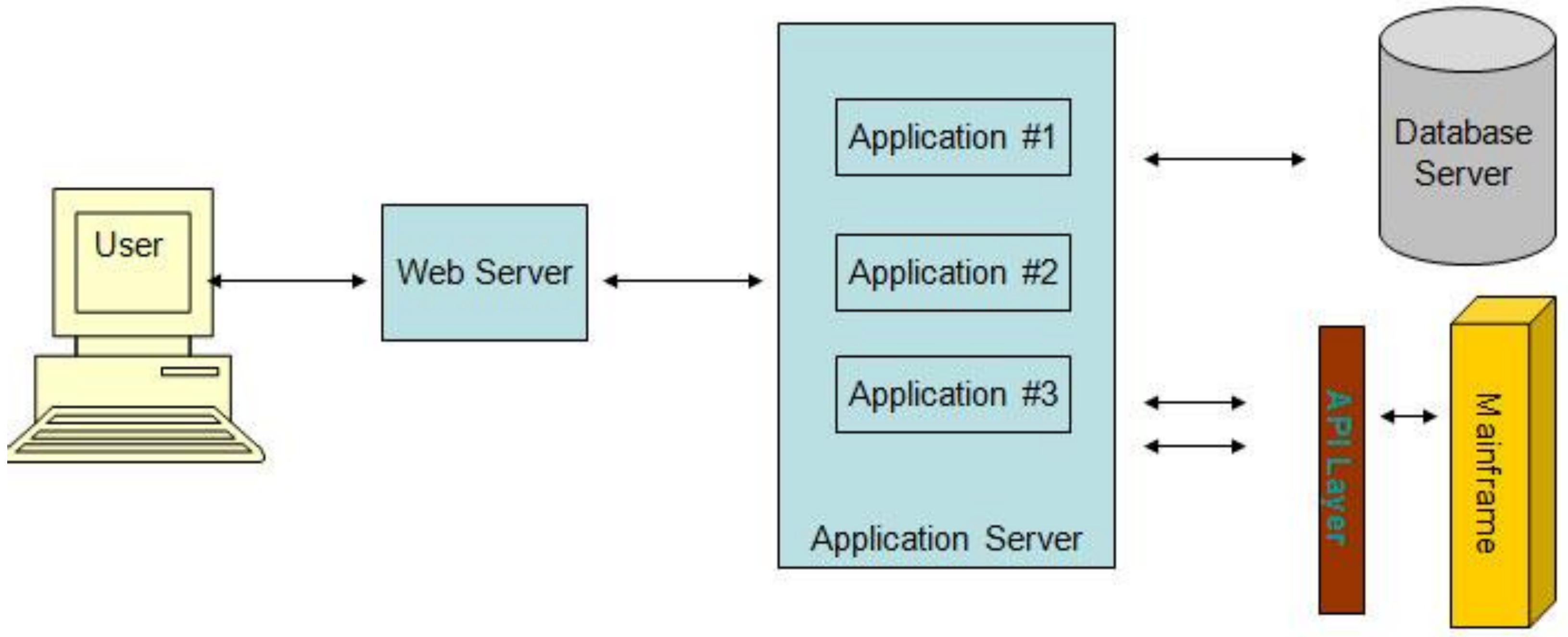
Microservices

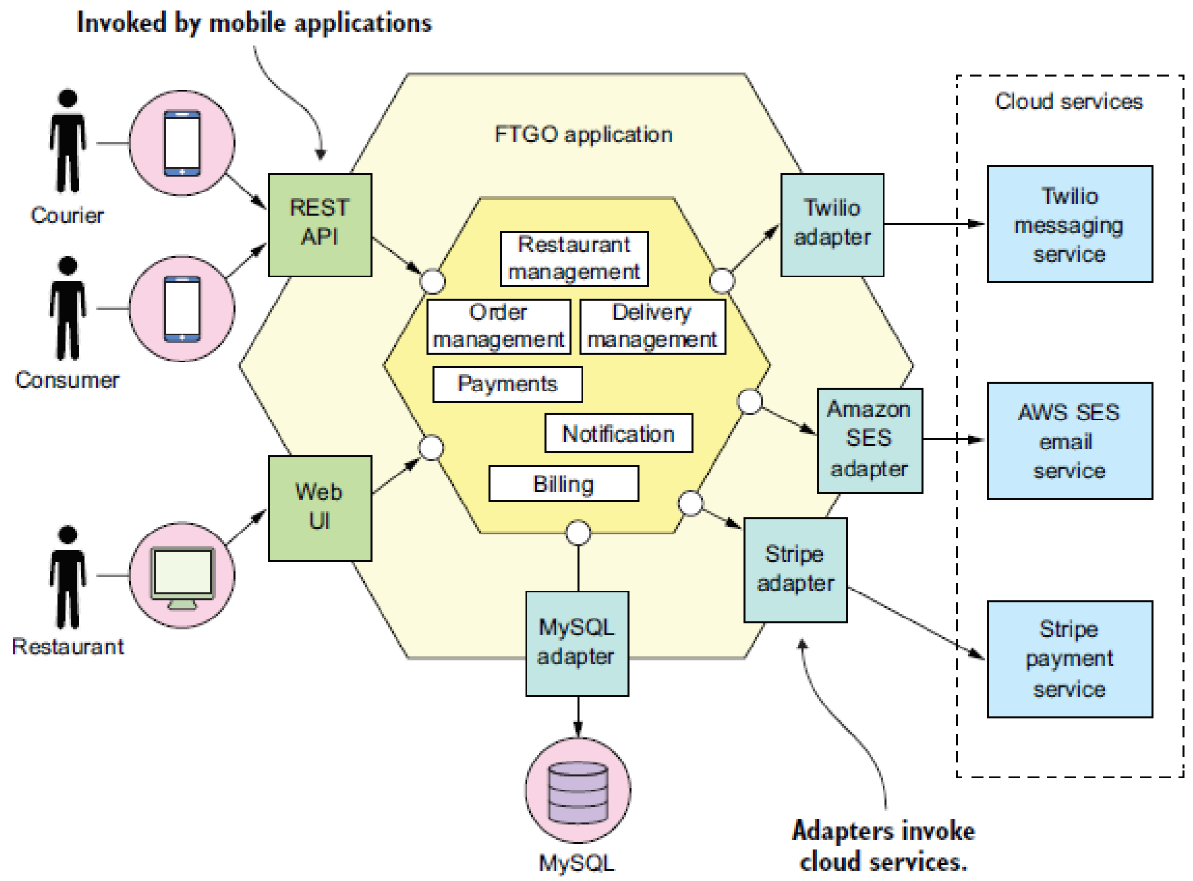
Monolithic applications with a multi-tiered architecture



Layered Architecture in Monolithic applications

- It organizes the application’s classes into the following tiers or layers:
- *Presentation layer*—Contains code that implements the user interface or external APIs
- *Business logic layer*—Contains the business logic
- *Persistence layer*—Implements the logic of interacting with the database
- The layered architecture is a great example of an architectural style, but it does have some significant drawbacks:
- *Single presentation layer*—It doesn’t represent the fact that an application is likely to be invoked by more than just a single system.
- *Single persistence layer*—It doesn’t represent the fact that an application is likely to interact with more than just a single database.
- *Defines the business logic layer as depending on the persistence layer*—In theory, this dependency prevents you from testing the business logic without the database.

