الجامعة السورية الخاصة
SYRIAN PRIVATE UNIVERSITY

مقرر بنيان البرمجيات

كلية الهندسة المعلوماتية

المحاضرة 6

Microservices Architecture

د. رياض سنبل

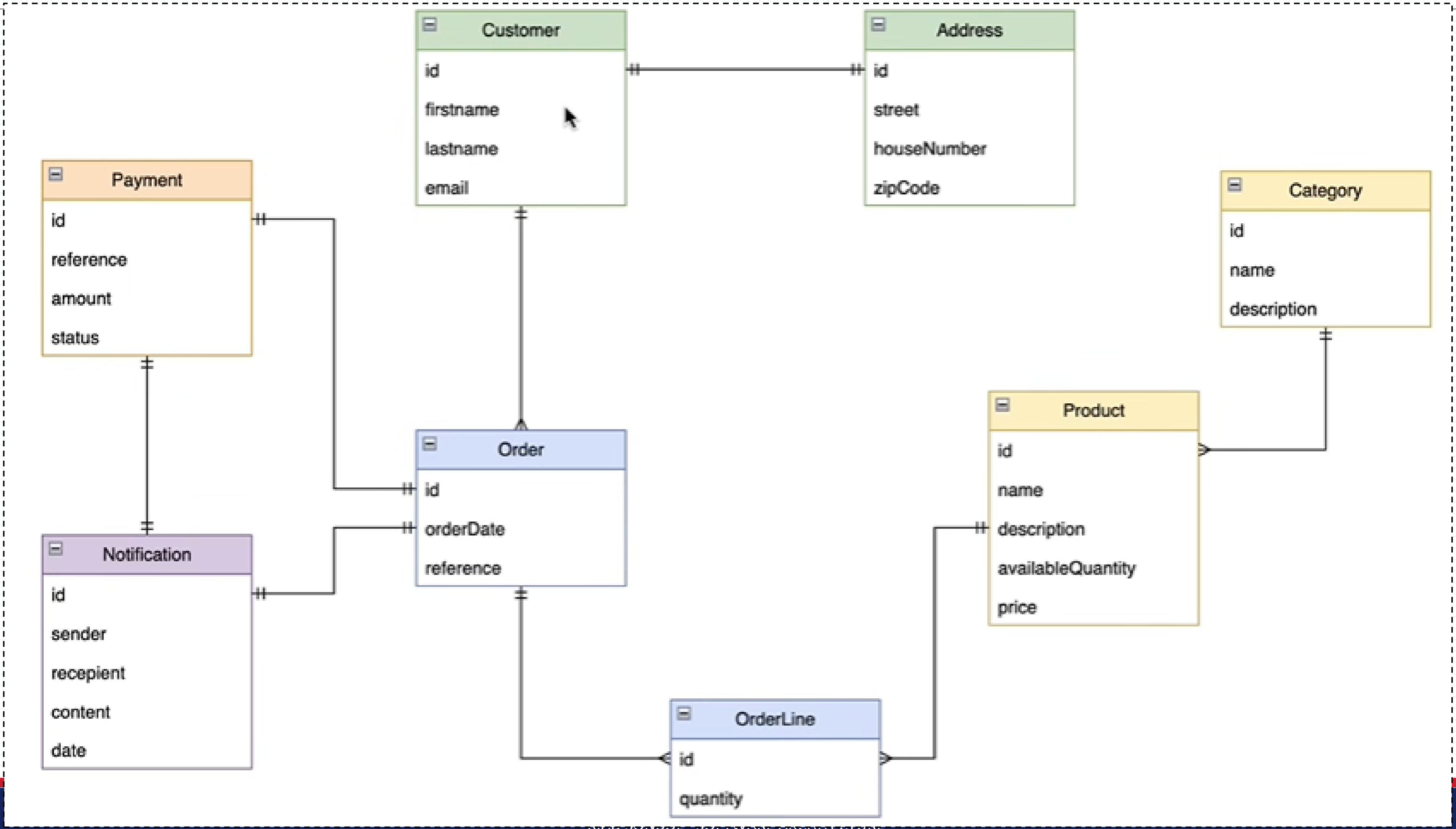
What is Domain-Driven Design?

- Domain-driven design (DDD) is a major software design approach, focusing on modeling software to match a domain according to input from that domain's experts.
- Under domain-driven design, the structure and language of software code (class names, class methods, class variables) should match the business domain
- Approach to software development focusing on domain complexity
- The term was coined by Eric Evans in his book of the same name published in 2003.

Why Use DDD?

