BITS Pilani
Pilani Campus

Microservices Contd.

Prof. Akanksha Bharadwaj
Asst. Professor, CSIS Department

innovate achieve lead

BITS Pilani
Pilani Campus

SE ZG583, Scalable Services
Lecture No. 6

Obstacles to decomposing an application into services

- Network latency
- Reduced availability due to synchronous communication
- Maintaining data consistency across services
- Obtaining a consistent view of the data
- God services preventing decomposition

Defining service APIs

Introduction

A service API operation exists for one of two reasons:

- **To Expose Business Capabilities** – The operation provides functionality that aligns with a business use case, such as processing orders, managing users, or retrieving product details.

- **To Support System Integration** – The operation facilitates interaction between services, allowing them to communicate, share data, and maintain consistency across the system.

Key Principles of Service APIs

- **Loose Coupling** – Minimize dependencies between services.
- **High Cohesion** – Each API should be specific to a microservice's functionality.
- **Standardized Communication** – Use REST, gRPC, GraphQL, or event-driven messaging.
- **Backward Compatibility** – Ensure versioning to prevent breaking changes.
- **Security** – Implement authentication, authorization, and encryption.

Assigning system operations to services

- Many system operations neatly map to a service, but sometimes the mapping is less obvious.

We can either

- Assign an operation to a service that needs the information provided by the operation

- Assign an operation to the service that has the information necessary to handle it

Example

- Mapping system operations to services in the FTGO application

Service	Operations
Consumer Service	createConsumer()
Order Service	createOrder()
Restaurant Service	findAvailableRestaurants()
Kitchen Service	■ acceptOrder() ■ noteOrderReadyForPickup()
Delivery Service	■ noteUpdatedLocation() ■ noteDeliveryPickedUp() ■ noteDeliveryDelivered()

Determining the APIs required to support collaboration between services

- Some system operations are handled entirely by a single service

- Some system operations span across multiple services.

Example

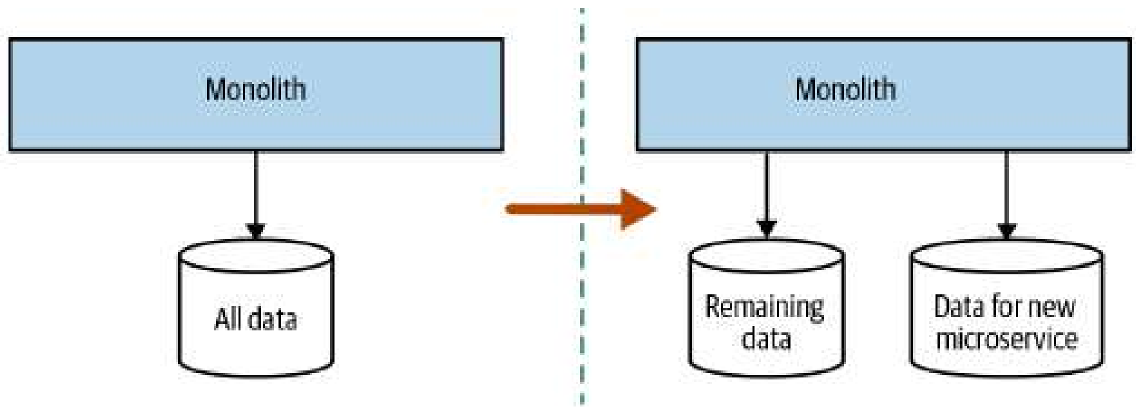
Service	Operations	Collaborators
Consumer Service	verifyConsumerDetails()	—
Order Service	createOrder()	■ Consumer Service verifyConsumerDetails() ■ Restaurant Service verifyOrderDetails() ■ Kitchen Service createTicket() ■ Accounting Service authorizeCard()
Restaurant Service	■ findAvailableRestaurants() ■ verifyOrderDetails()	—
Kitchen Service	■ createTicket() ■ acceptOrder() ■ noteOrderReadyForPickup()	■ Delivery Service scheduleDelivery()
Delivery Service	■ scheduleDelivery() ■ noteUpdatedLocation() ■ noteDeliveryPickedUp() ■ noteDeliveryDelivered()	—
Accounting Service	■ authorizeCard()	—

