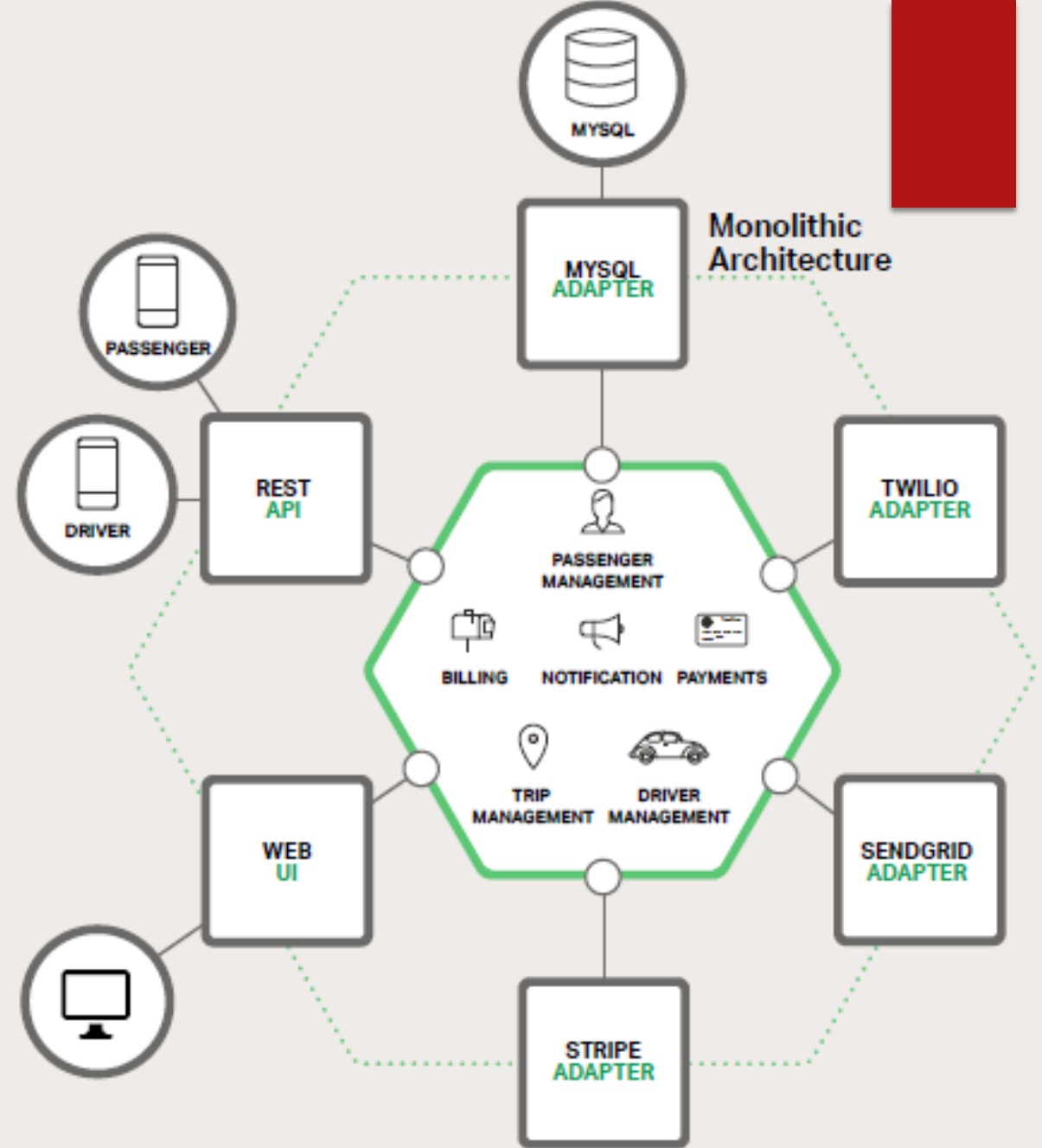


Monolithic Hell

- ▶ Successful applications have a habit of growing over time and eventually becoming huge.
 - ▶ After a few years, your small, simple application will have grown into a monstrous monolith.
- ▶ It's simply too large for any single developer to fully understand
 - ▶ fixing bugs and implementing new features correctly becomes difficult and time consuming
- ▶ The larger the application, the longer the start-up time is
 - ▶ large part of developer day will be spent waiting around and their productivity will suffer
- ▶ Increased Lead time
- ▶ Difficult to scale when different modules have conflicting resource requirements.
- ▶ Reliability
 - ▶ Because all modules are running within the same process, a bug in any module, such as a memory leak, can potentially bring down the entire process
 - ▶ If all instances of the application are identical, that bug will impact the availability of the entire application.
- ▶ Adaptability
 - ▶ Monolithic apps not open to adopt new frameworks and languages

Monolithic

- ▶ Core Business logic
- ▶ Core Adapters
- ▶ Core Services
- ▶ Messaging Components
- ▶ Single Large Database
- ▶ Single Deployment Pipeline

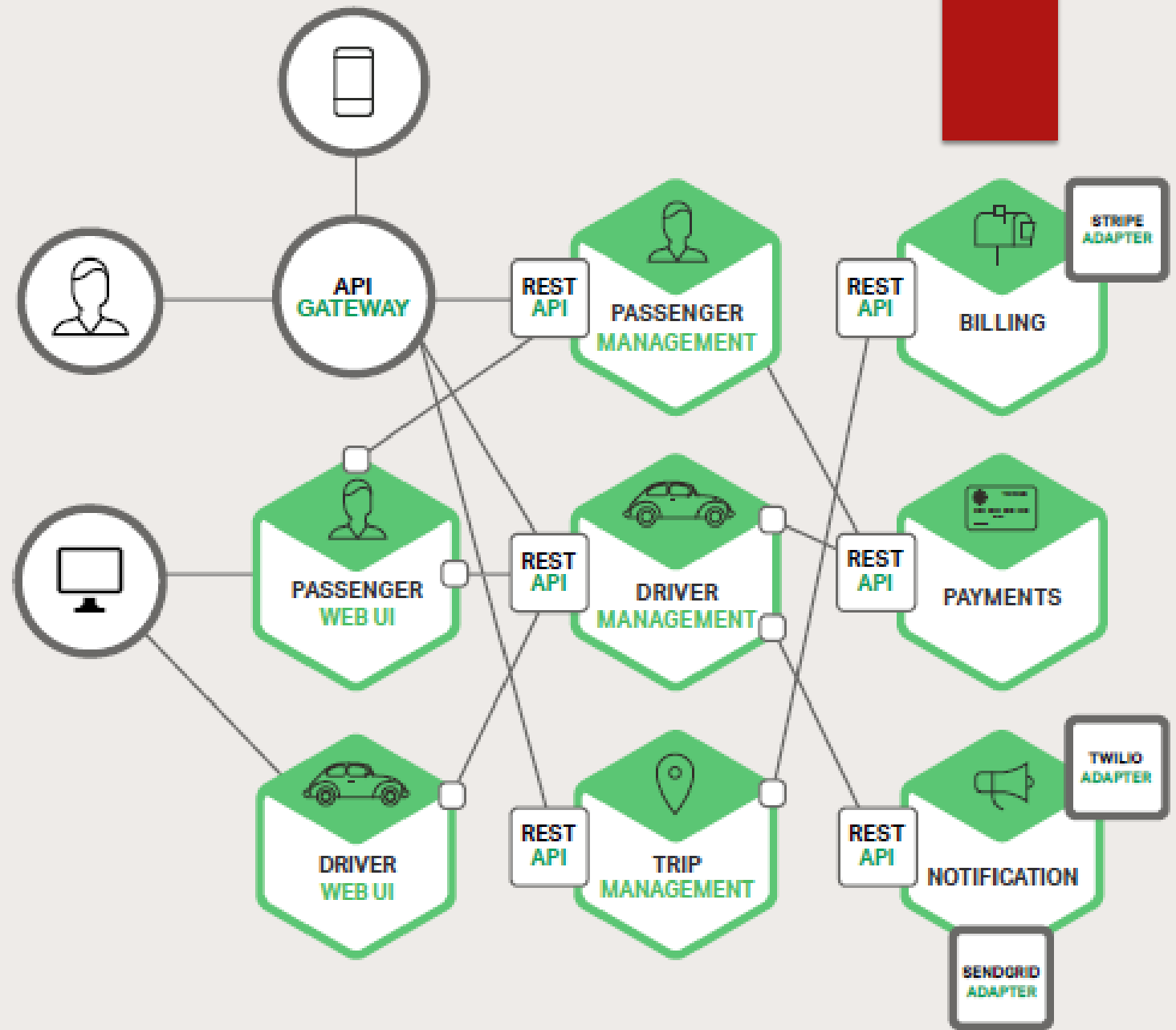


Partitioning Your Business into Services

- ▶ Application Decomposition
 - ▶ Decomposing by Functionality
 - ▶ Email processing
 - ▶ Decomposing by Maturity
 - ▶ Decomposing by Data-Access Pattern
 - ▶ Read intensive
 - ▶ write intensive
 - ▶ Decomposing by Context
- ▶ Application Aggregation
 - ▶ Aggregation for a Derived Functionalities
 - ▶ Aggregation for Business Intelligence
 - ▶ ETL, Fraud detection services
- ▶ Aggregation for Client Convenience
 - ▶ mobile façade service
- ▶ Aggregation to Aid System Performance
 - ▶ Cache Service

