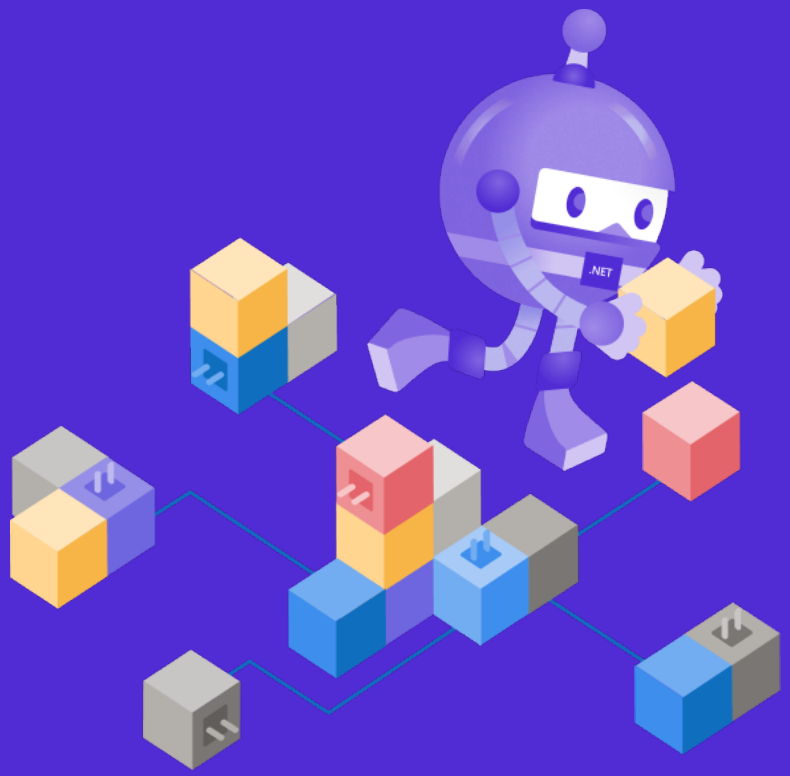
Microservice Communication

Rob Vettor



Microservices Communication

A large challenge moving to a microservice-based architecture is <u>*communication*</u>

We'll discuss...

Front-End
Client Communication

Backend
Service Communication

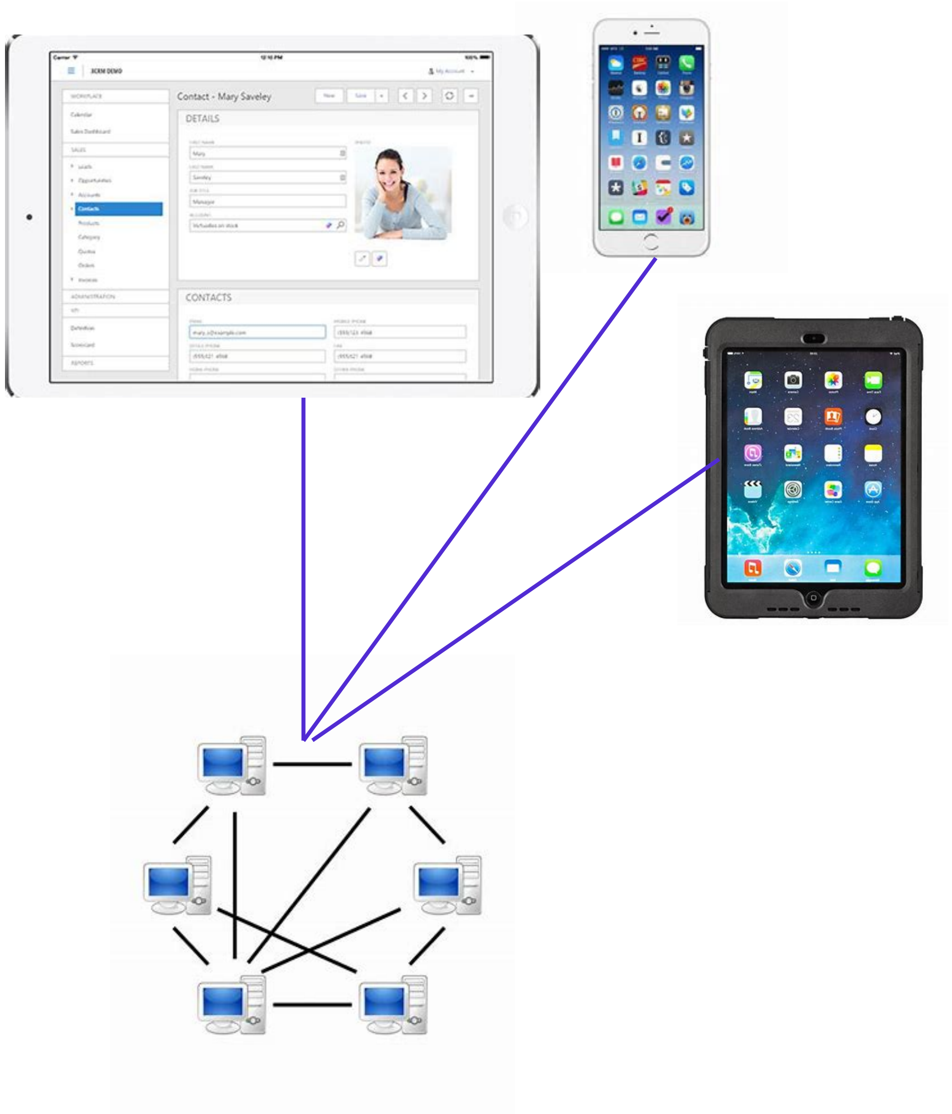
Front-end Client Communication

Front-End Clients

- Distributed applications typically have *decoupled* front-end clients

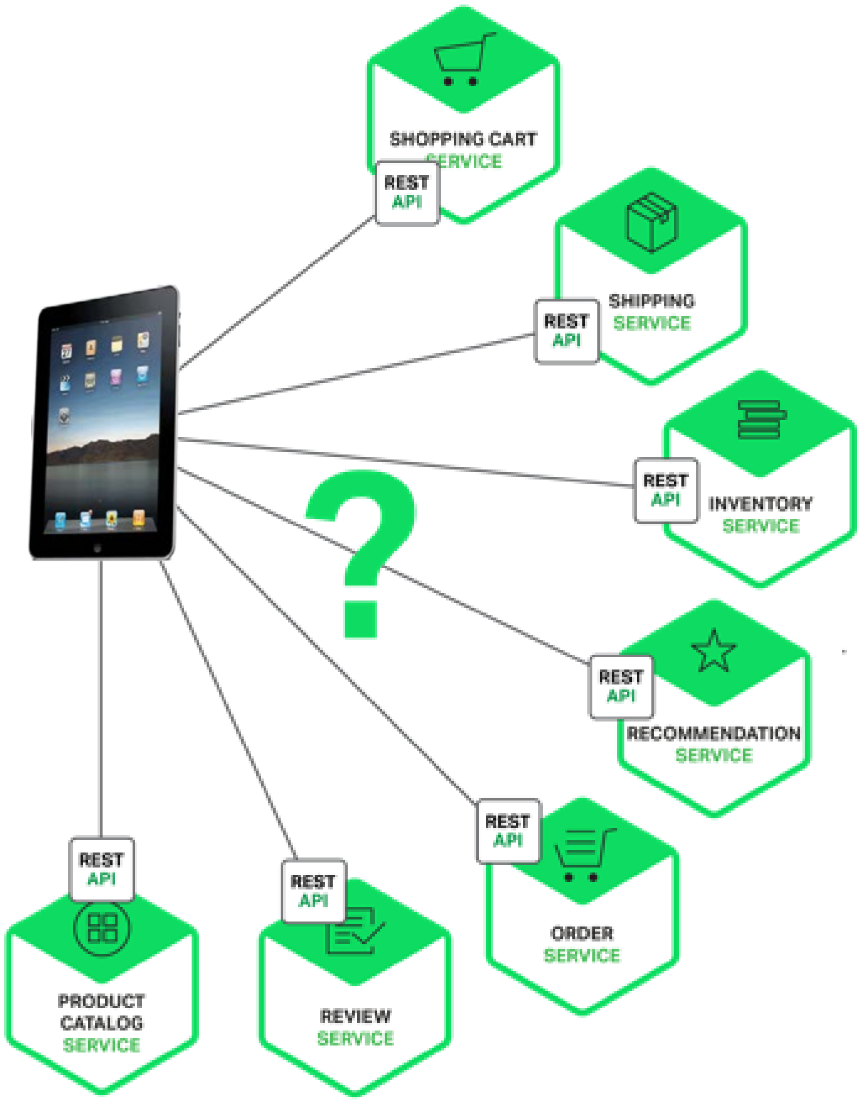
- What communication patterns are in play?
  - Direct client communication
  - API Gateway pattern

- Both commonly implemented with REST (HTTP), but gRPC is gaining popularity



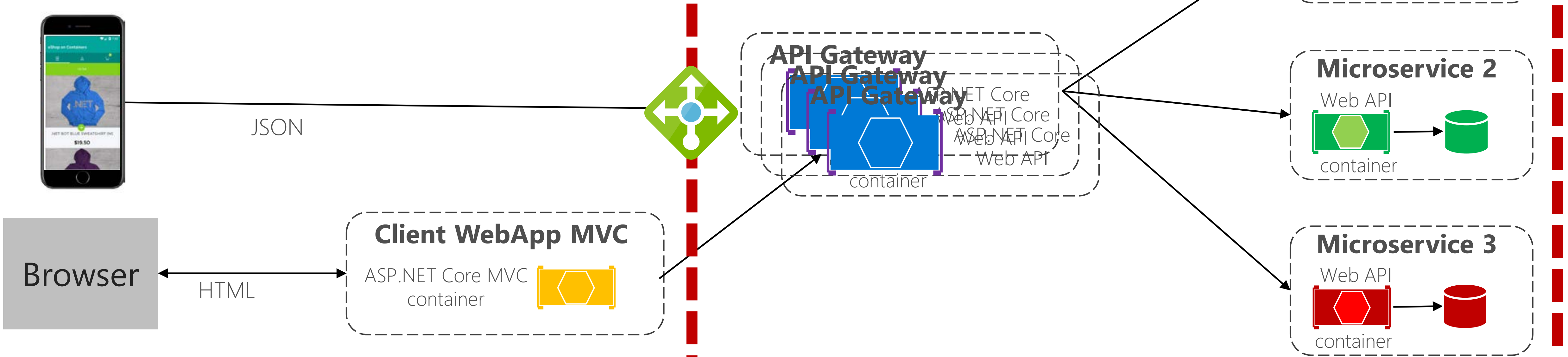
Direct Client Communication

- Client communicates directly with each service
- Simple, but far from ideal…
  - Client is tightly-coupled to backend core services
  - Client susceptible to service refactoring/partitioning
  - Client can become chatty – orchestrating multiple calls
  - Client becomes complex – often contains business logic
  - Each service must support full set of cross-cutting concerns
- Security concerns: Direct access exposes backend services and widens attack surface



