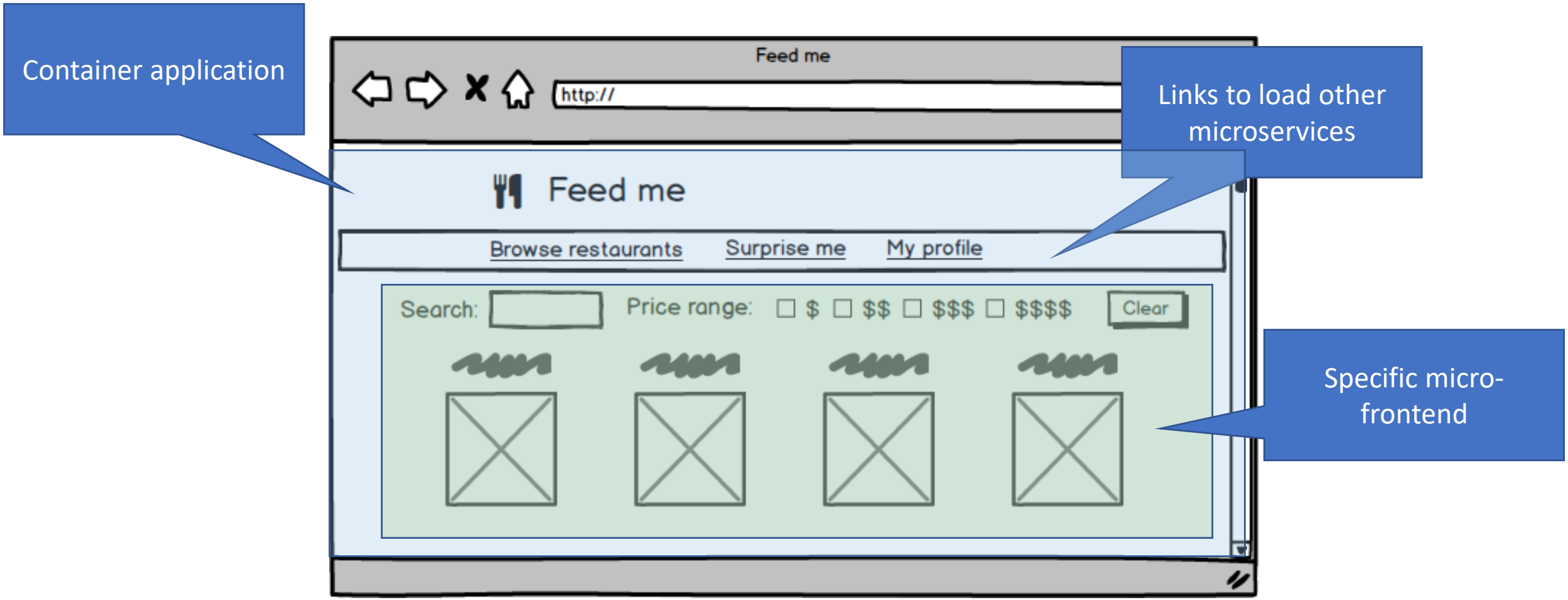


Micro-frontend per page

- Pros
  - Easy
  - Loose coupling
  - Robust
- Cons
  - Not optimal from user perspective
  - Technical redundancy