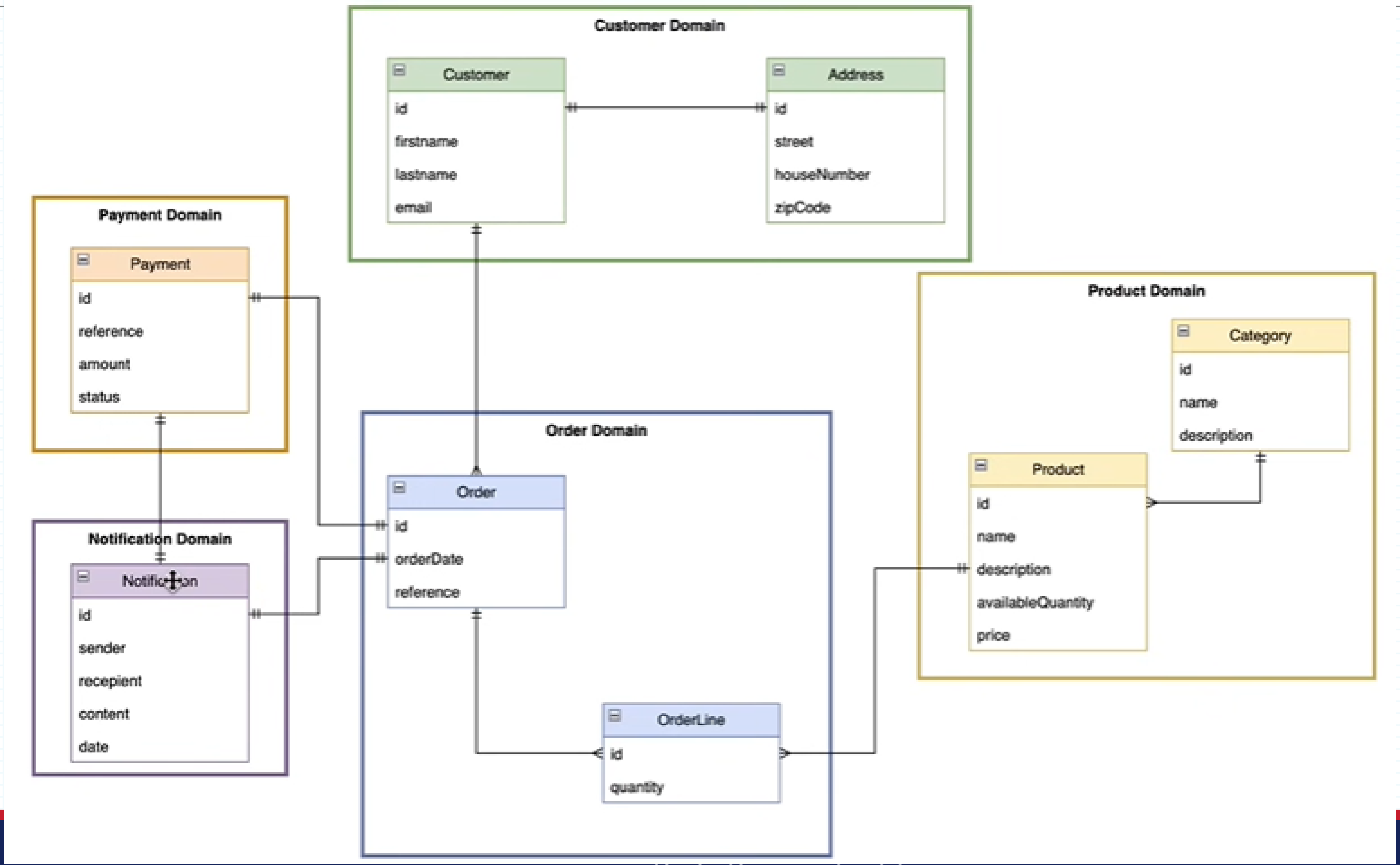
- **Manages complexity**: DDD breaks down complex business logic into smaller, understandable models within clearly defined boundaries.
- **Clear communication using domain language**: It promotes a shared vocabulary (ubiquitous language) between developers and domain experts to avoid misunderstandings.
- **Aligns software with business goals**: DDD ensures the software design reflects real-world business processes and priorities.
- **Foundation for microservices architecture**: Each bounded context in DDD can map naturally to a microservice, encouraging modular and scalable systems.