API Gateway Pattern

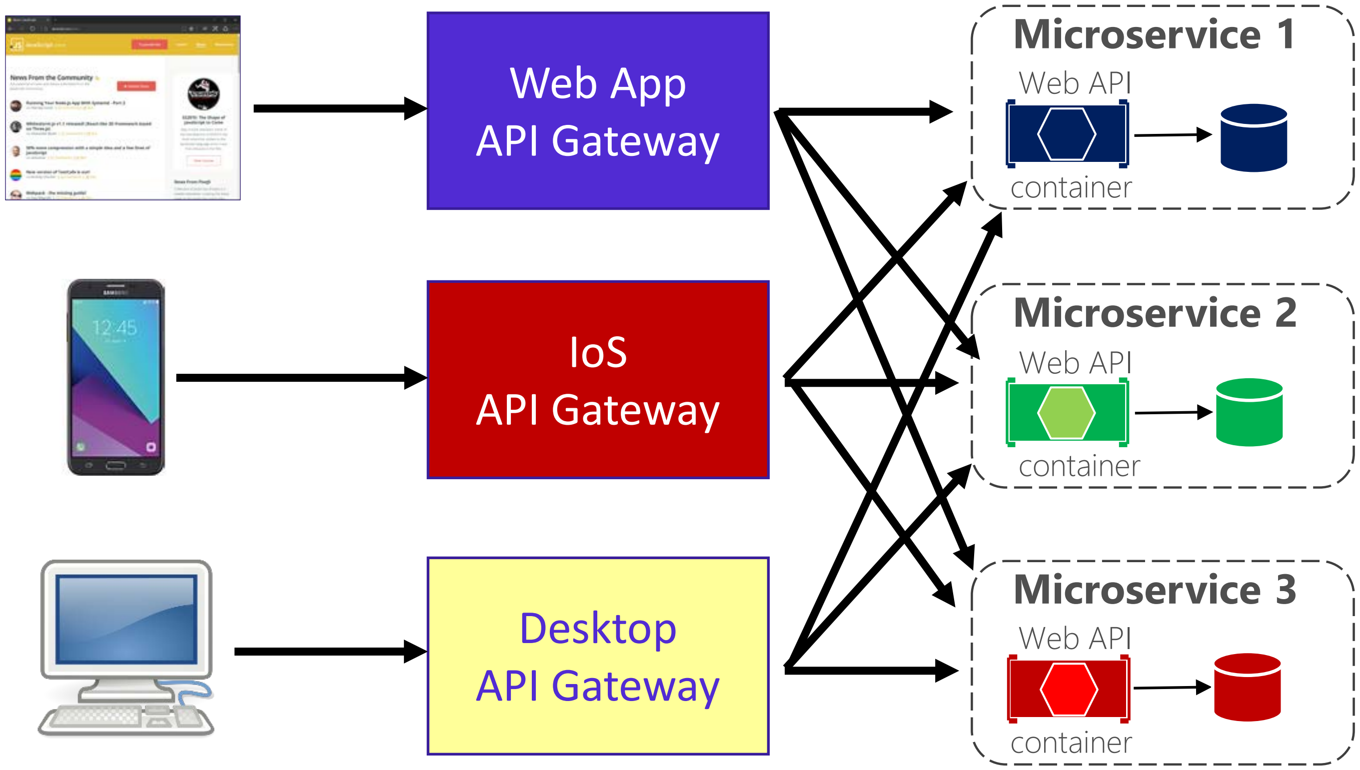
- Front-end service that encapsulates core backend services
  - Exposes single point of entry
  - Routes requests to backend services
  - Insulates client from internal partitioning and refactoring
  - Can fan requests across multiple backend services



- Keep gateway close to backend services to reduce latency
- Ensure availability to prevent single point of failure

Backends for Frontends Pattern

- Beware of *"overly-ambitious" gateways – especially in large systems*
- Consider multiple gateways
  - Separate by user experience
  - Separate by service category

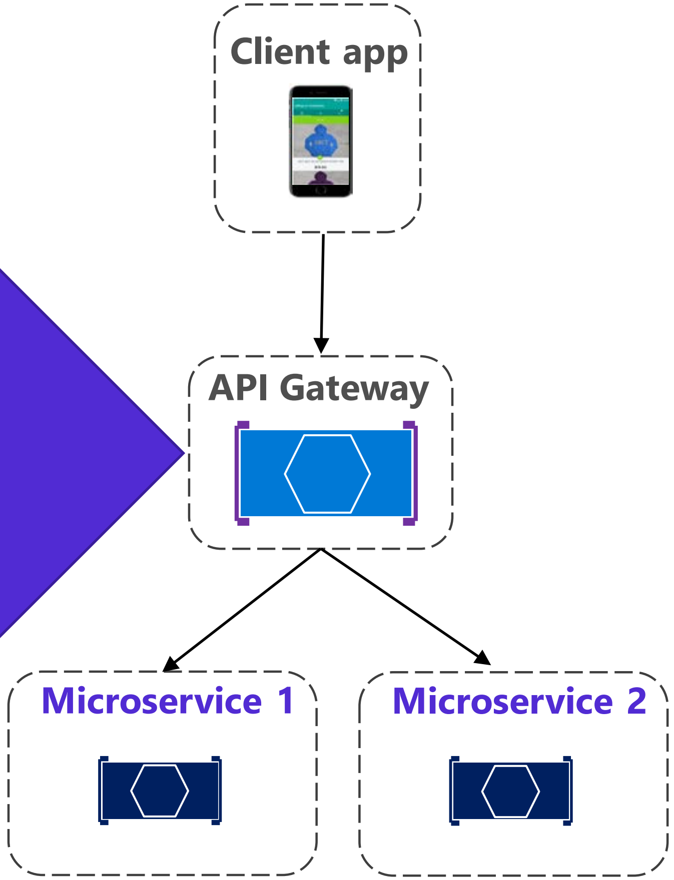


API Gateway Offloading Pattern

- Offload some cross-cutting concerns from backend services
  - Centralize in the gateway
  - Simplify backend services

<u>Centralize Cross-Cutting Concerns…</u>

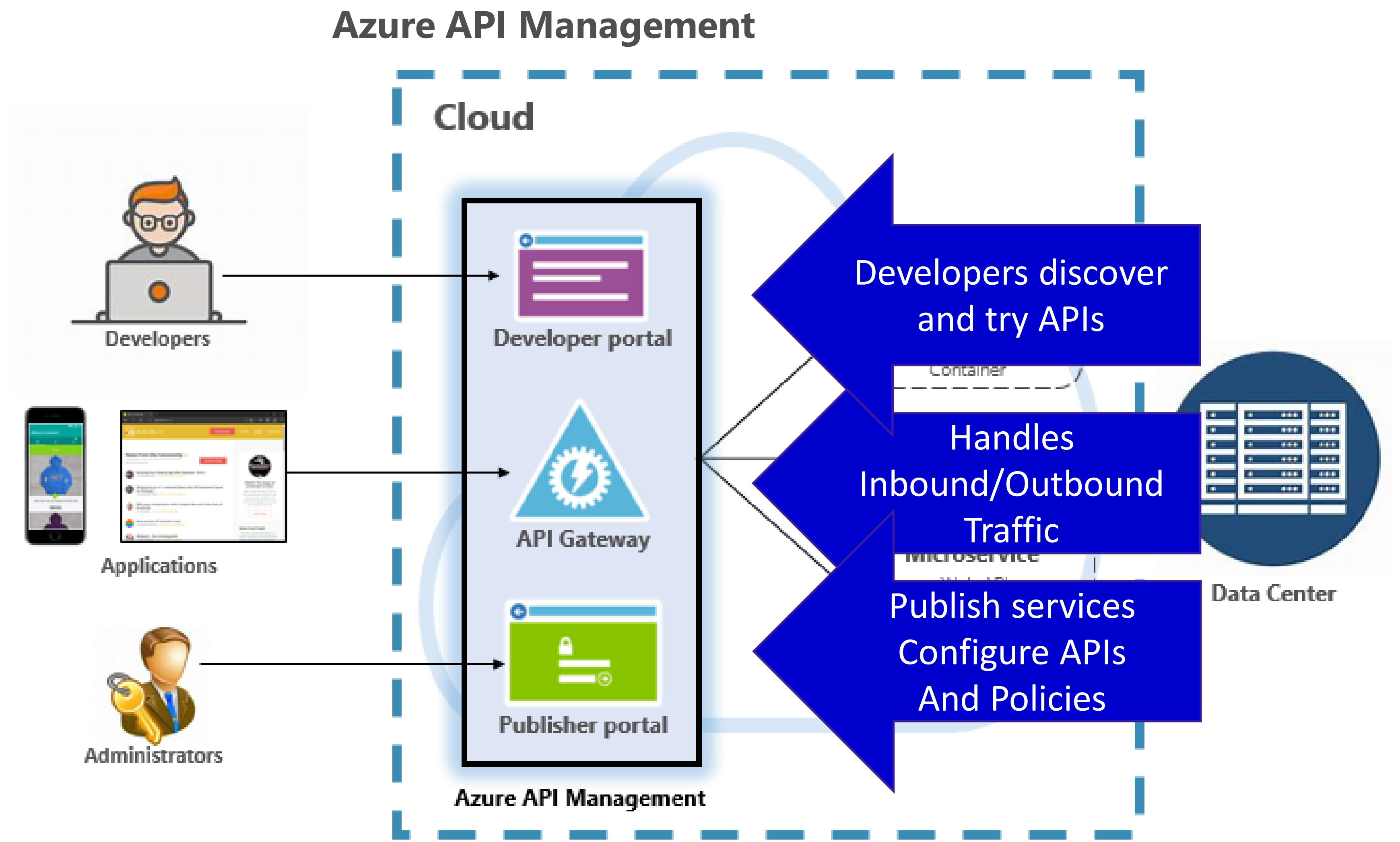
- Service discovery
- Correlation
- Response caching
- Resiliency logic
- Metering
- Throttling
- SSL termination
- Protocol translation