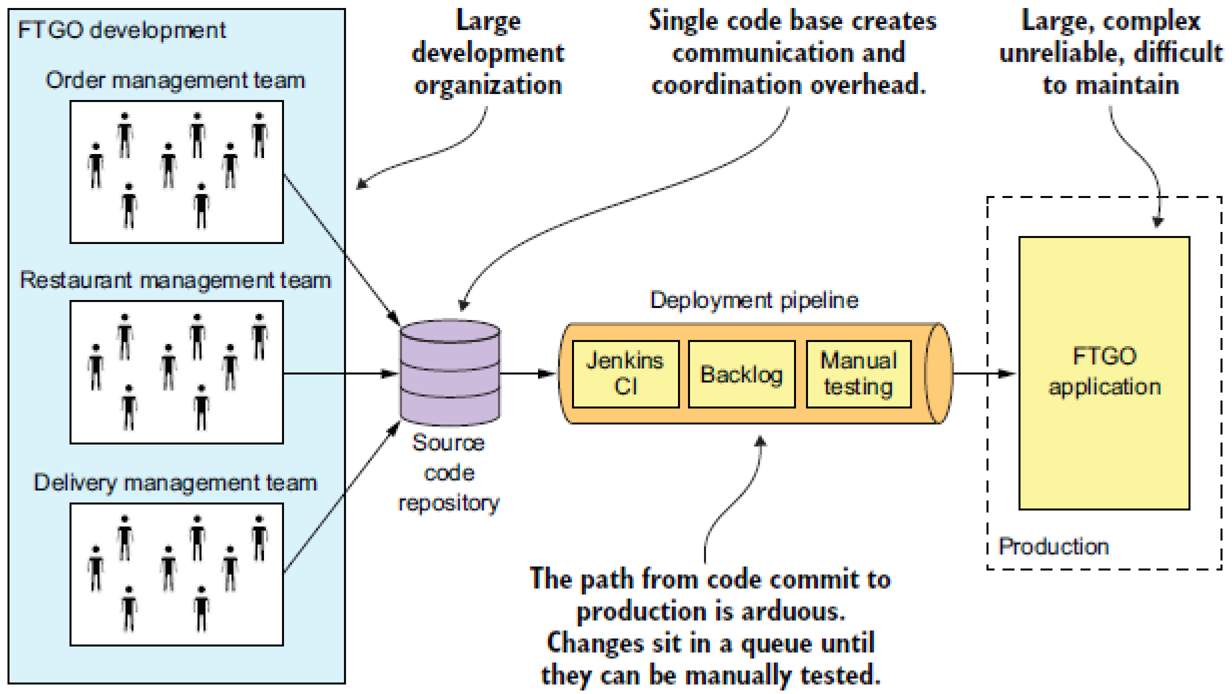
The Food to Go, Inc.(FTGO) Application



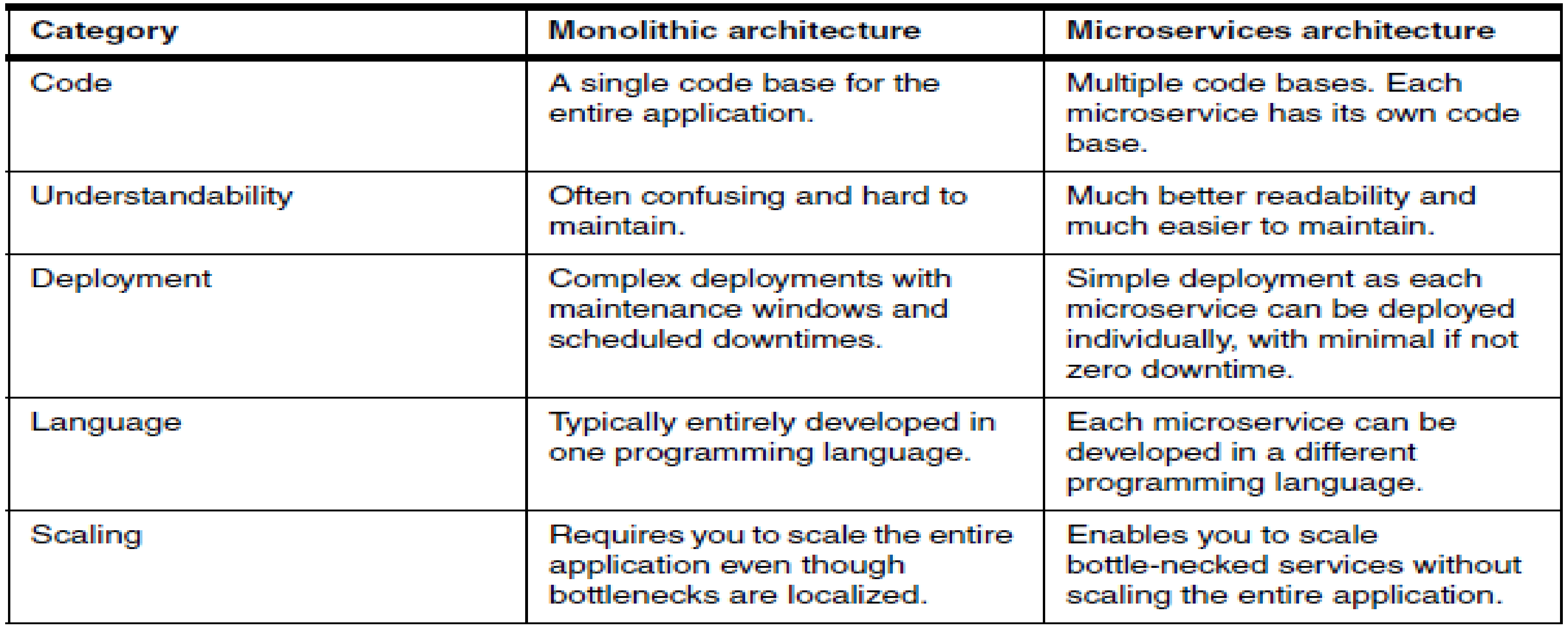
The benefits of the monolithic architecture

- ⍰ *Simple to develop*—IDEs and other developer tools are focused on building a single application.
- ⍰ *Easy to make radical changes to the application*—You can change the code and the database schema, build, and deploy.
- ⍰ *Straightforward to test*—The developers wrote end-to-end tests that launched the application, invoked the REST API, and tested the UI with Selenium.
- ⍰ *Straightforward to deploy*—All a developer had to do was copy the WAR file to a server that had Tomcat installed.
- ⍰ *Easy to scale*—FTGO ran multiple instances of the application behind a load balancer.

Drawbacks of Monolithic Application


FTGO development
Order management team
Restaurant management team
Delivery management team
Large development organization
Single code base creates communication and coordination overhead.
Large, complex unreliable, difficult to maintain
Source code repository
Deployment pipeline
Jenkins CI
Backlog
Manual testing
FTGO application
Production
The path from code commit to production is arduous. Changes sit in a queue until they can be manually tested.

Comparing microservices and monolithic architectures



Category	Monolithic architecture	Microservices architecture
Code	A single code base for the entire application.	Multiple code bases. Each microservice has its own code base.
Understandability	Often confusing and hard to maintain.	Much better readability and much easier to maintain.
Deployment	Complex deployments with maintenance windows and scheduled downtimes.	Simple deployment as each microservice can be deployed individually, with minimal if not zero downtime.
Language	Typically entirely developed in one programming language.	Each microservice can be developed in a different programming language.
Scaling	Requires you to scale the entire application even though bottlenecks are localized.	Enables you to scale bottle-necked services without scaling the entire application.

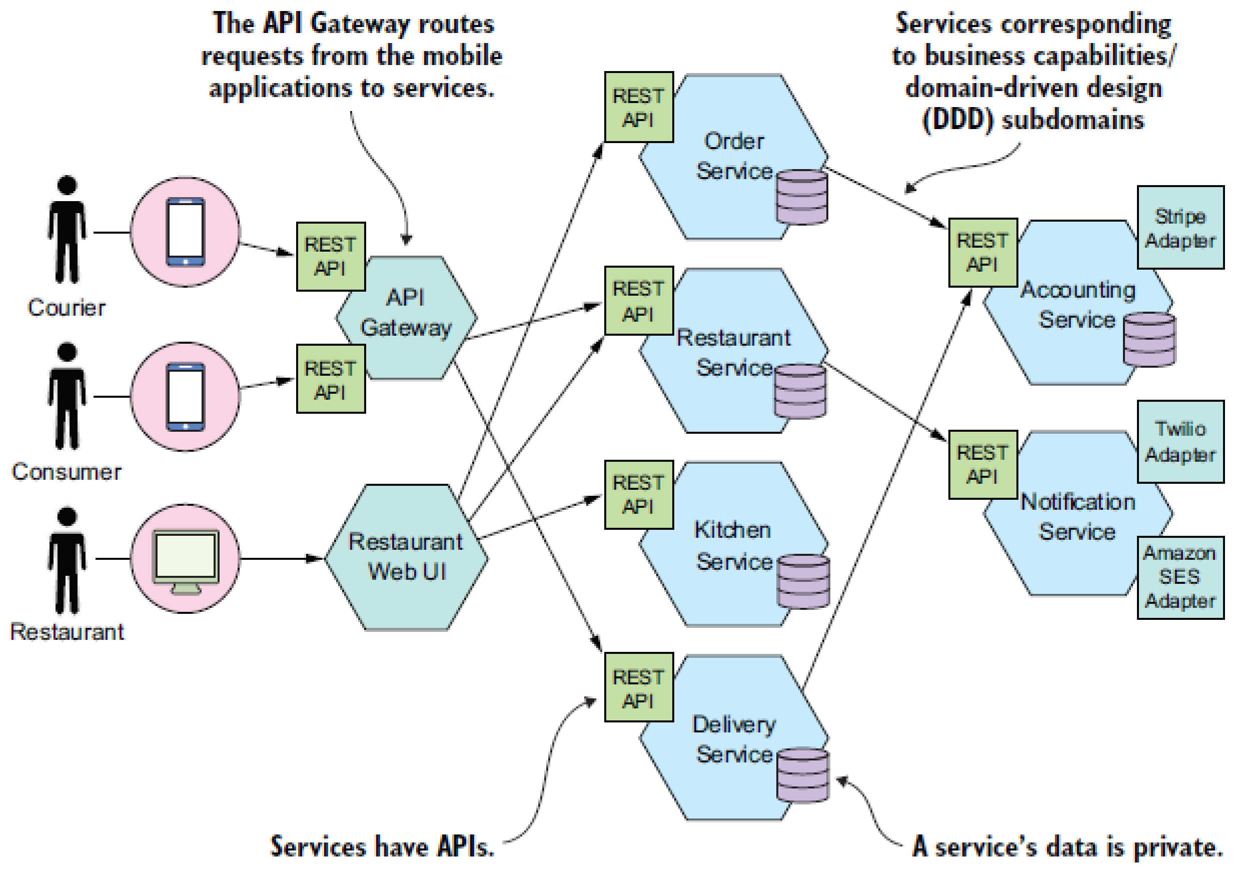
What are microservices

- Microservices is an architecture style, in which large complex software applications are composed of one or more services.
- Microservice can be deployed independently of one another and are loosely coupled.
- Each of these microservices focuses on completing one task only and does that one task really well. In all cases, that one task represents a small business capability.
- Microservices can be developed in any programming language.
- They communicate with each other using language-neutral application programming interfaces (APIs) such as Representational State Transfer (REST).
- Microservices also have a bounded context. They don’t need to know anything about underlying implementation or architecture of other microservices.

Characteristics of a Microservice

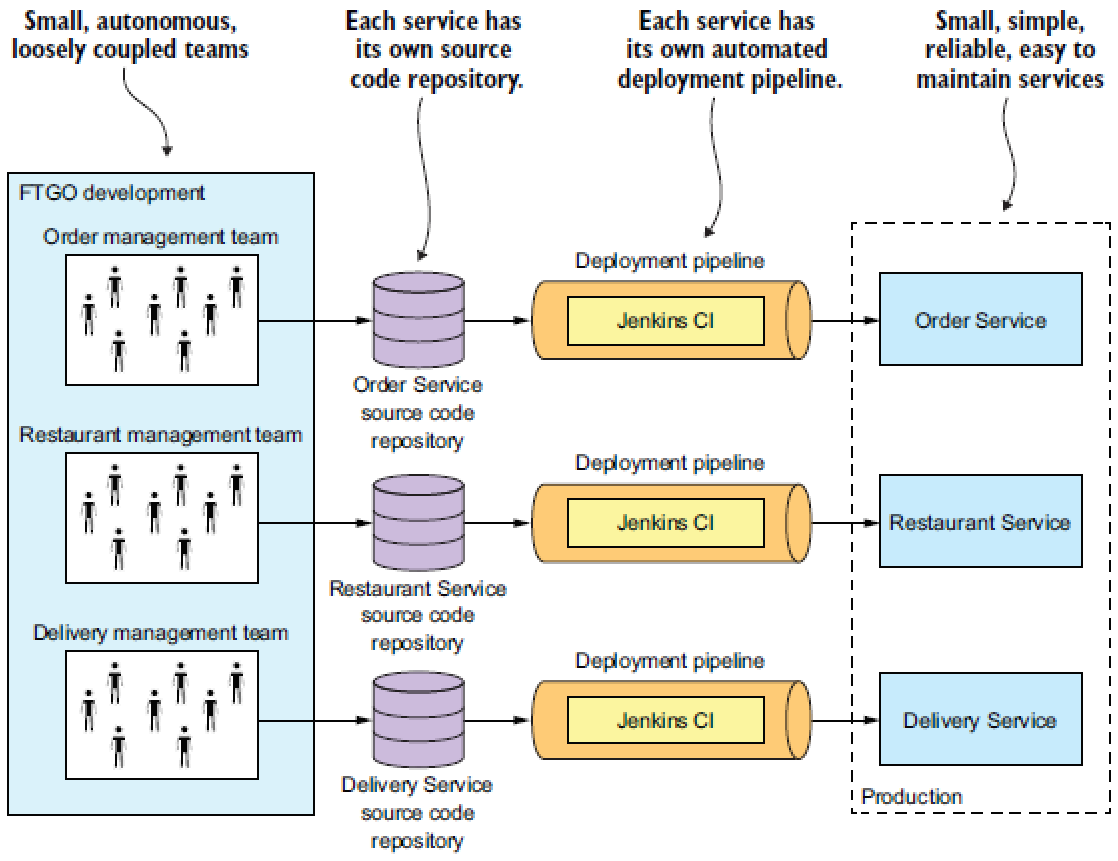
- **Small and focused**
- **Loosely coupled**
- **Language-neutral**
- **Bounded context**

Microservice architecture-based version of the FTGO application.