innovate achieve lead

BITS Pilani
Pilani Campus

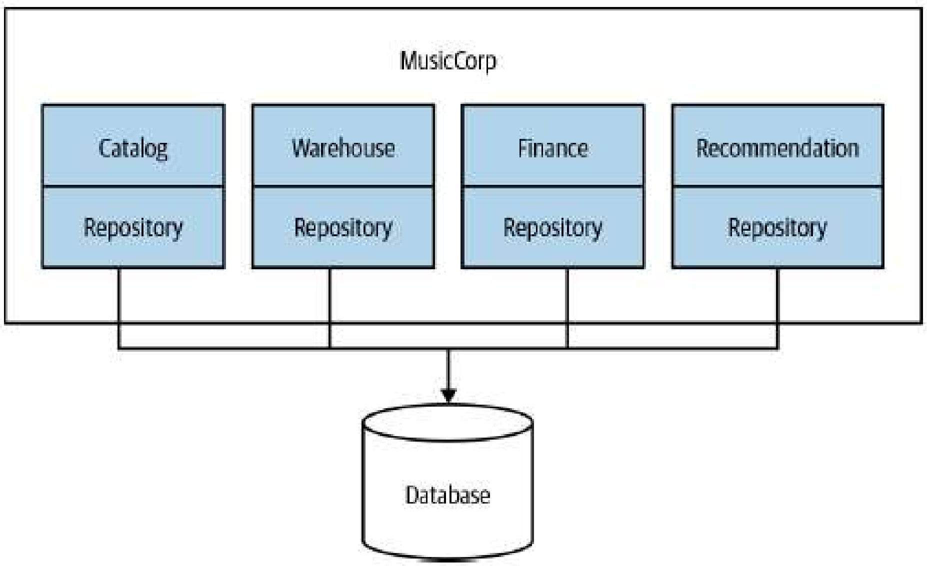
Transition from monolith to microservices