Demo: API Gateway

Azure API Management

- Managed service from Azure – Gateway as a Service



Azure API Management

- Can expose services from Azure, on-prem, and other public clouds
- Developers can apply *policies* to each endpoint to affect behavior…
  - Pre-built functionality can execute for each request
  - Applied to inbound calls, outbound calls, or invoked on error
- Policies include…
  - Restrict access
  - Throttle calls
  - Enable caching
  - Control the flow of a service
  - Transform data formats, such as XML to JSON
  - Custom policies

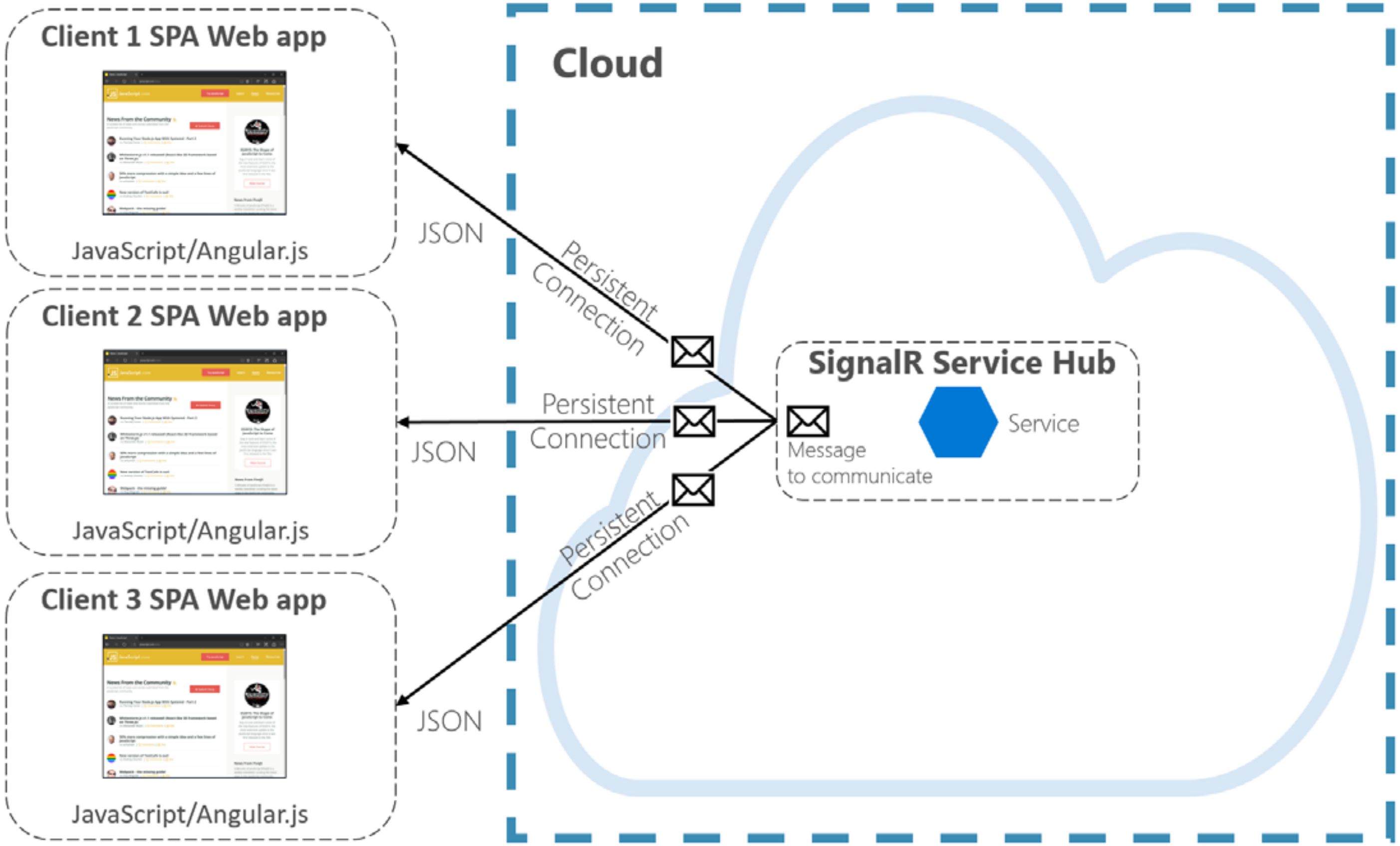
Ocelot

- Open source, .NET Core API Gateway
- Lightweight, extremely simple, scalable - provides numerous features
- Exposes functionality as a set of middleware hosted within an ASP.NET Core
  - Captures incoming Http Request
  - Forwards it through a set of pre-defined middleware, manipulating its state
  - Creates new HttpRequestMessage and forwards to downstream services

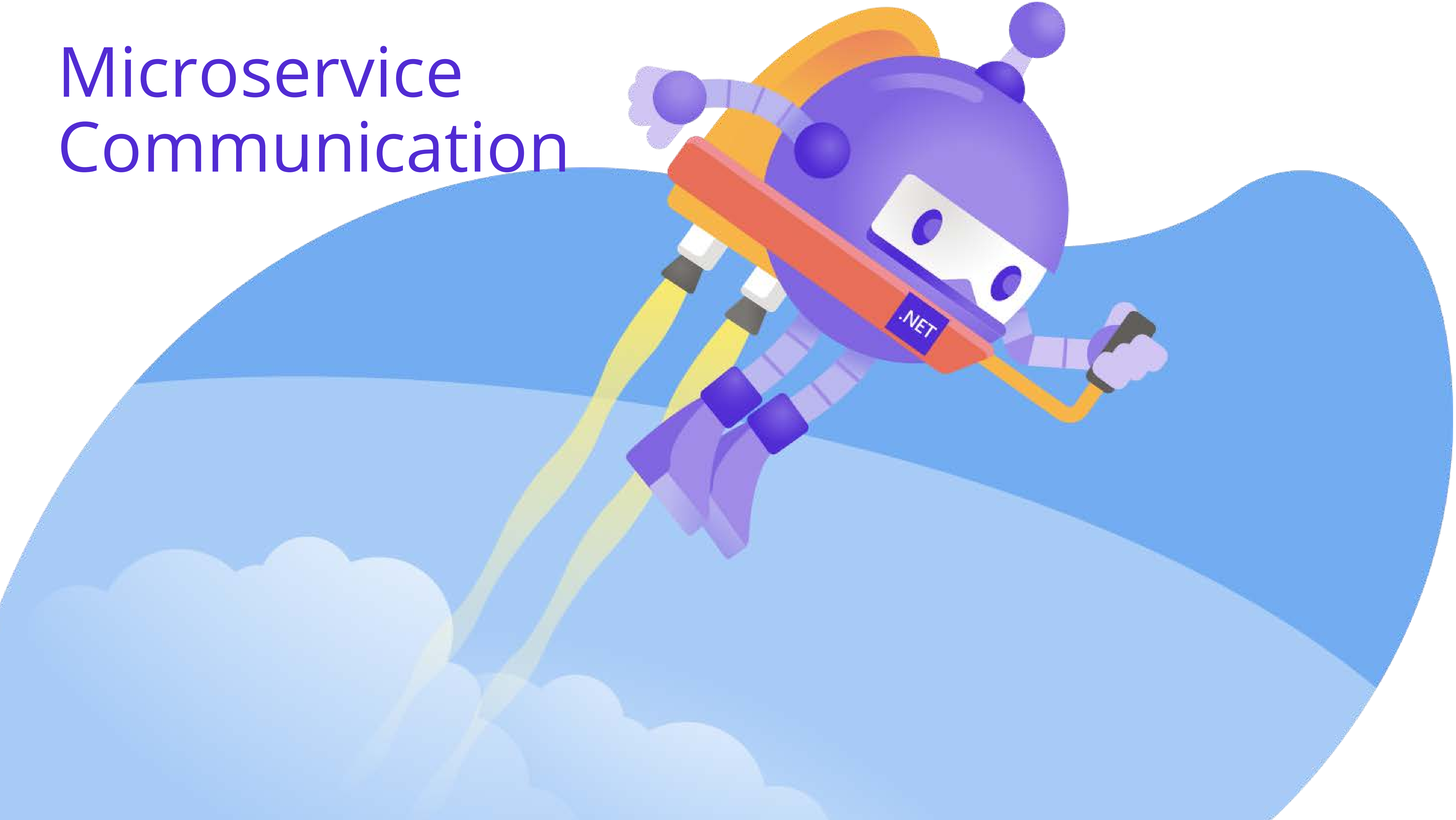
Ocelot Features	
Routing	Authentication
Request Aggregation	Authorization
Service Discovery with Consul & Eureka	Throttling
Load Balancing	Logging, Tracing
Correlation Pass-Through	Headers/Query String Transformation
Quality of Service	Custom Middleware