Example

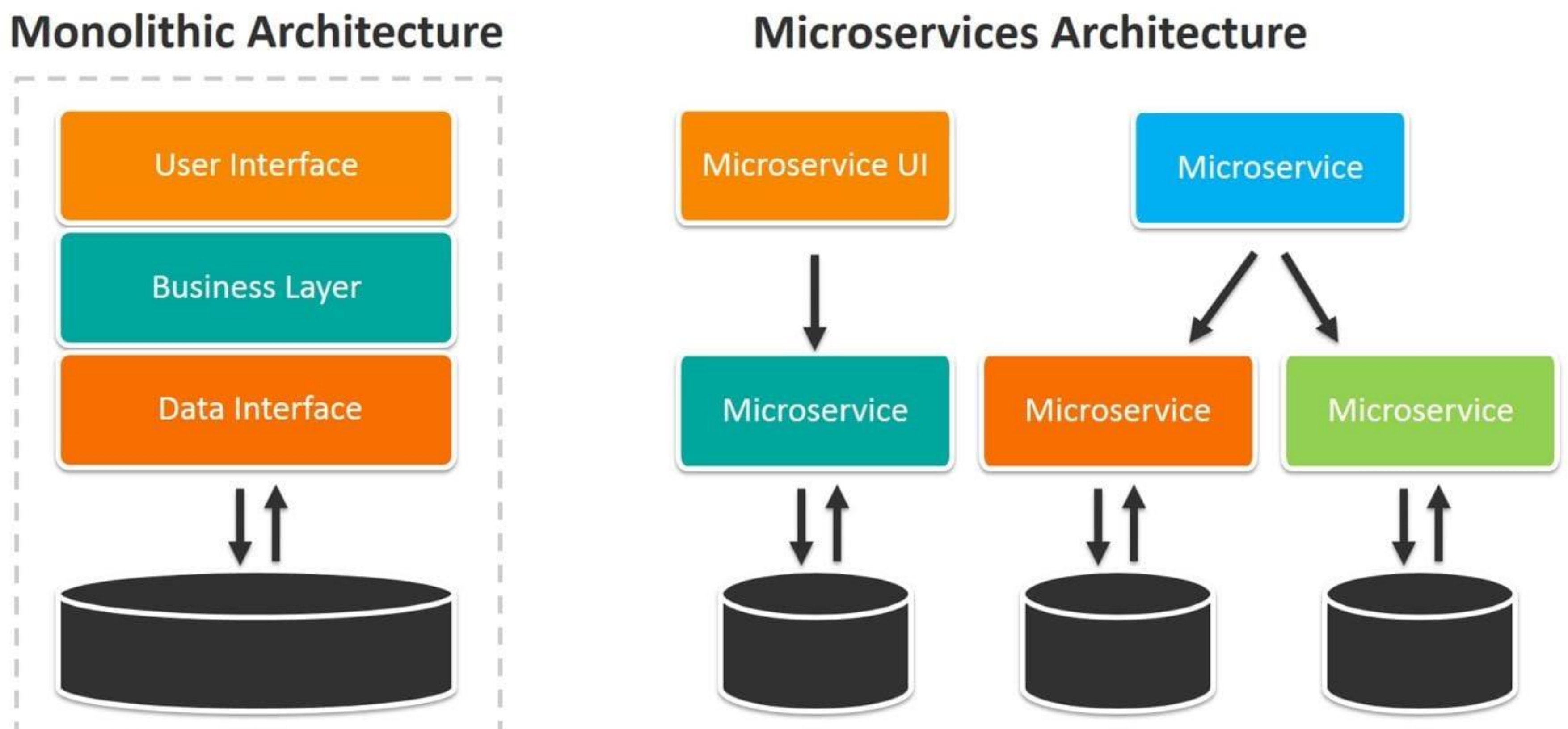


Customer
id
firstname
lastname
email
Address
id
street
houseNumber
zipCode
Payment
id
reference
amount
status
Category
id
name
description
Order
id
orderDate
reference
Product
id
name
description
availableQuantity
price
Notification
id
sender
recepient
content
date
OrderLine
id
quantity

Example


Customer Domain
Customer
id
firstname
lastname
email
Address
id
street
houseNumber
zipCode
Payment Domain
Payment
id
reference
amount
status
Product Domain
Category
id
name
description
Product
id
name
description
availableQuantity
price
Order Domain
Order
id
orderDate
reference
OrderLine
id
quantity
Notification Domain
Notification
id
sender
recepient
content
date