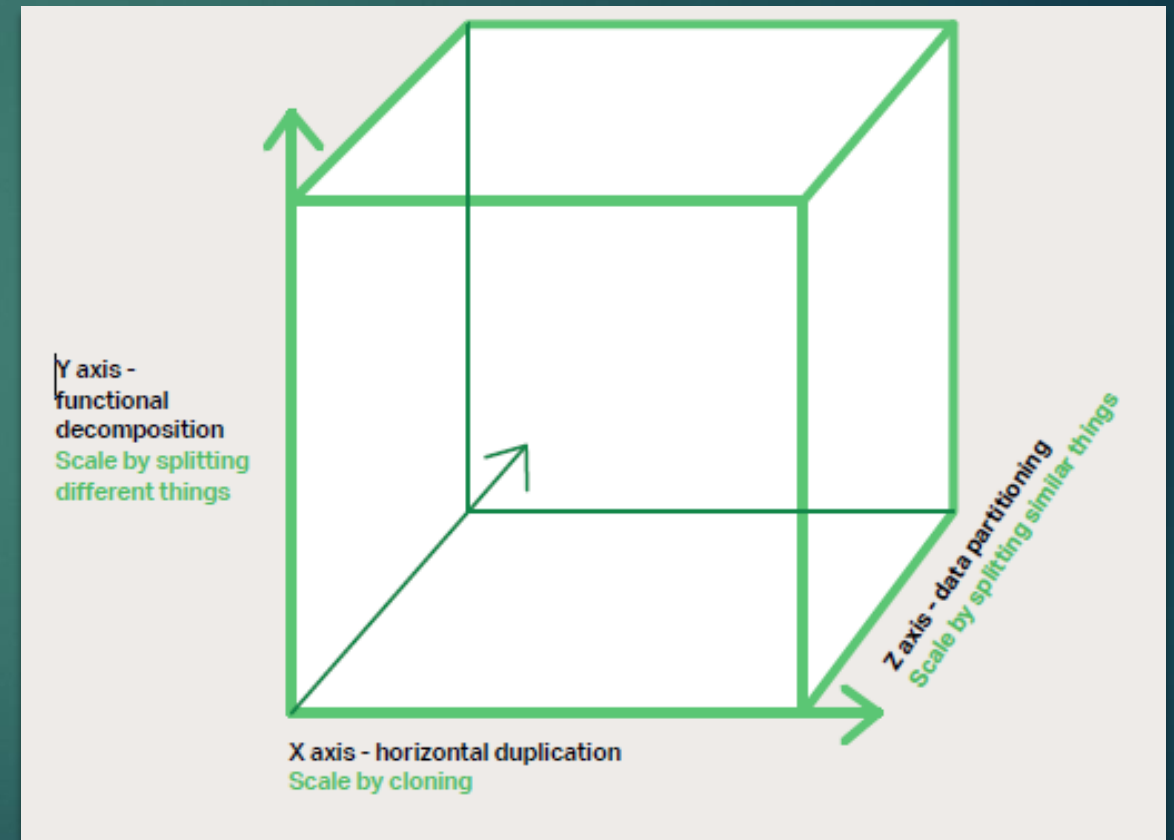
Microservices

- ▶ A service typically implements a set of **distinct** features or functionality
- ▶ Each microservice is a mini-application that has its own **hexagonal** architecture consisting of business logic along with various adapters



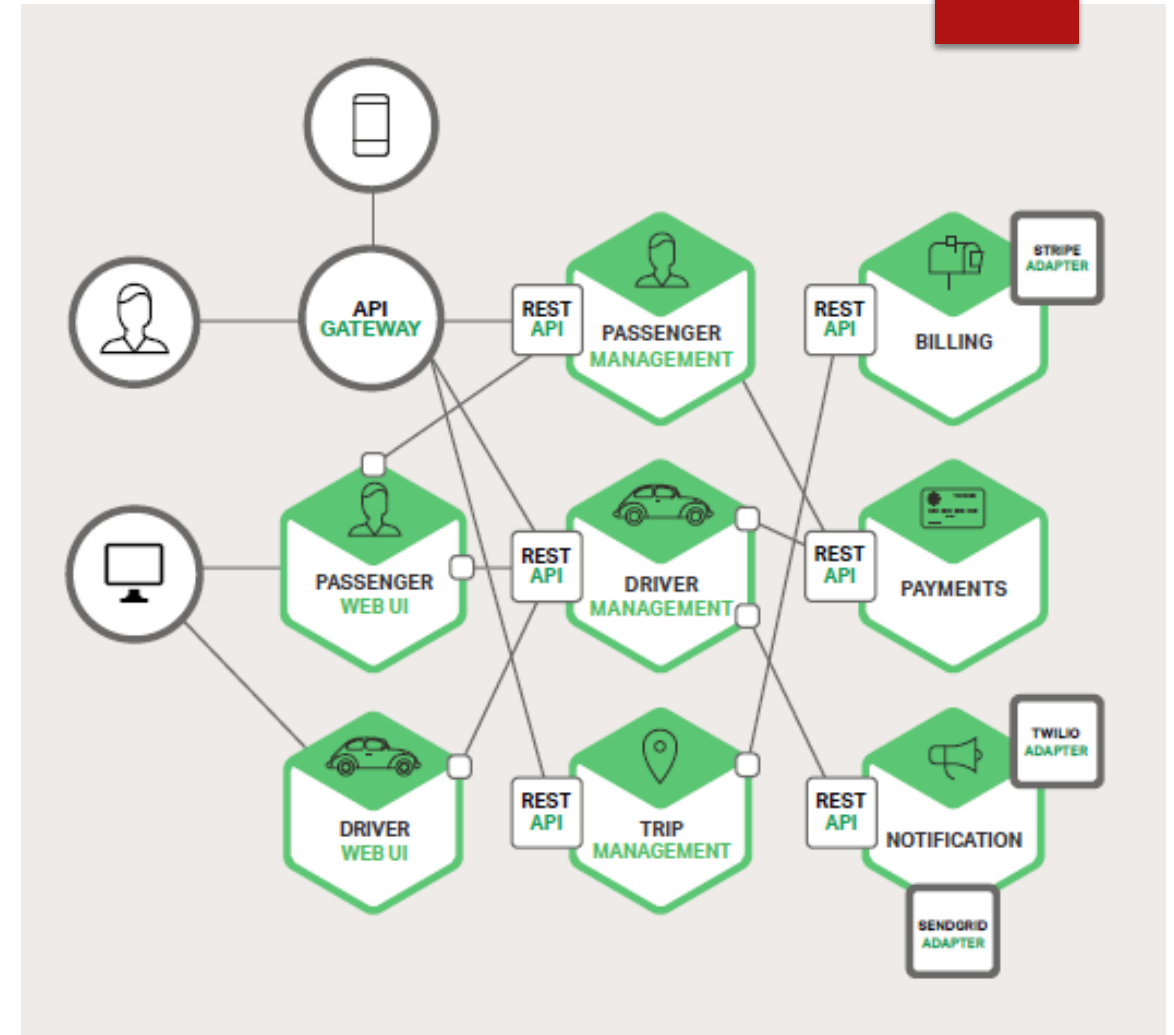
Scale cube

- ▶ The Microservices Architecture pattern corresponds to the Y-axis scaling of the Scale Cube
- ▶ X-axis scaling
 - ▶ Cloning
- ▶ Z-axis scaling
 - ▶ Data Partition