Real-time communication

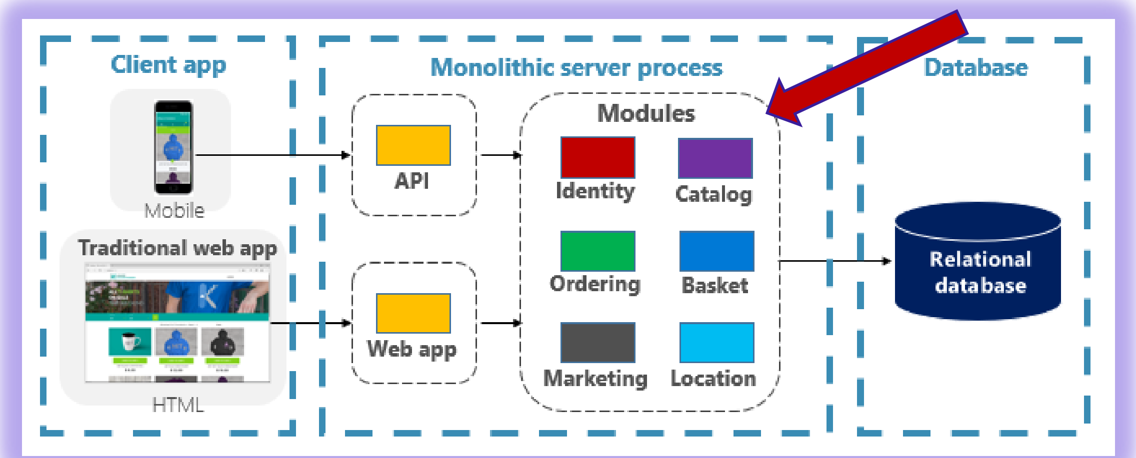
- *Azure SignalR Service* fully managed Azure service that simplifies real-time communication – abstracts the complex plumbing
- Register your client and server can *push data directly* to it
- Clients do not need to poll the server for updates



Microservice Communication



Monolithic Communication

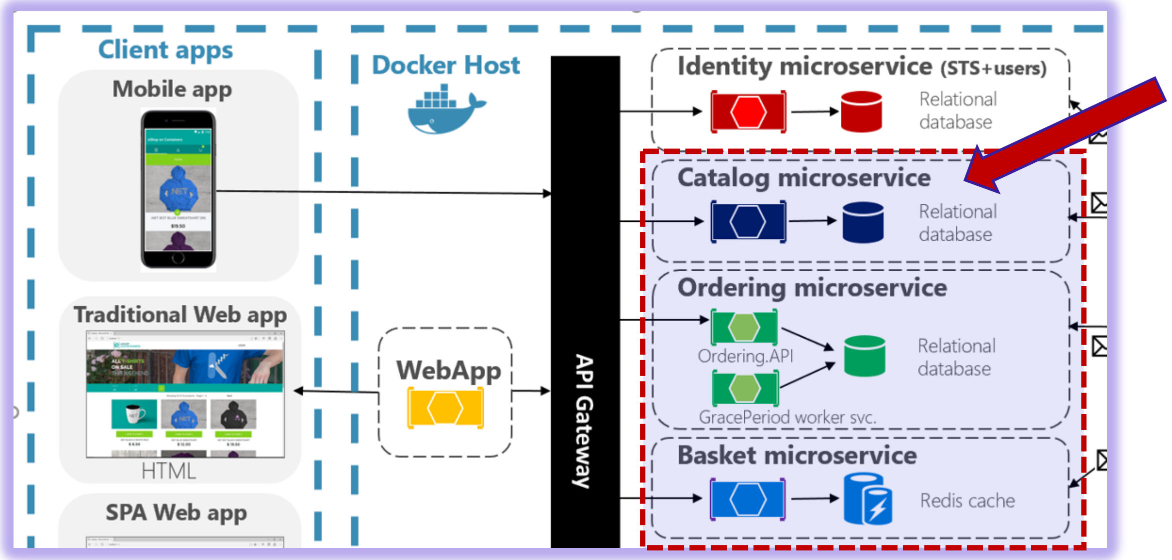



```
// One can directly call another
var CatalogComponent = new
CatalogComponent();
var result =
CatalogCommponent.GetItem(arg1);
```

- For a monolithic app…
  - Communication is straightforward
  - Code modules execute together in server process
  - Can be fast, but results in tightly-coupled code – which can be expensive to maintain, evolve, and scale

- What happens if you *transform* the in-process components to microservices?

Cloud-Native App



- Life dramatically changes…
  - Calls are now *out-of-process* and communicate across a *network*
    - In-process methods transform to service *endpoints*
    - Each exposes language-agnostic, multi-version *contract*
    - Each must serialize/deserialize arguments/payload ($$$ in memory and CPU)
  - Manage network latency, partitions, transient faults, and unpredictable timing
  - Must authenticate/authorize each service call and encrypt sensitive messages

Adopted from Jeffery Richter

How do microservices communicate with each other?

- Two approaches:

Request/Response Model (synchronous) communication

Publish/Subscribe Model (asynchronous) communication

- Depending on the *message type*…

Message Interaction Types

- Message interaction patterns include…

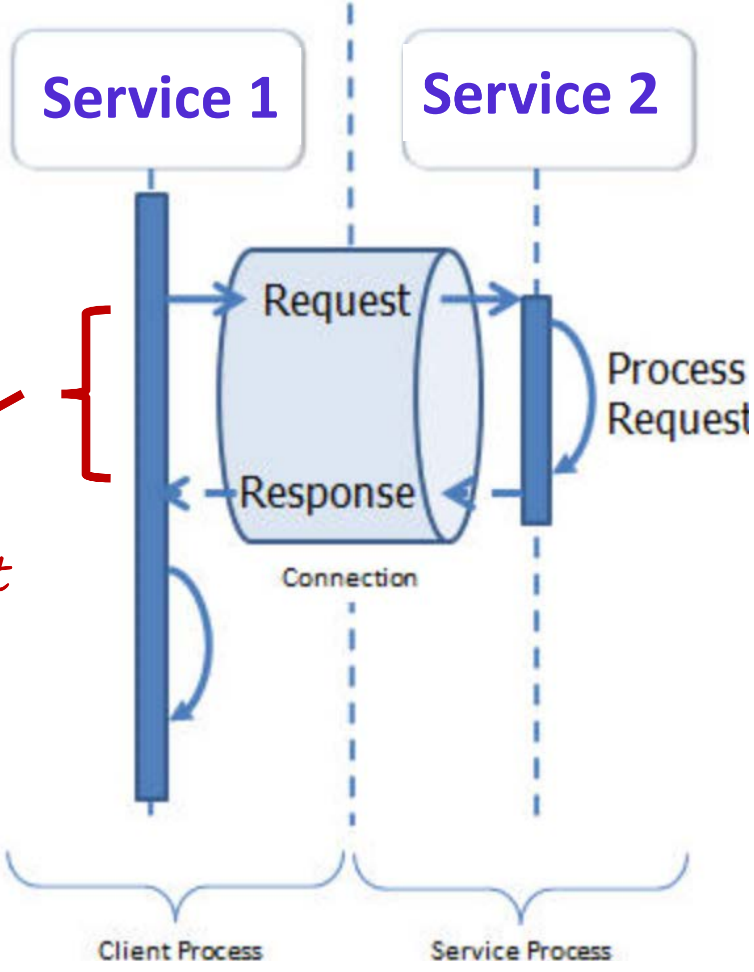
Query	Client needs response from a service
Command	Client needs a service to perform an action
Event	Service reacts to something that's happened in another service

- All three patterns are commonly implemented in cloud applications

Queries

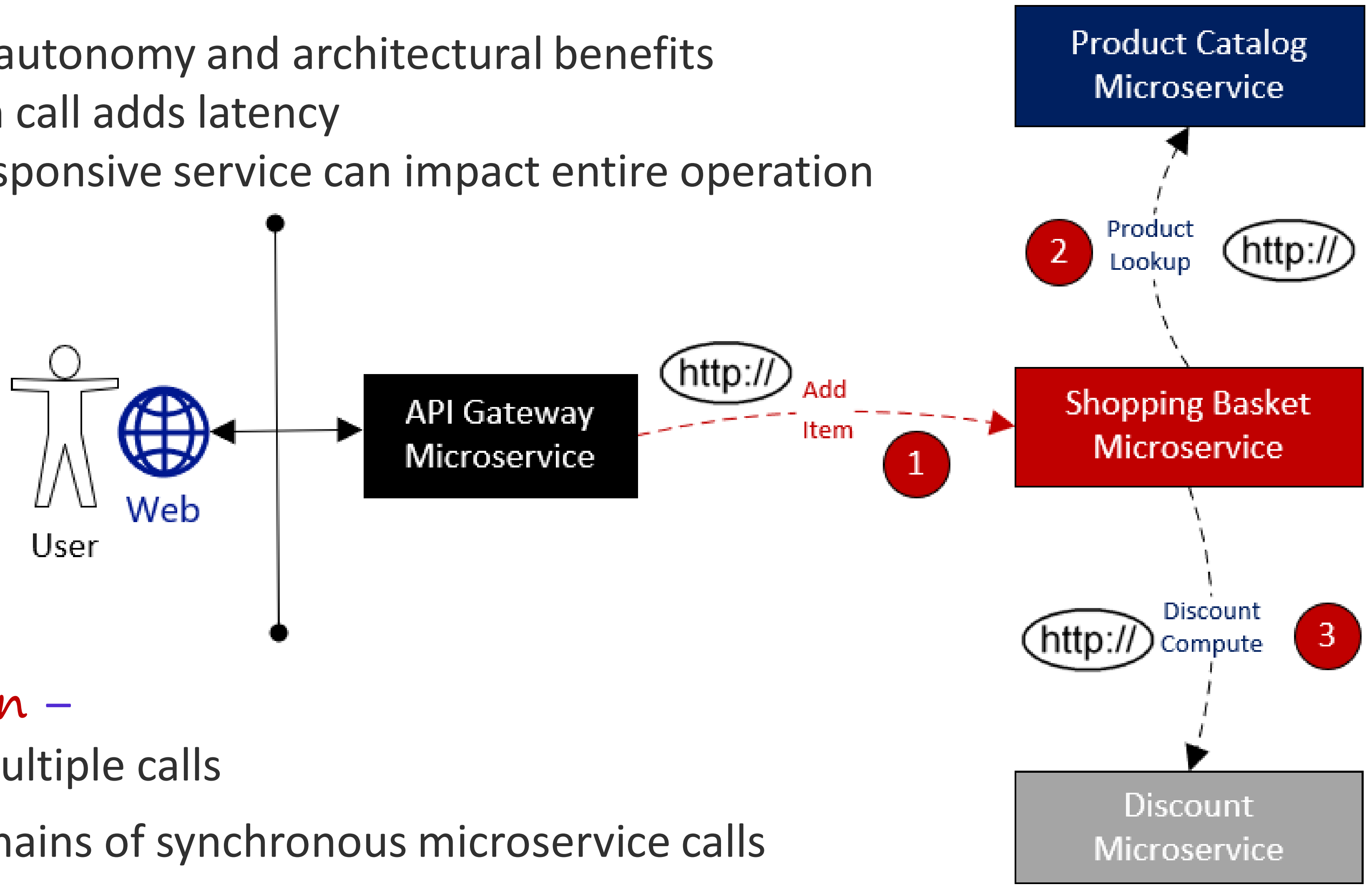
- Calling service has *dependency* upon another service for data or operation
- Needs *immediate response* to complete operation
- Simple to implement
- *Always synchronous*
- Implement async/await pattern to avoid blocking threads