Split the Database First

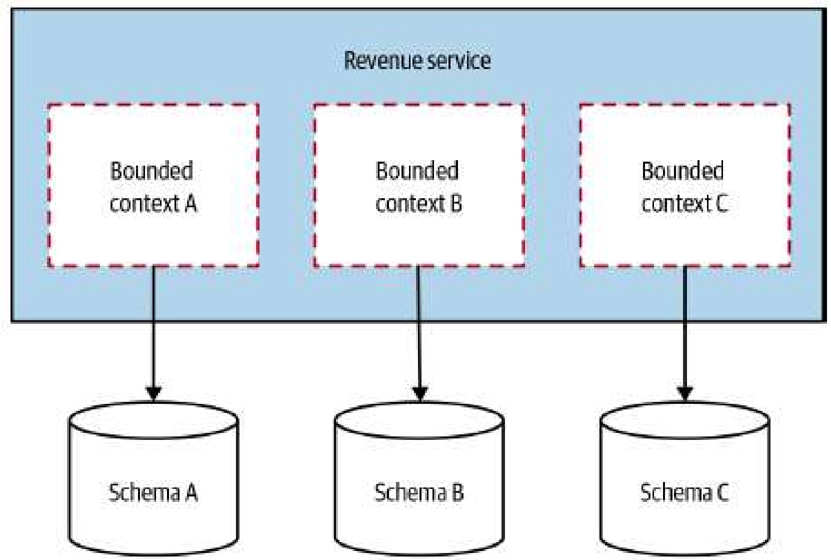


Split the Database First

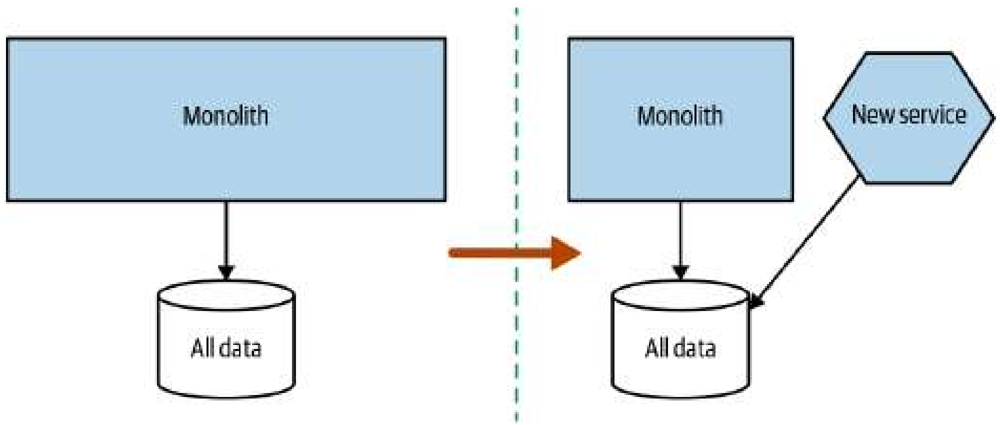
Pattern: Repository per bounded context



Pattern: Database per bounded context

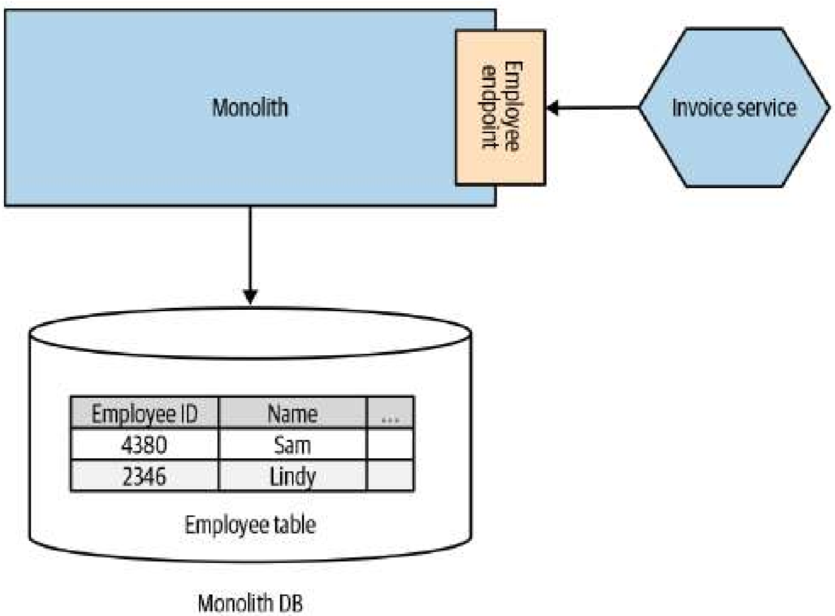


Split the Code First

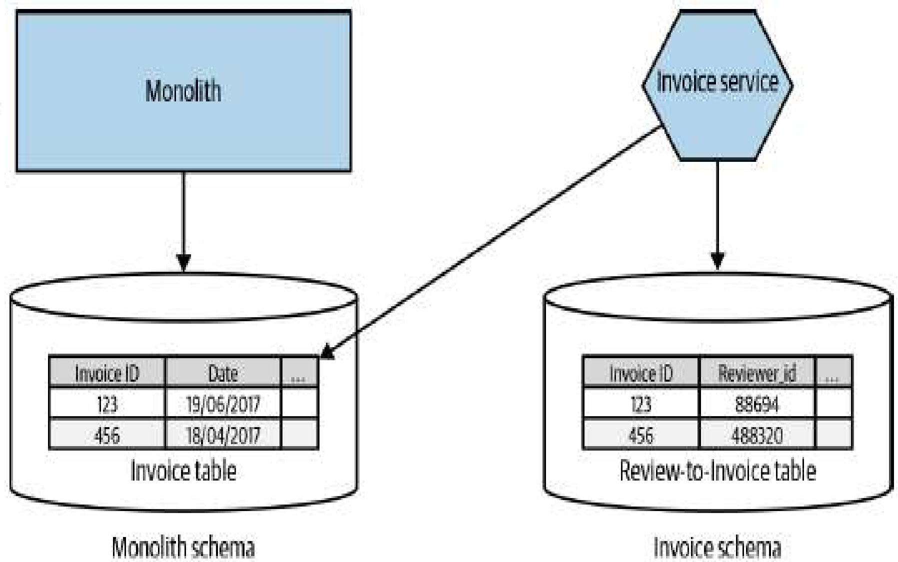


Split the Code First

Pattern: Monolith as data access layer



Pattern: Multi-schema storage



innovate achieve lead

BITS Pilani
Pilani Campus

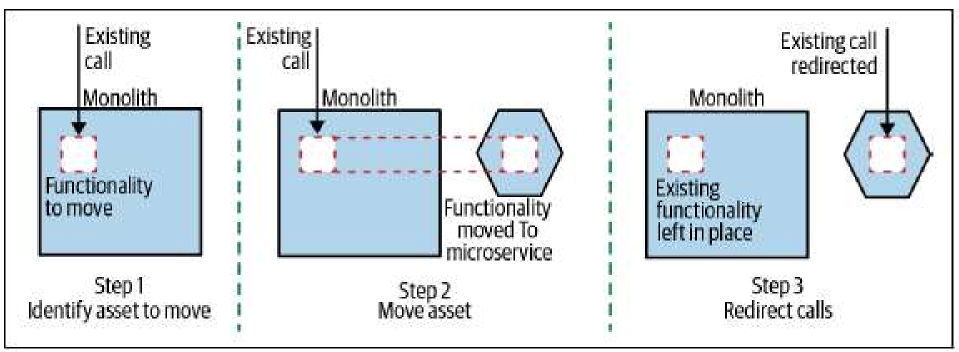
Patterns for Monolith to Microservices