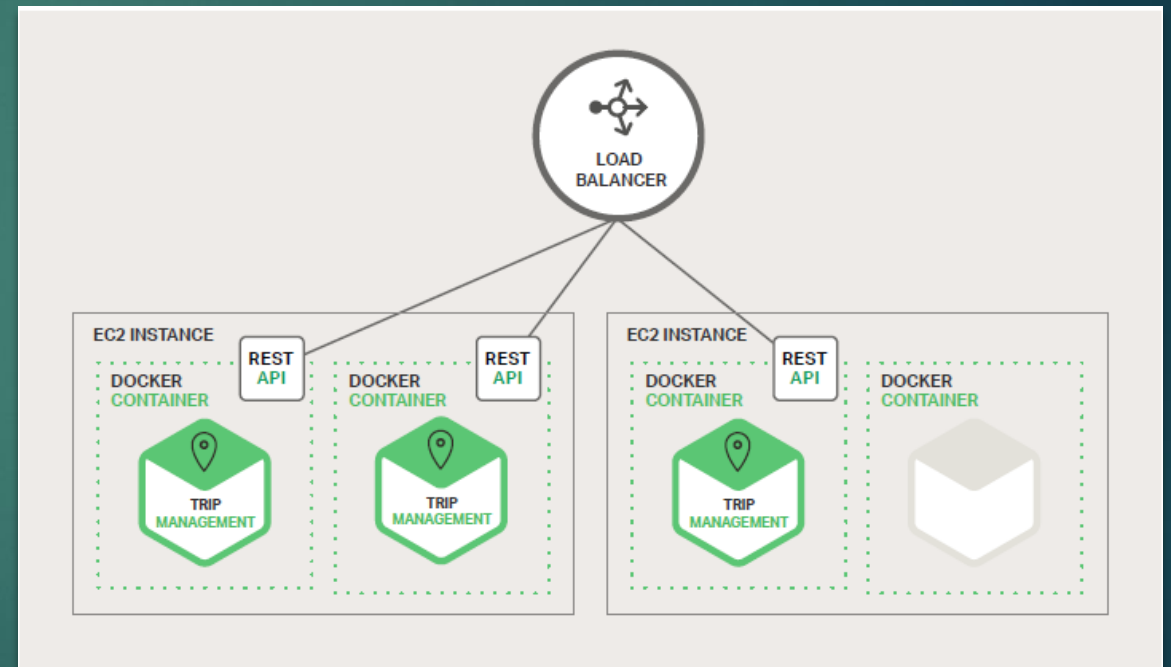
Y-axis scaling

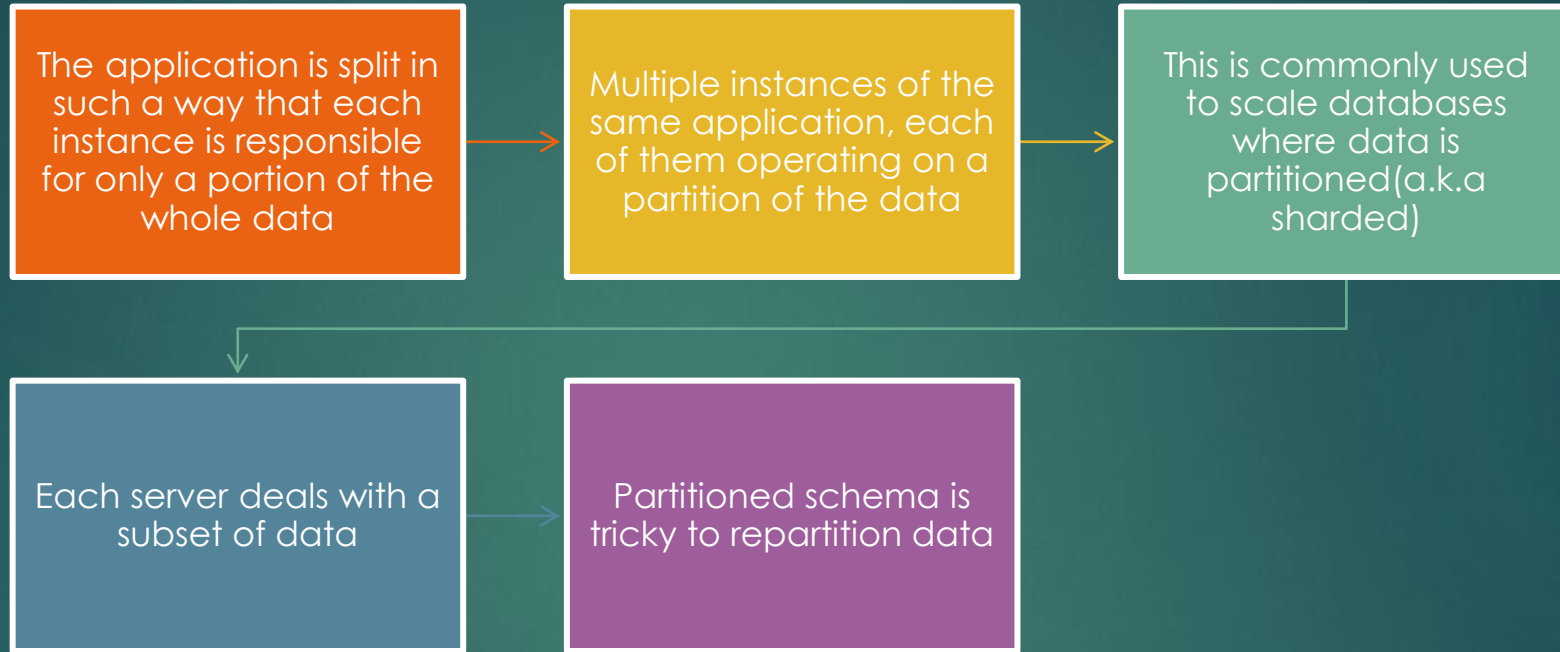
- ▶ Splitting the application into multiple, different services (microservices)
- ▶ Then more infra resources can be added to only the micro-service which is bottleneck in the architecture.
- ▶ Decomposing the application based on its functionalities, services, or use cases



X-axis scaling

- ▶ Running multiple copies of an application behind a load balancer
- ▶ Each copy can handle $1/N$ of the load
- ▶ Each copy accesses all the data, cache size will be higher.
- ▶ Doesn't solve increasing development and application complexity

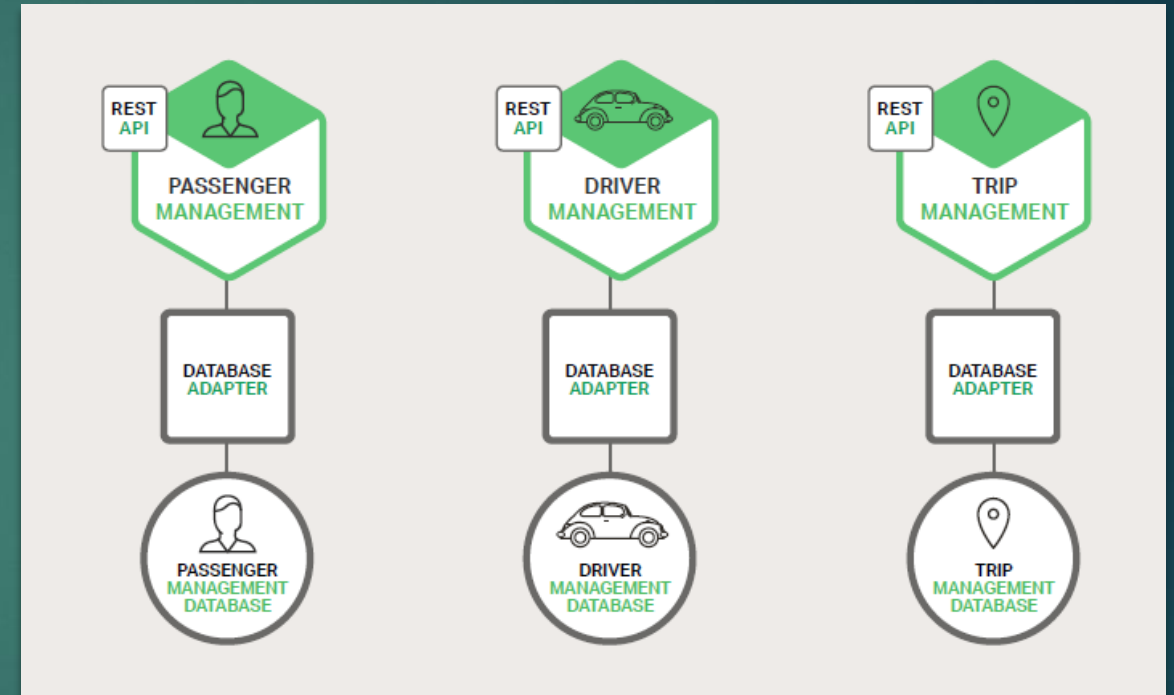




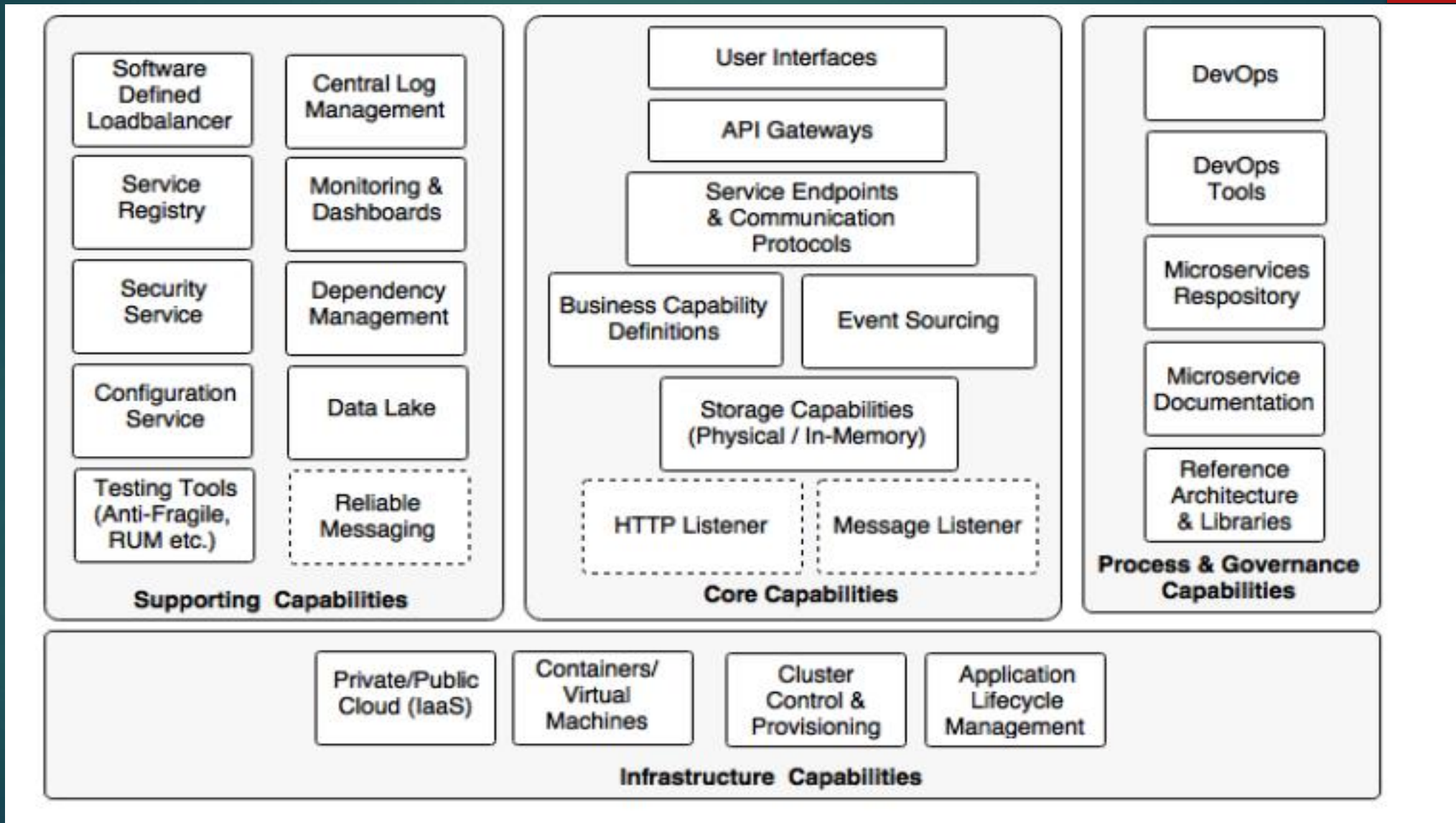
Z-axis scaling

Data Architecture (Z-axis)

- ▶ Each of the services has its own database.
- ▶ polyglot persistence architecture
 - ▶ A service can use a type of database that is best suited to its needs



Microservices Capability Model



API Gateway Pattern

Death Star Pitfall

- ▶ An architectural anti-pattern that occurs if services are highly coupled with each other
 - ▶ Intense Coupling
 - ▶ Hard To Make Changes
 - ▶ Maintenance Issues



How to Address Death Star Problem

▶ Bounded Context

- ▶ A bounded context is a set of tiny services (API, Background Jobs, Serverless Functions, etc) that work together to deliver requirements related to a sub-domain of a large software
- ▶ Technically speaking, the **Bounded Context** in DDD-speak is a specific boundary within your domain that your [Glossary from your Ubiquitous Language](#) can only apply – the idea being that different **Subdomains** may have competing or conflicting definitions of terms
- ▶ A **Bounded Context** is a system that fulfills the goals of the business in the real world.
- ▶ Any of our software systems (like a web service or web app) that operate as concrete instances in the Real World are considered **Bounded Contexts**
- ▶ one of the goals of Domain-Driven Designs is to have a one-to-one mapping from a **Subdomain** to a **Bounded Context**.

DDD in Minute

- ▶ Addresses poor collaboration between domain experts and software development teams
- ▶ DDD aims to increase the success rates by bridging collaboration and communication gap b/w Domain Experts and Software Development Teams
- ▶ How they Share Knowledge Fluently ?
 - ▶ **Ubiquitous Language – Shared** business-domain-oriented language
 - ▶ Language resemble the business domain, its terms, entities, and processes
 - ▶ **All Communication ,Documentation and Code should use Ubiquitous Language**
 - ▶ The Ubiquitous Language becomes the model of the business domain implemented in code

Ubiquitous Language term should have exactly one meaning

- Words often have different meanings in different contexts

How to overcome this hurdle ?