2.2 Single Container Application

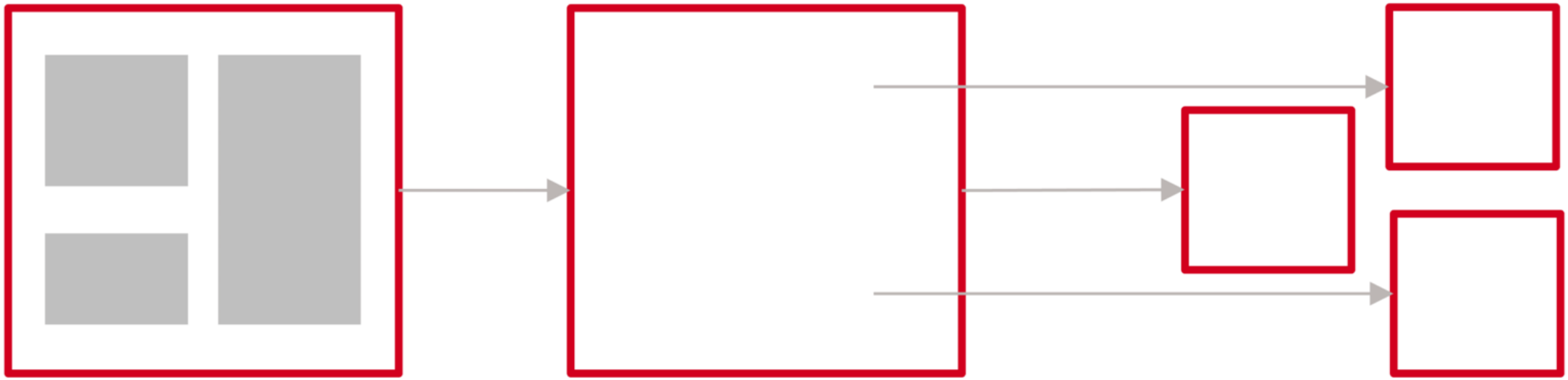
2.2 Single Container Application



2.2 Single Container Application

- Possible techniques
  - Server-side template composition
  - Build time integration
  - Runtime integration
    - IFrames
    - Web components
    - Single SPA
    - Webpack Module Federation

Server-side template composition



SSI/ESI

- Supported by IIS
- Uses the concept of directives, e.g. #include -> which the server replaces with a file or result of a request

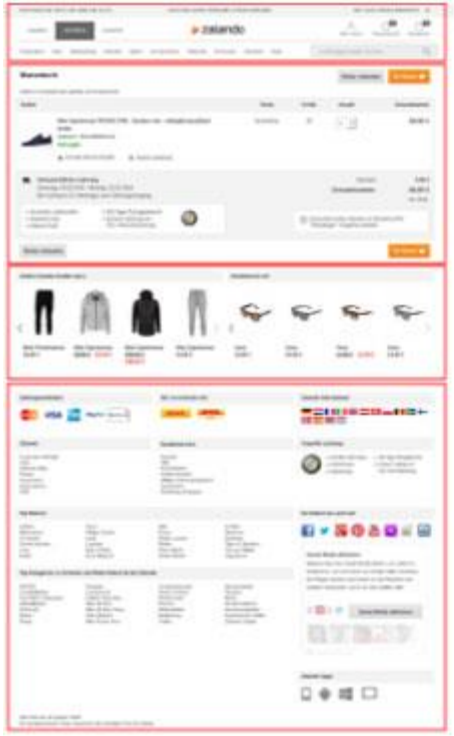
```
<html>
  ...
  <body class="decide_layout">
    ...
    <aside class="decide_recos">
      <!--#include virtual="/inspire/fragment/recommendations/porsche" -->
    </aside>
  </body>
</html>
```

Custom frameworks

Tailor

Podium

Easy server side composition of microfrontends



```
<html>
<head>
  <fragment src="http://assets.domain.com">
</head>
<body>
  <fragment src="http://header.domain.com">
  <fragment src="http://cart.domain.com" primary>
  <fragment src="http://reco.domain.com" async>
  <fragment src="http://footer.domain.com" async>
</body>
</html>
```

- https://www.oreilly.com/content/better-streaming-layouts-for-frontend-microservices-with-tailor/
- https://podium-lib.io/

Server-side template composition

- Pros
  - Minimises the time until the end-user sees something in the browser
  - Works with Javascript disabled
- Cons
  - Less flexible than other approaches

Build time integration

- Publish each micro frontend as a package, and have the container application include them all as library dependencies.

```
{
  "name": "@feed-me/container",
  "version": "1.0.0",
  "description": "A food delivery web app",
  "dependencies": {
    "@feed-me/browse-restaurants": "^1.2.3",
    "@feed-me/order-food": "^4.5.6",
    "@feed-me/user-profile": "^7.8.9"
  }
}
```

Build time integration

- This approach means that we have to re-compile and release every single micro frontend in order to release a change to any individual part of the product.
- Don’t do this!

