Step by Step: .NET Microservices

Using Kubernetes, RabbitMQ & Grpc

Episode 3 – Season 4

This course will

- Focus on the Technical aspects of building .NET Microservices
- Cover theory as we go
- Take a pragmatic / balanced approach (but follow best practices)
- Be fully step by step – that’s why it’s long!
- Be as practical and interesting as possible
- Acknowledge that Microservices are hard!

This course does not…

- Focus on Domain Driven Design, Bounded Contexts etc.
- Build out redundant “business logic” – our use case is simple
- Demonize the Monolith / band-wagon jump
- Cover EVERYTHING… there was a balance to be struck

What we'll cover (& what you'll learn)

INTRO & THEORY

- Course Overview
- Pre requisites
- Ingredients / Tooling
- What are Microservices?
- Our Services
- Solution Architecture
- Service Architecture

STARTING THE PLATFORM SERVICE

- Overview
- Scaffolding
- Data Layer
- Controller and & Actions

DOCKER & KUBERNETES

- Review of Docker
- Containerize Platform Service
- Pushing to Docker Hub
- Intro to Kubernetes
- Kubernetes Architecture
- Deploy Platform Service
- External Network Access

What we'll cover (& what you'll learn)

STARTING THE COMMANDS SERVICE

- Scaffolding
- Controller & Action
- Synchronous & Asynchronous Messaging
- Adding a HTTP Client
- Deploy service to Kubernetes
- Internal Networking
- API Gateway

SQL SERVER

- Persistent Volume Claims
- Kubernetes Secrets
- Deploy SQL Server to Kubernetes
- Revisit Platform Service

MULTI-RESOURCE API

- Review of Endpoints for Commands Service
- Data Layer
- Controllers & Actions

What we'll cover (& what you'll learn)

MESSAGE BUS / RABBITMQ

- Solution Architecture Review
- RabbitMQ Overview
- Deploy RabbitMQ to Kubernetes
- Test

ASYNCHRONOUS MESSAGEING

- Adding a Message Bus Publisher to Platform Service
- Event Processing
- Adding an Event Listener to the Commands Service

GRPC

- Overview of GRPC
- Final Kubernetes networking
- Adding gRPC server to Platforms service
- Creating a "proto" file
- Adding a gRPC client to the Commands service
- Deploy & Test

Prerequisites

We will do everything step by step here, but:

- Experience with building .NET (Core) REST APIs in C#
- Understanding of Docker & associated concepts
- Dependency Injection in C#
- Use of Async / Await

Would be useful!

Ingredients

- VS Code Text Editor (free)
- .NET 5 (free)
- Docker Desktop (Running Kubernetes) (free)
- An account on Docker Hub (free)
- Insomnia or Postman (free)
- A “decent” level of local hardware (not usually free)

My PC Spec

- We'll be spinning up a lot of containers (in Kubernetes)
- Intel i7
- 32Gb Memory
  - 16Gb will be fine
  - Less than 8Gb and you'll start to struggle
- 1TB SSD

Free Kubernetes Cheat Sheet

- Docker & Kubernetes Commands
- Kubernetes Application Architecture Reference
- Kubernetes Object Glossary
- Go to: https://dotnetplaybook.com/
    - Subscribe & we'll email you with a download link

Other course resources

- Code is available on GitHub
- Course Outline in description
  - Jump to / revisit sections
- Course Slides – (Patreon $5 supporters)
  - patreon.com/binarythistle

Microservices

What are they and how are they saving the world?

The Single Responsibility Principle

"Gather together those things that change for the same reason, and separate those things that change for different reasons."

Robert C. Martin

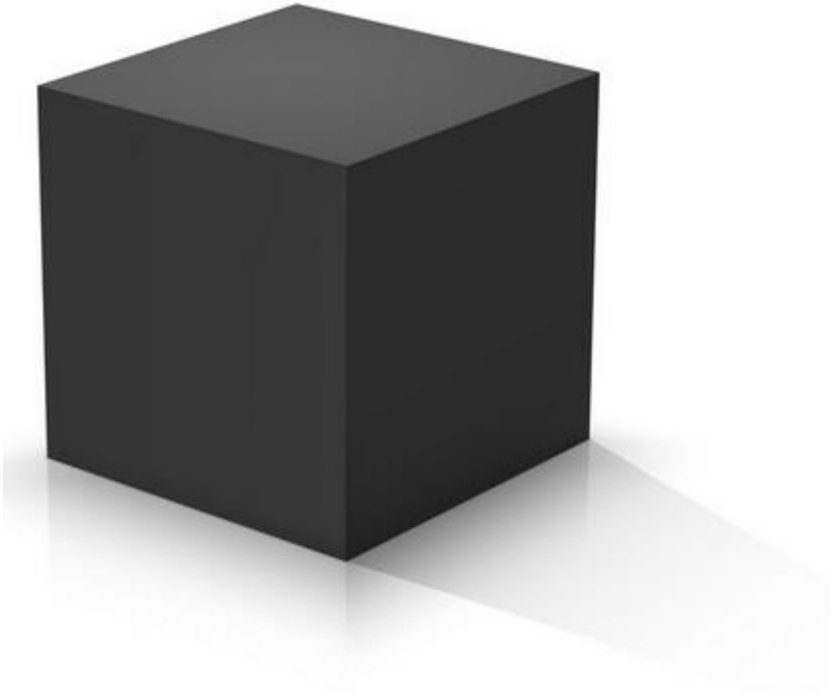
What are “Microservices”

- Small (2 pizza team, 2 weeks to build)
- Responsible for doing 1 thing well
- Organisationally aligned
- Form part of the (distributed) whole
- Self-contained / Autonomous

A True Story…

I know of large monolithic CRM system, that:

- Services millions of customs
- Evolved over many years
- Built on a single, proprietary tech stack
- Managed by 1 “out-sourced” partner