- DDD requires each language to have a strict applicability context
 - Bounded Context

Bounded Context

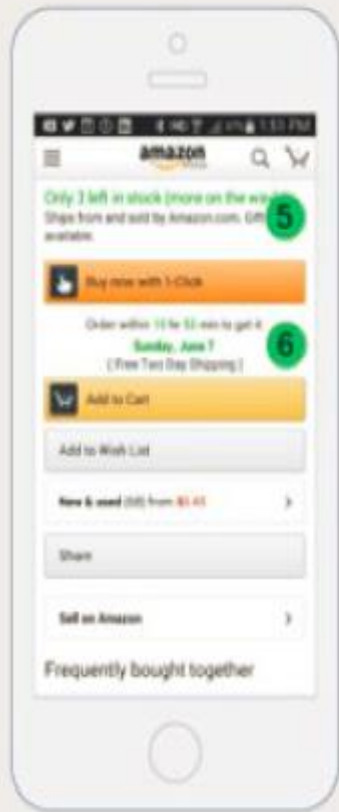
- Defines a boundary, inside of which a Ubiquitous Language can be used freely, Outside of it, the language's terms may have different meanings
- Each Bounded Context defines the applicability context of a specific model.
- A model is only valid in its Bounded Context

Ubiquitous Language Issues

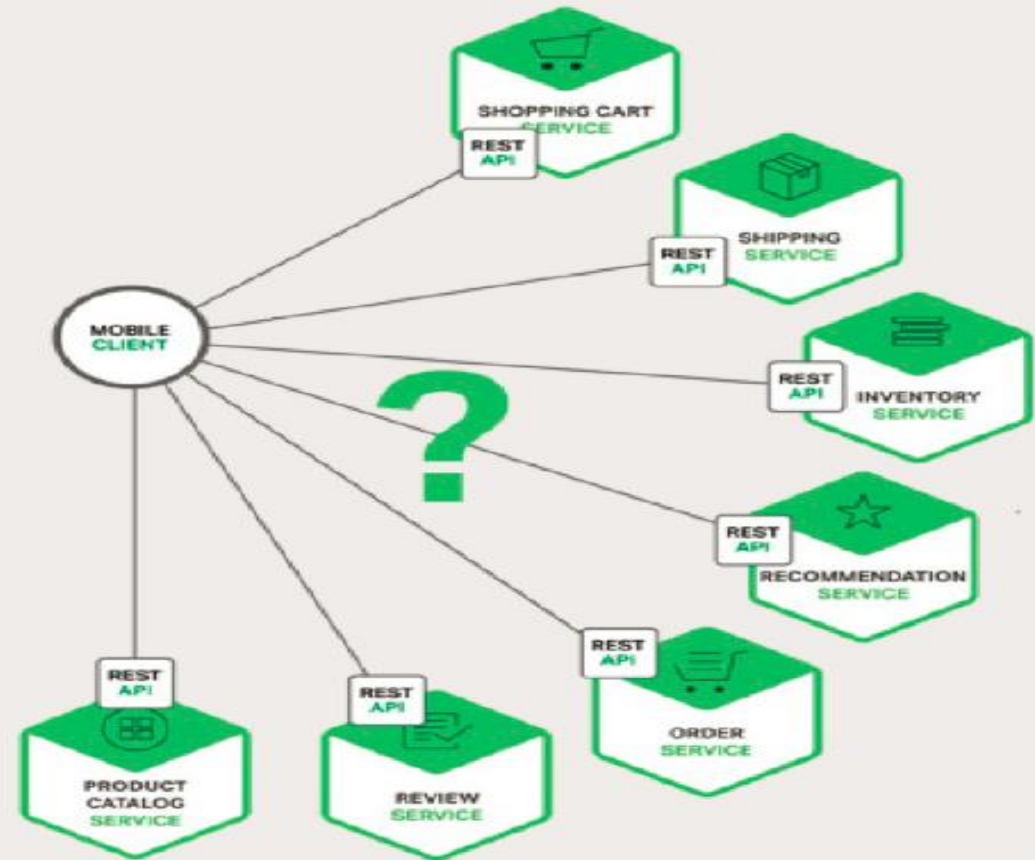
Low Coupling b/w Bounded Contexts

- ▶ Data governed by a bounded context would be exposed through main Web APIs
 - ▶ Front-end applications need to communicate with several endpoints (Bad due to latency and TLS factors).
 - ▶ If a single transaction involves several sub-domains, client-side applications need to communicate with all of them.
 - ▶ Cross cutting concerns (Authentication, Logging, Security, Monitoring) needs to be implemented across all web APIs.
 - ▶ Each web API needs to have its own SSL and Domain name (Added Cost).
 - ▶ Cross-Origin Resource Sharing (CORS) gets more complex

Direct Client-to-Microservice Communication



1. ORDER HISTORY
2. REVIEWS
3. BASIC PRODUCT INFO
4. RECOMMENDATION
5. INVENTORY
6. SHIPPING

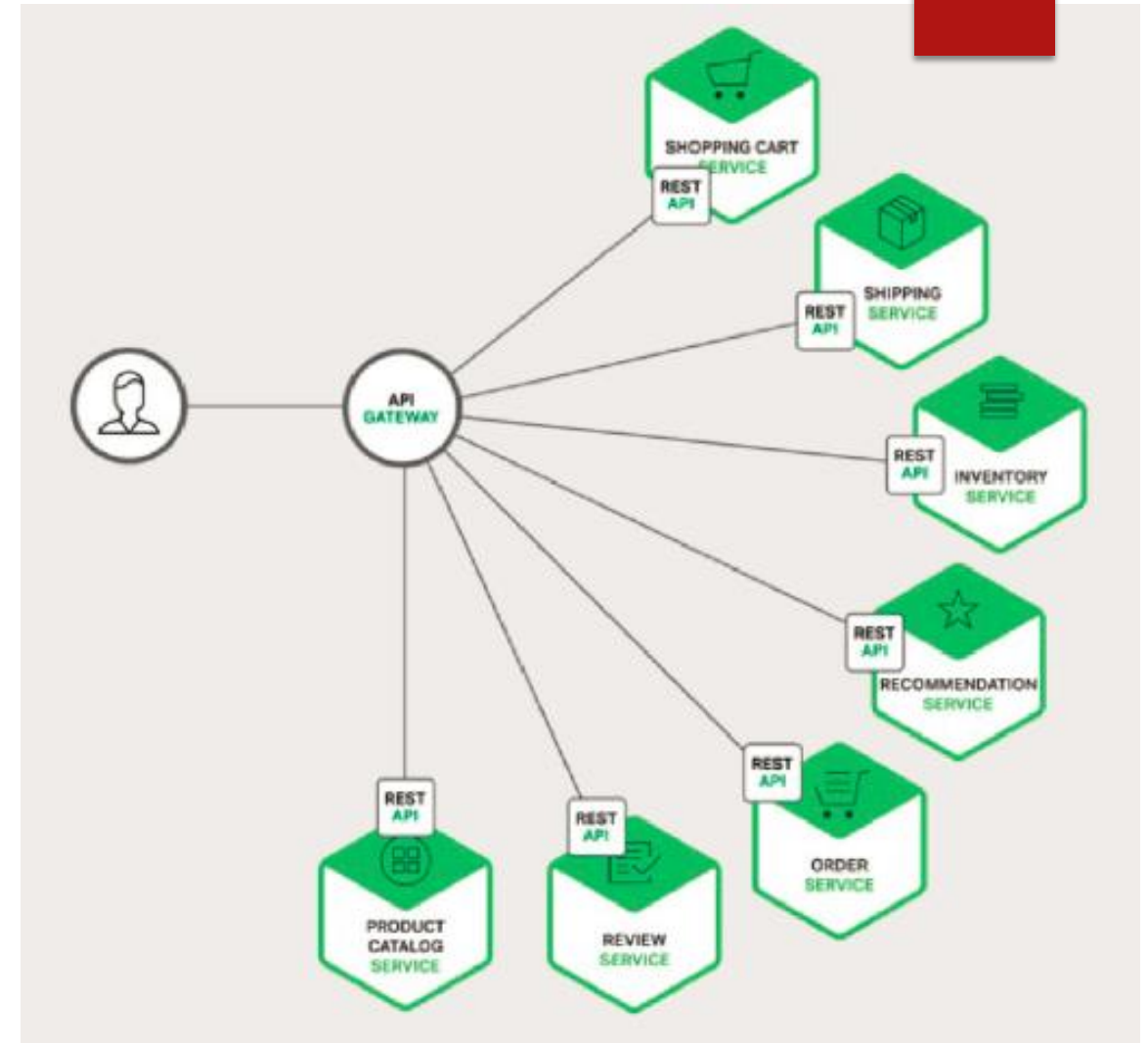


Direct Client-to-Microservice Communication

- ▶ Client could make that many requests over a LAN
- ▶ Incompatible Protocols , Protocols may not be web friendly
 - ▶ One service might use Thrift binary RPC while another
 - ▶ Service might use the AMQP messaging protocol
- ▶ Direct Communication makes difficult to refactor the microservices

API Gateway Pattern

- ▶ Request Routing
- ▶ Composition
- ▶ Protocol Translation
- ▶ Aggregating the results.
- ▶ Encapsulates the internal structure of the application
- ▶ API gateways are mainly useful in encapsulating the existence of the bounded context APIs from clients



Api Gateway - Design issues

- ▶ Performance and Scalability
 - ▶ Build the API Gateway on a platform that supports asynchronous, non-blocking I/O.
 - ▶ JVM
 - ▶ NIO-based frameworks such Natty
 - ▶ Vertx
 - ▶ Spring Reactor
 - ▶ Jboss undertow
 - ▶ Node.js
 - ▶ NGINX Plus

Api Gateway - Design issues

- ▶ API Composition
 - ▶ Dependencies between requests
 - ▶ The API Gateway might first need to validate the request by calling an authentication service before routing the request to a backend service
 - ▶ Not –feasible to implement using Callback hell
 - ▶ Requires Reactive Programming Model
 - ▶ Promises
 - ▶ CompletableFuture in java 8
 - ▶ Reactive Extensions. Net
 - ▶ Future in Scala,