Runtime integration

- IFrames, Web components,…

IFrames

```
<html>
  <head>
    <title>Feed me!</title>
  </head>
  <body>
    <h1>Welcome to Feed me!</h1>

    <iframe id="micro-frontend-container"></iframe>

    <script type="text/javascript">
      const microFrontendsByRoute = {
        '/': 'https://browse.example.com/index.html',
        '/order-food': 'https://order.example.com/index.html',
        '/user-profile': 'https://profile.example.com/index.html',
      };

      const iframe = document.getElementById('micro-frontend-container');
      iframe.src = microFrontendsByRoute[window.location.pathname];
    </script>
  </body>
</html>
```

IFrames

- Pros
  - Easy
  - Offer a good degree of isolation
- Cons
  - Less flexible than other approaches
  - Layout constraints
  - Can be difficult to build integrations between different parts of the application(routing, history, deep-linking)
  - Challenging to making your page fully responsive.

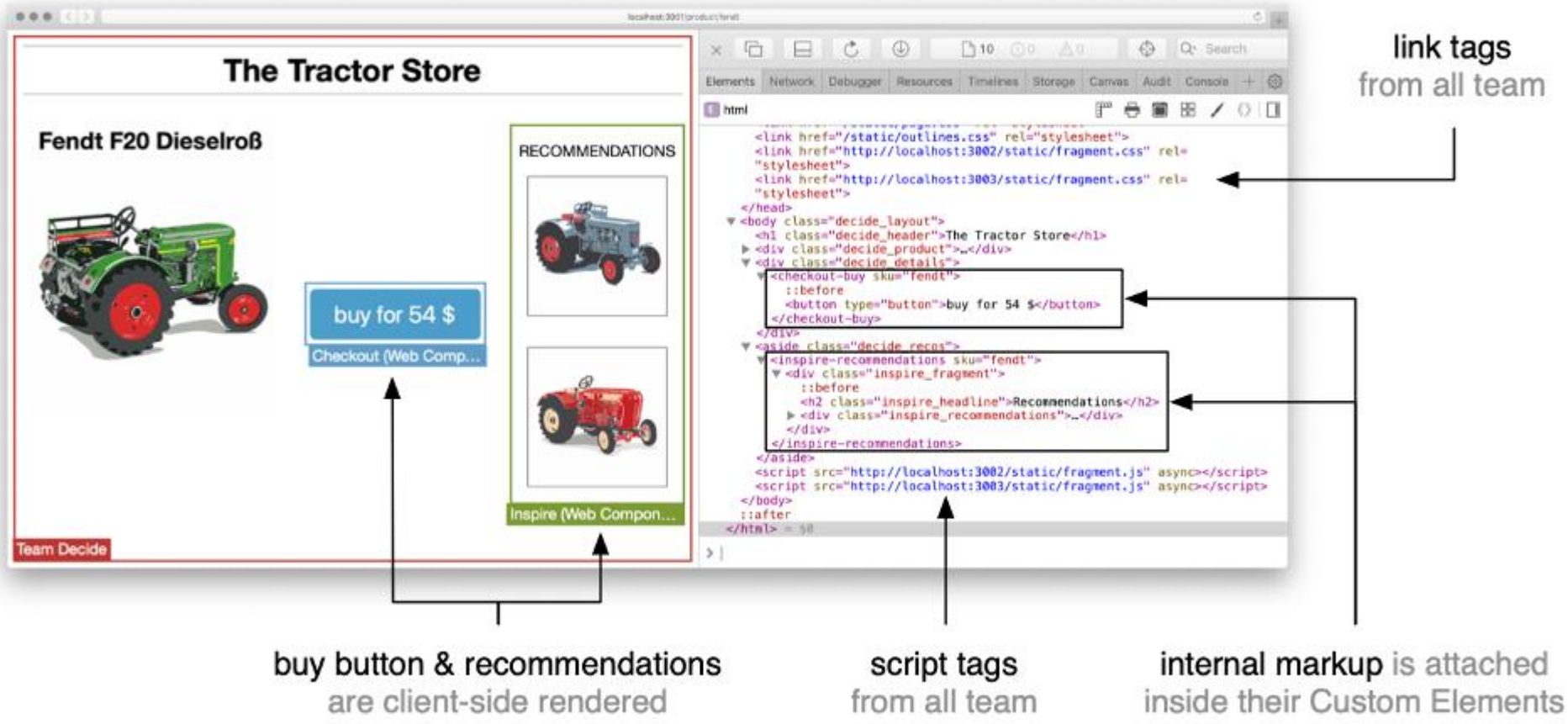
Web Components

- Web Components comprises of 3 concepts:

1. **Custom Elements:** Set of JavaScript APIs that allow you to create your own HTML elements and define the way they behave.
2. **Shadow DOM: A p**rivate DOM that is scoped to your component only and capable of isolating CSS and JavaScript.
3. **HTML templates.** New HTML tags that allows us to create templates for your components.

- In simpler terms, you can create your own HTML selector, like <select>, <h1> etc.

Web Components



Web Components

- Pros
  - Web Standard
  - Extra isolation features(shadow dom, custom elements)
  - Can be implemented using framework of your choice (React, Angular, Blazor, …)
- Cons
  - Javascript should be enabled
  - No server-side rendering option
  - Browser support

Single SPA

- Single-Spa is a framework for bringing together multiple JavaScript microfrontends in a frontend application. Architecting your frontend using single-spa enables many benefits, such as:
  - Use multiple frameworks on the same page without page refreshing (React, Angular, Ember, or whatever you're using)
  - Deploy your microfrontends independently
  - Write code using a new framework, without rewriting your existing app
  - Lazy load code for improved initial load time