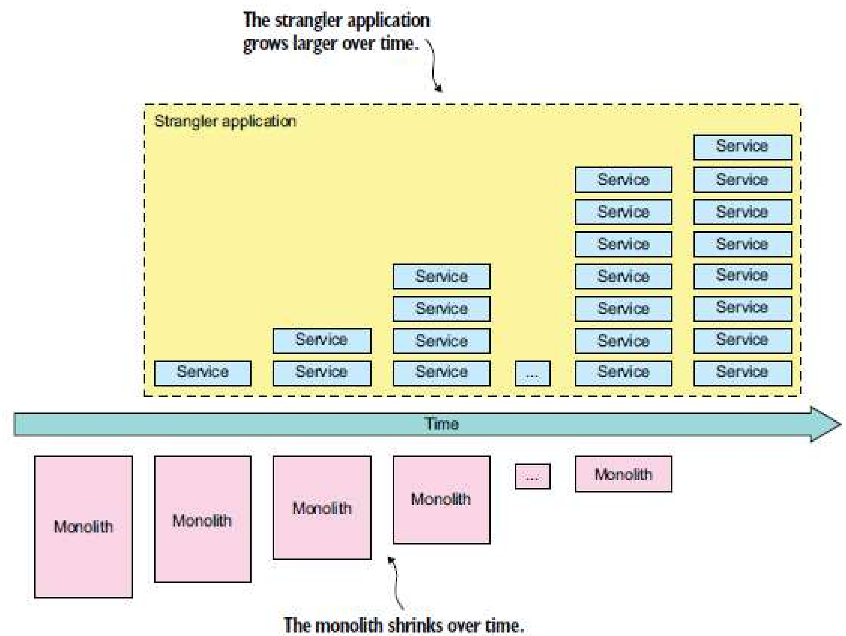
Rebuild From Scratch

- One of the biggest challenges for us was having a good understanding of the legacy system.
- Can not use the system until complete
- Longer duration required

Strangler Pattern

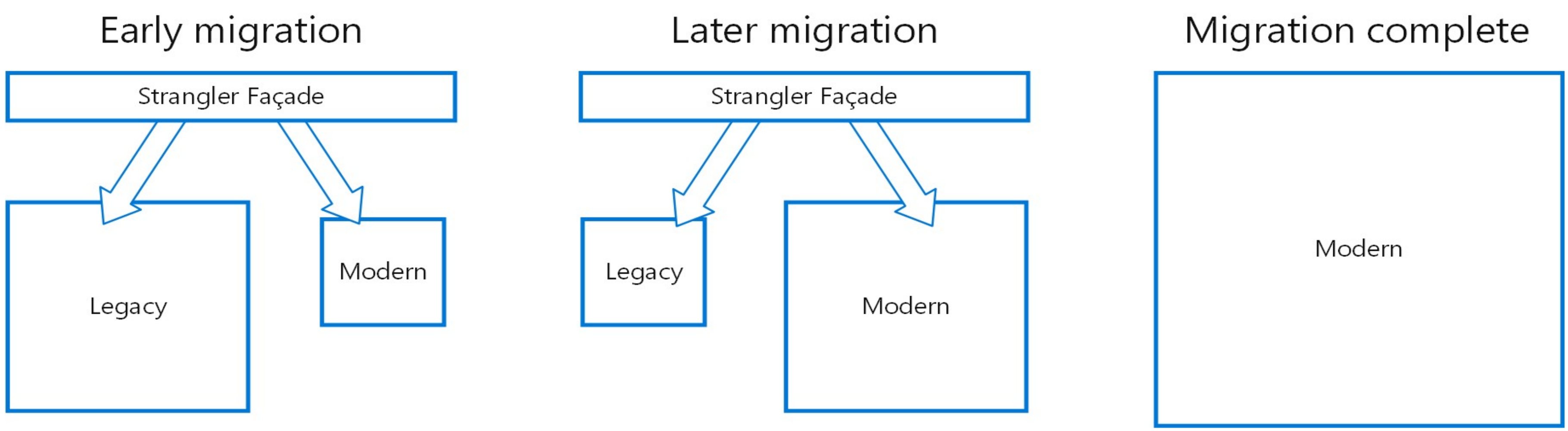
- The Strangler Pattern is a popular design pattern to incrementally transform your monolithic application into microservices by replacing a particular functionality with a new service.
- Any new feature to be added is done as part of the new service