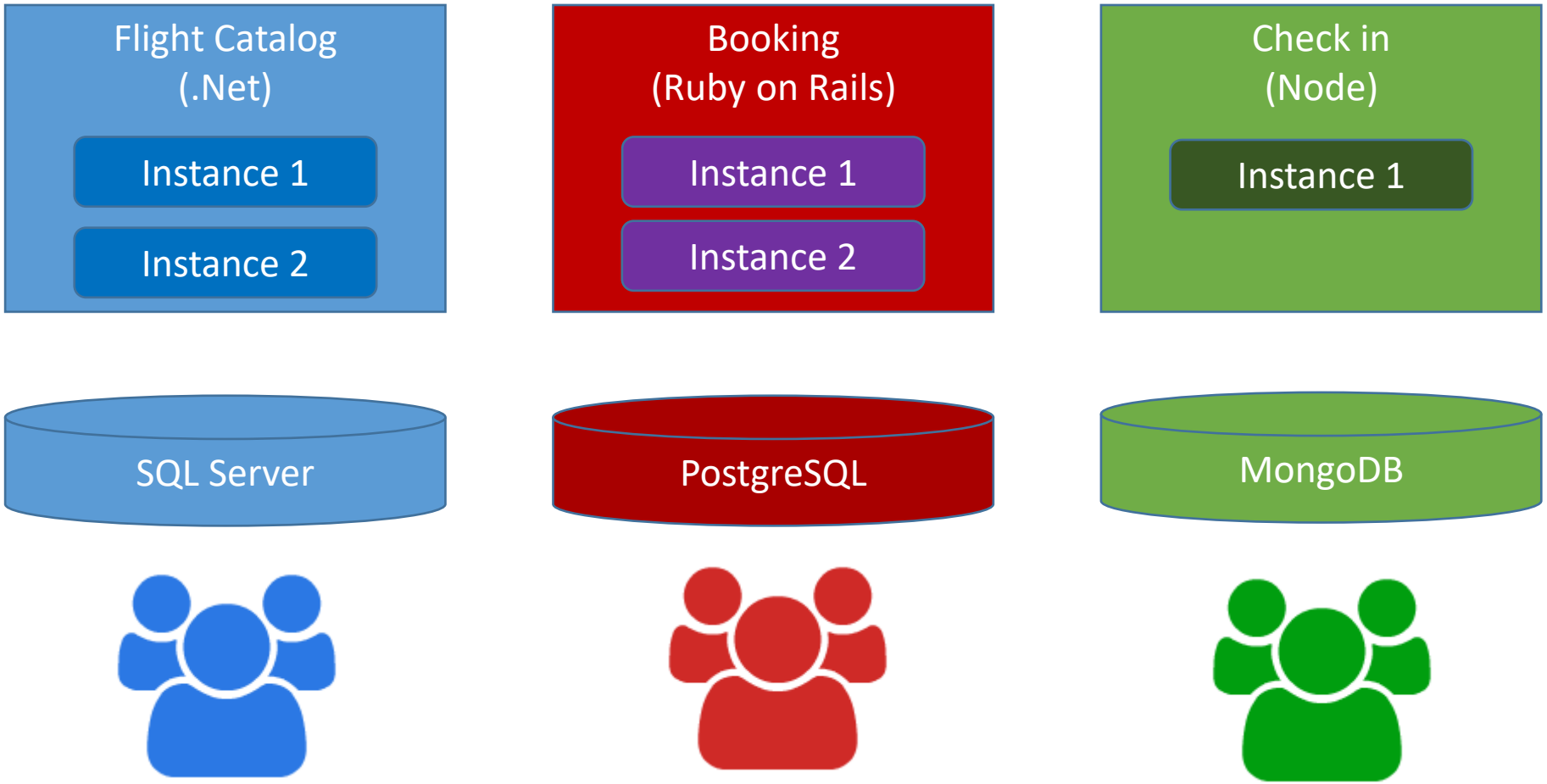
This system is…

- Very difficult to “change”
  - Change cycles are months in duration
  - Massive amounts of (frequently manual) testing
- Difficult to Scale
- Locked in
  - Technology terms
  - Intellectual Property terms (an external party held the cards)

Benefits of Microservices

- Easier to change & deploy (small and decoupled)
- Can be built using different technologies
- Increased organisational ownership & alignment
- Resilient: 1 service can break the, others will continue to run
- Scalable: You can scale out only the services you need to
- Built to be highly replaceable / swappable

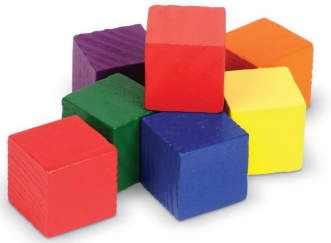
An Example


Flight Catalog
(.Net)
Instance 1
Instance 2
Booking
(Ruby on Rails)
Instance 1
Instance 2
Check in
(Node)
Instance 1
SQL Server
PostgreSQL
MongoDB
Team: Nessie
Country: Scotland
Team: The Eagles
Country: USA
Team: Cobra
Country: India

So..?

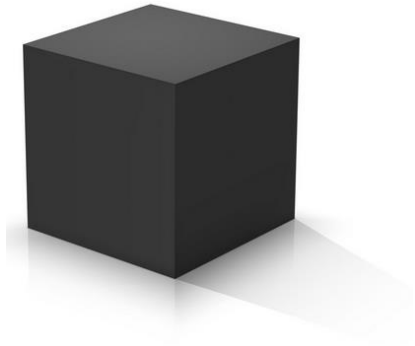
Microservices

Monoliths



Microservices

- Are difficult to implement
- Can result in “analysis paralysis”
- Need strong domain knowledge
- Distributed – shock horror the network can fail
- Paradoxically will always be coupled to something…



Monoliths

- Simpler to implement
- Can use CI/CD, daily deploys, small changes etc.
- Allow you to familiarise yourself with the domain
- Can have 2 or 3 “big” services
- Not as reliant on network

Our Services

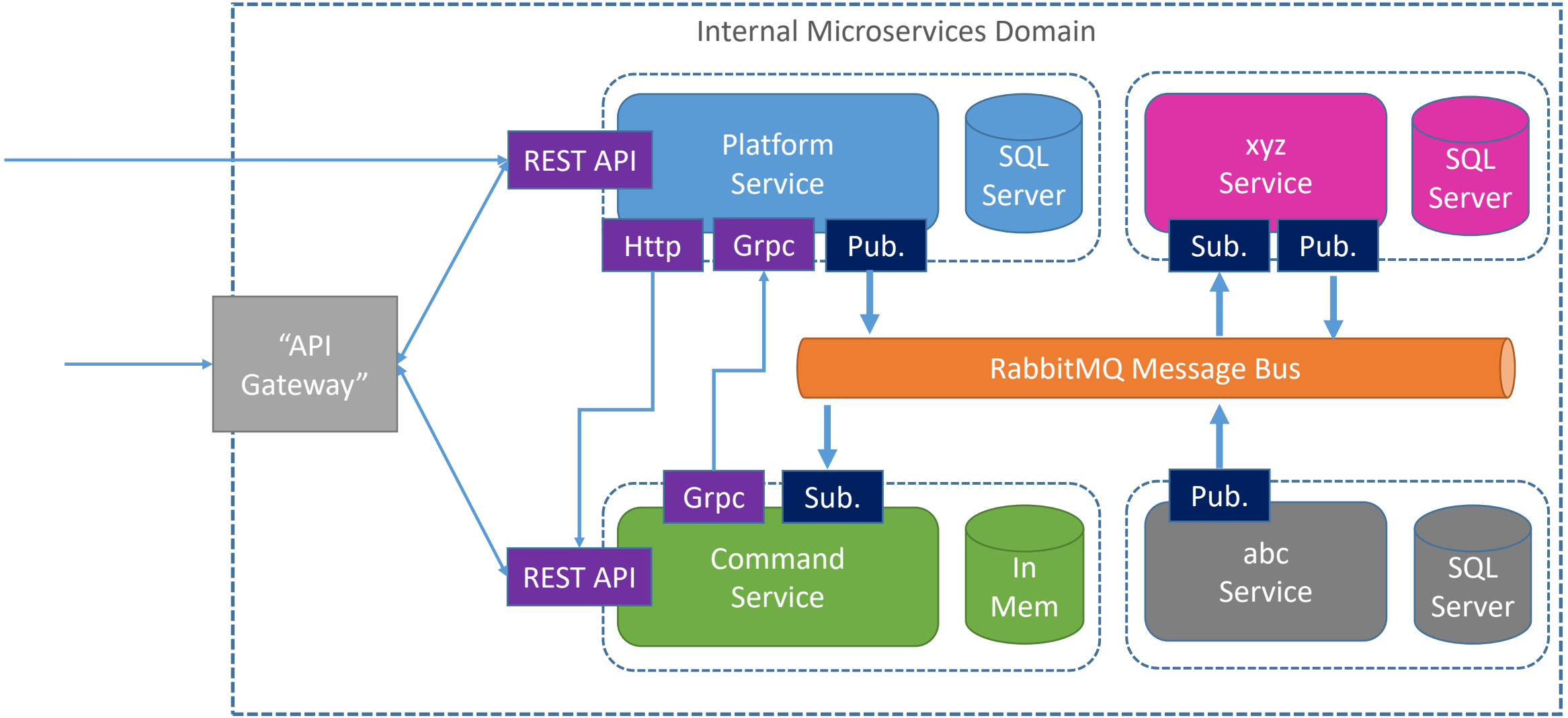
The “Platform” Service

- Function as an “Asset Register”
- Track all the platforms / systems in the company
- Built by the Infrastructure Team
- Used by:
  - Infrastructure Team
  - Technical Support Team
  - Engineering
  - Accounting
  - Procurement