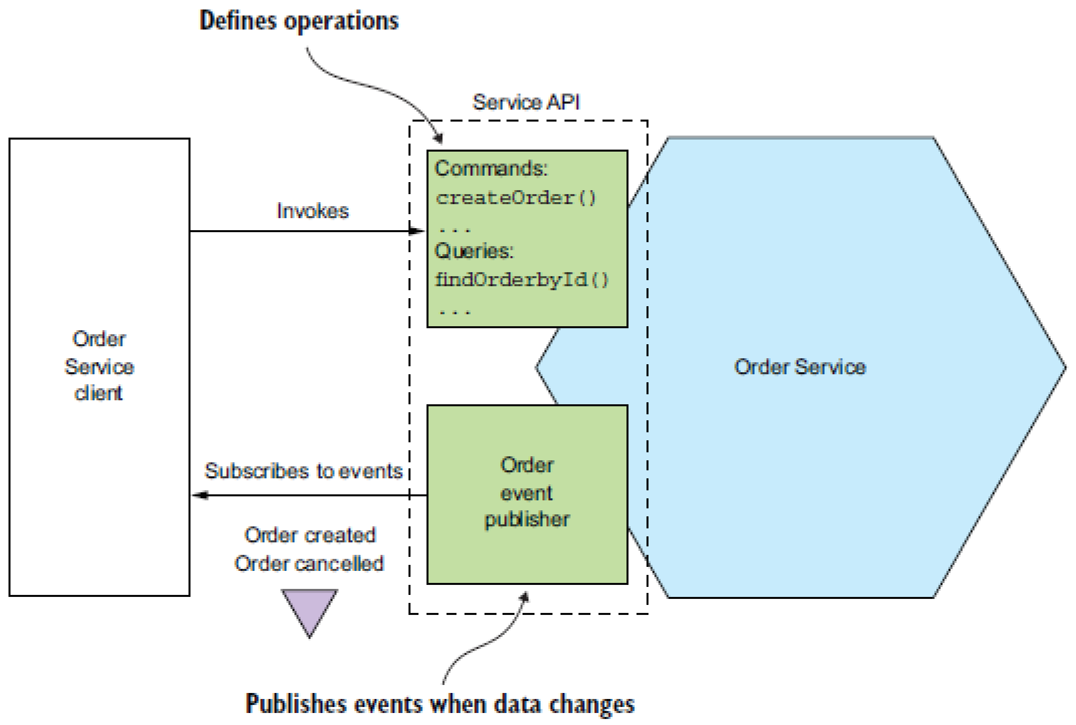
Benefits of the microservice architecture

- The microservice architecture has the following benefits:
- ⍰ It enables the continuous delivery and deployment of large, complex applications.
- ⍰ Services are small and easily maintained.
- ⍰ Services are independently deployable.
- ⍰ Services are independently scalable.
- ⍰ The microservice architecture enables teams to be autonomous.
- ⍰ It allows easy experimenting and adoption of new technologies.
- ⍰ It has better fault isolation.


Small, autonomous, loosely coupled teams
Each service has its own source code repository.
Each service has its own automated deployment pipeline.
Small, simple, reliable, easy to maintain services
FTGO development
Order management team
Restaurant management team
Delivery management team
Order Service source code repository
Restaurant Service source code repository
Delivery Service source code repository
Deployment pipeline
Jenkins CI
Deployment pipeline
Jenkins CI
Deployment pipeline
Jenkins CI
Order Service
Restaurant Service
Delivery Service
Production

WHAT IS A SERVICE?

- A *service* is a standalone, independently deployable software component that implements some useful functionality.
- A service has an API that provides its clients access to its functionality.
- There are two types of operations: commands and queries.
- The API consists of commands, queries, and events.
    - A command, such as `createOrder()`, performs actions and updates data.
    - A query, such as `findOrderById()`, retrieves data.
    - A service also publishes events, such as `OrderCreated`, which are consumed by its clients.
- A service’s API encapsulates its internal implementation.


Defines operations
Service API
Commands:
createOrder()
...
Queries:
findOrderbyId()
...
Invokes
Order Service client
Order Service
Subscribes to events
Order event publisher
Order created
Order cancelled
Publishes events when data changes

Interprocess communication in a microservice architecture

- There are a variety of client-service interaction styles which can be categorized in two dimensions.
- The first dimension is whether the interaction is one-to-one or one-to-many:
- *One-to-one*—Each client request is processed by exactly one service.
- *One-to-many*—Each request is processed by multiple services.
- The second dimension is whether the interaction is synchronous or asynchronous:
- *Synchronous*—The client expects a timely response from the service and might even block while it waits.
- *Asynchronous*—The client doesn’t block, and the response, if any, isn’t necessarily sent immediately.

	one-to-one	one-to-many
Synchronous	Request/response	—
Asynchronous	Asynchronous request/response One-way notifications	Publish/subscribe Publish/async responses

IPC Patterns-

- Use REST for *Communicating using the synchronous Remote procedure invocation pattern*
- *Event/Message Driven/Message Broker- Communicating using the Asynchronous messaging pattern*

Microservices Design Patterns

- Decomposition Patterns
  - **Decompose by Business Capability**
  - **Decompose by Subdomain**
  - **Strangler Pattern**
- Integration Patterns
  - **API Gateway Pattern**
  - **Aggregator Pattern**
  - **Client-Side UI Composition Pattern**
- Databse Patterns
  - **Database per Service**
  - **Shared Database per Service**
  - **Command Query Responsibility Segregation (CQRS)**
  - **Saga Pattern**
- Obervability Pattrns
  - **Log Aggregation**
  - **Performance Metrics**
  - **Distributed Tracing**
  - **Health Check**
- Cross Cutting Concerns Patterns
  - **External Configuration**
  - **Service Discovery Pattern**
  - **Circuit Breaker Pattern**

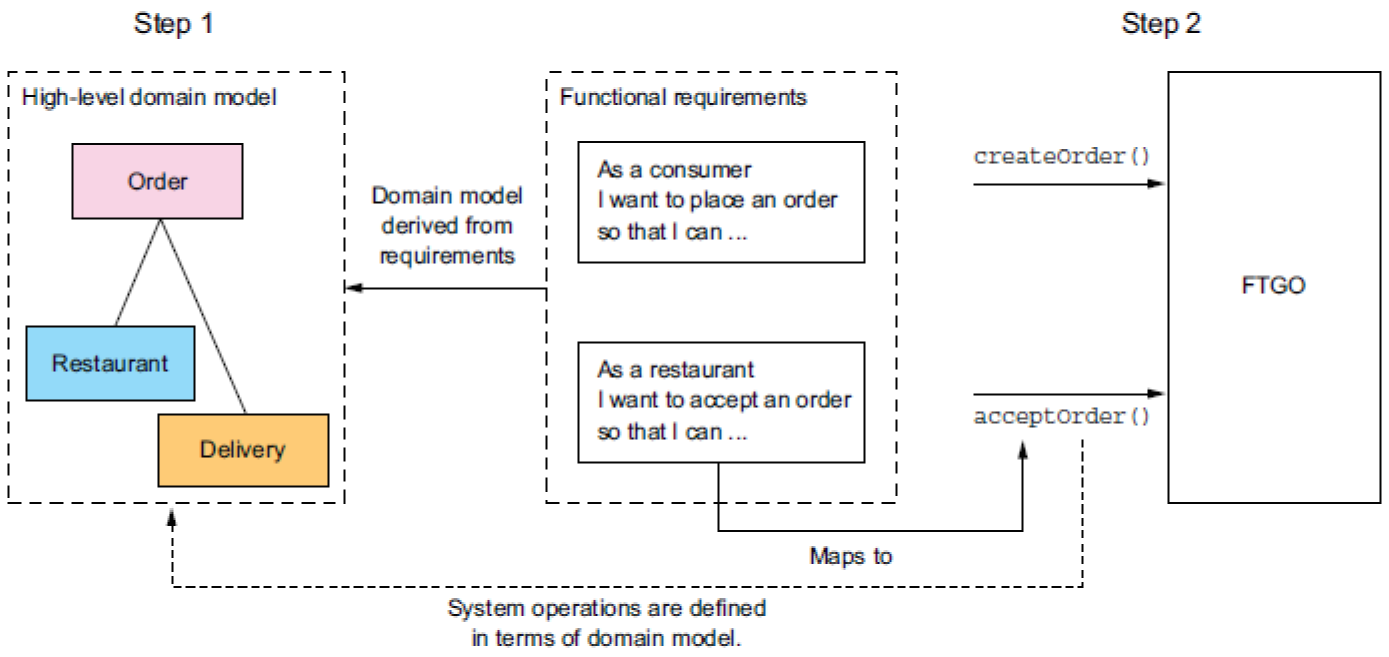
Decomposition Patterns

Decompose by Business Capability-Problem

- Microservices is all about making services loosely coupled, applying the single responsibility principle.
- However, breaking an application into smaller pieces has to be done logically.
- How do we decompose an application into small services?

Decompose by Business Capability-Solution

- Each business capability can be thought of as a service, except it’s business-oriented rather than technical.
- The set of capabilities for a given business depend on the type of business.
- For example, the capabilities of an insurance company typically include sales, marketing, underwriting, claims processing, billing, compliance, etc.


Step 1
Step 2
High-level domain model
Order
Restaurant
Delivery
Domain model derived from requirements
Functional requirements
As a consumer I want to place an order so that I can ...
As a restaurant I want to accept an order so that I can ...
createOrder()
acceptOrder()
FTGO
Maps to
System operations are defined in terms of domain model.