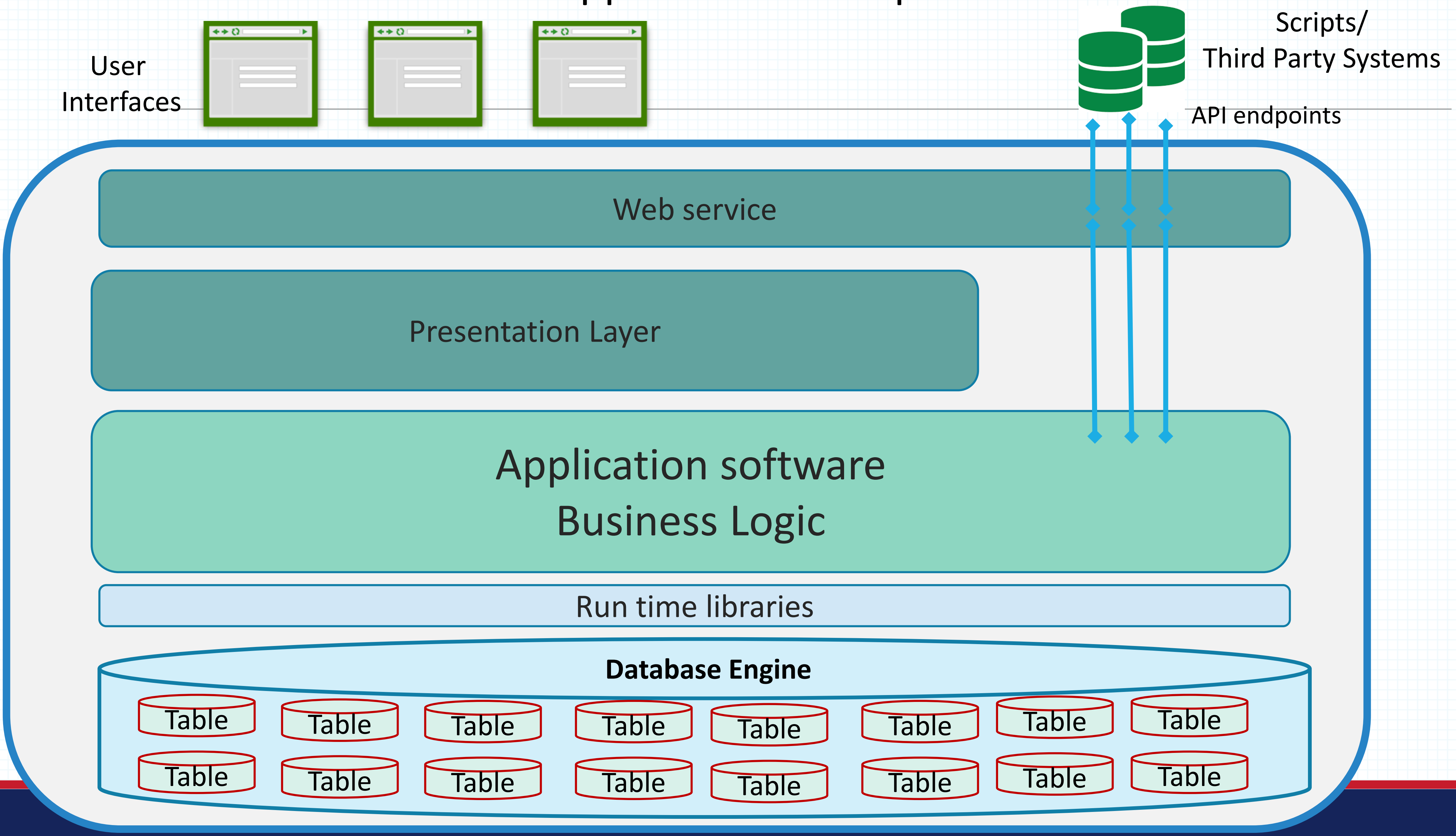
Monolithic vs Microservices



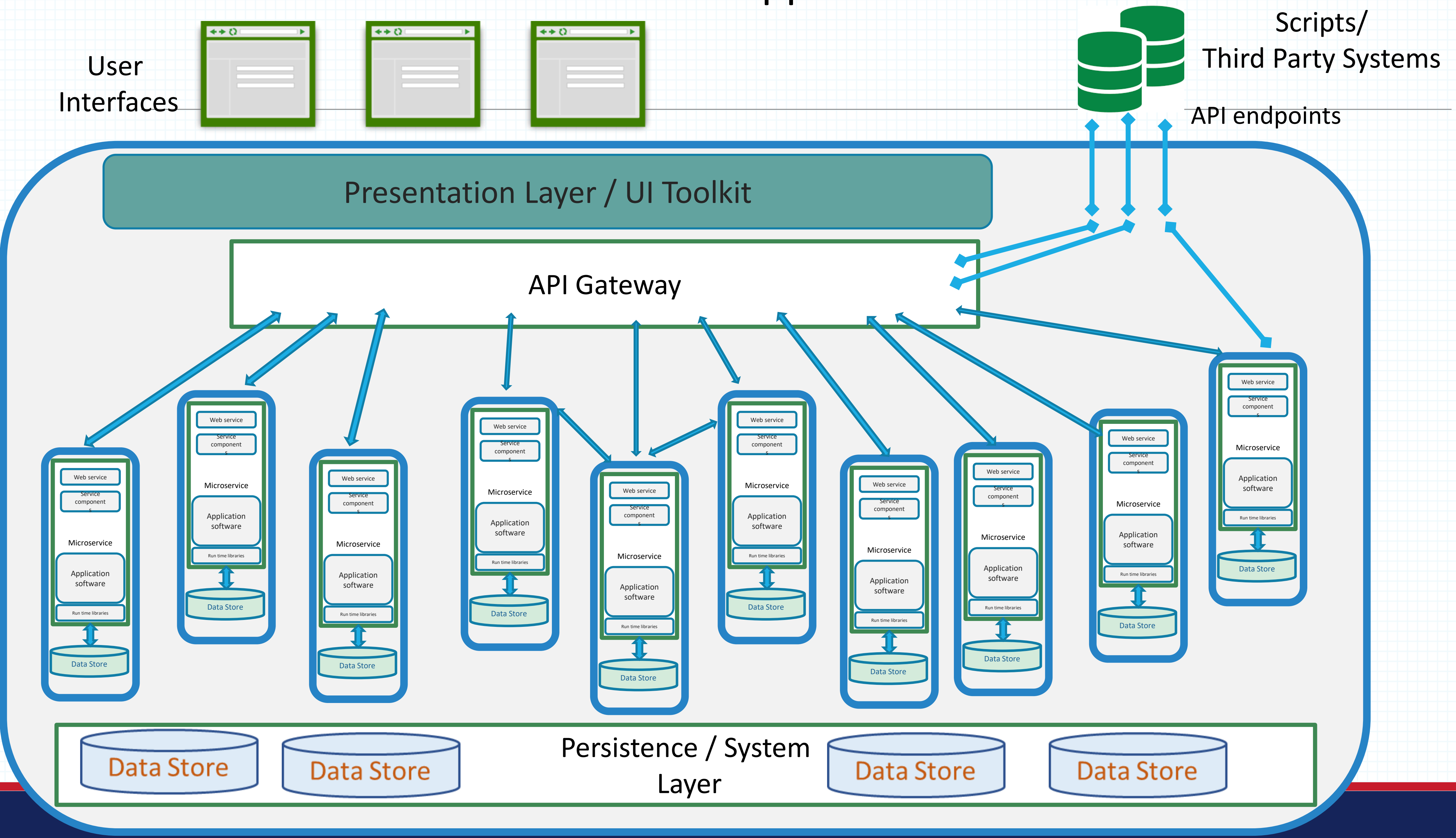
deployed as a single bundle of executables and libraries on a unified platform

Multiple independent software components orchestrated to form a unified application