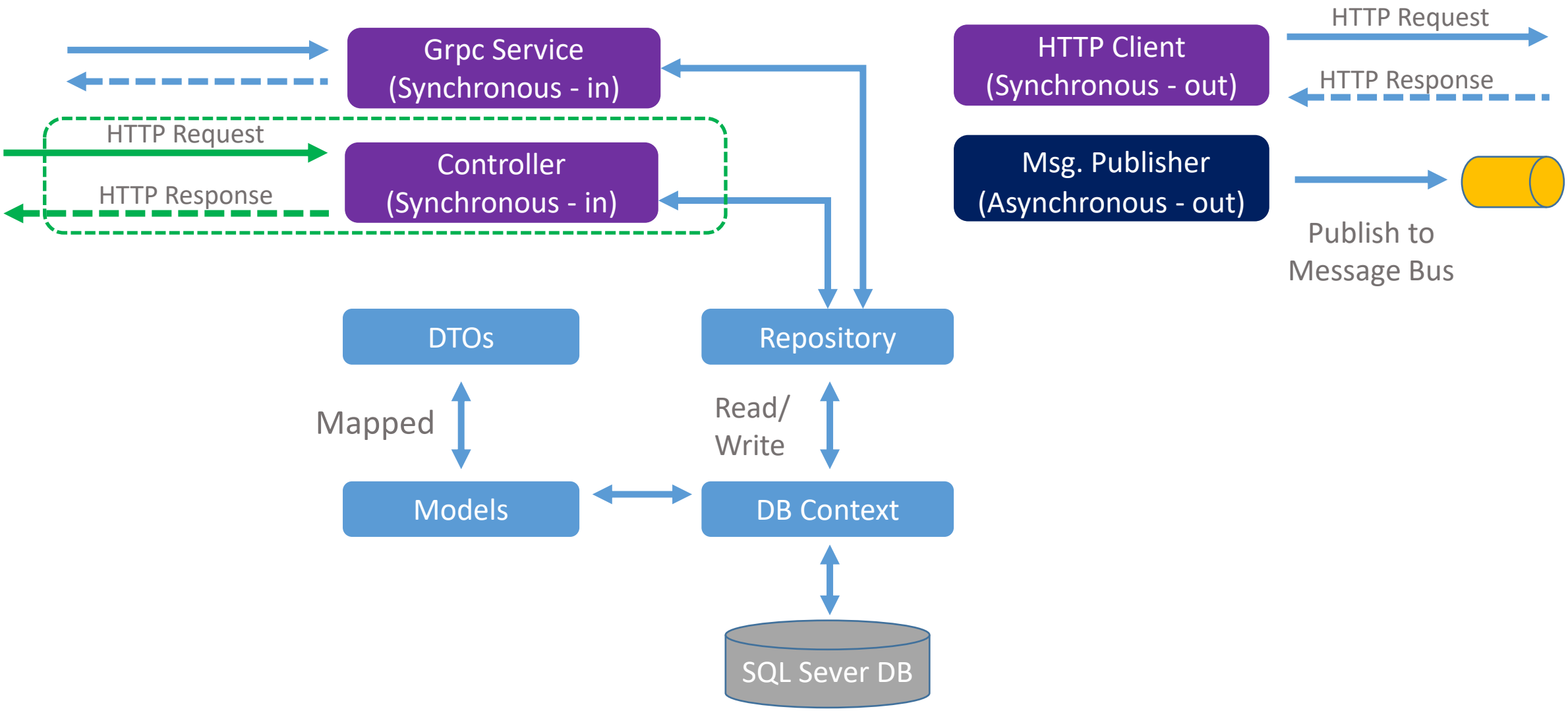
The “Commands” Service

- Function as a repository of command line augments for given Platforms
- Aid in the automation of support processes
- Built by the Technical Support Team
- Used By:
  - Technical Support Team
  - Infrastructure Team
  - Engineering

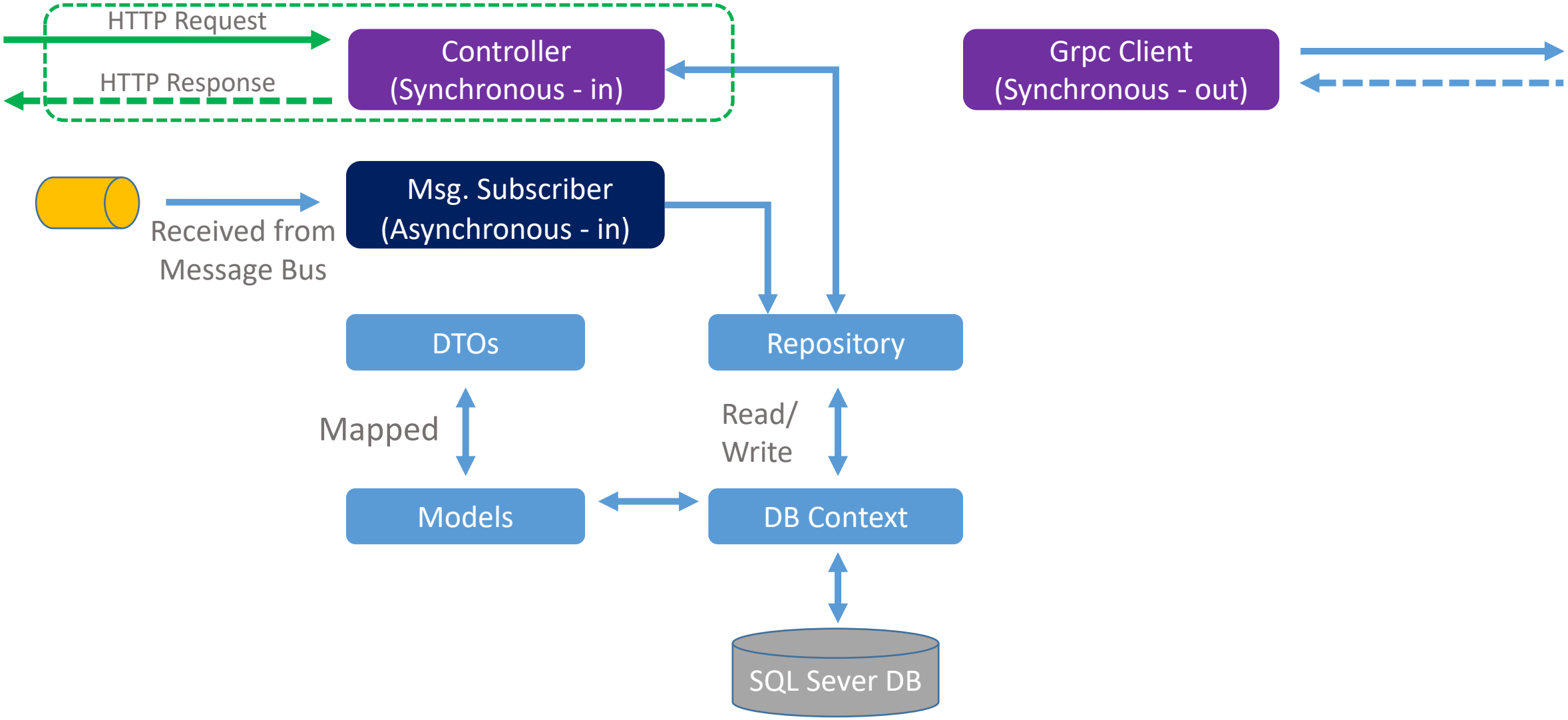
Solution Architecture



Platform Service Architecture



Command Service Architecture



Start Coding!

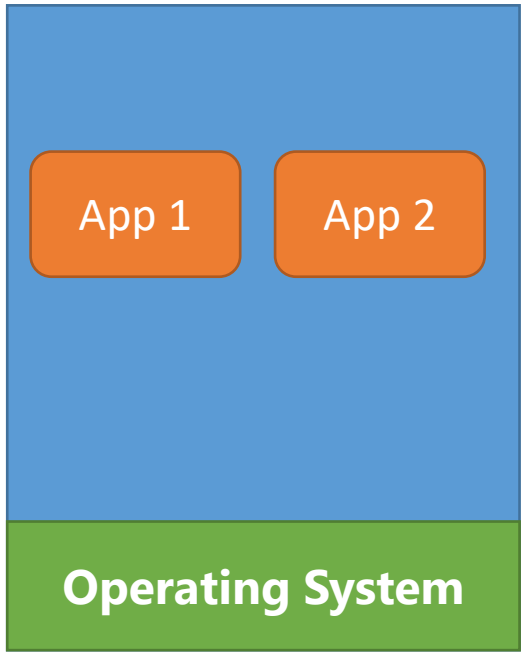
Platform Service

Docker

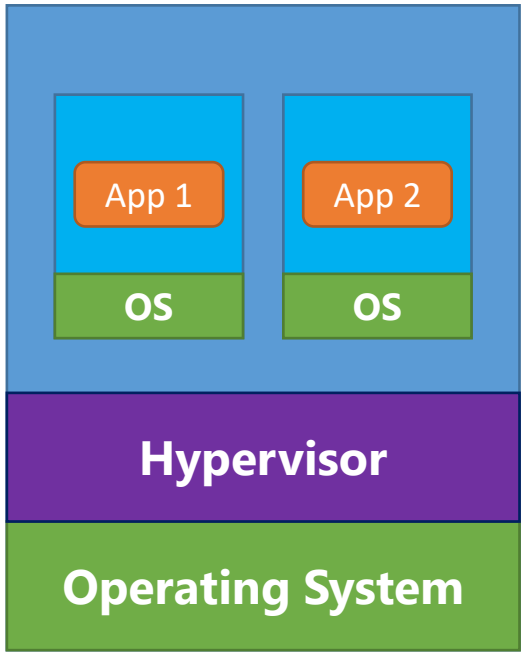
A Quick Review

What is Docker?