Service Invocation

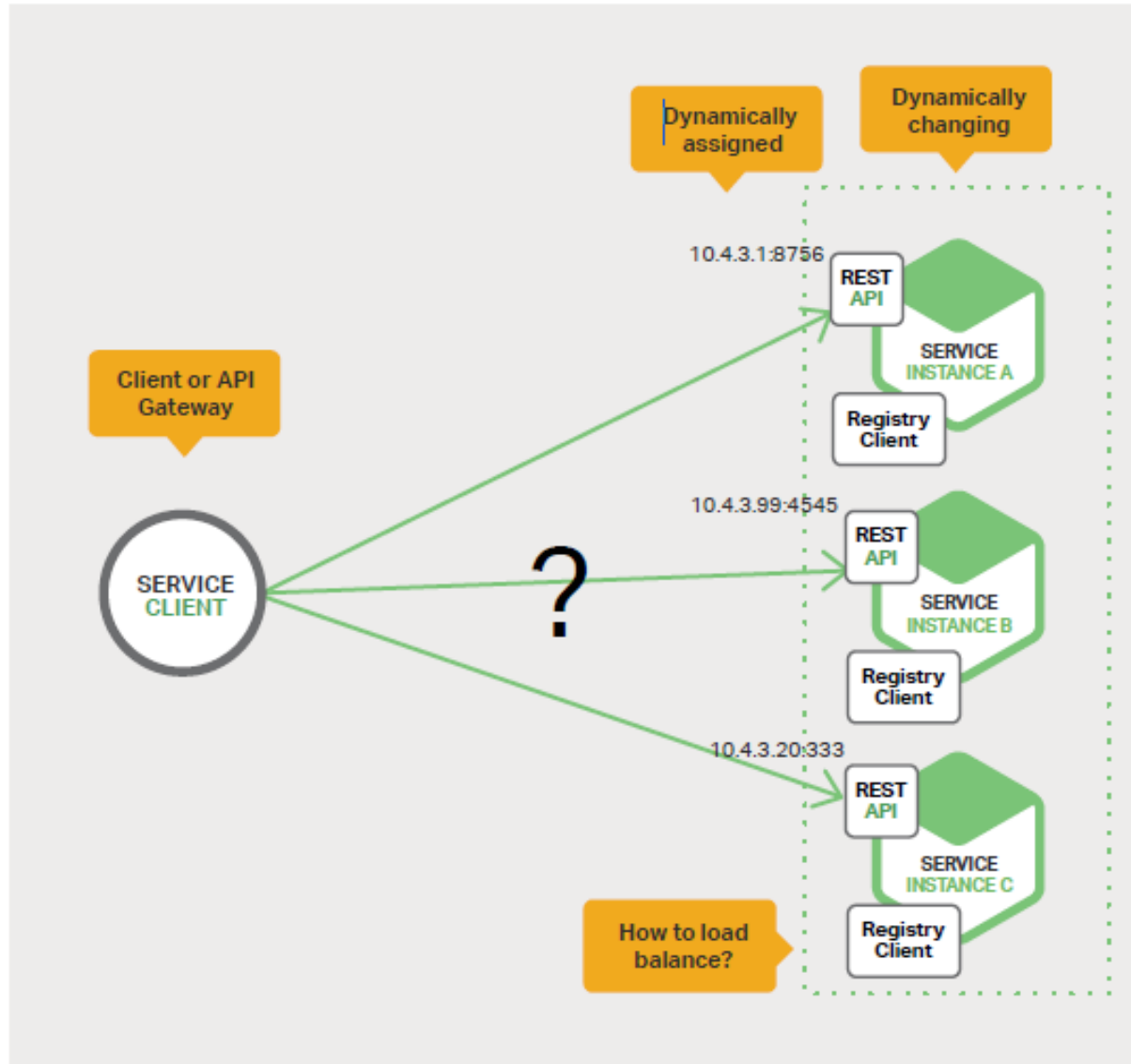
- ▶ Inter-process communication mechanism
- ▶ Styles of inter-process communication
 - ▶ An asynchronous, messaging-based mechanism
 - ▶ Broker Less
 - ▶ JMS , AMQP , ZeroMq
 - ▶ Synchronous Communication
 - ▶ HTTP or Thrift

API gateway pattern drawbacks

- ▶ Increased complexity - the API gateway is yet another moving part that must be developed, deployed and managed
- ▶ Increased response time due to the additional network hop through the API gateway

API – Gateway Related Patterns

- ▶ The API gateway must use either the [Client-side Discovery pattern](#) or [Server-side Discovery pattern](#) to route requests to available service instances.
- ▶ An API Gateway will use a [Circuit Breaker](#) to invoke services
- ▶ An API gateway often implements the [API Composition pattern](#)



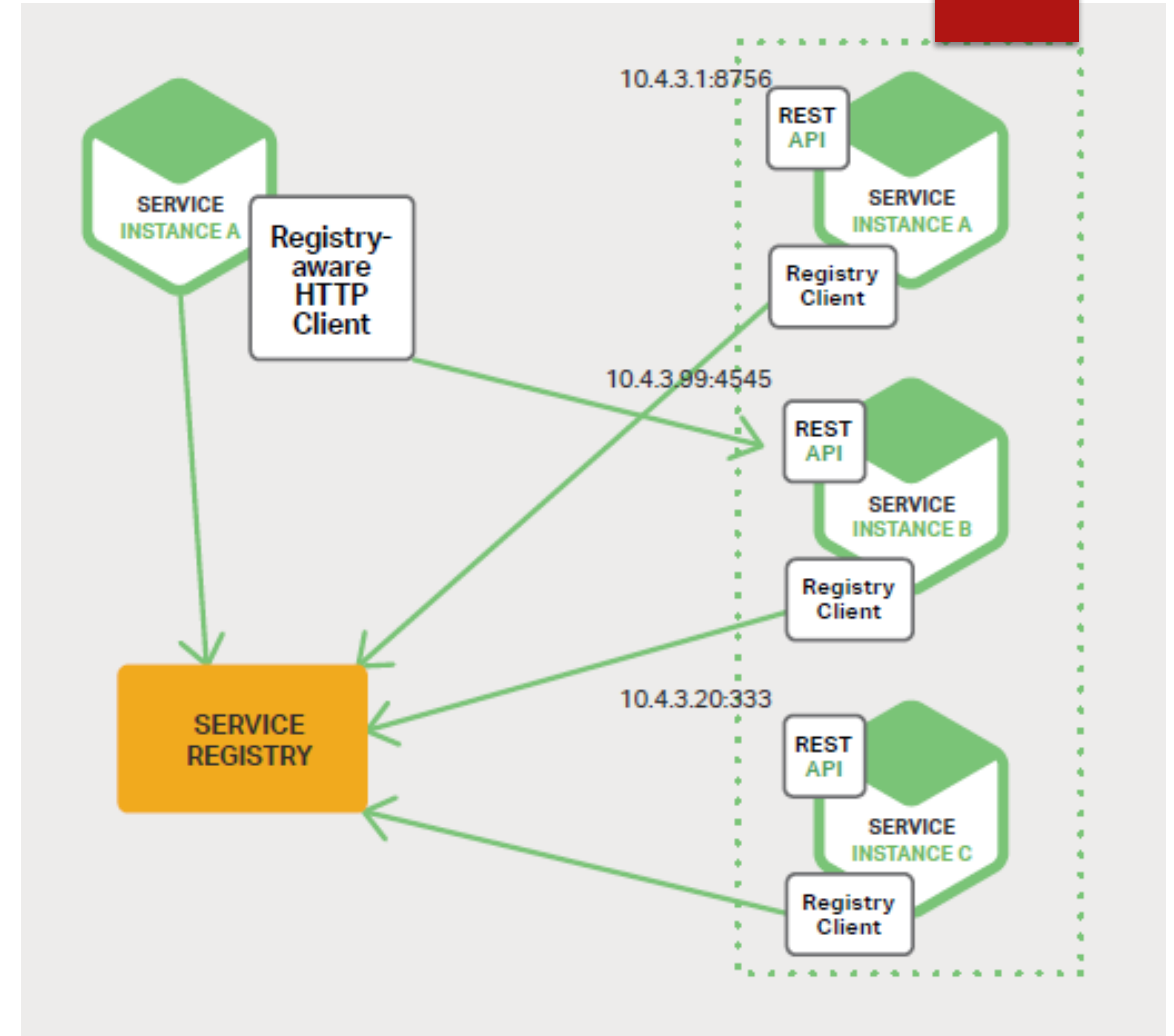
Why Service Discovery

Service Discovery

- ▶ The API Gateway needs to know the location (IP address and port) of each microservice with which it communicates
 - ▶ Don't hardwire the locations
- ▶ Server-side-discovery
- ▶ Client-side-discovery
- ▶ Use Service Registry

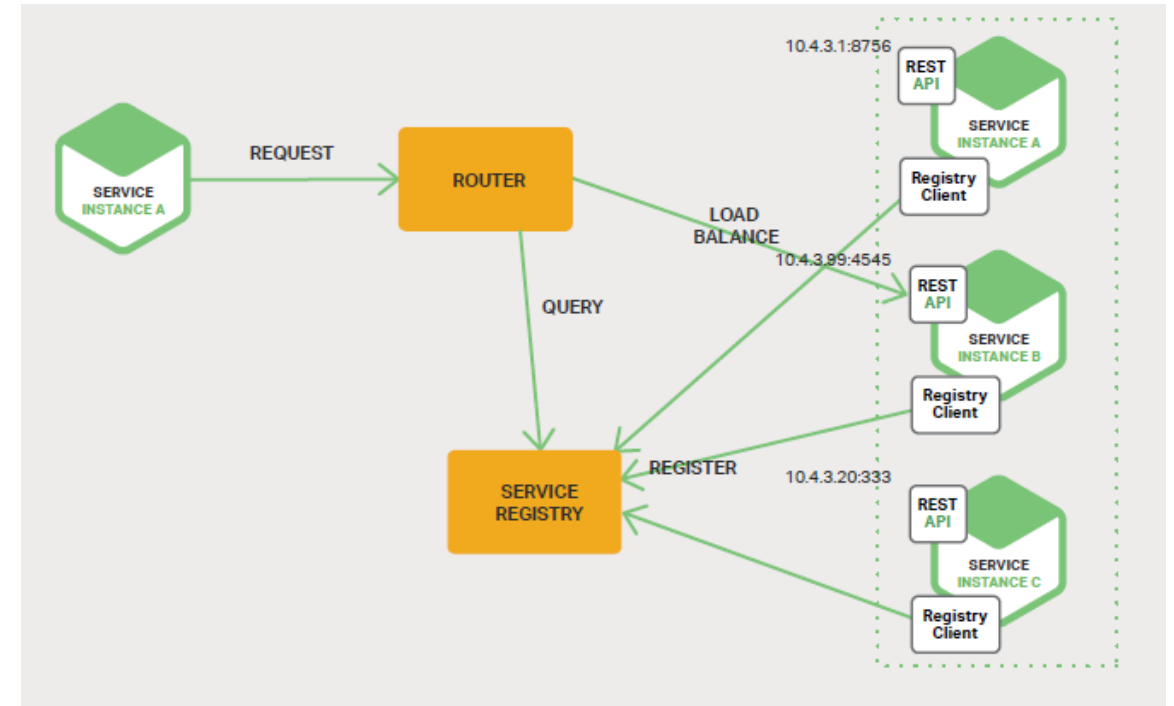
The Client-Side Discovery Pattern

- ▶ Client is responsible for determining the network locations of available service instances and load balancing requests across them
- ▶ Client uses a service registry
- ▶ The client then uses a load-balancing algorithm to select one of the available service instances and makes a request.
- ▶ This pattern is relatively straightforward and, except for the service registry



The Server-Side Discovery Pattern

- ▶ The client makes a request to a service via a load balancer
- ▶ The load balancer queries the service registry and routes each request to an available service instance
- ▶ Details of discovery are abstracted away from the client.