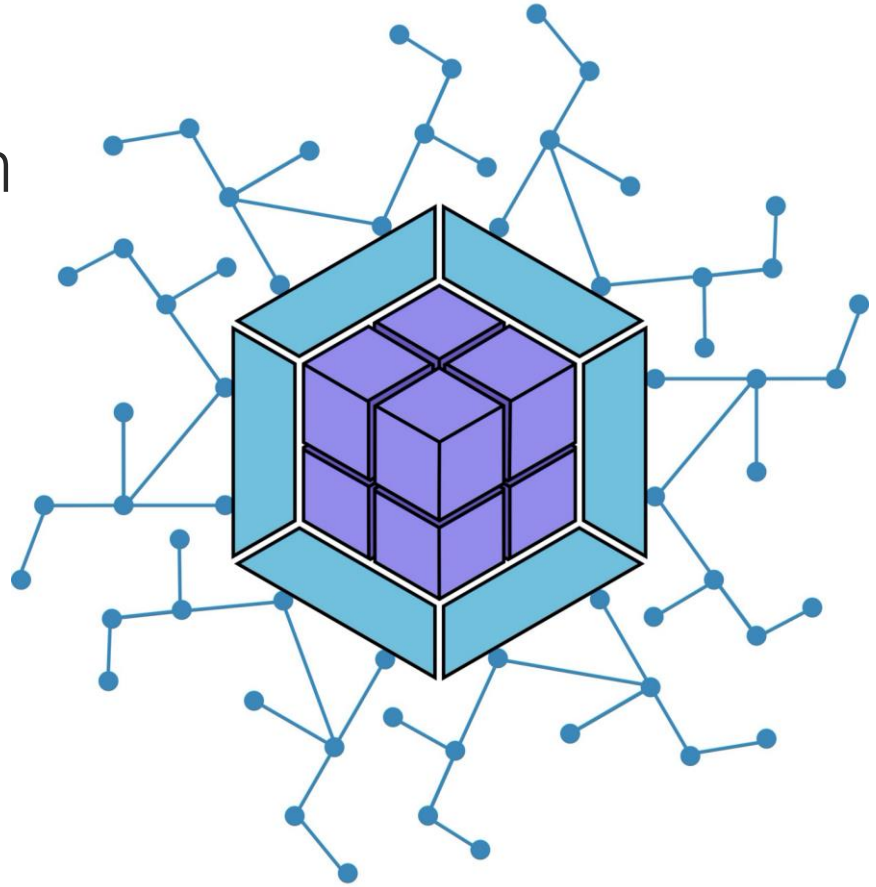
https://single-spa.js.org/

Single-SPA

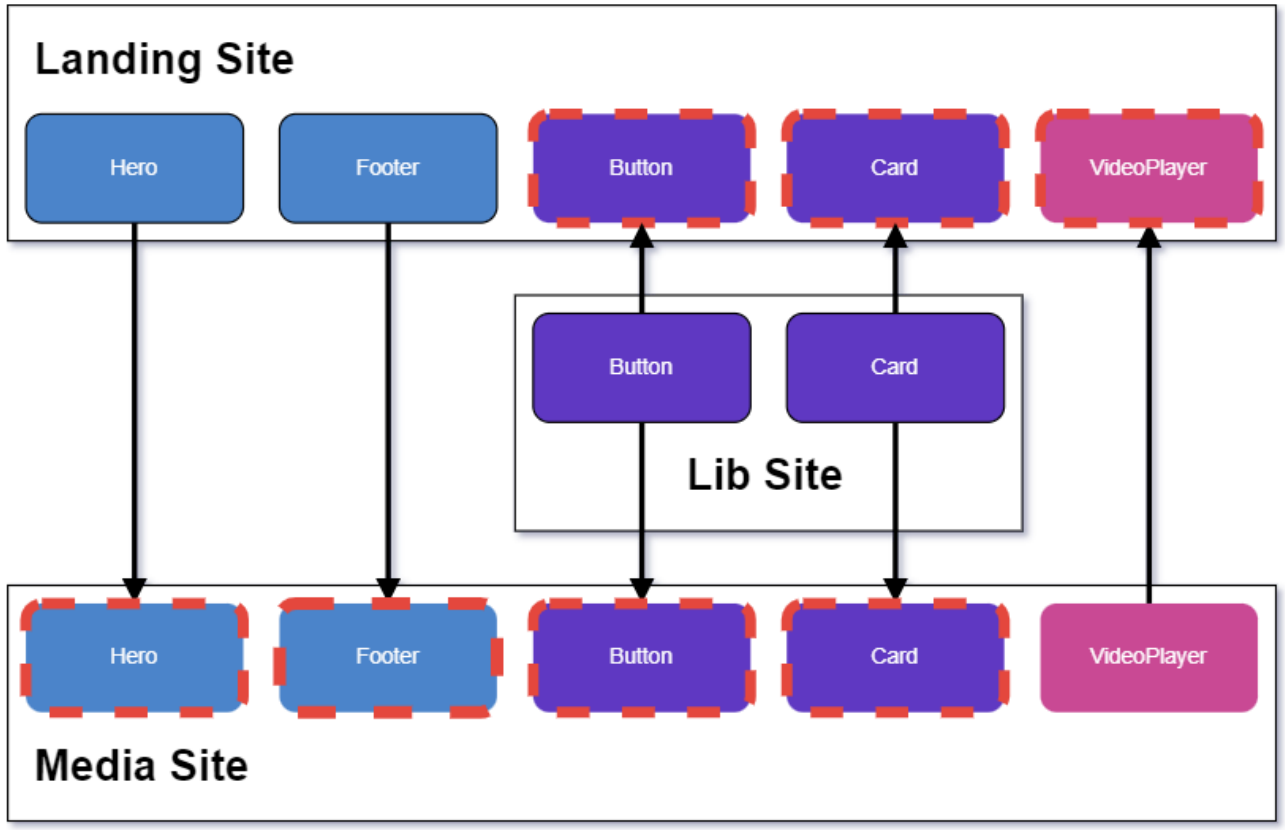
- Pros
  - Use several Javascript frameworks in parallel
  - Great user experience
- Cons
  - Javascript should be enabled

Webpack 5 Module Federation

- **Module federation**: same idea as Apollo Federation GraphQL federation — but applied to JavaScript modules. In the browser and in node.js.
- https://module-federation.github.io/