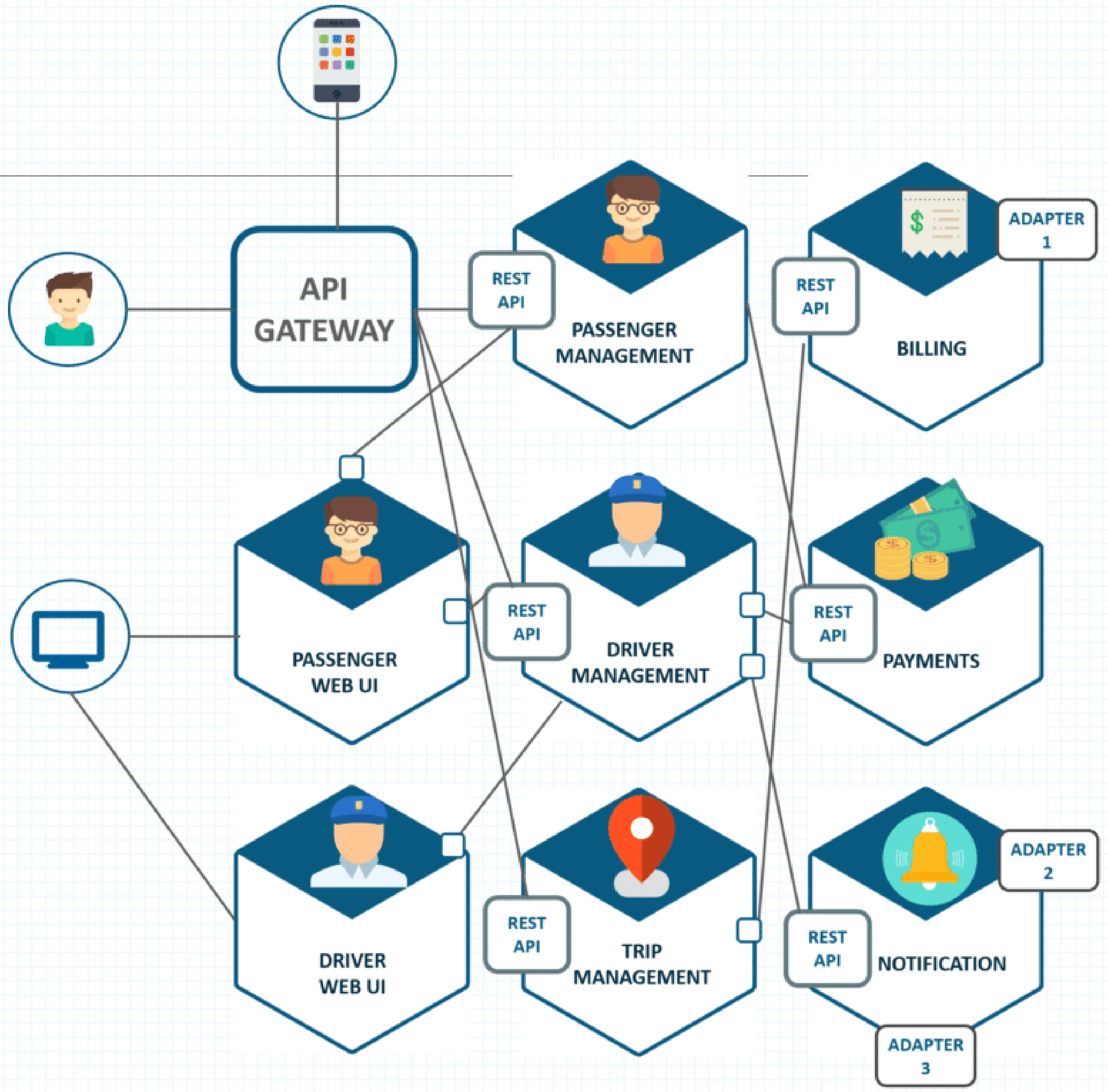
Monolithic Application Conceptual Model



Microservices-based Application



Example


API GATEWAY
REST API
PASSENGER MANAGEMENT
REST API
BILLING
ADAPTER 1
PASSENGER WEB UI
REST API
DRIVER MANAGEMENT
REST API
PAYMENTS
DRIVER WEB UI
REST API
TRIP MANAGEMENT
REST API
NOTIFICATION
ADAPTER 2
ADAPTER 3

Why Microservices!

- Successful applications often live a very long time + **Technology changes**

⇒ Need to be able to easily “modernize” application!

⇒ Need to deliver changes rapidly, frequently, and reliably.

- More complex Applications.. App **keep growing up**

⇒Need to divide team

⇒Need to improve testability, deployability, maintainability, modularity, evolvability.

Characteristics of Microservices

- **Independently deployable**: Each microservice can be developed, deployed, and updated without affecting other services.
- **Decentralized data management**: Every microservice owns and manages its own database, avoiding tight data coupling between services.
- **Technology agnostic**: Microservices can be built using different programming languages, databases, and frameworks, depending on the service’s needs.
- **Focused on a single business capability**: Each microservice is designed to handle one specific business function, making the system modular and easier to understand.

But.. Loose coupling is essential!

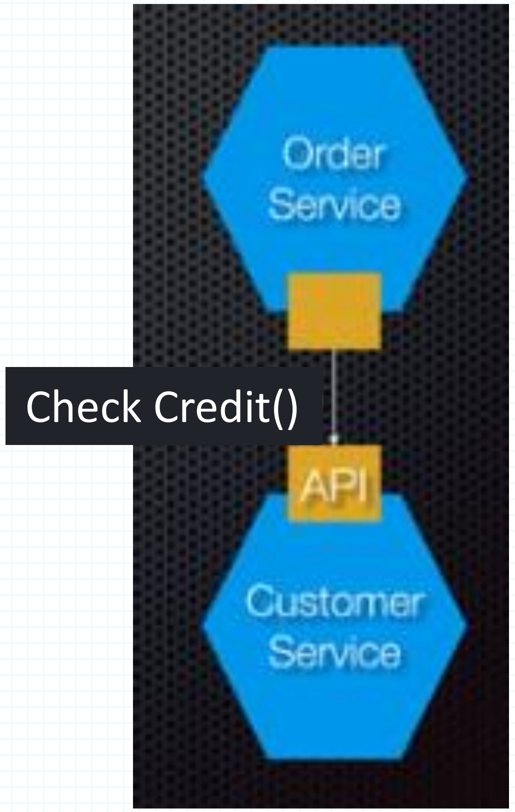
- **Service collaborate.. Thus:**

⇒ Design Time Coupling:

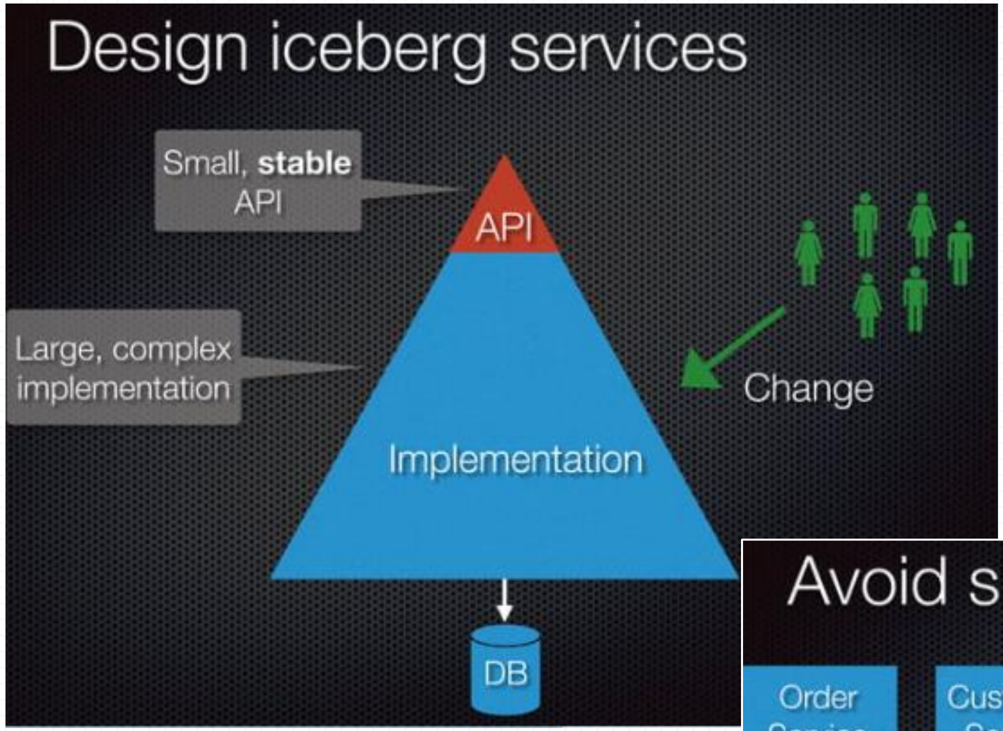
Change Service A -> change Service B

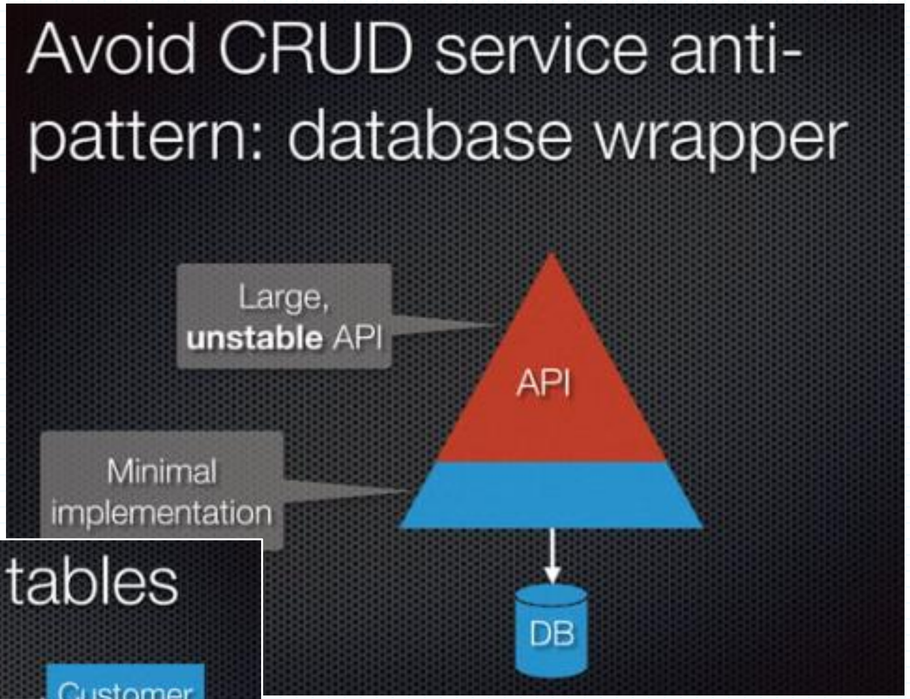
⇒ Runtime Coupling:

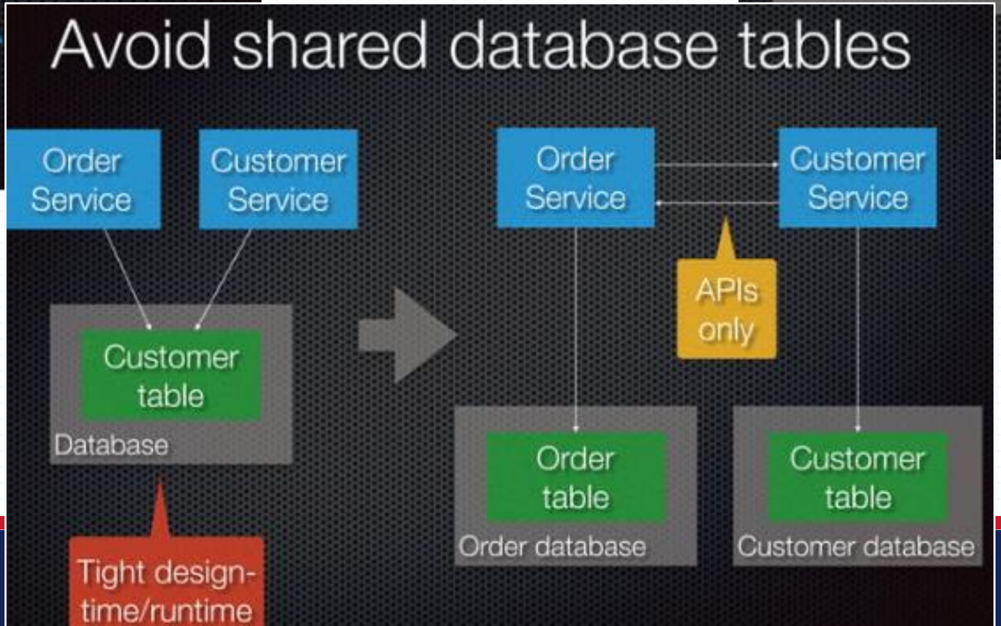
Service A cannot respond to a synchronous request until service B responds



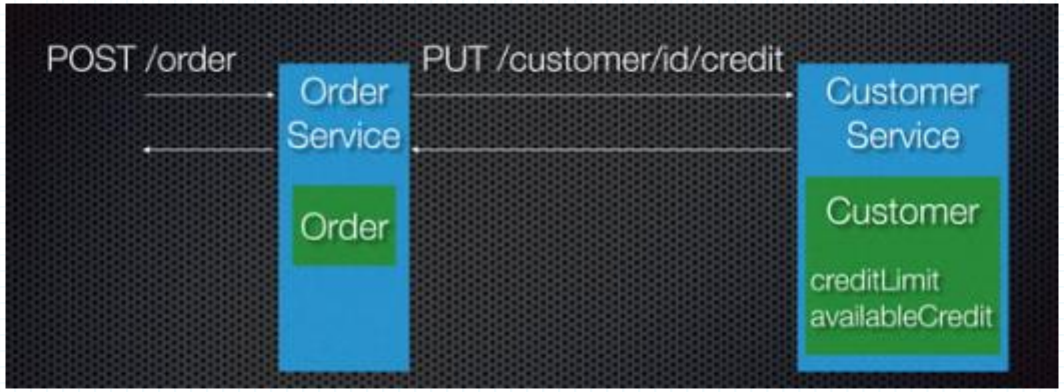
Design Time Coupling… Solutions

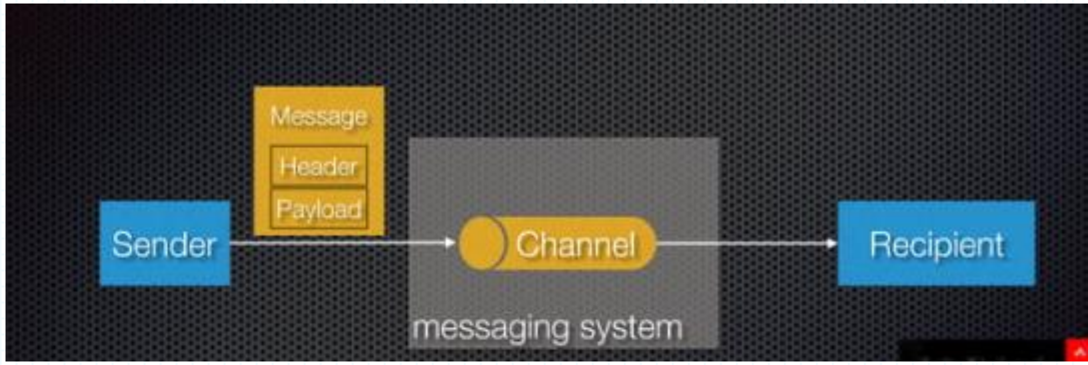






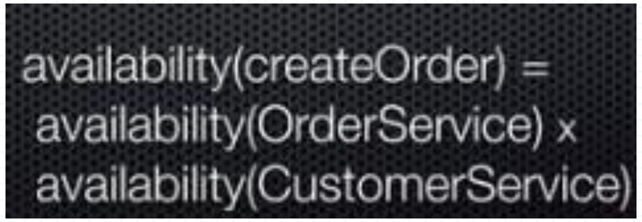
Runtime Coupling … Solution





Reduce Availability!