The Service Registry

- ▶ It is a database containing the network locations of service instances
- ▶ A service registry needs to be highly available and up to date
- ▶ Clients can cache network locations obtained from the service registry
- ▶ Some service registry consists of a cluster of servers that use a replication protocol to maintain consistency

examples of service registries`

- ▶ Netflix Eureka
 - ▶ Provides a REST API for registering and querying service instances.
 - ▶ Service instance registers its network location using a POST request. Every 30 seconds it must refresh its registration using a PUT request
- ▶ Etcd
 - ▶ A highly available, distributed, consistent, key-value store that is used for shared configuration and service discovery
 - ▶ Kubernetes and Cloud Foundry uses etcd
- ▶ Consul
 - ▶ It provides an API that allows clients to register and discover services
 - ▶ Consul can perform health checks to determine service availability
- ▶ Apache ZooKeeper
 - ▶ A widely used, high-performance coordination service for distributed applications
- ▶ Kubernetes, Marathon, Azure and AWS do not have an explicit service registry. Instead, the service registry is just a built-in part of the infrastructure

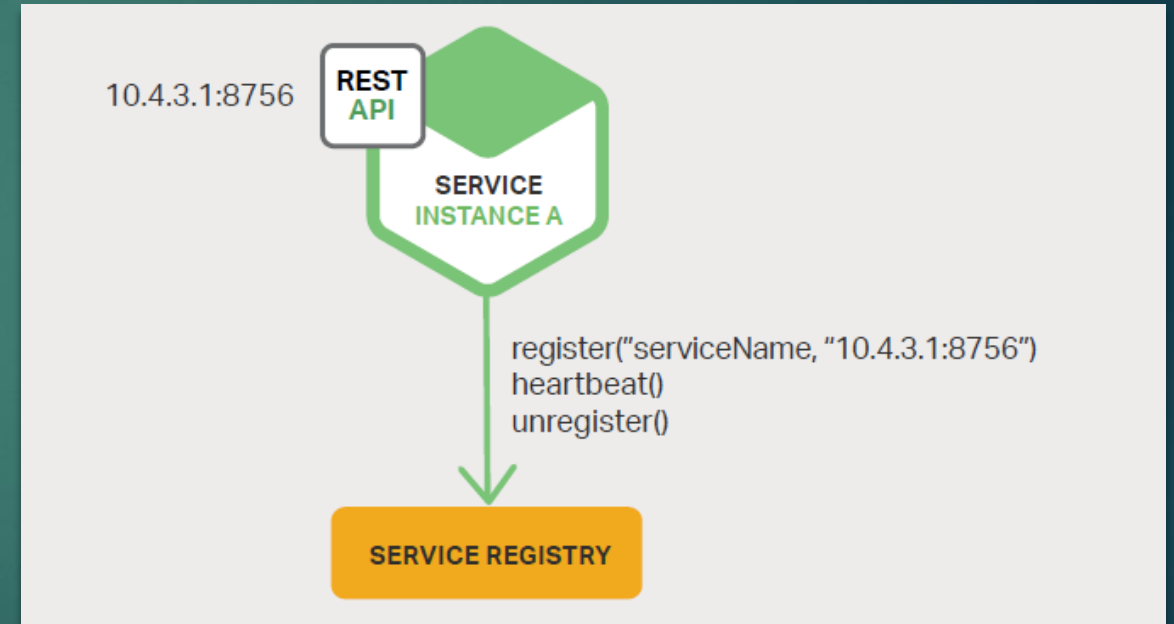
Service Registration Options

Self
Registration
Pattern

Third Party
Registration
Pattern

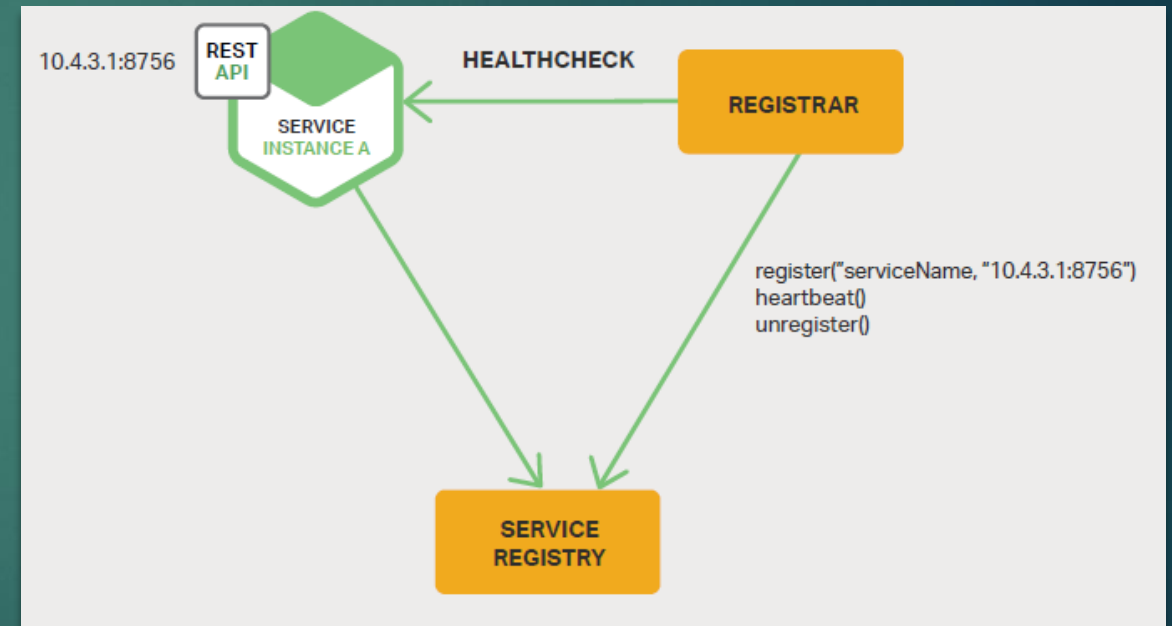
Self Registration Pattern

- ▶ A service instance is responsible for registering and unregistering itself with the service registry
- ▶ A service instance sends heartbeat requests to prevent its registration from expiring
- ▶ Couples the service instances to the service registry



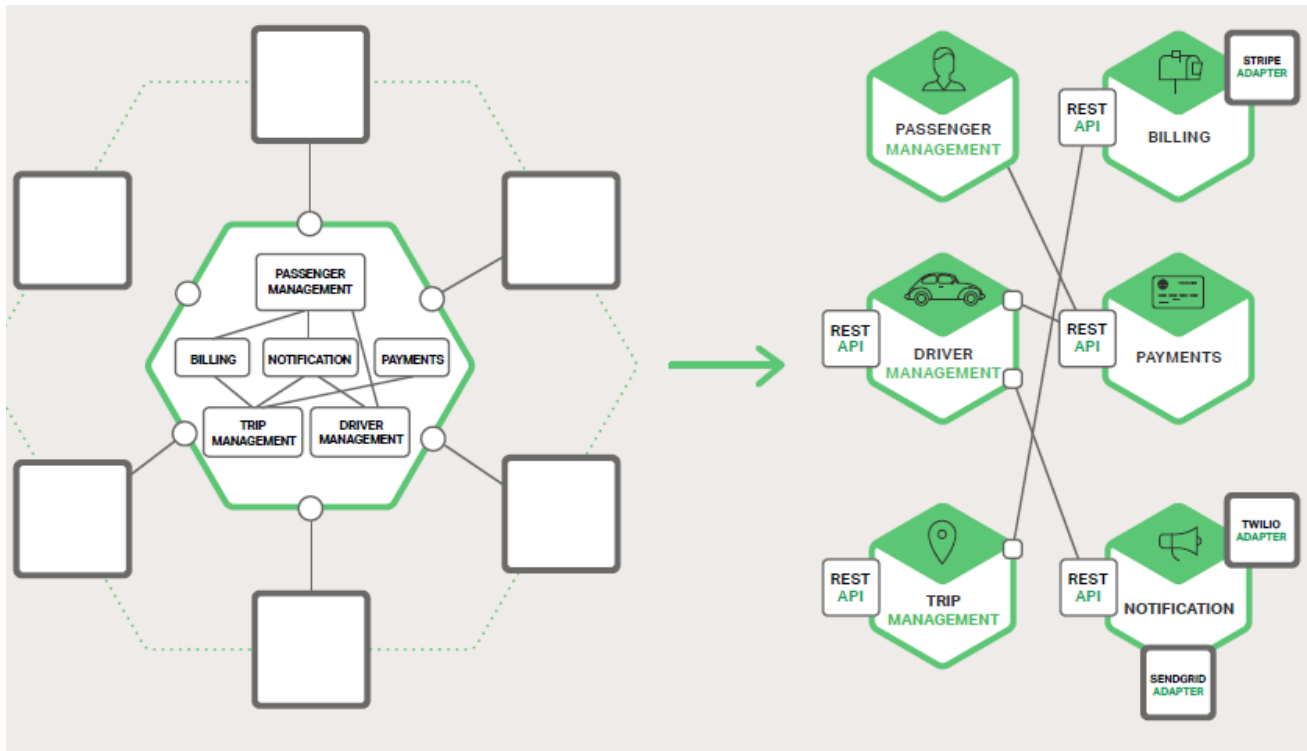
The Third-Party Registration Pattern

- ▶ A system component known as the *service registrar* handles the registration
- ▶ The service registrar tracks changes to the set of running instances by either polling the deployment environment or subscribing to events
 - ▶ When it notices a newly available service instance, it registers the instance with the service registry
- ▶ The service registrar unregisters terminated service instances.
- ▶ Services are decoupled from the service registry



Handling Partial Failures

- ▶ Partial Failure
 - ▶ This issue arises in all distributed systems whenever one service calls another service that is either responding slowly or is unavailable
- ▶ Ex:- if the recommendation service is unresponsive in the product details scenario, the API Gateway should return the rest of the product details to the client since they are still useful to the user
- ▶ The API Gateway could also return cached data if that is available
 - ▶ For example, since product prices change infrequently, the API Gateway could return cached pricing data if the pricing service is unavailable
- ▶ Use Circuit Breaker Pattern



Inter Process Communication

Interaction Styles and Interaction Types

- ▶ Interaction – Styles

- ▶ One to One

- ▶ Each client request is processed by exactly one service instance

- ▶ One-to-many

- ▶ Each request is processed by multiple service instances

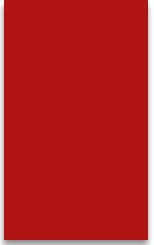
- ▶ Interaction Types

- ▶ Synchronous

- ▶ The client expects a timely response from the service and might even block while it waits.