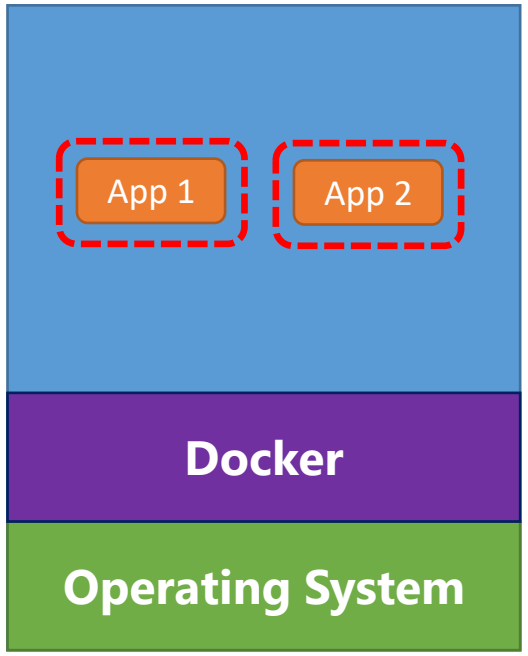
Docker is a **containerization** platform, meaning that it enables you to **package** your applications into **images** and run them as "**containers**" on any platform that can run Docker.



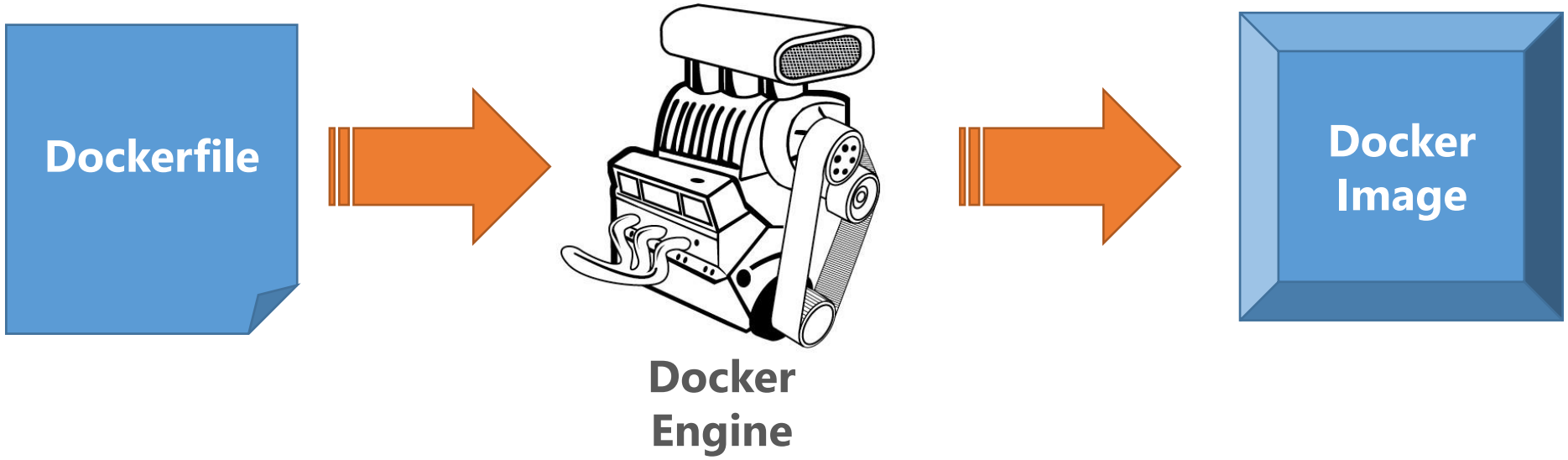
Physical Machine



Virtual Machine
(OS Virtualisation)



Container (Docker)
(App Virtualisation)


Dockerfile
Docker
Engine
Docker
Image

Kubernetes

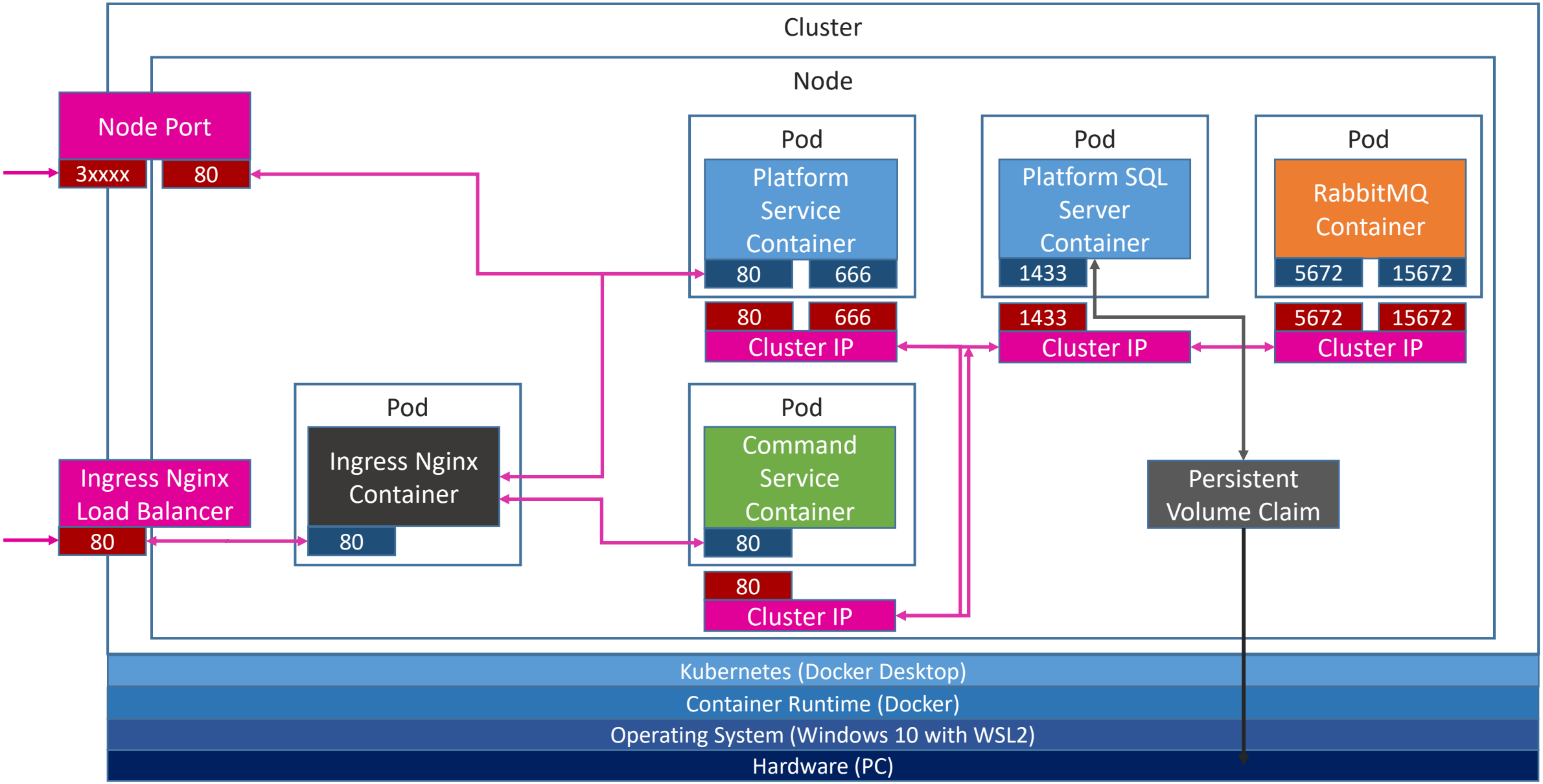
An Introduction


Kubernetes
Container
Container
Container

Kubernetes

- Built by Google now maintained by the Cloud Native Foundation
- Often referred to as “K8S”
- Container Orchestrator
- Huge subject area!
- 2 broad user profiles
  - Developer
  - Administrator