Decompose by Sub Domain- Problem

- Decomposing an application using business capabilities might be a good start, but you will come across so-called "God Classes" which will not be easy to decompose.
- These classes will be common among multiple services.
- For example, the Order class will be used in Order Management, Order Taking, Order Delivery, etc.
- How do we decompose them?

Decompose by Sub Domain- Solution

- For the "God Classes" issue, DDD (Domain-Driven Design) comes to the rescue.
- It uses subdomains and bounded context concepts to solve this problem.
- DDD breaks the whole domain model created for the enterprise into subdomains.
- Each subdomain will have a model, and the scope of that model will be called the bounded context.
- Each microservice will be developed around the bounded context.

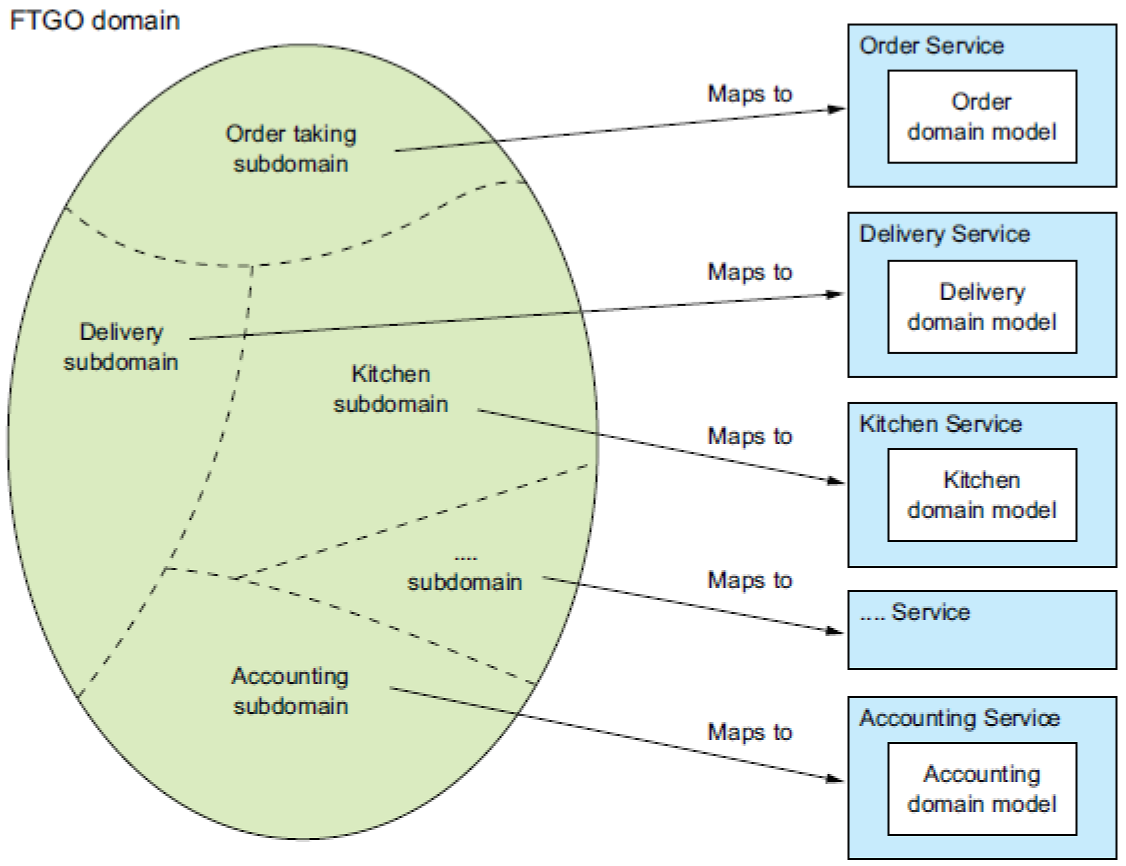


Figure 2.9 From subdomains to services: each subdomain of the FTGO application domain is mapped to a service, which has its own domain model.

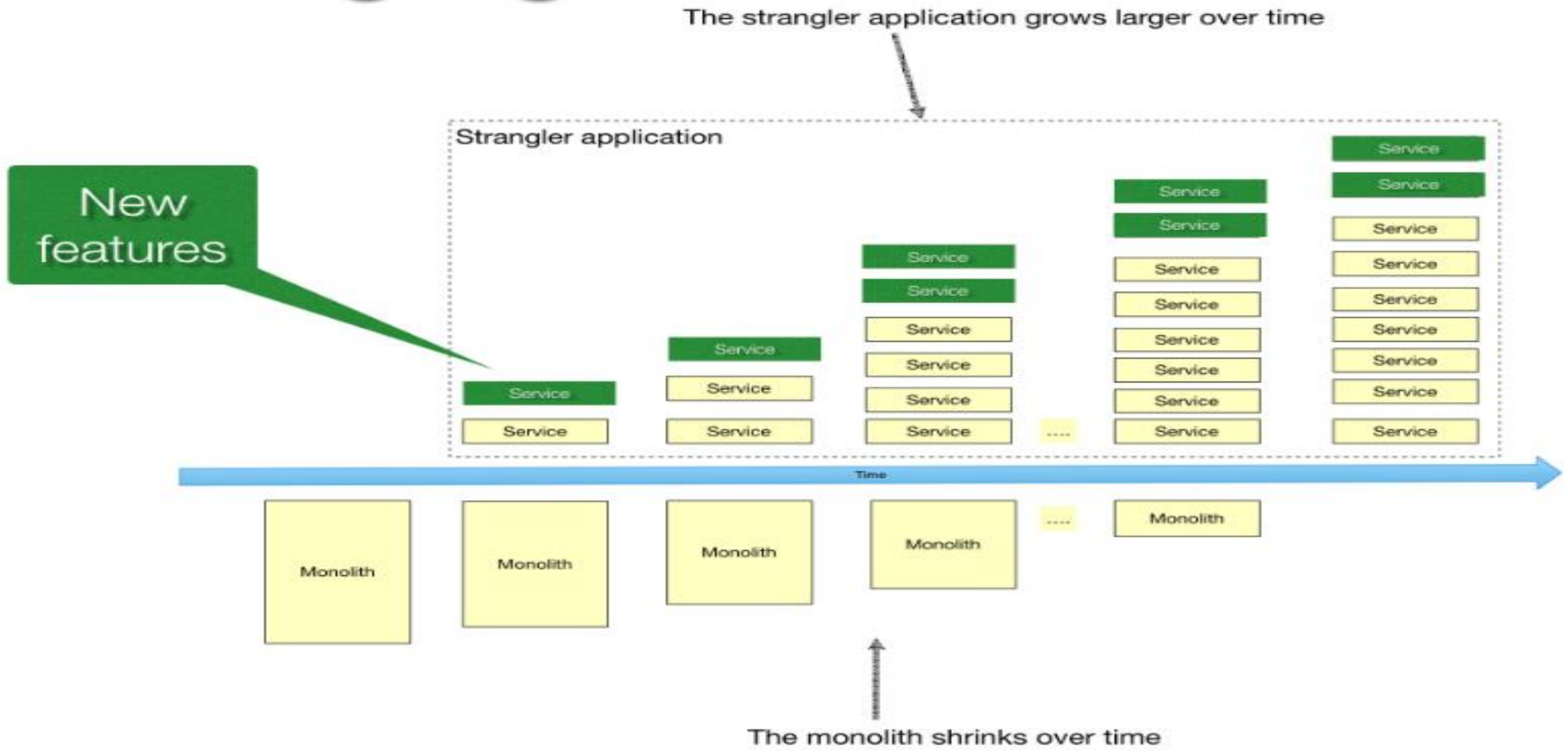
Strangler Pattern /Vine Pattern- **Problem**

- So far, the design patterns we talked about were decomposing applications for greenfield, but 80% of the work we do is with brownfield applications, which are big, monolithic applications.
- Applying all the above design patterns to them will be difficult because breaking them into smaller pieces at the same time it's being used live is a big task.
- The only way to decompose big monolithic applications is by following the Vine Pattern or the Strangler Pattern.

Strangler Pattern /Vine Pattern- **Solution**

- The Strangler pattern is based on an analogy to a vine that strangles a tree that it’s wrapped around.
- This solution works well with web applications, where a call goes back and forth, and for each URI call, a service can be broken into different domains and hosted as separate services.
- The idea is to do it one domain at a time. This creates two separate applications that live side by side in the same URI space.
- Modernize an application by incrementally developing a new (strangler) application around the legacy application.
- Eventually, the newly refactored application “strangles” or replaces the original application until finally you can shut off the monolithic application.
- The strangler application consists of two types of services.
  - First, there are services that implement functionality that previously resided in the monolith.
  - Second, there are services that implement new features.

Strangling the monolith


The strangler application grows larger over time
Strangler application
New features
Service
Time
Monolith
.....
The monolith shrinks over time