Client process waits...
Cannot continue until it receives a response or timeout



Direct Synchronous Communication

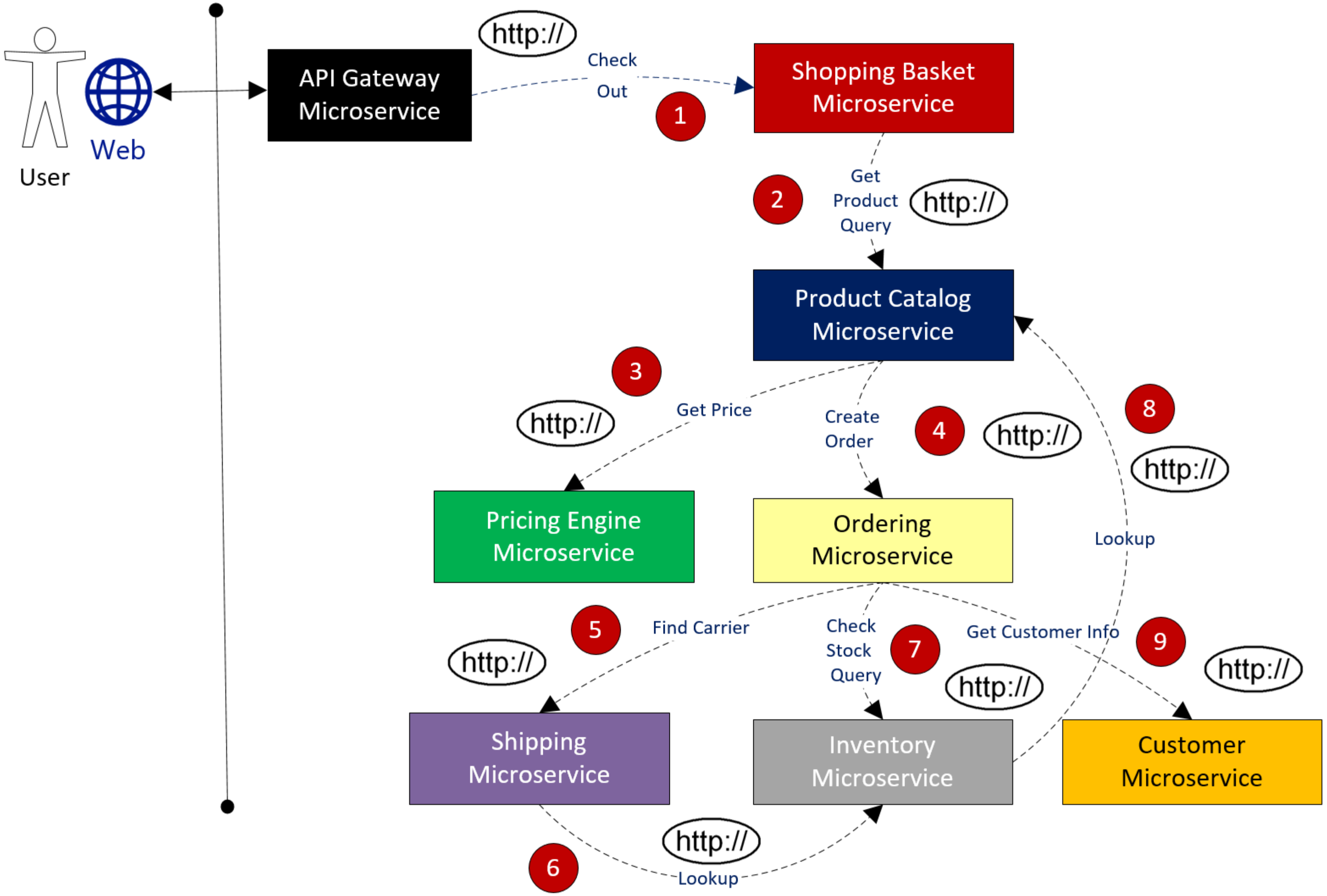
- Couples services, reducing autonomy and architectural benefits
- Impact performance – each call adds latency
- Impact reliability – an unresponsive service can impact entire operation



- *Approach with caution* – especially when invoking multiple calls
- Can lead to long complex chains of synchronous microservice calls

Chaining HTTP Calls

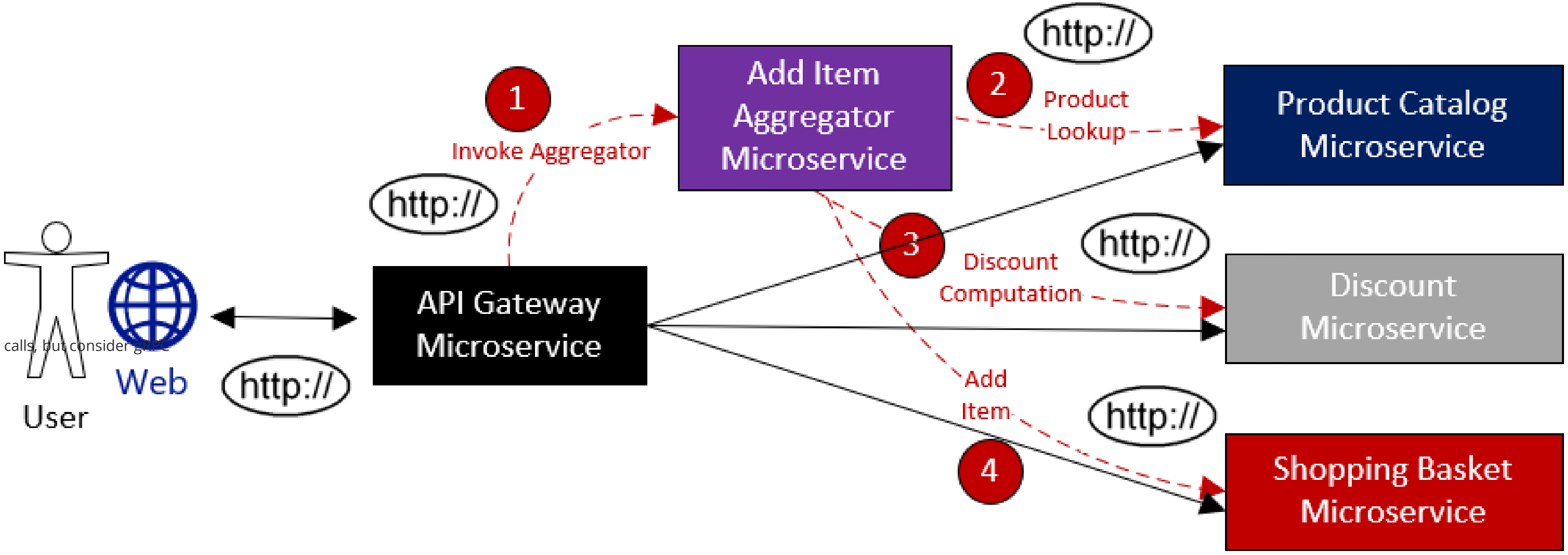
- Deep chaining or nesting of HTTP calls - simple to implement, but an antipattern



Aggregator Pattern

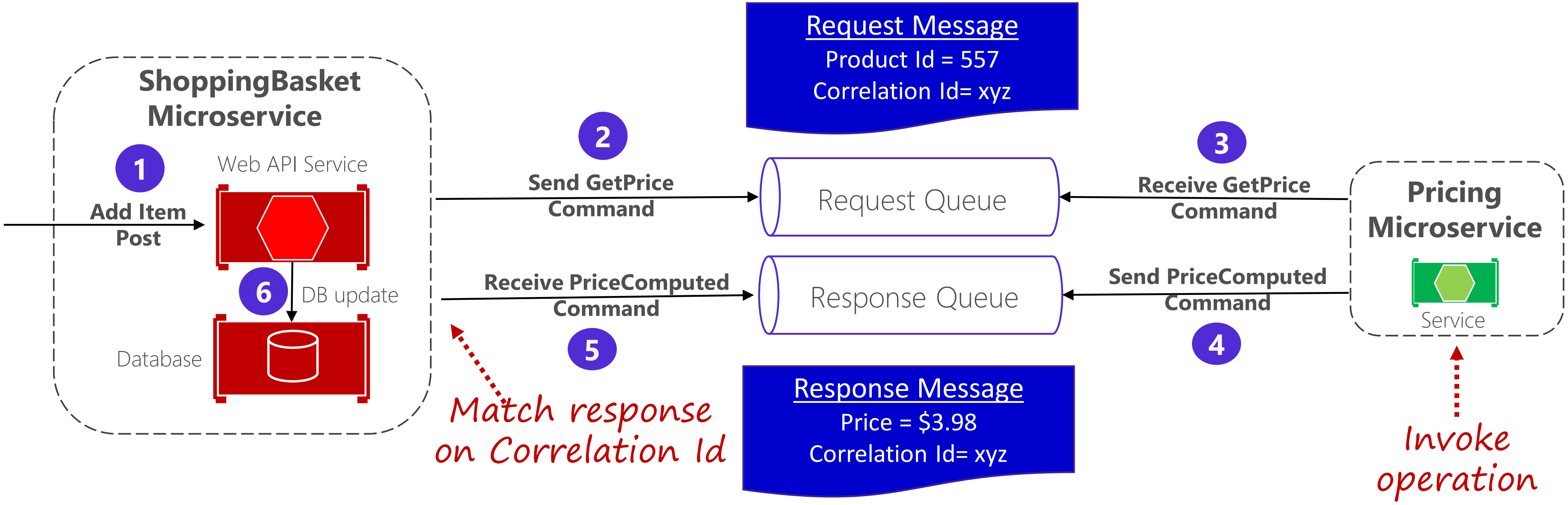
- *<u>Aggregator microservice</u>* orchestrates a business process
- Orchestrates calls across multiple backend services and aggregates data