Webpack 5 Module Federation

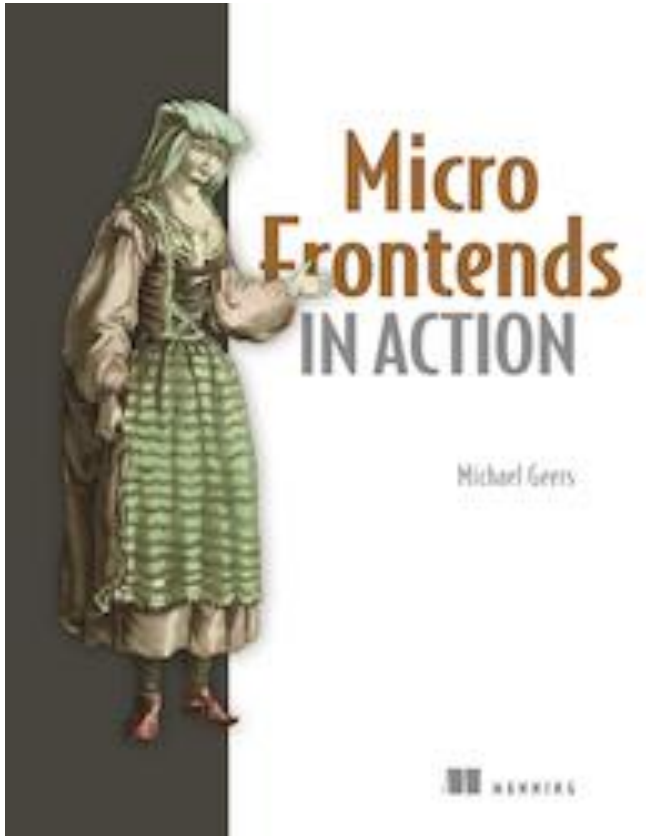


Teams can consume components at runtime instead of as part of their build pipeline

Webpack 5 Module Federation

- Pros
  - Easy to share code
  - Framework agnostic
  - Great user experience
- Cons
  - Complexity
  - You cannot mix frameworks(web components as a workaround)
  - Works only for newer frameworks

Looking for code?



- https://github.com/tpeczek/Demo.AspNetCore.MicroFrontendsInAction

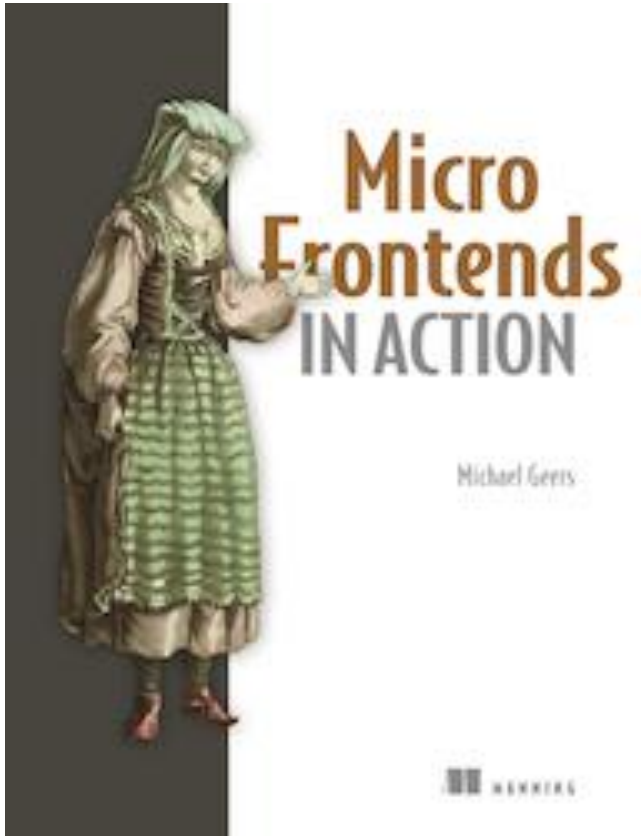
- **Tomasz Pęczek**
- https://www.tpeczek.com/2022/07/micro-frontends-in-action-with-aspnet.html