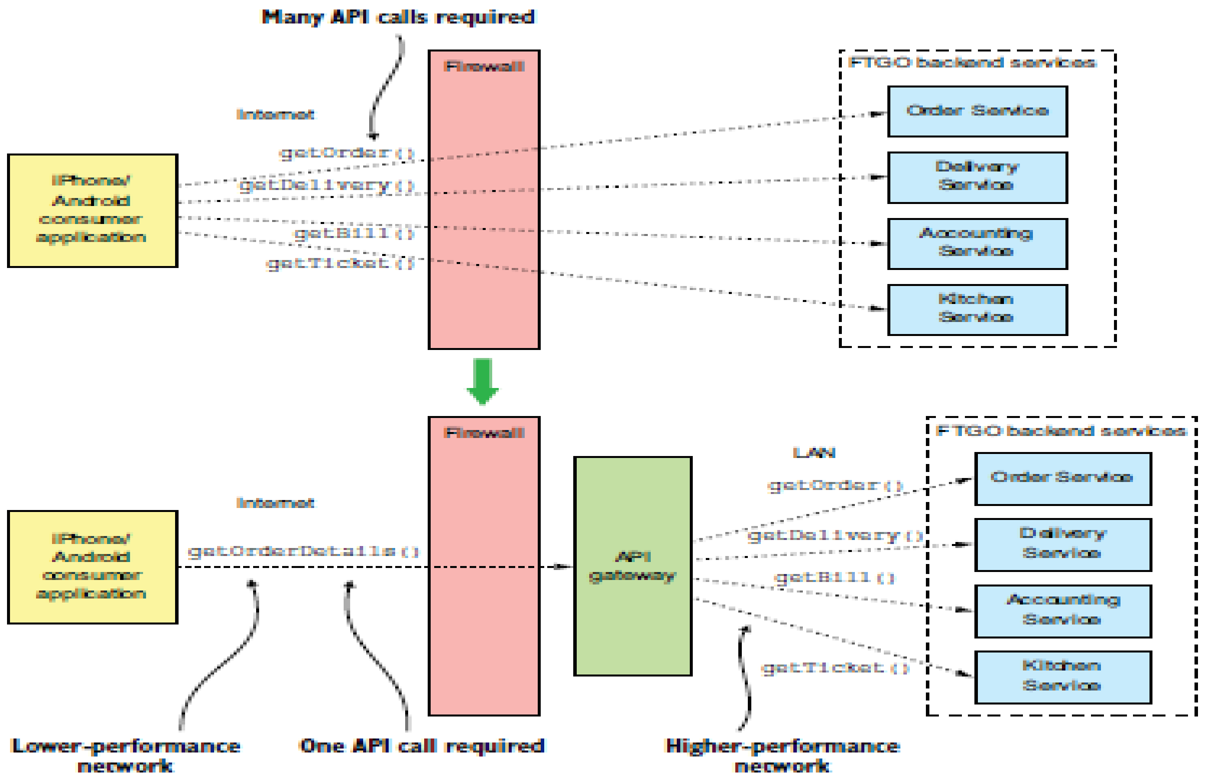
Integration Patterns

API Gateway Pattern-Problem

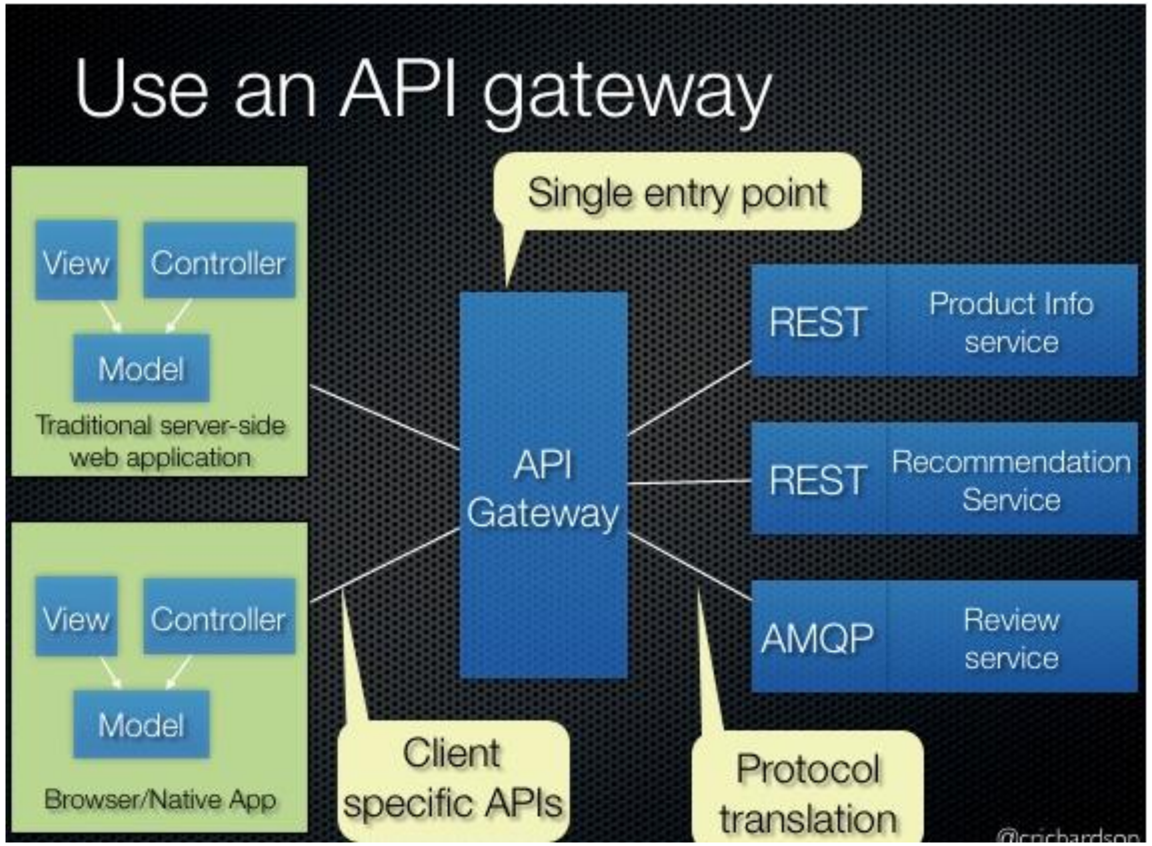
- The granularity of APIs provided by microservices is often different than what a client needs.
- Microservices typically provide fine-grained APIs, which means that clients need to interact with multiple services.
- How to call multiple microservices abstracting producer information.
  - A client needing the details for a product needs to fetch data from numerous services.
- Different clients need different data.
  - The desktop browser version of a product details page desktop is typically more elaborate then the mobile version.
- Network performance is different for different types of clients.
- The number of service instances and their locations (host+port) changes dynamically
- Partitioning into services can change over time and should be hidden from clients
- Services might use a diverse set of protocols, some of which might not be web friendly.
- On different channels (like desktop, mobile, and tablets), apps need different data to respond for the same backend service, as the UI might be different.
- Different consumers might need a different format of the responses from reusable microservices. Who will do the data transformation or field manipulation?
- How to handle different type of Protocols some of which might not be supported by producer microservice.

API Gateway Pattern-Solution

- An API Gateway is the single point of entry for any microservice call.
- The API gateway is responsible for request routing, API composition, and protocol translation.
- All API requests from external clients first go to the API gateway, which routes some requests to the appropriate service.
- The API gateway handles other requests using the API composition pattern and by invoking multiple services and aggregating the results.
- It may also translate between client-friendly protocols such as HTTP and WebSockets and client-unfriendly protocols used by the services.

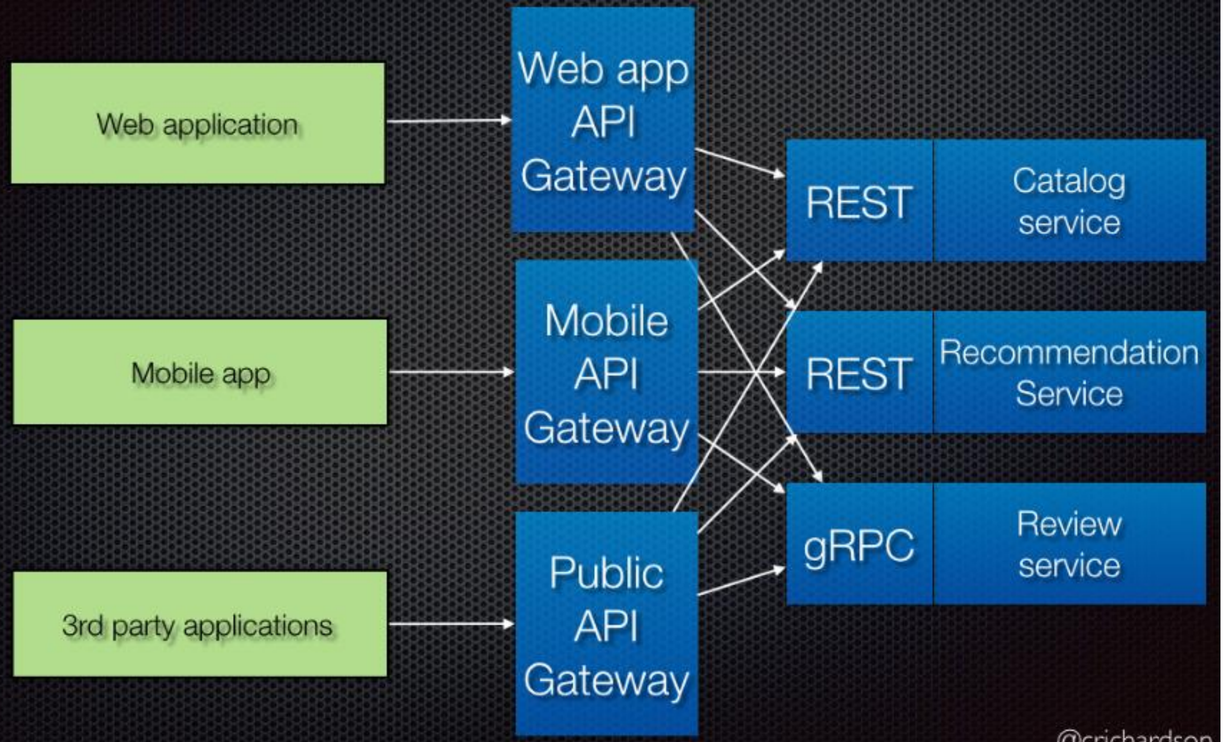


Many API calls required
Internet
Firewall
FTGO backend services
Order Service
Delivery Service
Accounting Service
Kitchen Service
iPhone/ Android consumer application
getOrder()
getDelivery()
getBill()
getTicket()
Firewall
Internet
getOrderDetails()
API gateway
LAN
FTGO backend services
Order Service
Delivery Service
Accounting Service
Kitchen Service
getOrder()
getDelivery()
getBill()
getTicket()
iPhone/ Android consumer application
Lower-performance network
One API call required
Higher-performance network