- Centralizes an operation
- Can include business logic
- Does *<u>not "chain" calls</u>*
- Implements synchronous HTTP calls, but consider gRPC



Request-Reply Pattern

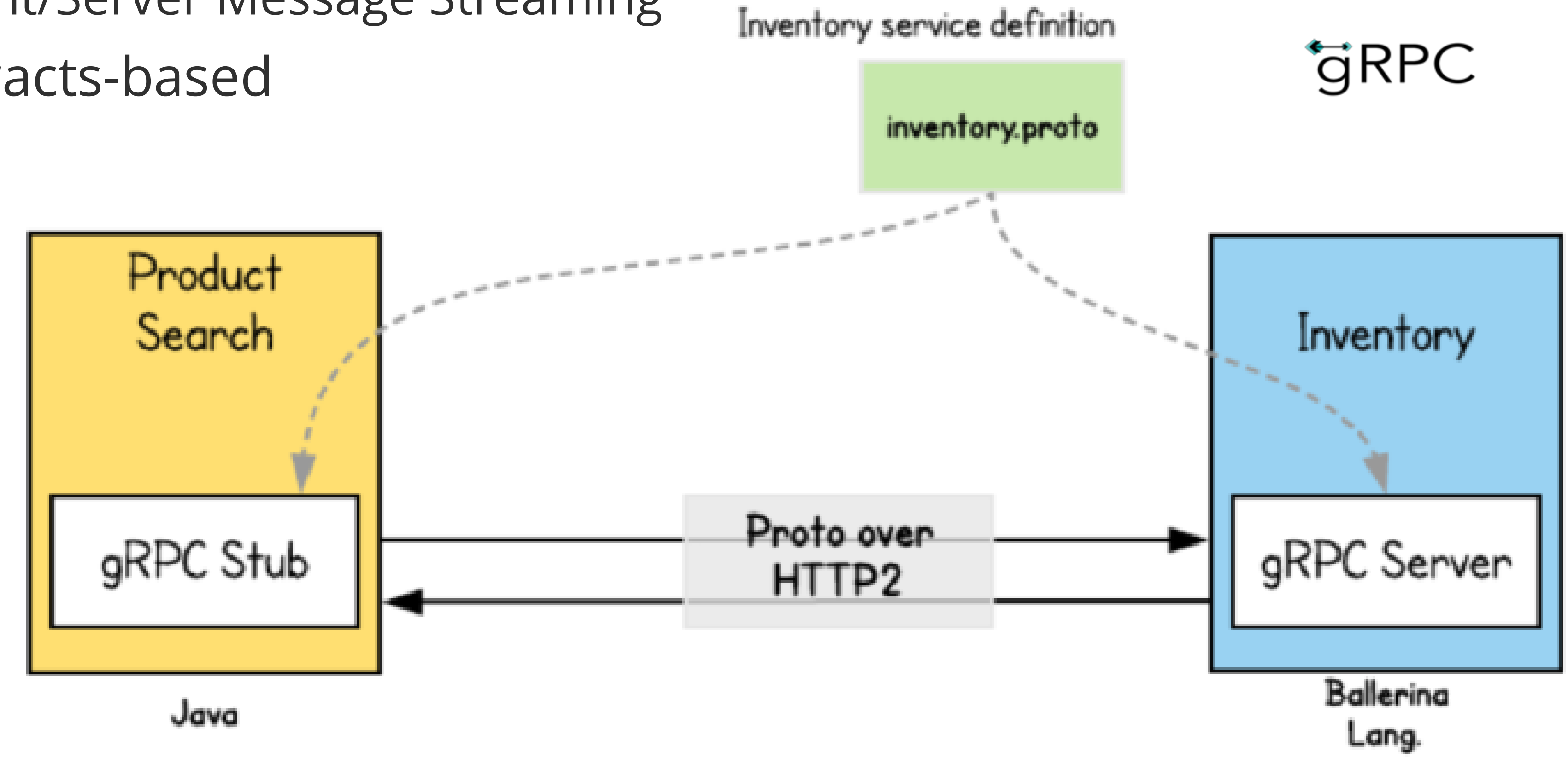
- Referred to as *Sync over Async*
  - Client communication synchronous/backend communication asynchronous



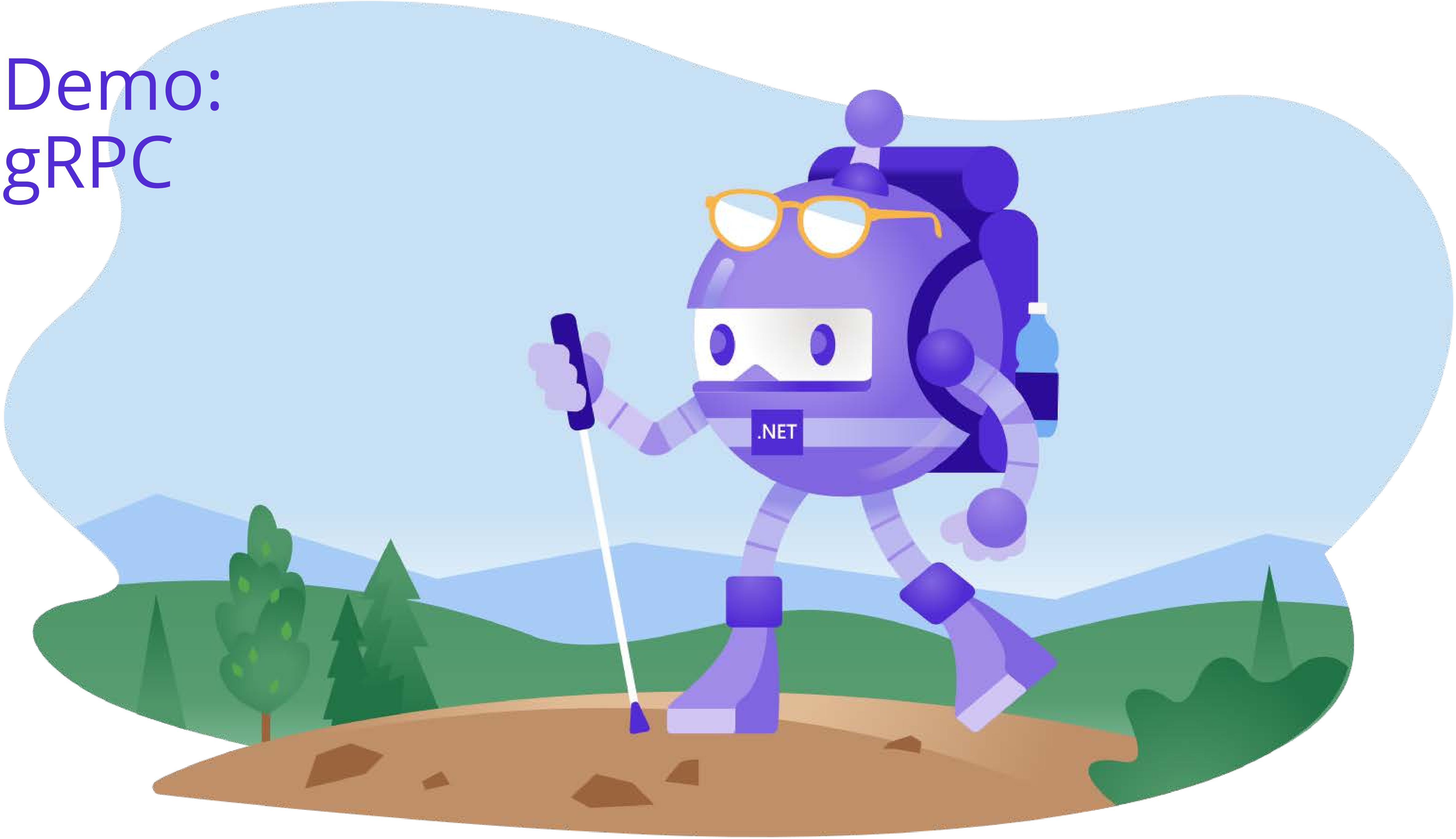
- Decouples calls among backend services
- Adds complexity and not ideal for “awaiting” UI calls
- Consider for systems that implement two-way, real-time communication (SignalR)

gRPC

- High performance, highly scalable, standards based, open source general purpose RPC Framework
  - gRPC: ‘**g**’ for Google, **RPC** for Remote Procedure Calls
  - Binary Data Representation (compact)
  - Takes advantage of the HTTP/2 feature set
  - Bi-directional Client/Server Message Streaming
- Endpoints are contracts-based