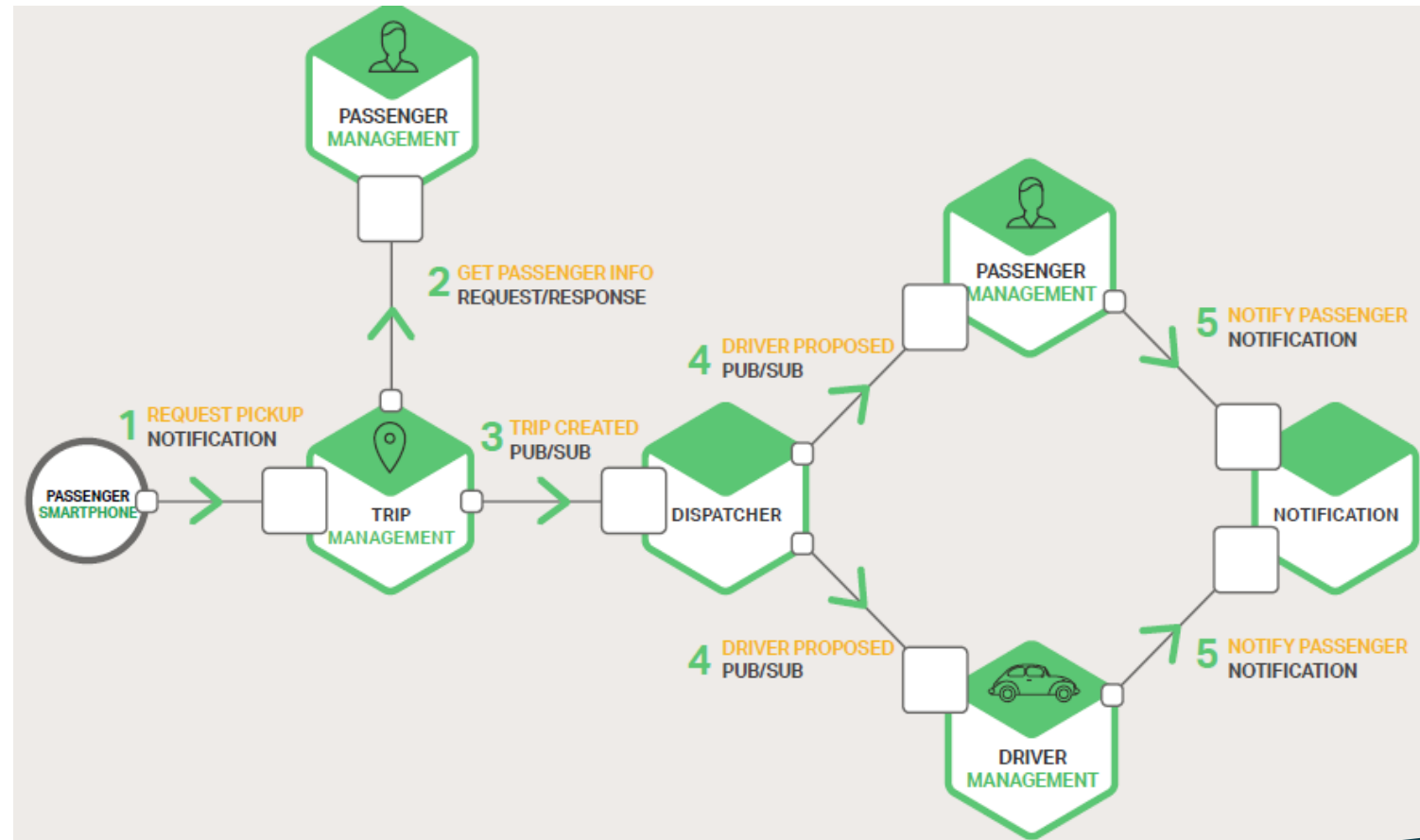
- ▶ Asynchronous

- ▶ The client doesn't block while waiting for a response, and the response, if any, isn't necessarily sent immediately



	ONE-TO-ONE	ONE-TO-MANY
SYNCHRONOUS	Request/response	—
ASYNCHRONOUS	Notification	Publish/subscribe
	Request/async response	Publish/async responses

Interaction communication Style



IPC mechanisms for service interactions

Handling Partial Failure

- ▶ Network timeouts
 - ▶ Never block indefinitely and always use timeouts when waiting for a response
 - ▶ Using timeouts ensures that resources are never tied up indefinitely
- ▶ Limiting the number of outstanding requests
 - ▶ Impose an upper bound on the number of outstanding requests that a client can have with a particular service
- ▶ Circuit Breaker Pattern
 - ▶ Track the number of successful and failed requests
- ▶ Provide fallbacks
 - ▶ Return cached data or a default value, such as an empty set of recommendations
- ▶ Hystrix , Poly ,

Strategies for Handling Partial Failures

- ▶ Retry
 - ▶ Allows callers to retry operations in the expectation that many faults are transient and may self-correct: the operation may succeed if retried, possibly after a short delay.
 - ▶ Exponential backoff
 - ▶ Jitter
- ▶ Circuit Breaker
 - ▶ Prevents calls when a configurable fault threshold is exceeded
- ▶ Bulkhead Isolation
 - ▶ - A parallelism limit - around one stream of calls
- ▶ Timeout
 - ▶ Allows callers to walk away from a pending call
- ▶ Fallback

What is HttpClient factory?

- ▶ Allows you to name and configure logical HttpClient
- ▶ Provides configurable logging (via ILogger) for all requests and responses performed by clients created with the factory;
- ▶ Provides a simple API for adding middleware to outgoing calls, be that for logging, authorization, service discovery, or resilience with Polly.
- ▶ Manages the pooling and lifetime of underlying HttpClientMessageHandler instances
- ▶ Consumption Patterns
 - ▶ Basic usage
 - ▶ Named clients
 - ▶ Typed clients
 - ▶ Generated clients

IPC Technologies

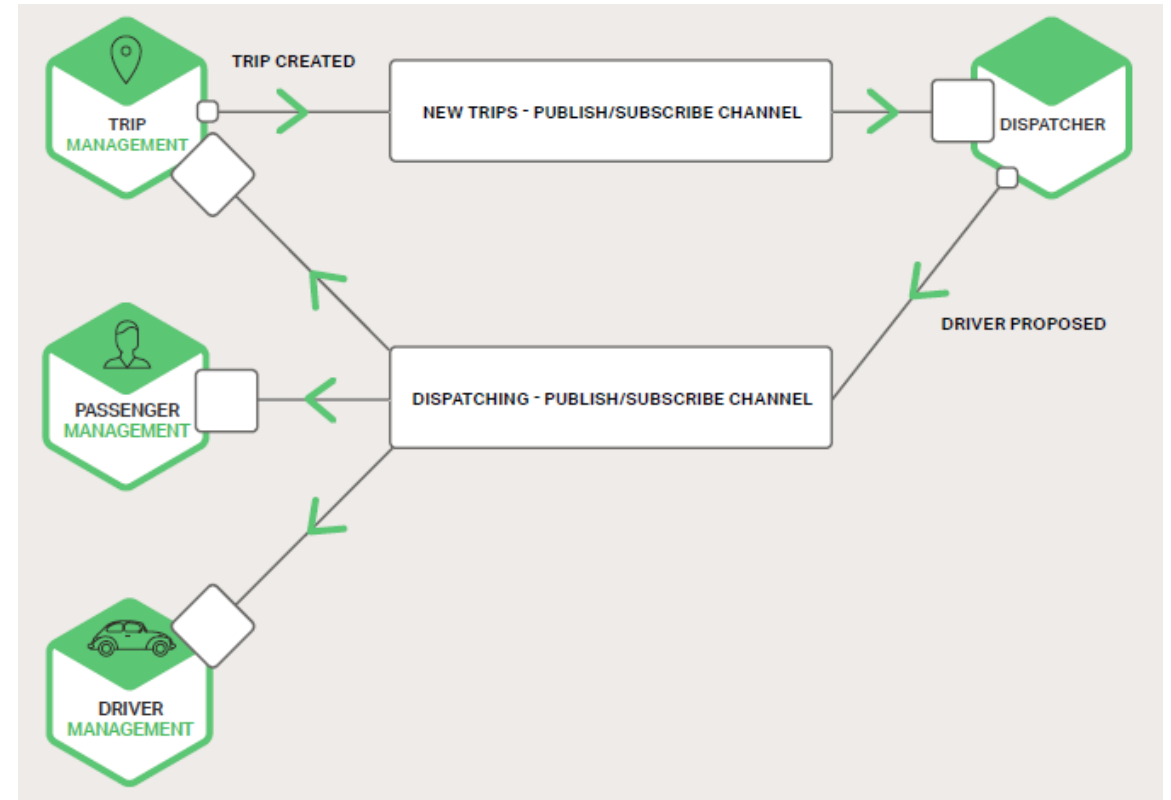
- ▶ Synchronous Request / Reply
- ▶ Asynchronous Message Based Communication

Asynchronous, Message-Based Communication

- ▶ Point To Point Communication
 - ▶ A point-to-point channel delivers a message to exactly one of the consumers that are reading from the channel
- ▶ Publish and Subscribe
 - ▶ A publish-subscribe channel delivers each message to all of the attached consumers.