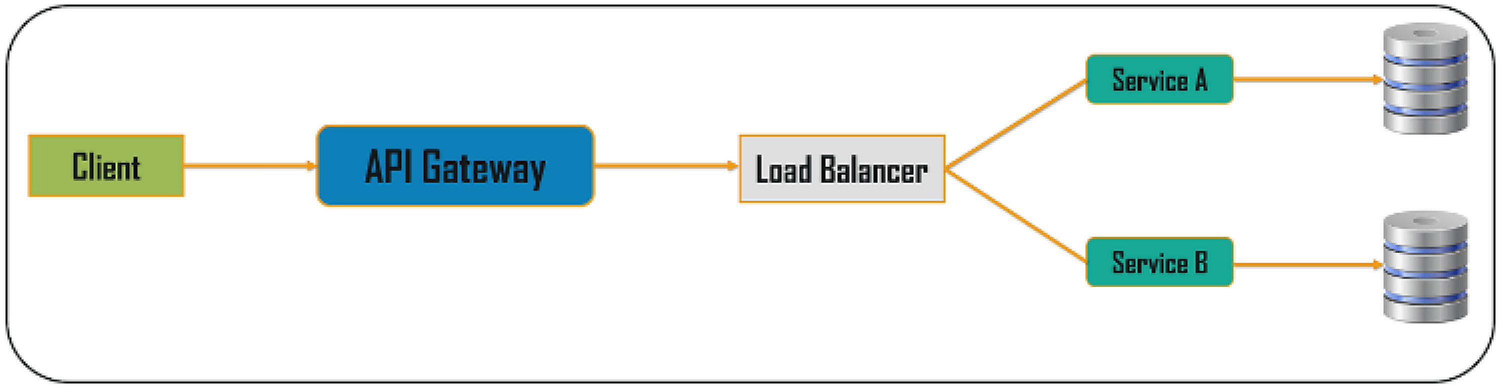
Use an API gateway

Variation: Backends for frontends



@crichardson

API Gateway



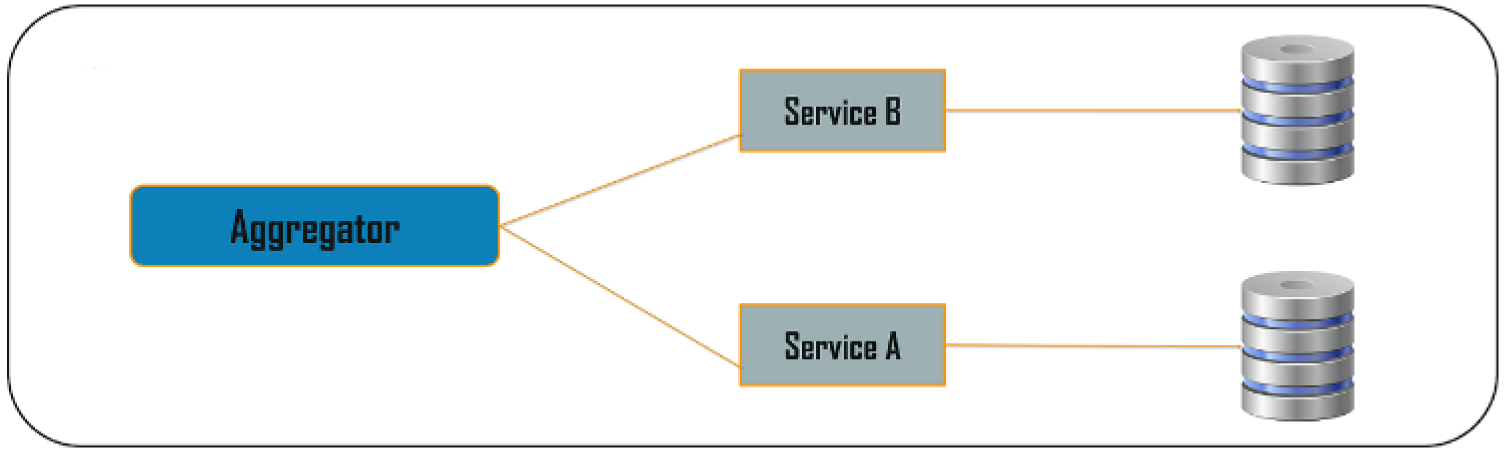
Aggregator Pattern - Problem

- When breaking the business functionality into several smaller logical pieces of code, it becomes necessary to think about how to collaborate the data returned by each service.
- This responsibility cannot be left with the consumer, as then it might need to understand the internal implementation of the producer application.

Aggregator Pattern - Solution

- Aggregator in the computing world refers to a website or program that collects related items of data and displays them.
- Aggregator is a basic web page which invokes various services to get the required information or achieve the required functionality.
- The Aggregate Design Pattern is based on the DRY principle. It talks about how we can aggregate the data from different services and then send the final response to the consumer. This can be done in two ways:
  - A **composite microservice** will make calls to all the required microservices, consolidate the data, and transform the data before sending back.
  - An **API Gateway** can also partition the request to multiple microservices and aggregate the data before sending it to the consumer.
- It is recommended if any business logic is to be applied, then choose a composite microservice. Otherwise, the API Gateway is the established solution.

Aggregator



Client-Side UI Composition Pattern

Client-Side UI Composition Pattern-Problem

- When services are developed by decomposing business capabilities/subdomains, the services responsible for user experience have to pull data from several microservices.
- In the monolithic world, there used to be only one call from the UI to a backend service to retrieve all data and refresh/submit the UI page.
- However, now it won't be the same.

Client-Side UI Composition Pattern-Solution

- With microservices, the UI has to be designed as a skeleton with multiple sections/regions of the screen/page.
- Each section will make a call to an individual backend microservice to pull the data. That is called composing UI components specific to service. Frameworks like AngularJS and ReactJS help to do that easily.
- These screens are known as Single Page Applications (SPA).
- This enables the app to refresh a particular region of the screen instead of the whole page.

Database Patterns

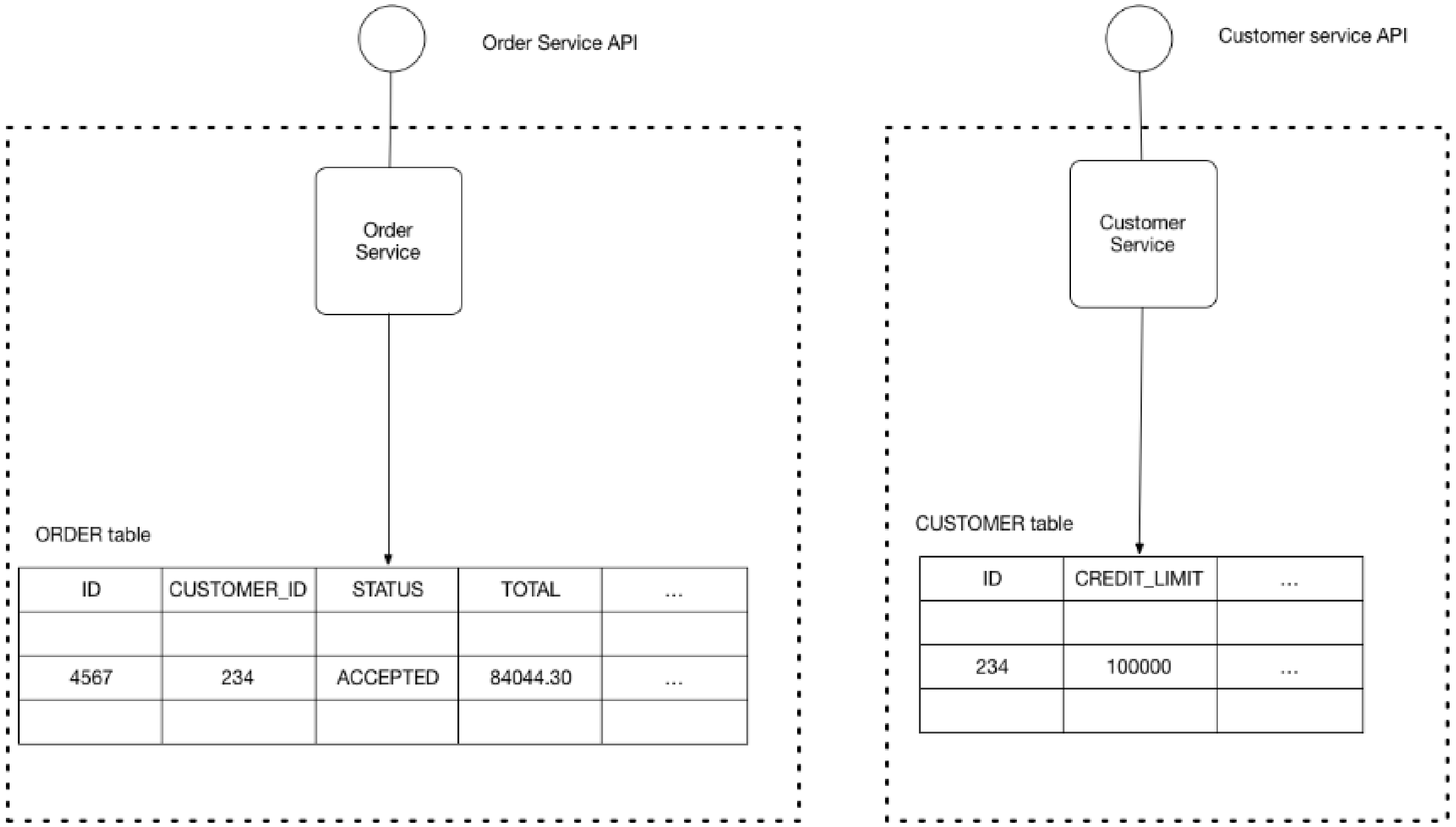
Database Per Service-Problem

- Duplication of data and inconsistency
- Different services have different kinds of storage requirements
- Business transactions may enforce invariants that span multiple services.
- Some business transactions need to query data that is owned by multiple services.
- Databases must sometimes be replicated and sharded in order to scale.
- Different services have different data storage requirements.
- De-normalization of data

Database Per Service-Solution

- Can use the following strategies:
  - Private-tables-per-service – each service owns a set of tables that must only be accessed by that service
  - Schema-per-service – each service has a database schema that’s private to that service
  - Database-server-per-service – each service has it’s own database server.

Database Per Service



ORDER table

ID	CUSTOMER_ID	STATUS	TOTAL	...
4567	234	ACCEPTED	84044.30	...

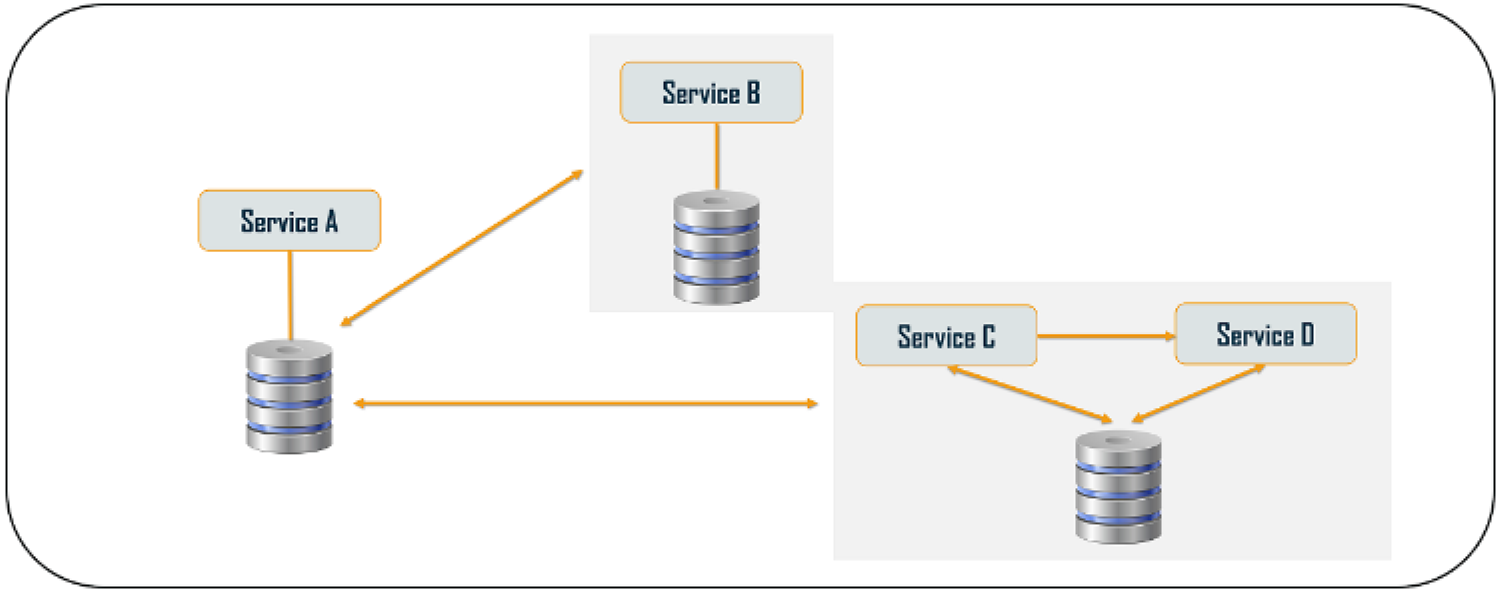
CUSTOMER table

ID	CREDIT_LIMIT	...
234	100000	...