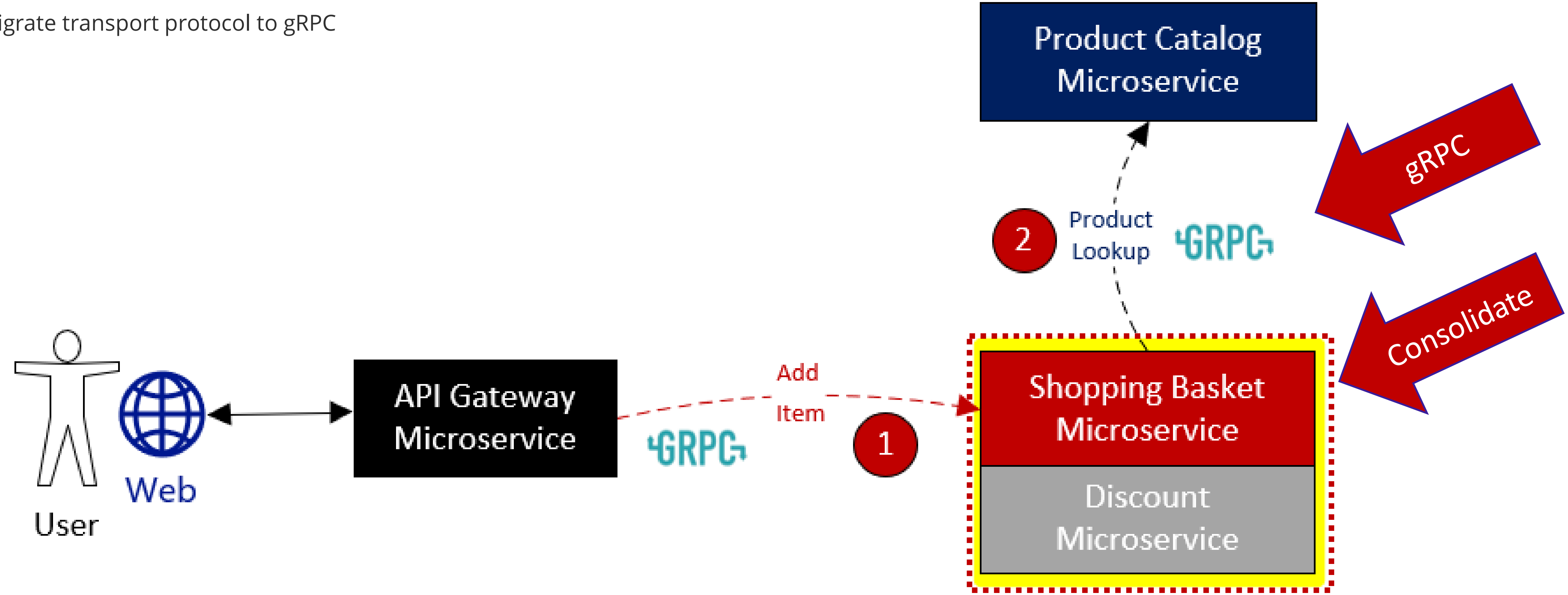
Demo: gRPC



Optimize

- Consolidate Basket and Discount services
  - Tightly integrated
  - Share data
  - High volumes of interservice messages

- Migrate transport protocol to gRPC



Microservice Communication:

Commands and Events

Message Interaction Types

- How do distributed, cloud-based apps/services communicate with each other?
- Message interaction patterns include…

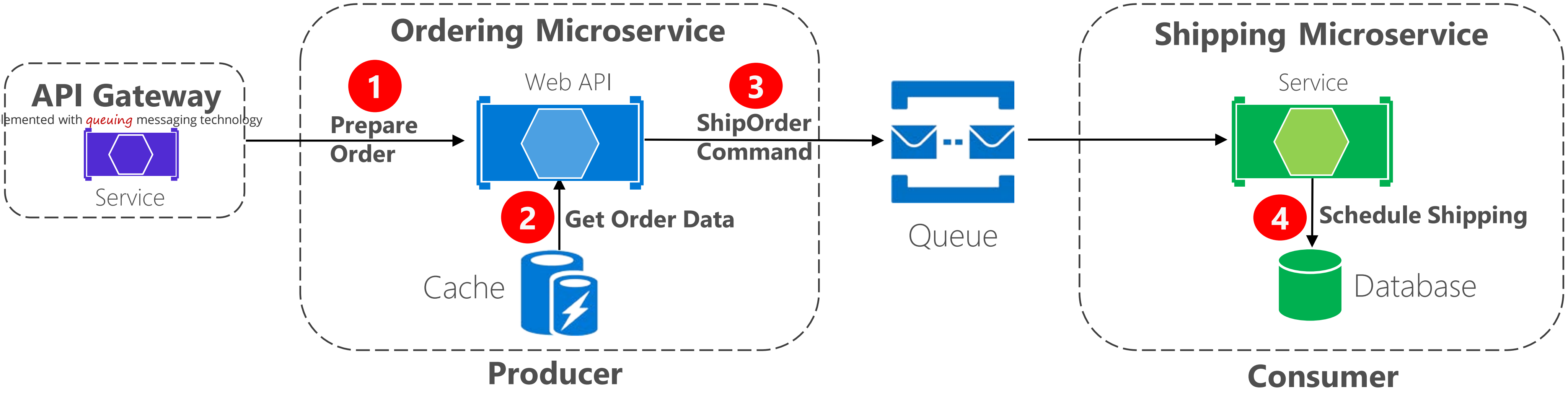
Query	Client needs response from a service
Command	Client needs a service to perform an action
Event	Service reacts to something that's happened in another service

- All three patterns are commonly implemented in cloud applications

Commands

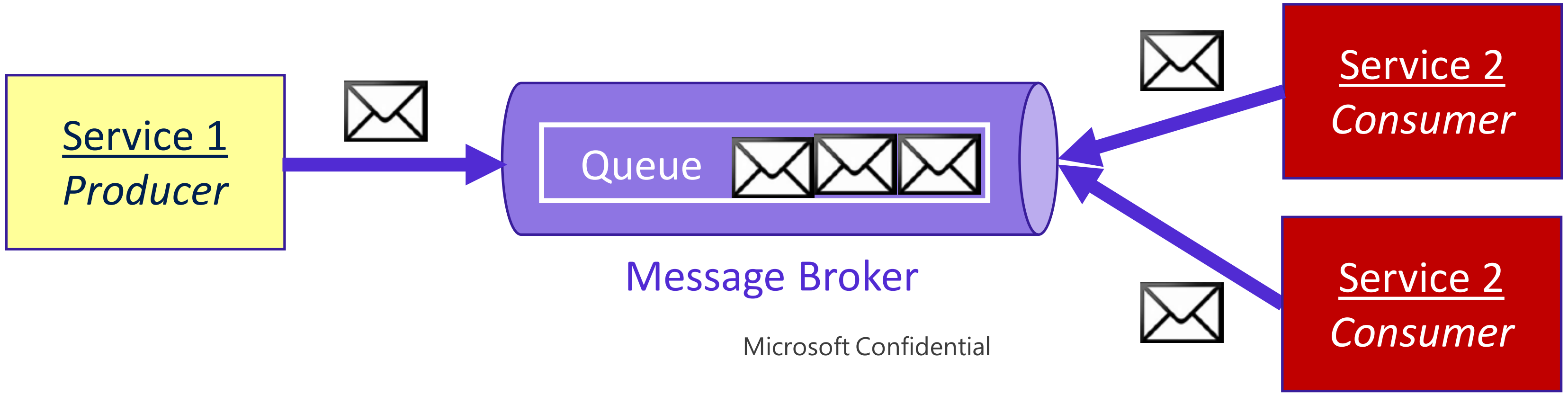
- Producer/consumer relationship
  - Producer needs consumer to perform an action
  - Producer fires message and forgets
  - Communication moves from synchronous to *asynchronous* exchange
  - Removes cross-service dependency

- Typically implemented with *queuing* messaging technology



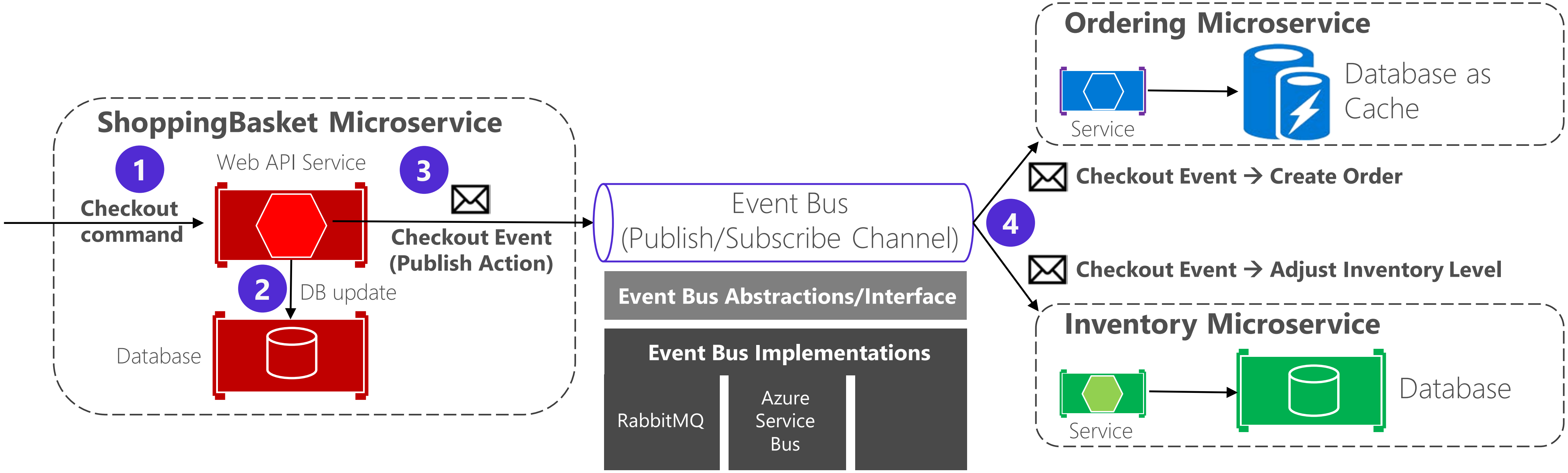
Queue

- An intermediary construct through which a producer passes a message to a consumer
- Implements a *point-to-point* messaging pattern
- Guarantees each message processed by *at least one consumer*
- Producers and consumers not aware of each other – both have dependency on queue
- Can scale-out one service without affecting the other
- Technologies can be disparate on each side
- Consumer does always not need to run – messages persisted in queue until processed by consumer