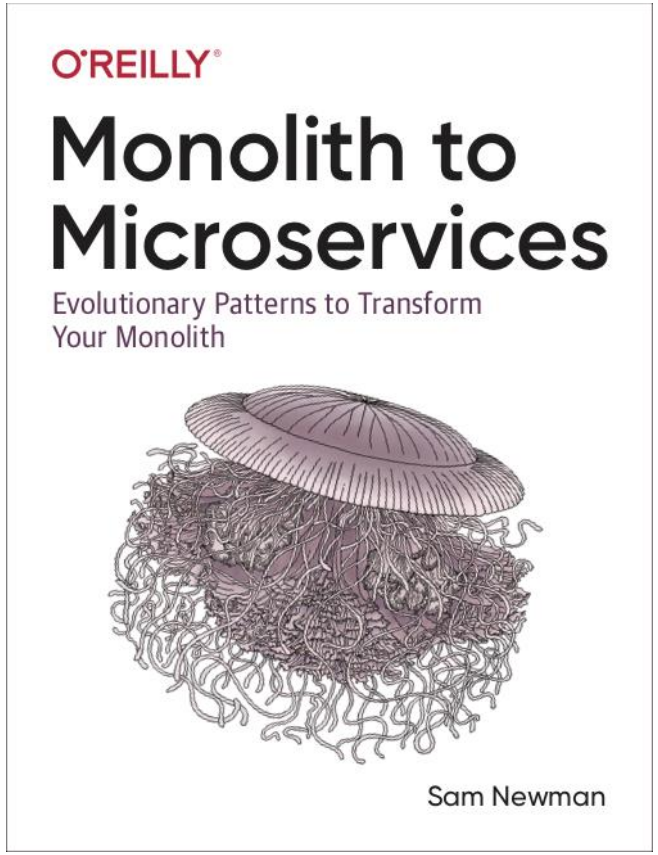
Summary

Summary

- 2 approaches to handle the “last mile” problem
- Approach 1 – Frontend Monolith
  - Direct client-to-microservice communication
  - Api Gateway(Routing, Offloading, Aggregation)
  - Backend for Frontend
- Approach 2 – Frontend Integration
  - Server-side template composition
  - Build-time integration(don’t do this)
  - Run-time integration

Additional resources





Additional resources

- Chris Richardson. Pattern: API Gateway / Backend for Front-End https://microservices.io/patterns/apigateway.html
- API Gateway pattern https://docs.microsoft.com/azure/architecture/microservices/gateway
- Aggregation and composition pattern https://microservices.io/patterns/data/api-composition.html
- Azure API Management https://azure.microsoft.com/services/api-management/
- Udi Dahan. Service Oriented Composition https://udidahan.com/2014/07/30/service-oriented-composition-with-video/
- Clemens Vasters. Messaging and Microservices at GOTO 2016 (video) https://www.youtube.com/watch?v=rXi5CLjIQ9k
- API Gateway in a Nutshell (ASP.NET Core API Gateway Tutorial Series) https://www.pogsdotnet.com/2018/08/api-gateway-in-nutshell.html

Additional resources

- Micro Frontends (Martin Fowler's blog) https://martinfowler.com/articles/micro-frontends.html
- Micro Frontends (Michael Geers site) https://micro-frontends.org/
- Composite UI using ASP.NET (Particular's Workshop) https://github.com/Particular/Workshop/tree/master/demos/asp-net-core
- Ruben Oostinga. The Monolithic Frontend in the Microservices Architecture https://xebia.com/blog/the-monolithic-frontend-in-the-microservices-architecture/
- Mauro Servienti. The secret of better UI composition https://particular.net/blog/secret-of-better-ui-composition
- Viktor Farcic. Including Front-End Web Components Into Microservices https://technologyconversations.com/2015/08/09/including-front-end-web-components-intomicroservices/
- Managing Frontend in the Microservices Architecture https://allegro.tech/2016/03/Managing-Frontend-in-the-microservices-architecture.html

Thank you!


codit


involved


Capgemini


Xpirit


team talent


ORDINA


AllPhi
Enabling the next generation software developers


AXXES


Microsoft