Shared Database Per Service-Solution

- To solve the issue of de-normalization, you can choose shared databases per service, to align more than one database for each microservice.
- This will help you gather data, for the monolithic applications which are broken down into microservices.
- But, you have to keep in mind that, you have to limit these databases to 2-3 microservices; else, scaling these services will be a problem.

Database Per Service

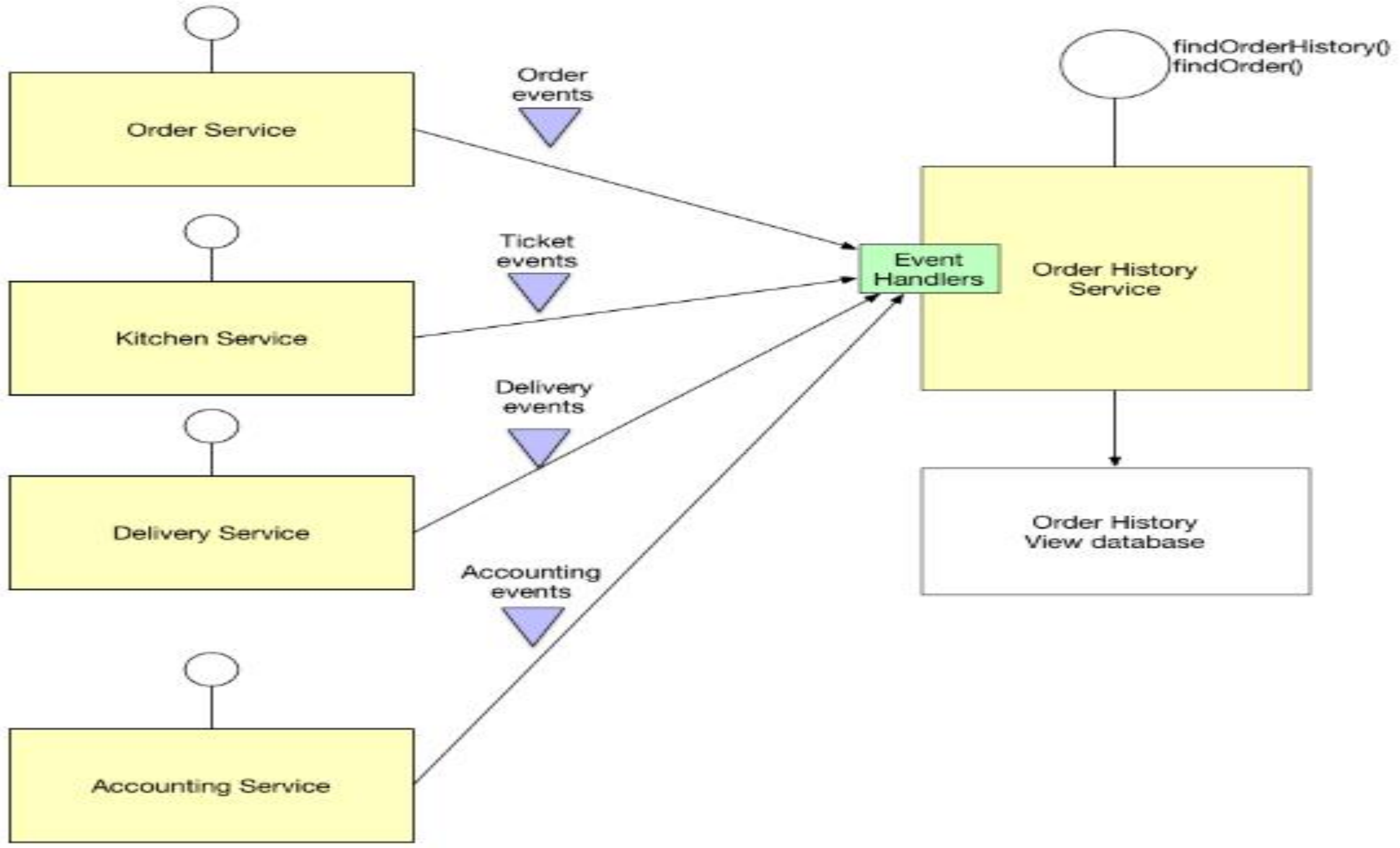


Command Query Responsibility Segregation (CQRS)-Problem

- How to implement a query that retrieves data from multiple services in a microservice architecture?

Command Query Responsibility Segregation (CQRS)-Solution

- CQRS suggests splitting the application into two parts — the command side and the query side.
- The command side handles the Create, Update, and Delete requests.
- The query side handles the query part by using the materialized views.



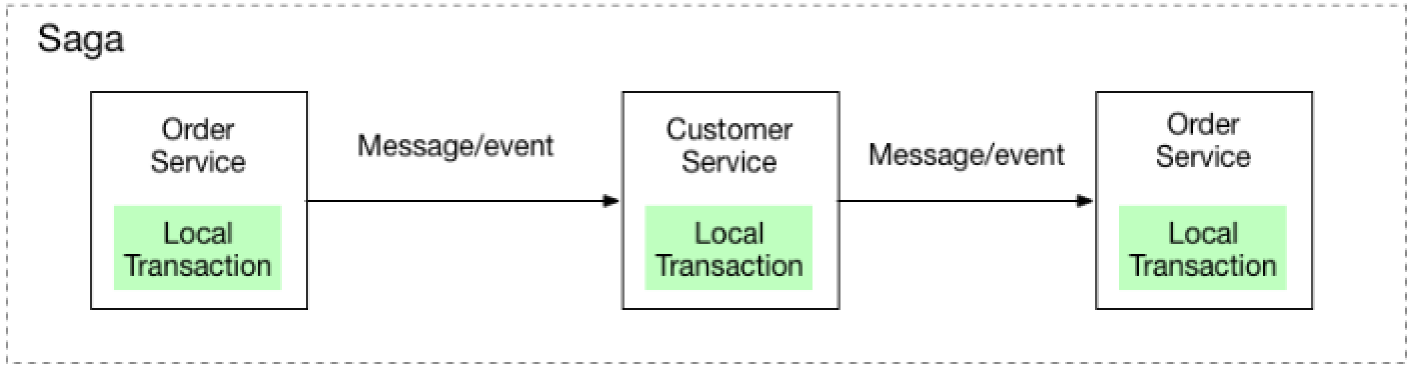
Order Service
Kitchen Service
Delivery Service
Accounting Service
Order events
Ticket events
Delivery events
Accounting events
Event Handlers
Order History Service
findOrderHistory()
findOrder()
Order History View database

Saga Pattern-Problem

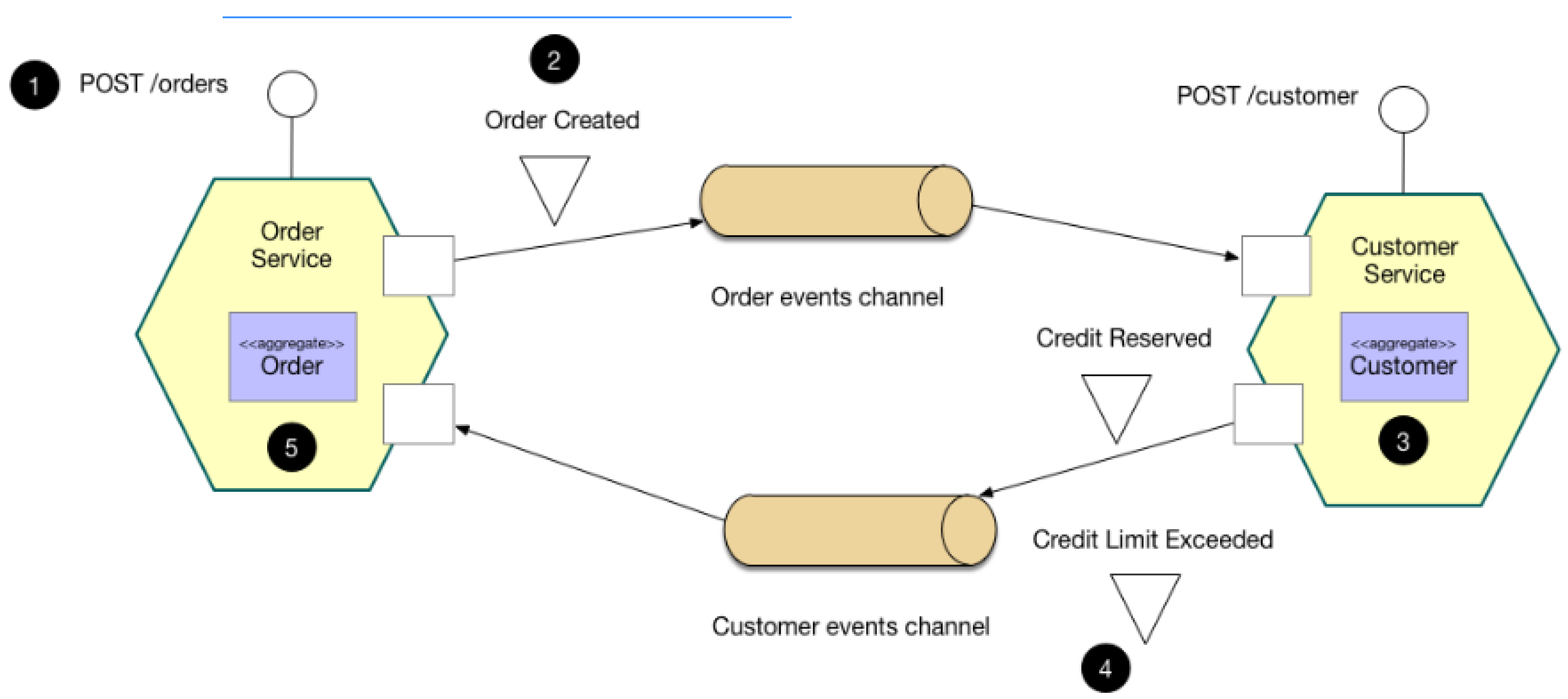
- When each service has its own database and a business transaction spans multiple services, how do we ensure data consistency across services?
- For example, for an e-commerce application where customers have a credit limit, the application must ensure that a new order will not exceed the customer’s credit limit. Since Orders and Customers are in different databases, the application cannot simply use a local ACID transaction.
- How to implement transactions that span services?