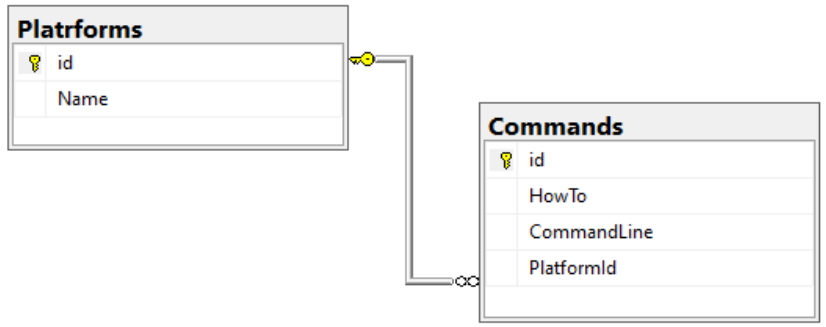
Kubernetes Architecture



Multi-Resources URIs

{ REST:API }

Working with Command Resources



- Platform is the “parent”
- We get to commands via their platform

Action	Verb		Controller
Get all Platforms	GET	/api/c/platforms	Platform
Get all Commands for a Platform	GET	/api/c/platforms/{platformId}/commands	Command
Get a Command for a Platform	GET	/api/c/platforms/{platformId}/commands/{commandId}	Command
Create a Command for a Platform	POST	/api/c/platforms/{platformId}/commands/	Command

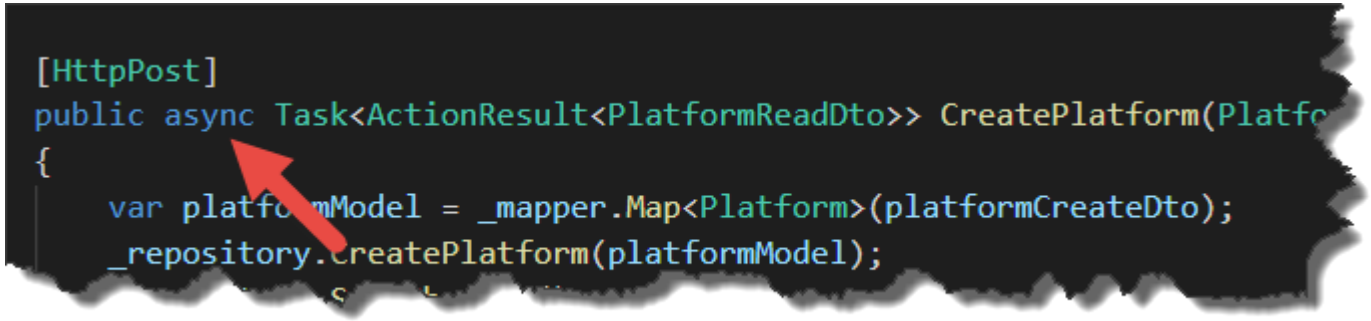
Messaging

Synchronous & Asynchronous Messaging

Synchronous Messaging

- Request / Response Cycle
- Requester will “wait” for response
- Externally facing services usually synchronous (e.g. http requests)
- Services *usually* need to “know” about each other
- We are using 2 forms:
  - Http
  - Grpc

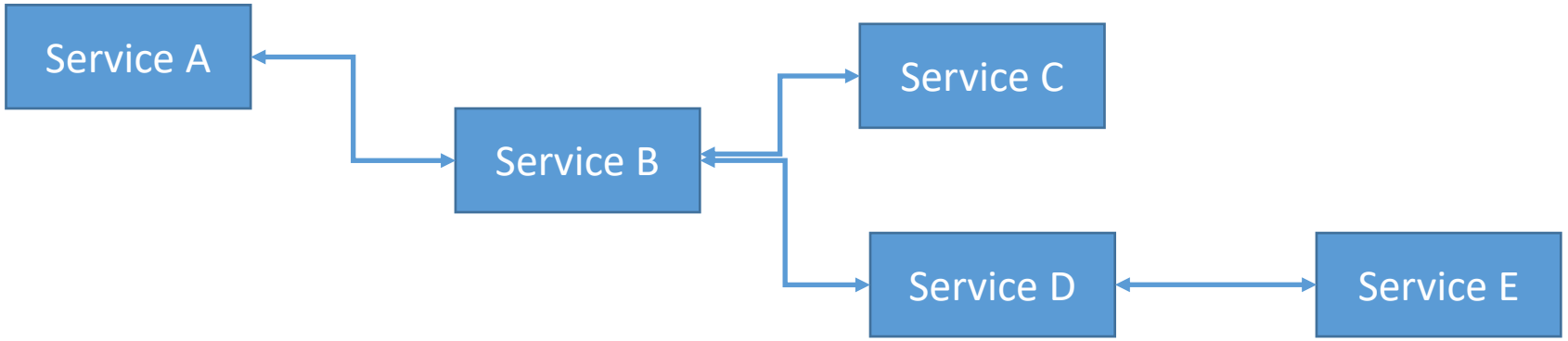
Wait! What if I mark http actions as Async?



- From a messaging perspective this method is still synchronous
- The client still has to wait for a response
- Async in this context (the C# language) means that the ***action will not wait*** for a long running operation
- It will hand back it's thread to the thread pool, where it can be reused
- When the operation finishes it will re-acquire a thread and complete, (and respond back to the requestor)
- So Async here is about thread exhaustion – the requestor still has to wait (the call is synchronous)

Synchronous messaging between services

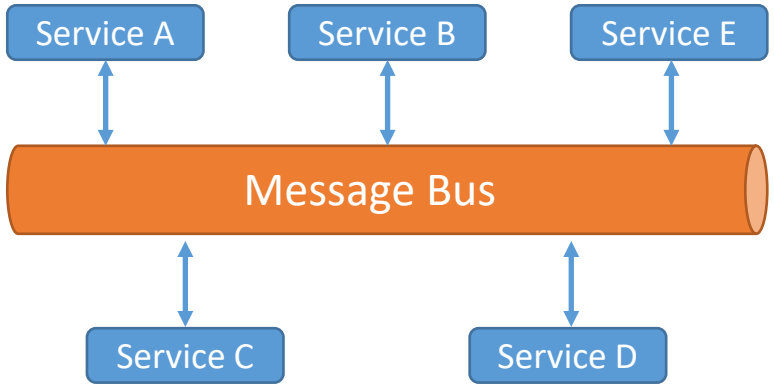
- Can and does occur – we will implement, however..
- It does tend to pair services, (couple them), creating a dependency
- Could lead to long dependency chains



Asynchronous Messaging

- No Request / Response Cycle
- Requester does not wait
- Event model, e.g. publish –subscribe
- Typically used between services
- Event bus is often used (we'll be using RabbitMQ)
- Services don't need to know about each other, just the bus
- Introduces its own range of complexities – not a magic bullet

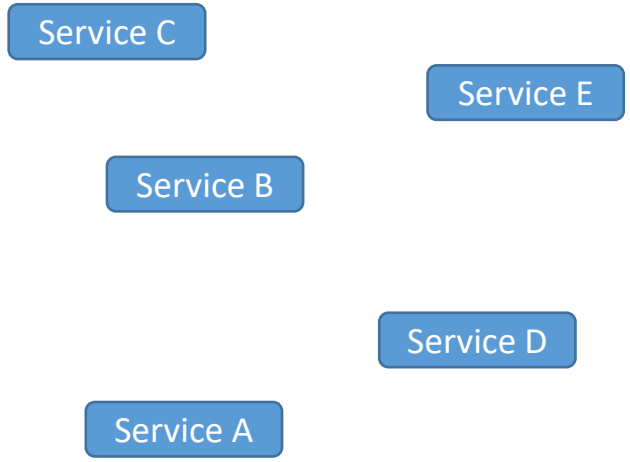
Wait! Isn't the event bus a Monolith?



- To some extent yes
- Internal comms would cease if the message bus goes down
- Services will still operate and work externally
- Should be treated as a first class citizen, similar to:
  - Network, physical storage, power etc
- Message bus should be clustered, with message persistence etc.
- Services should implement some kind of retry policy
- Aim for Smart Services, stupid pipes.

The Microservice dependency paradox

- **Question:** Is a group of completely independent, autonomous “services” a microservices architecture?
- **Answer:** Probably more a collection of min-monoliths…



The Paradox: While we aim to have autonomous, decoupled services, in order to maximise the benefits of a microservices architecture, services *need* dependencies.

RabbitMQ

Overview

What is RabbitMQ?