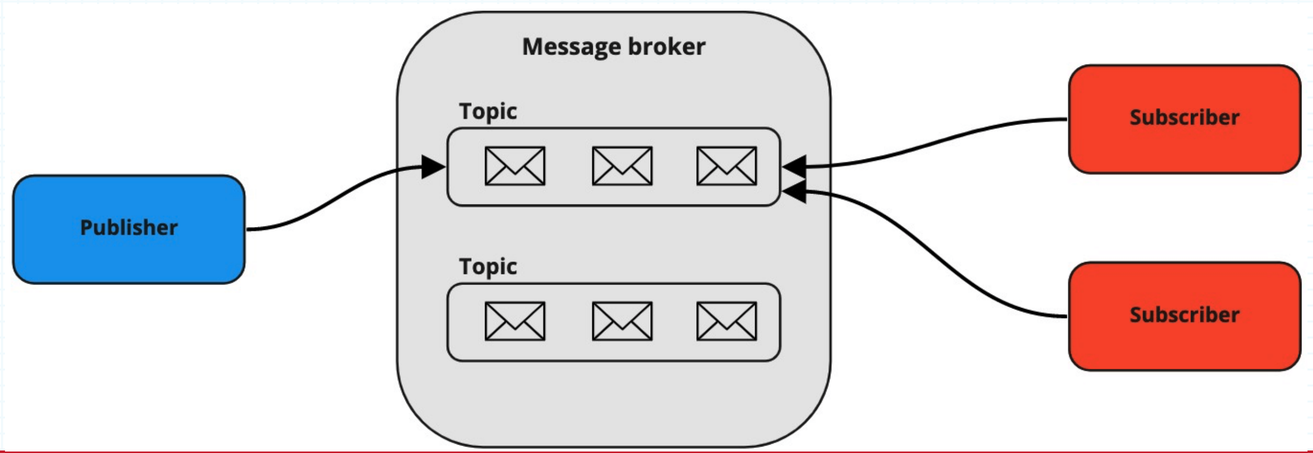
Use Async Messaging



availability(createOrder) =
availability(OrderService) x
availability(CustomerService)

What is a Message Broker?

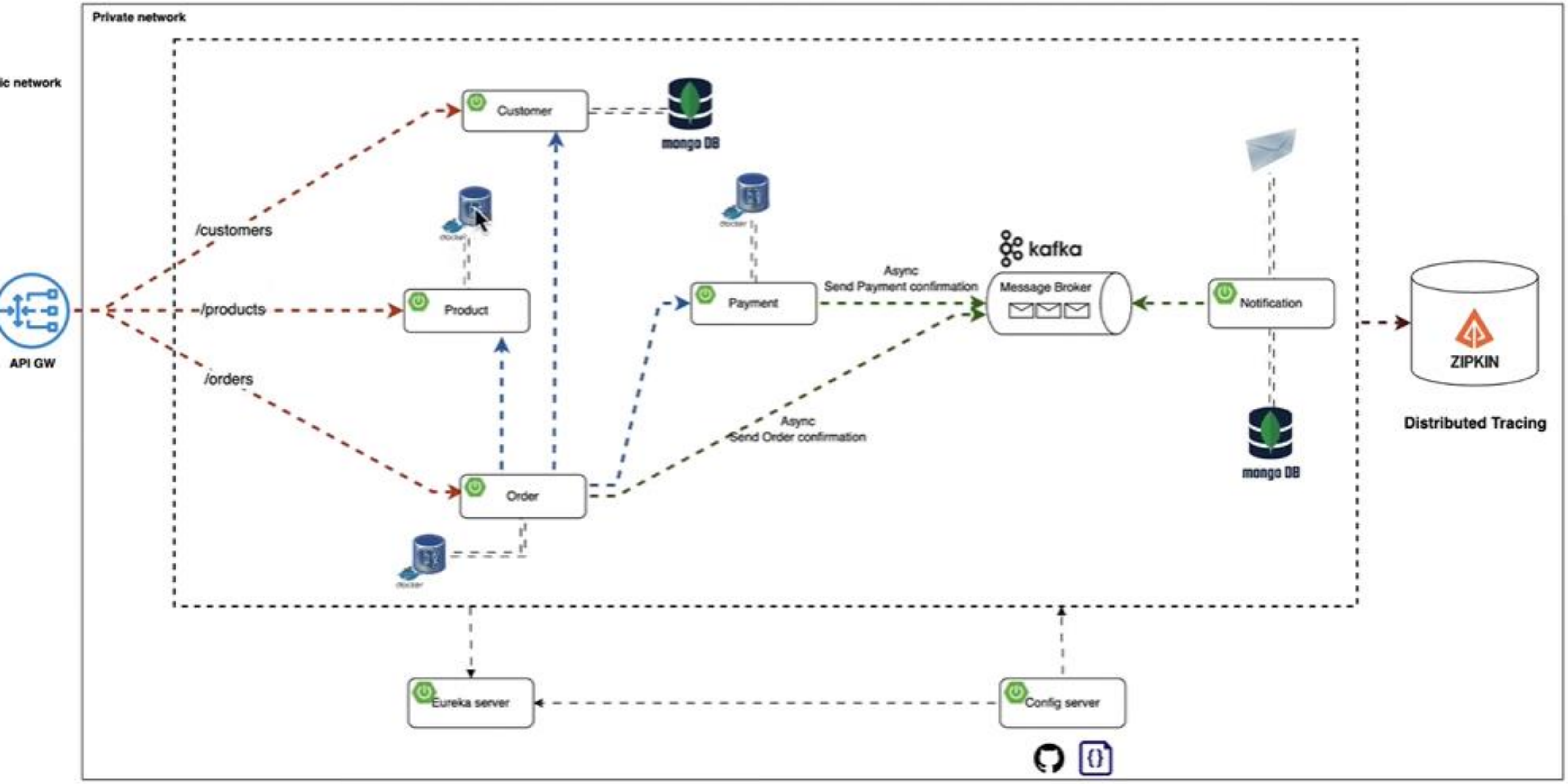
- **Middleware for message transmission between services**: A message broker acts as an intermediary that routes messages between services to enable smooth and reliable communication.
- **Decouples sender and receiver**: It allows services to interact without needing to know each other directly, increasing flexibility and resilience.
- **Supports asynchronous communication**: Message brokers let services send and receive messages without blocking, enabling faster and more scalable systems.



Popular Message Brokers

- RabbitMQ: Queue-based, AMQP support

- Apache Kafka: Distributed, high-throughput, stream processing


Private network
ic network
API GW
/customers
/products
/orders
Customer
mongo DB
Product
Payment
Order
Async
Send Payment confirmation
kafka
Message Broker
Notification
mongo DB
Async
Send Order confirmation
ZIPKIN
Distributed Tracing
Eureka server
Config server