Saga Pattern-Solution

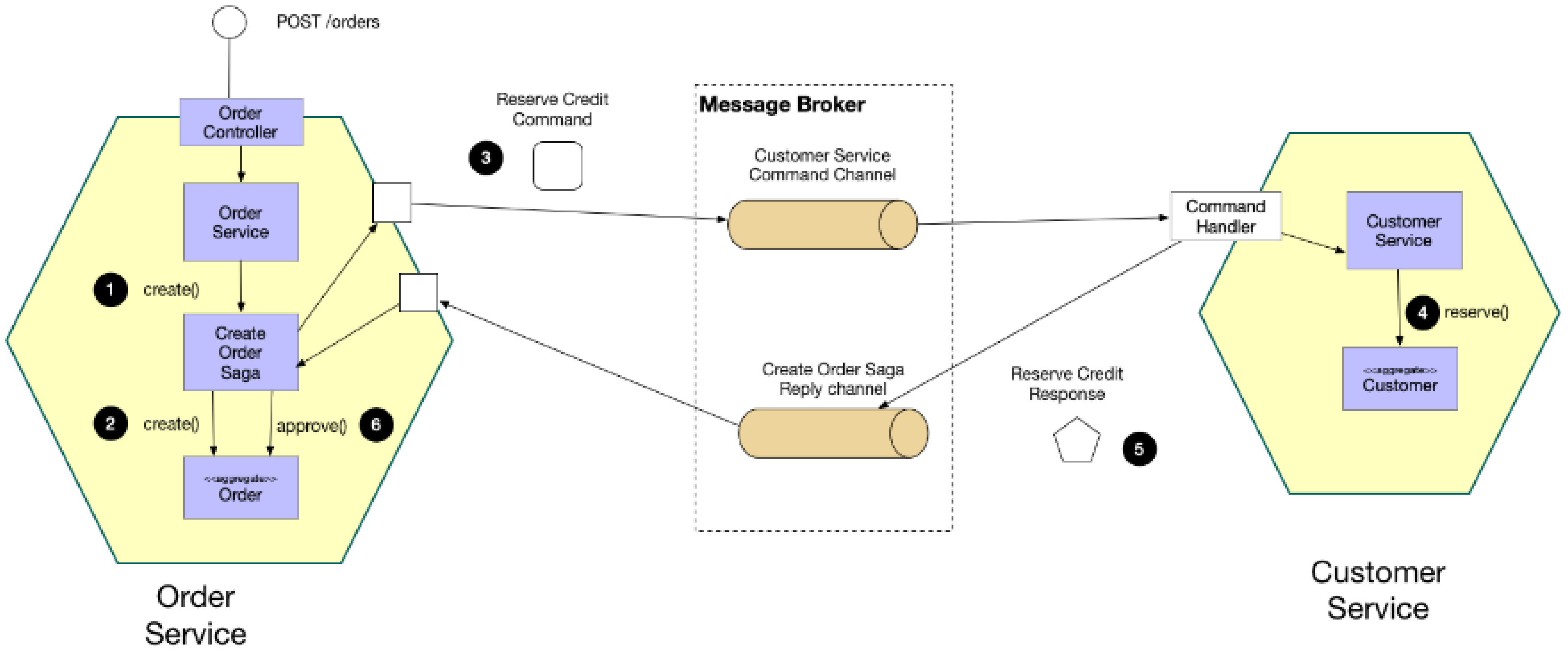
- Implement each business transaction that spans multiple services is a saga.
- A saga is a sequence of local transactions.
- It enables an application to maintain data consistency across multiple services without using distributed transactions
- Each local transaction updates the database and publishes a message or event to trigger the next local transaction in the saga.
- If a local transaction fails because it violates a business rule then the saga executes a series of compensating transactions that undo the changes that were made by the preceding local transactions.
- There are two ways of coordination sagas:
    - Choreography - each local transaction publishes domain events that trigger local transactions in other services
    - Orchestration - an orchestrator (object) tells the participants what local transactions to execute


Saga
Order Service
Local Transaction
Message/event
Customer Service
Local Transaction
Message/event
Order Service
Local Transaction

Choreography-based saga



Orchestration-based saga



Observability Patterns

Log Aggregation-Problems

- The application consists of multiple services and service instances that are running on multiple machines. Requests often span multiple service instances.
- Each service instance generates writes information about what it is doing to a log file in a standardized format.
- The log file contains errors, warnings, information and debug messages.
- How to understand the behavior of an application and troubleshoot problems?
- Any solution should have minimal runtime overhead

Log Aggregation-Solution

- Use a centralized logging service that aggregates logs from each service instance.
- The users can search and analyze the logs.
- They can configure alerts that are triggered when certain messages appear in the logs.

Performance Metrics-Problem

- When the service portfolio increases due to microservice architecture, it becomes critical to keep a watch on the transactions so that patterns can be monitored and alerts sent when an issue happens.
- How should we collect metrics to monitor application perfomance?

Performance Metrics-Solution

- A metrics service is required to gather statistics about individual operations.
- It should aggregate the metrics of an application service, which provides reporting and alerting.
- There are two models for aggregating metrics:
  - Push — the service pushes metrics to the metrics service e.g. NewRelic, AppDynamics
  - Pull — the metrics services pulls metrics from the service e.g. Prometheus

Distributed Tracing-Problem

- In microservice architecture, requests often span multiple services.
- Each service handles a request by performing one or more operations across multiple services.
- Then, how do we trace a request end-to-end to troubleshoot the problem?

Distributed Tracing-Solution

- We need a service which
  - Assigns each external request a unique external request id.
  - Passes the external request id to all services.
  - Includes the external request id in all log messages.
  - Records information (e.g. start time, end time) about the requests and operations performed when handling an external request in a centralized service.

Health Check-Problem

- Sometimes a service instance can be incapable of handling requests yet still be running.
- For example, it might have ran out of database connections.
- When this occurs, the monitoring system should generate a alert.
- Also, the load balancer or service registry should not route requests to the failed service instance.
- How to detect that a running service instance is unable to handle requests?

Health Check-Solution

- A service has an health check API endpoint (e.g. HTTP /health) that returns the health of the service.
- The API endpoint handler performs various checks, such as
  - the status of the connections to the infrastructure services used by the service instance
  - the status of the host, e.g. disk space
  - application specific logic
- A health check client - a monitoring service, service registry or load balancer - periodically invokes the endpoint to check the health of the service instance.

Cross-Cutting Concern Patterns

External Configuration-Problem

- A service must be provided with configuration data that tells it how to connect to the external/3rd party services.
    - For example, the database network location and credentials
- A service must run in multiple environments - dev, test, qa, staging, production - without modification and/or recompilation
- Different environments have different instances of the external/3rd party services, e.g. QA database vs. production database, test credit card processing account vs. production credit card processing account
- For each environment like dev, QA, UAT, prod, the endpoint URL or some configuration properties might be different.
- A change in any of those properties might require a re-build and re-deploy of the service.
- How to enable a service to run in multiple environments without modification?

External Configuration-Solution

- Externalize all application configuration including the database credentials and network location.
- On startup, a service reads the configuration from an external source, e.g. OS environment variables, etc.

Service Discovery Pattern-problem

- Each instance of a service exposes a remote API such as HTTP/REST, or Thrift etc. at a particular location (host and port)
- The number of services instances and their locations changes dynamically.
- Virtual machines and containers are usually assigned dynamic IP addresses.
- The number of services instances might vary dynamically. For example, an EC2 Autoscaling Group adjusts the number of instances based on load.
- So how does the consumer or router know all the available service instances and locations?

Service Discovery Pattern-Solution

- A service registry needs to be created which will keep the metadata of each producer service.
- A service instance should register to the registry when starting and should de-register when shutting down.
- The consumer or router should query the registry and find out the location of the service.
- The registry also needs to do a health check of the producer service to ensure that only working instances of the services are available to be consumed through it.
- There are two types of service discovery: client-side and server-side.
- An example of client-side discovery is Netflix Eureka and an example of server-side discovery is AWS ALB.

Circuit Breaker Pattern-Problem

- Services sometimes collaborate when handling requests.
- When one service synchronously invokes another there is always the possibility that the other service is unavailable or is exhibiting such high latency it is essentially unusable.
- Precious resources such as threads might be consumed in the caller while waiting for the other service to respond. This might lead to resource exhaustion, which would make the calling service unable to handle other requests.
- The failure of one service can potentially cascade to other services throughout the application.
- How to prevent a network or service failure from cascading to other services?

Circuit Breaker Pattern-Solution

- The consumer should invoke a remote service via a proxy that behaves in a similar fashion to an electrical circuit breaker.
- When the number of consecutive failures crosses a threshold, the circuit breaker trips, and for the duration of a timeout period, all attempts to invoke the remote service will fail immediately.
- After the timeout expires the circuit breaker allows a limited number of test requests to pass through. If those requests succeed, the circuit breaker resumes normal operation. Otherwise, if there is a failure, the timeout period begins again.
- Netflix Hystrix is a good implementation of the circuit breaker pattern. It also helps you to define a fallback mechanism which can be used when the circuit breaker trips.

References

- https://dzone.com/articles/design-patterns-for-microservices
- https://www.edureka.co/blog/microservices-design-patterns
- https://microservices.io/patterns/data/shared-database.html
- Microservices Patterns- Chris Richardson
- https://towardsdatascience.com/effective-microservices-10-best-practices-c6e4ba0c6ee2