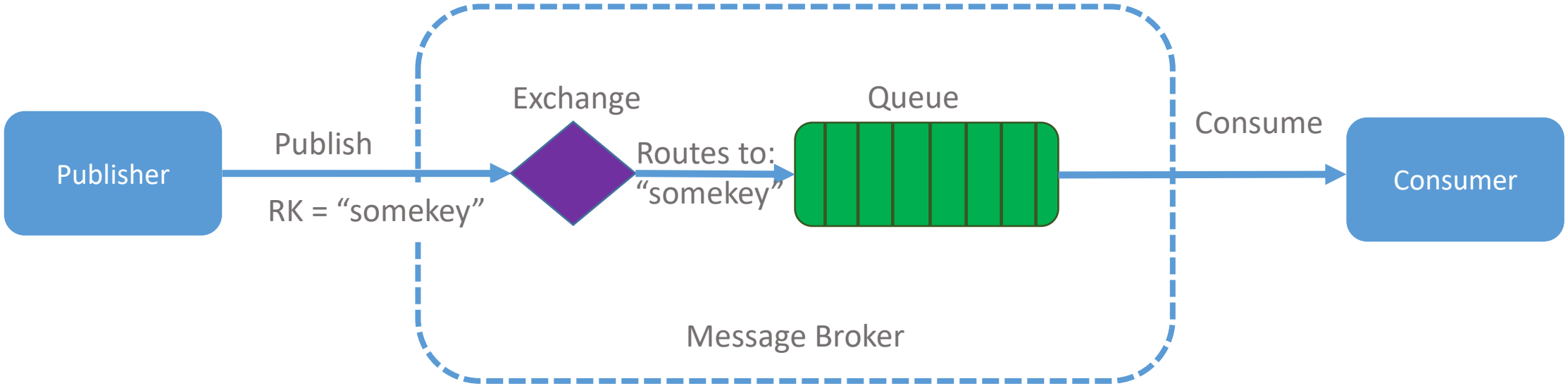
- A Message Broker – it accepts and forwards messages
- Messages are sent by Producers (or Publishers)
- Messages are received by Consumers (or Subscribers)
- Messages are stored on Queues (essentially a message buffer)
- Exchanges can be used to add “routing” functionality
- Uses Advanced Message Queuing Protocol (AMQP) & others

Exchanges

4 Types of Exchange

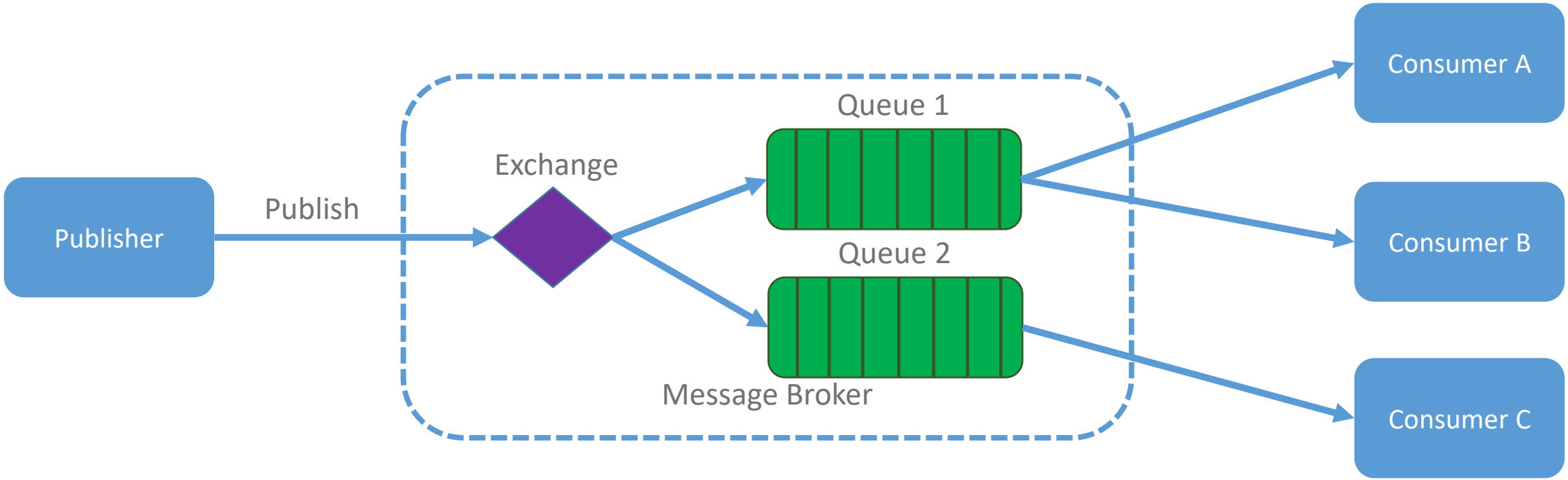
- Direct Exchange
- Fanout Exchange
- Topic Exchange
- Header Exchange

RabbitMQ Direct Exchange



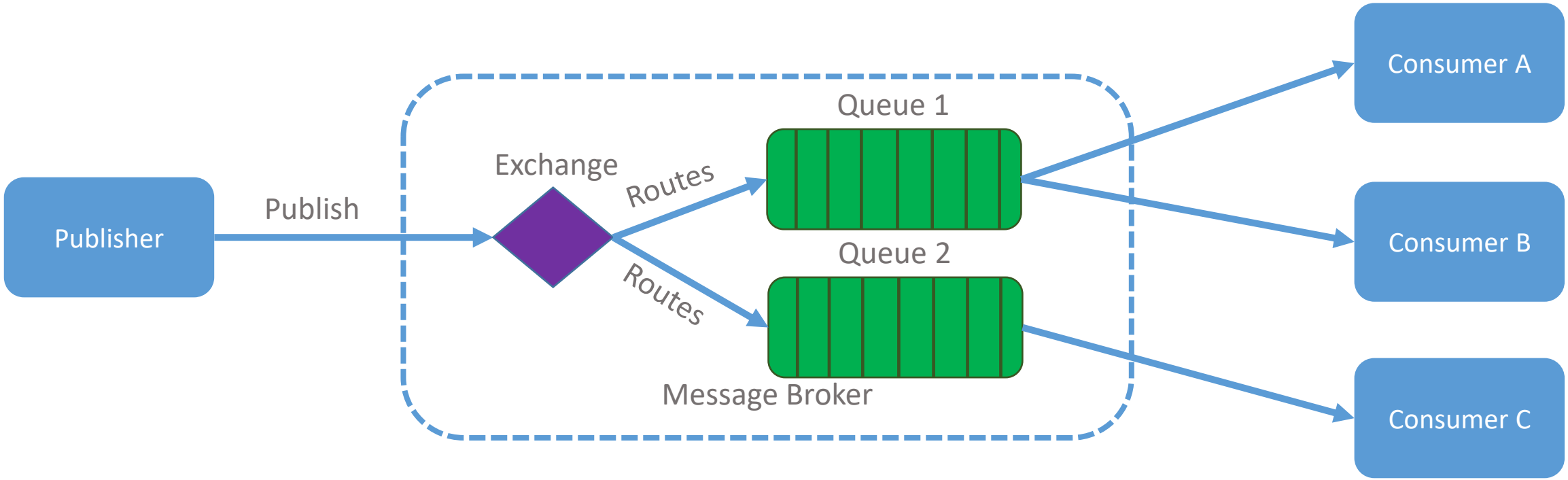
- Delivers Messages to queues based on a routing key
- Ideal for “direct” or unicast messaging

RabbitMQ Fanout Exchange



- Delivers Messages to all Queues that are bound to the exchange
- It ignores the routing key
- Ideal for broadcast messages

RabbitMQ Topic Exchange



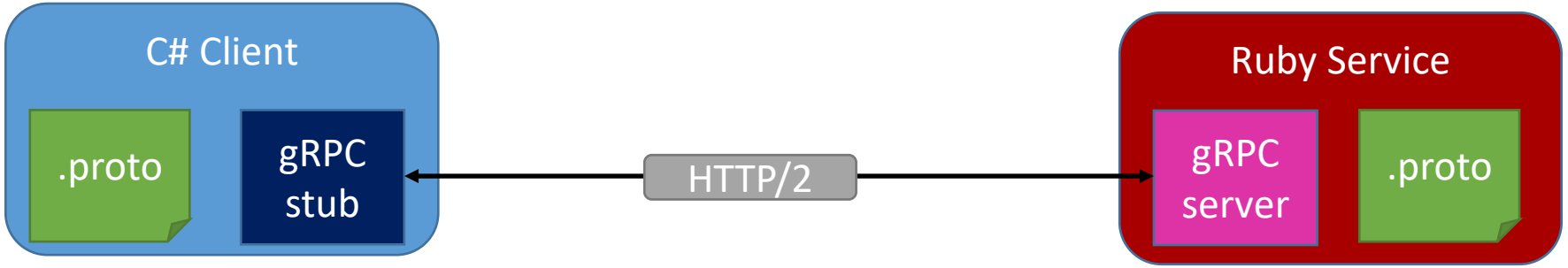
- Routes messages to 1 or more queues based on the routing key (and patterns)
- Used for Multicast messaging
- Implements various Publisher / Subscriber Patterns

gRPC

What is it, and why should we use it?