Publisher – Subscriber

- ▶ RabbitMQ
- ▶ Apache Kafka
- ▶ Apache ActiveMQ
- ▶ NSQ



Advantages and Disadvantages of using messaging:

Advantages

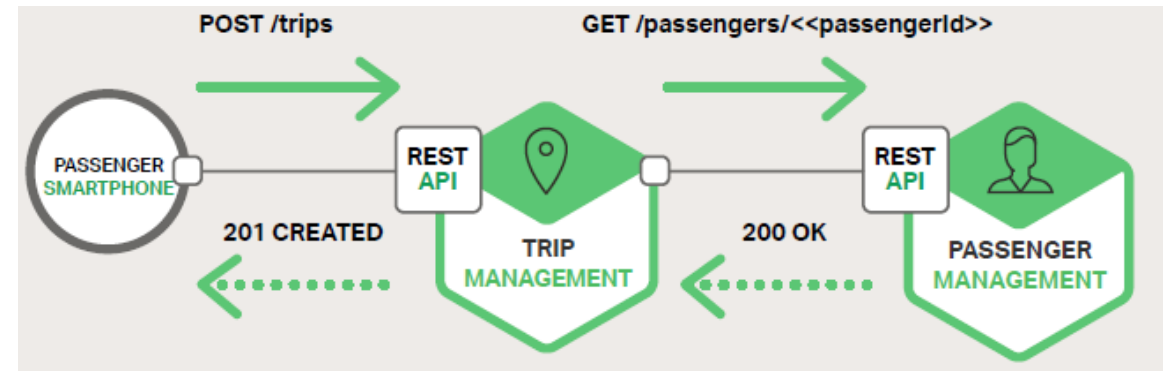
- ▶ Decouples the client from the service
- ▶ Message buffering
- ▶ Flexible client-service interactions
- ▶ Explicit inter-process communication

Disadvantages

- ▶ Additional operational complexity
- ▶ Complexity of implementing request/response-based interaction
 - ▶ Each request message must contain a reply channel identifier and a correlation identifier

Synchronous, Request/Response IPC

- ▶ Protocols
 - ▶ HTTP based REST
 - ▶ Thrift



HTTP-based APIs are RESTful ?

- ▶ Level 0

- ▶ Clients of a level 0 API invoke the service by making HTTP POST requests to its sole URL endpoint. Each request specifies the action to perform, the target of the action (for example, the business object), and any parameters.

- ▶ Level 1

- ▶ A level 1 API supports the idea of resources. To perform an action on a resource, a client makes a POST request that specifies the action to perform and any parameters.

- ▶ Level 2

- ▶ A level 2 API uses HTTP verbs to perform actions: GET to retrieve, POST to create, and PUT to update. The request query parameters and body, if any, specify the action's parameters. This enables services to leverage web infrastructure such as caching for GET requests

- ▶ Level 3

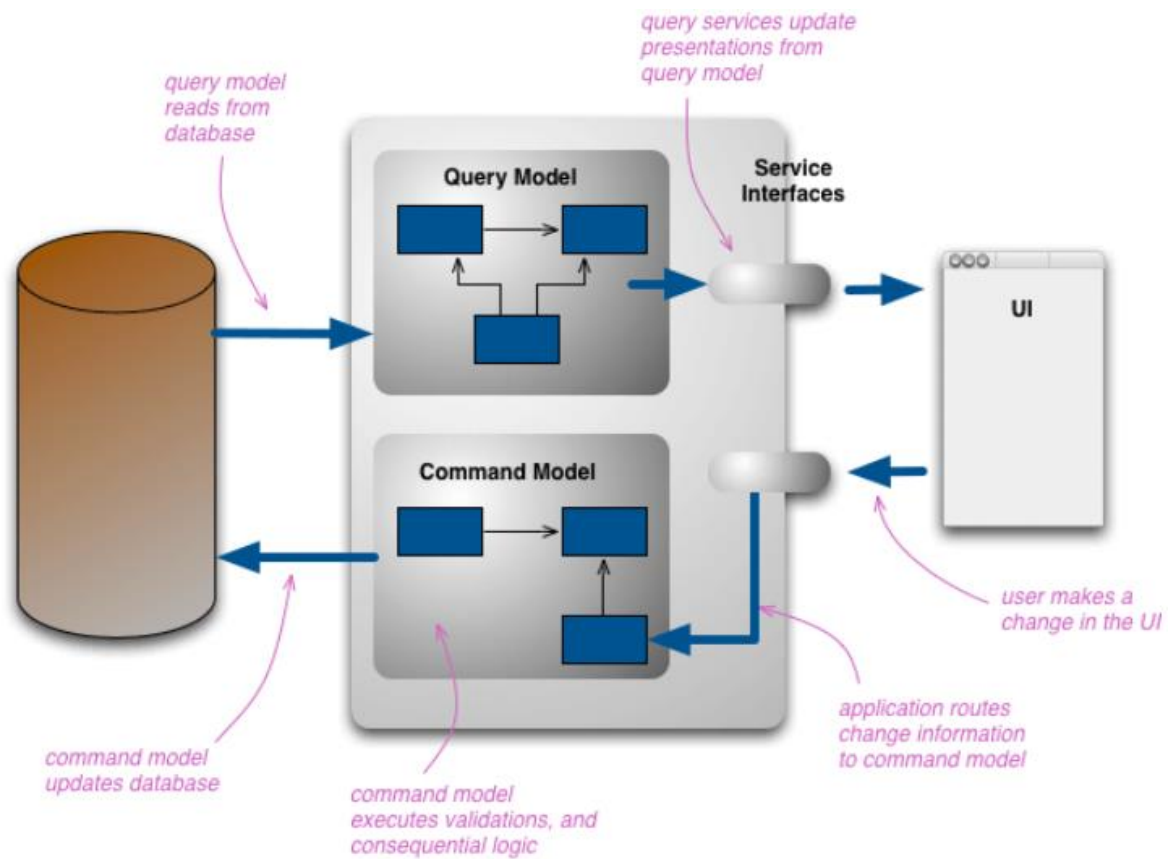
- ▶ The design of a level 3 API is based on the HATEOAS (Hypertext As The Engine Of Application State).
 - ▶ The representation of a resource returned by a GET request contains links for performing the allowable actions on that resource

Event Sourcing and CQRS

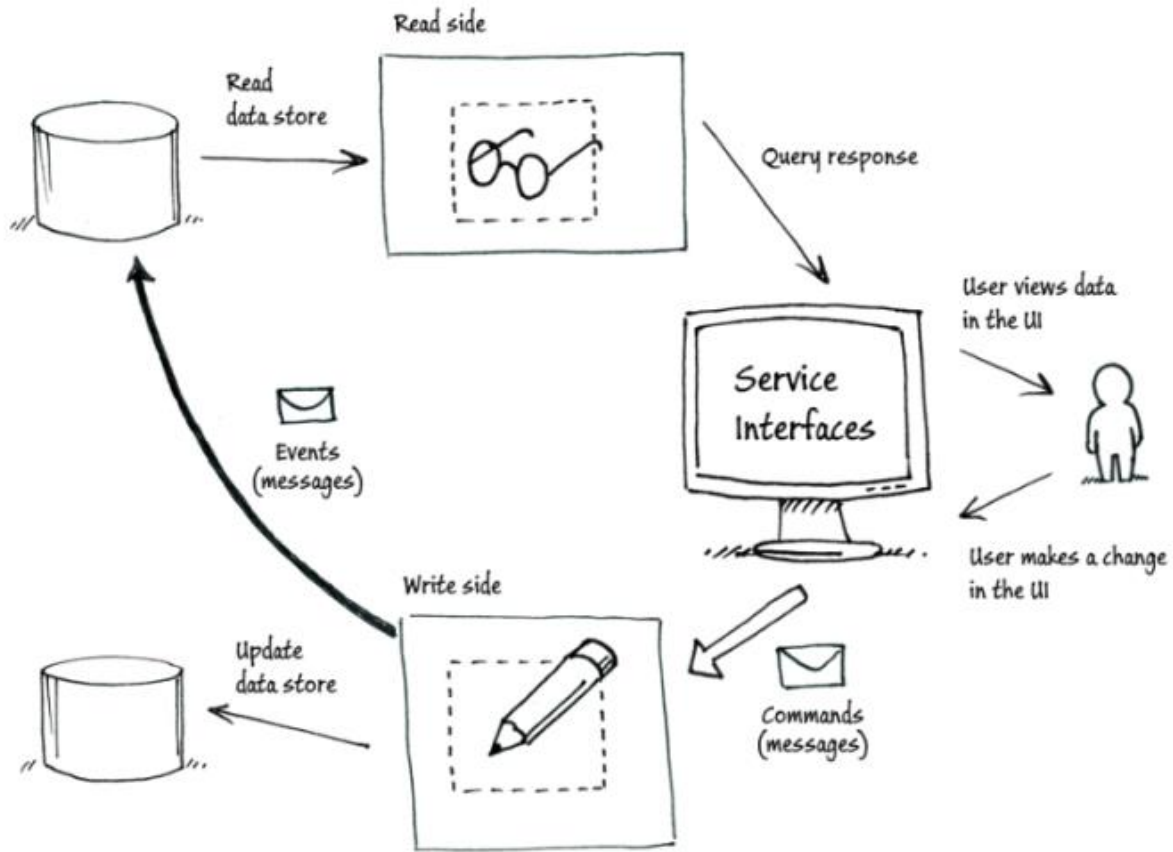
- ▶ Separates the concern of state management from the concern of receiving stimuli that result in state changes.
- ▶ Event Source System Concerns
- ▶ Ordering and Proper time management
 - ▶ Event streams are ordered
 - ▶ Performing calculations against the same set of events but in a different sequence will produce different output
- ▶ Idempotent
 - ▶ Any function that operates on an event stream must always return the exact same result for identical ordered event streams
- ▶ Isolated
 - ▶ Any function that produces a result based on an event stream cannot make use of external information. All data required for calculations must be present in the events

Eventual Consistency

- ▶ sacrifice immediate consistency for scale
- ▶ *complexity leak*
 - ▶ The internal workings (or limitations) of our system could leak out of our service and force our clients to bear the burden of additional complexity



CQRS



Source: [Introducing Event Sourcing from MSDN](#)

EventSourcing