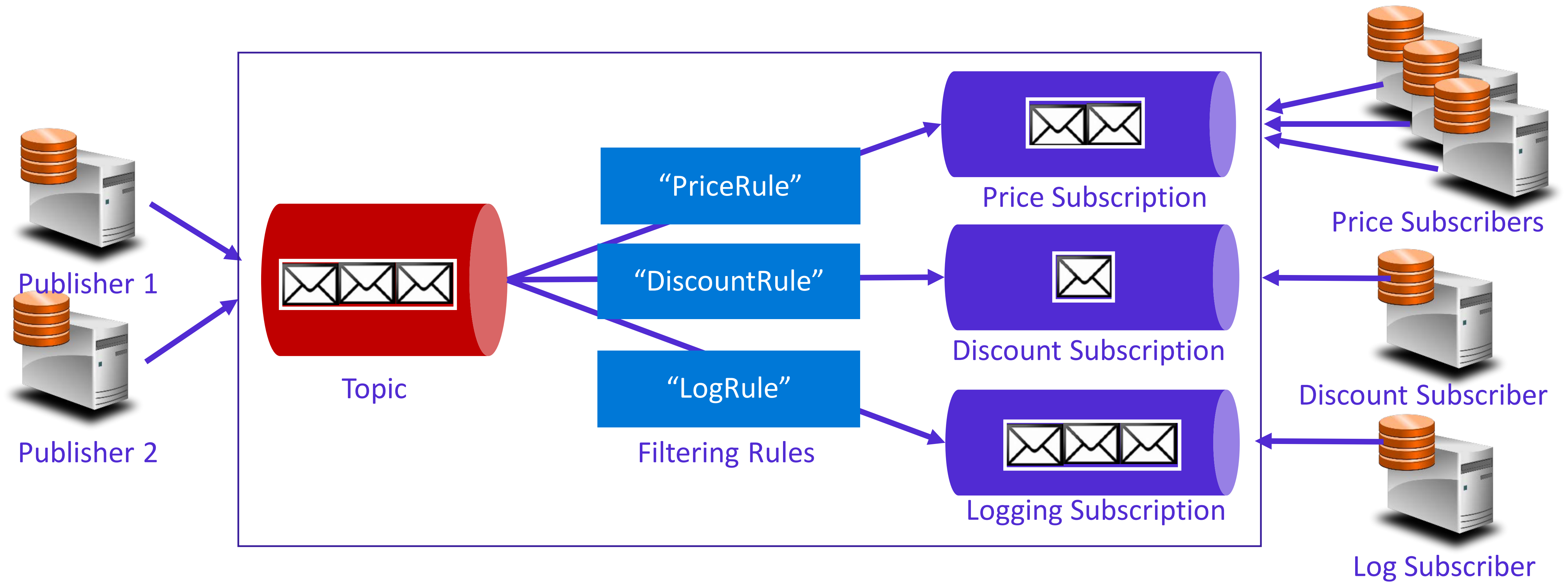
Events

- Publisher/subscriber relationship
  - Publisher raises event upon a state change
  - Subscribers can respond to event - without cross-service dependency
  - Services are unaware of each other
  - Embraces asynchronous communication exchange
- Typically implemented with *topic* messaging technology

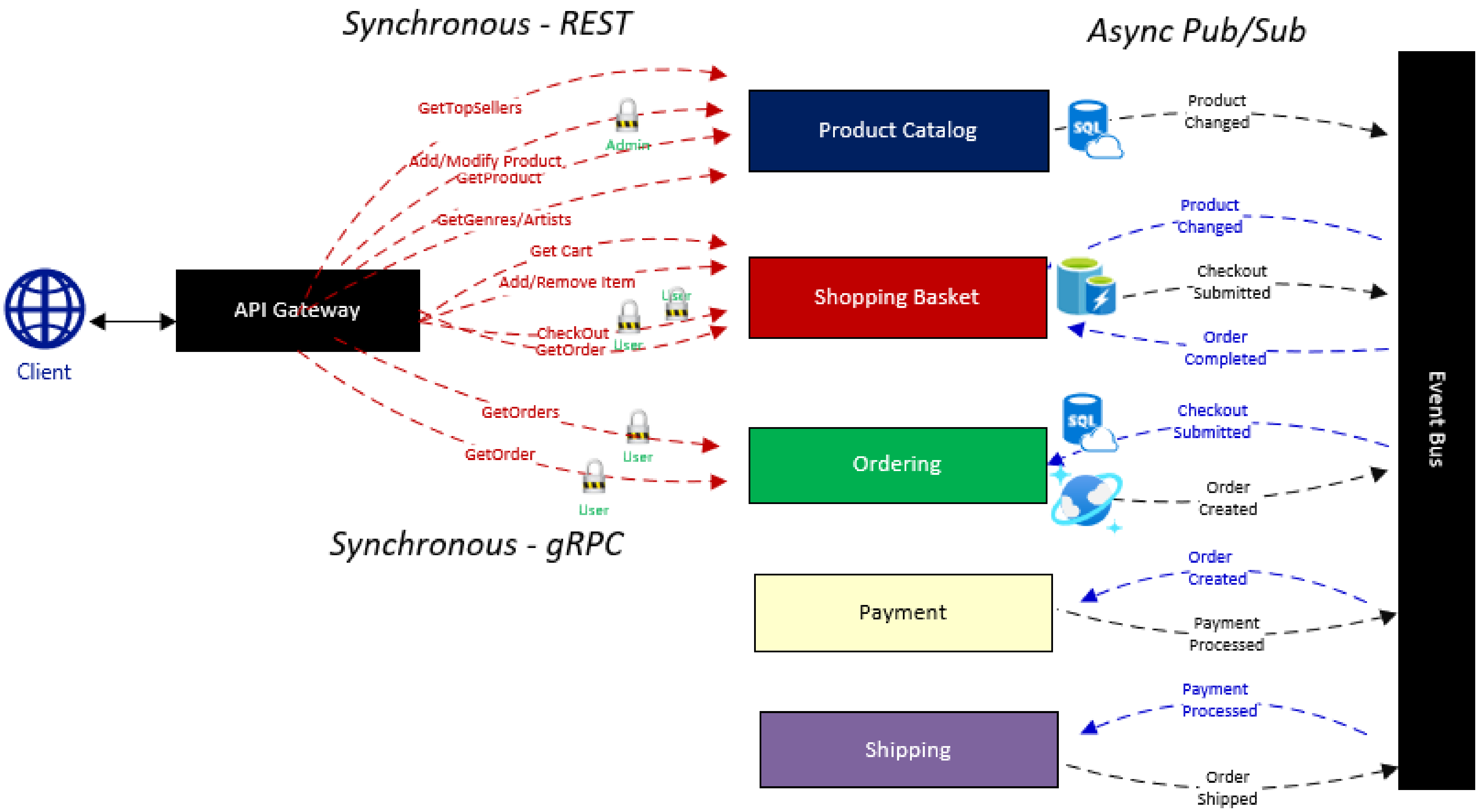


Topic

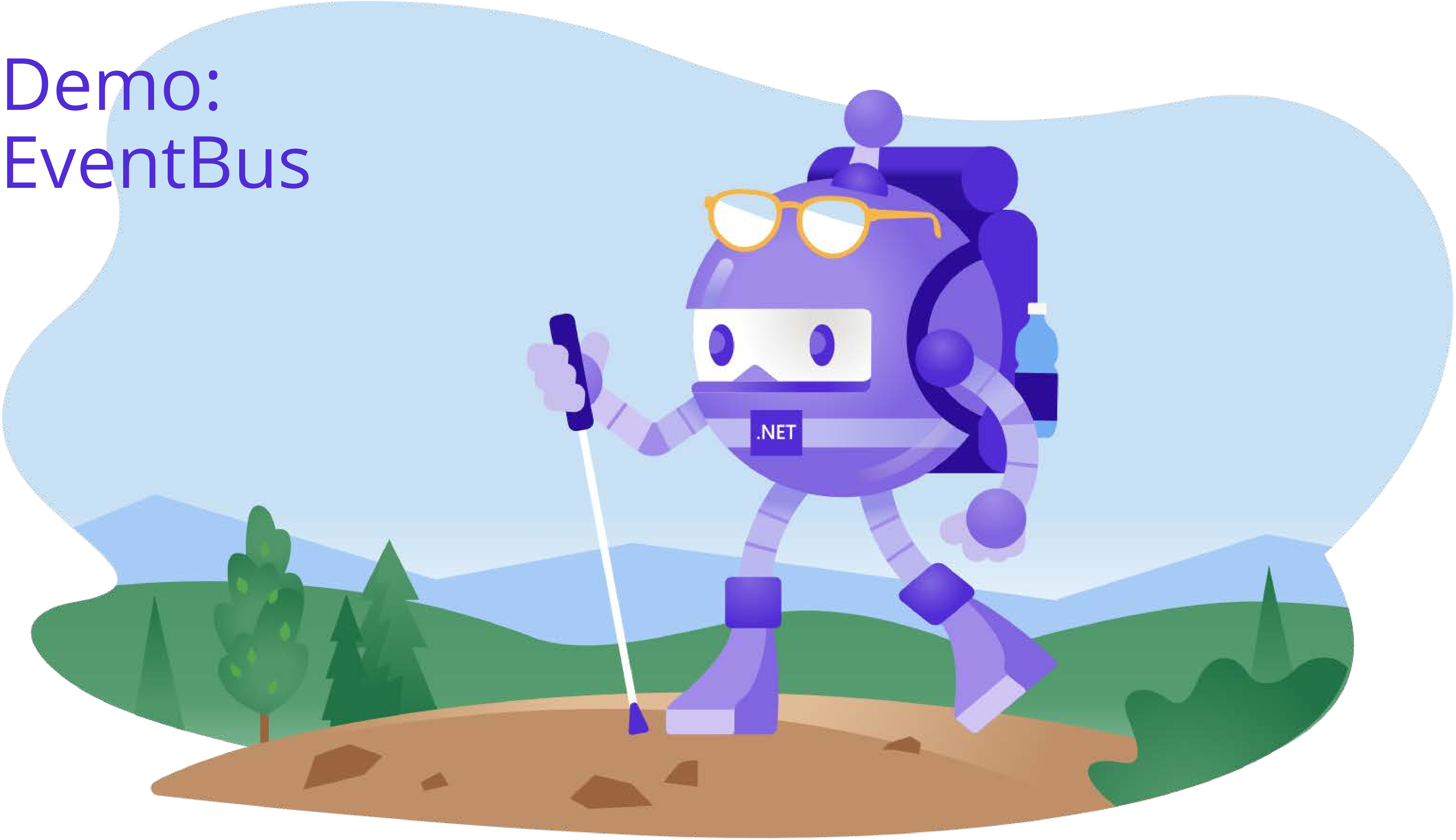
- Supports a *one-to-many* messaging pattern to multiple subscribers
  - Publisher service sends message to a topic
  - Subscribing services receive only those message types for which they register
  - Clear separation: Both have dependency on message broker



Communication in the Reference App



Demo: EventBus



Microsoft