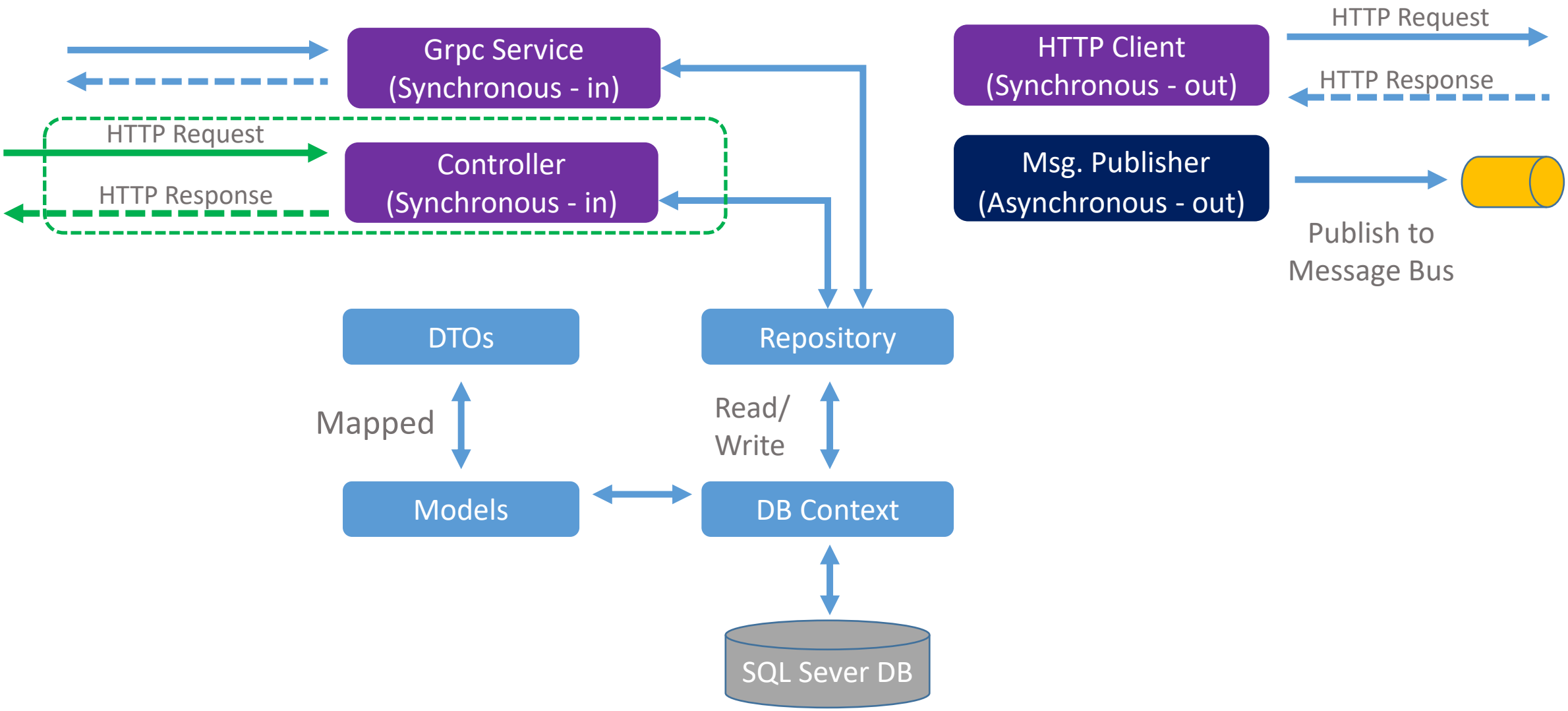
gRPC

- “Google” Remote Procedure Call
- Uses HTTP/2 protocol to transport binary messages (inc. TLS)
- Focused on high performance
- Relies on “Protocol Buffers” (aka Protobuf) to defined the contract between end points
- Multi-language support (C# client can call a Ruby service)
- Frequently used as a method of service to service communication
- Complex use of HTTP/ 2 prohibits use of gRPC in browser-based apps (would require a proxy)

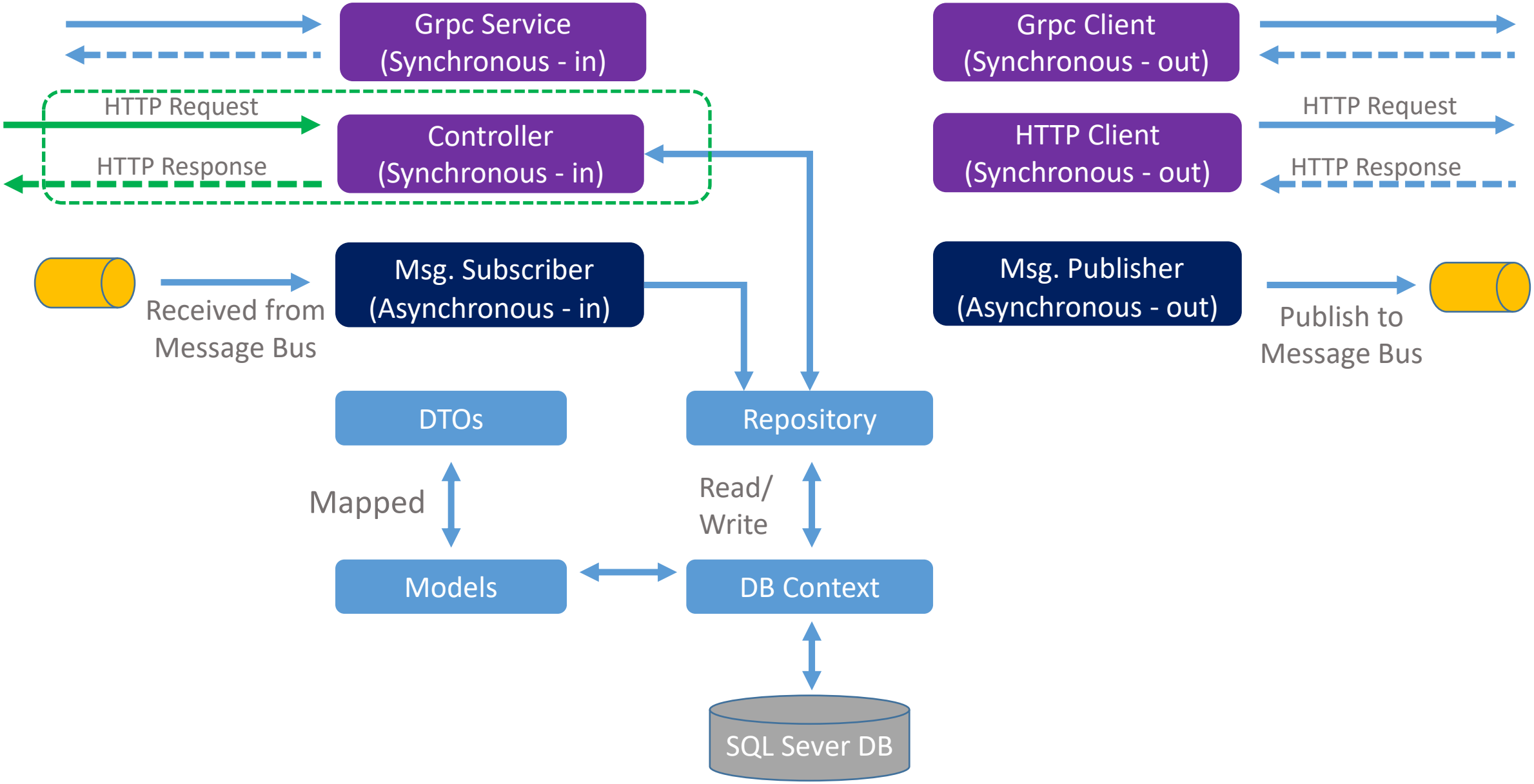


Appendix

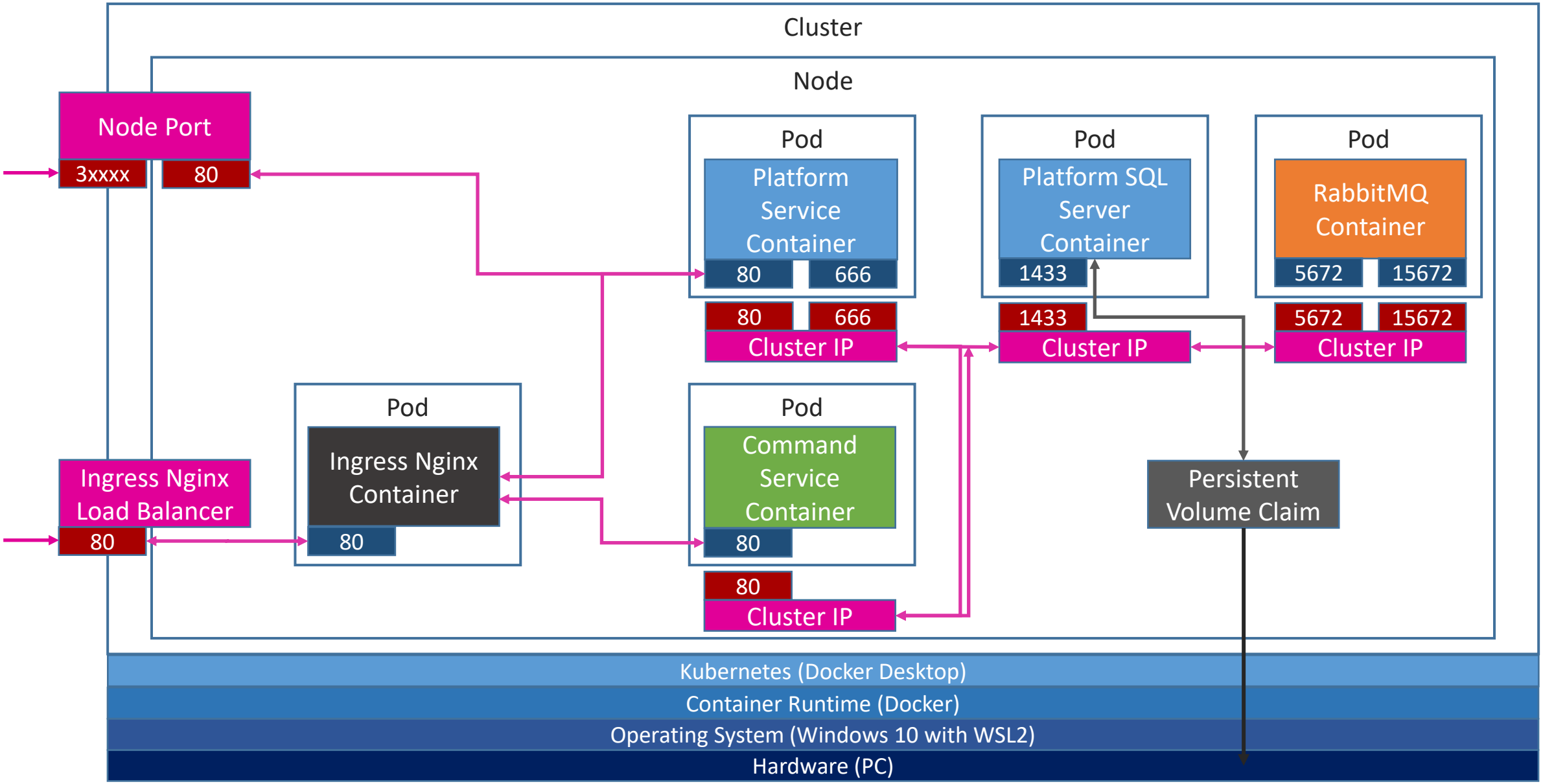
Platform Service Architecture



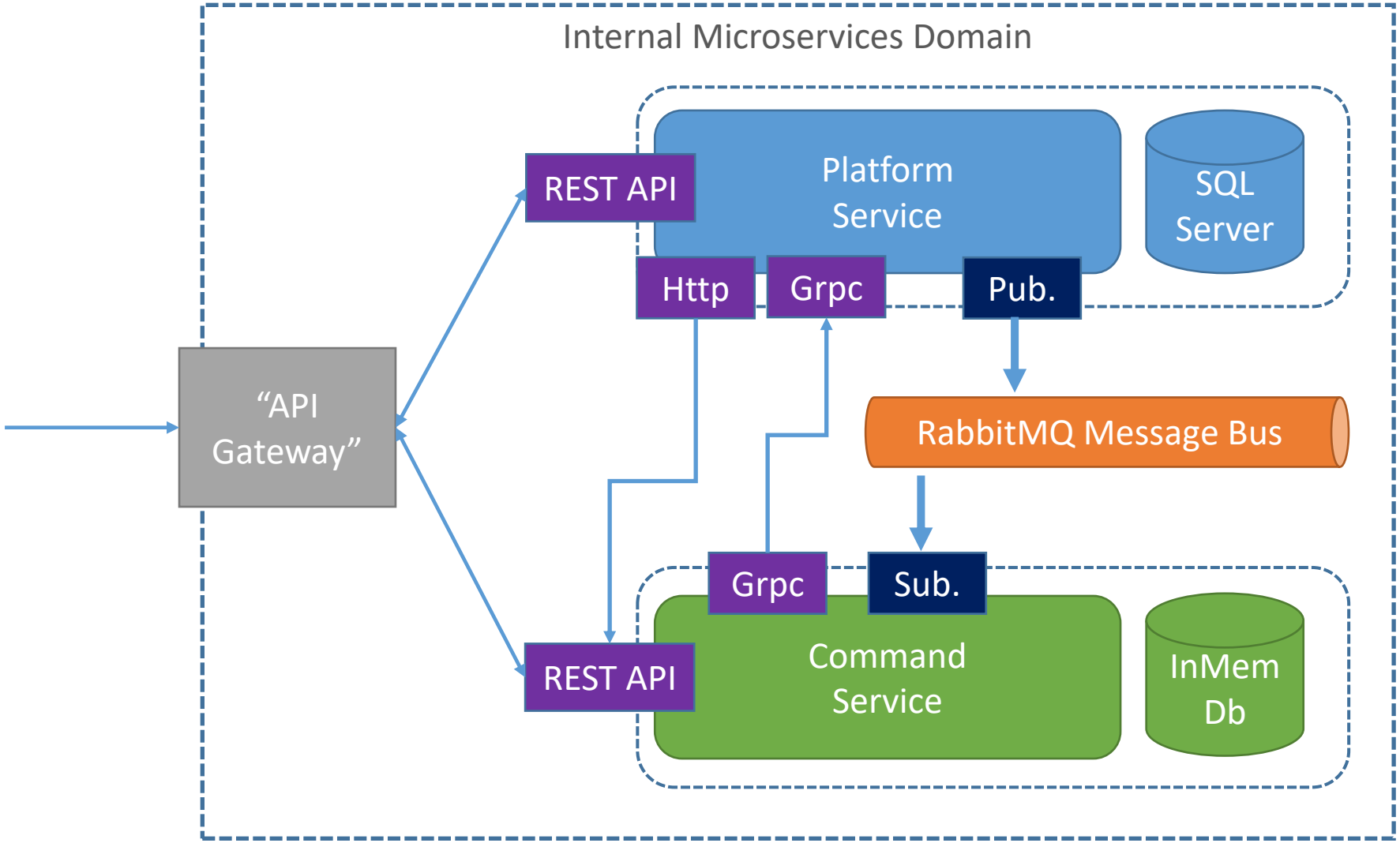
Combined Service Architecture



Kubernetes Architecture



Solution Architecture



Moving Forward

- Introduction of HTTPS / TLS
- Revisit Event Processor / Eventing
- More elaborate use-case
- Service Discovery

Thank You!