Strangler Pattern

Steps involved in transition



Strangler Pattern: Issues

- What component to start with?
- How to handle services and data stores that are potentially used by both new and legacy systems?
- Migration

When not to use Strangler Pattern?

- When requests to the back-end system cannot be intercepted.
- For smaller systems where the complexity of wholesale replacement is low.

innovate achieve lead

BITS Pilani
Pilani Campus

Communication Protocols

Aspects of communication

Communication Type

- **Synchronous protocol:** The client sends a request and waits for a response from the service.
- **Asynchronous protocol:** The client code or message sender usually doesn't wait for a response.

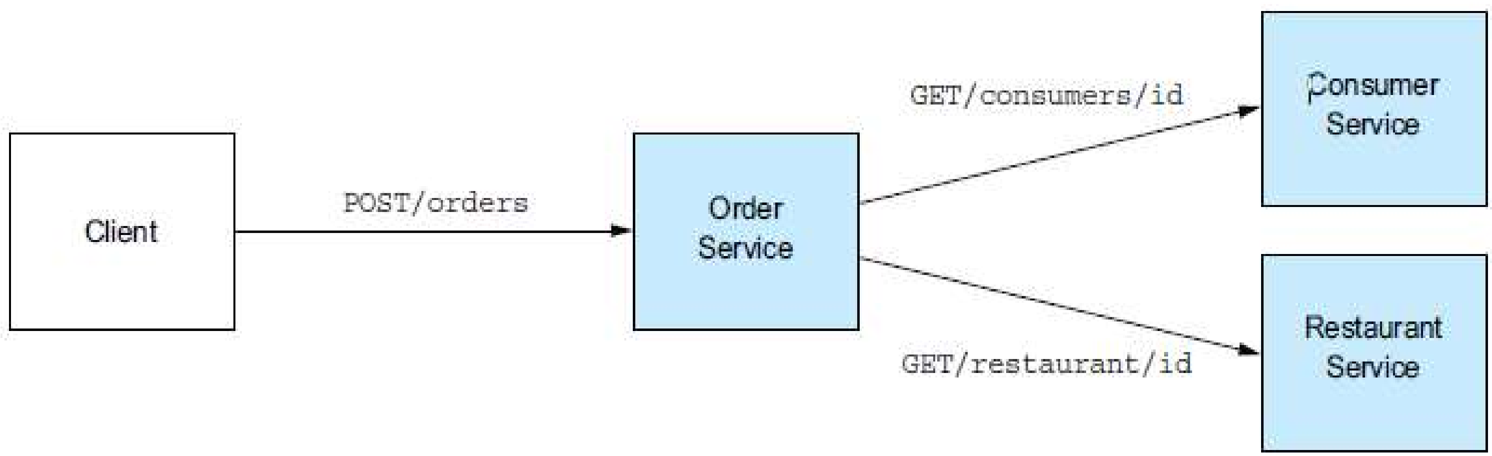
Number of Receivers

- Single receiver
- Multiple receivers

Synchronous communication

- REST is an extremely popular IPC mechanism under this category

Example: FTGO CreateOrder request



Representational state transfer (REST)

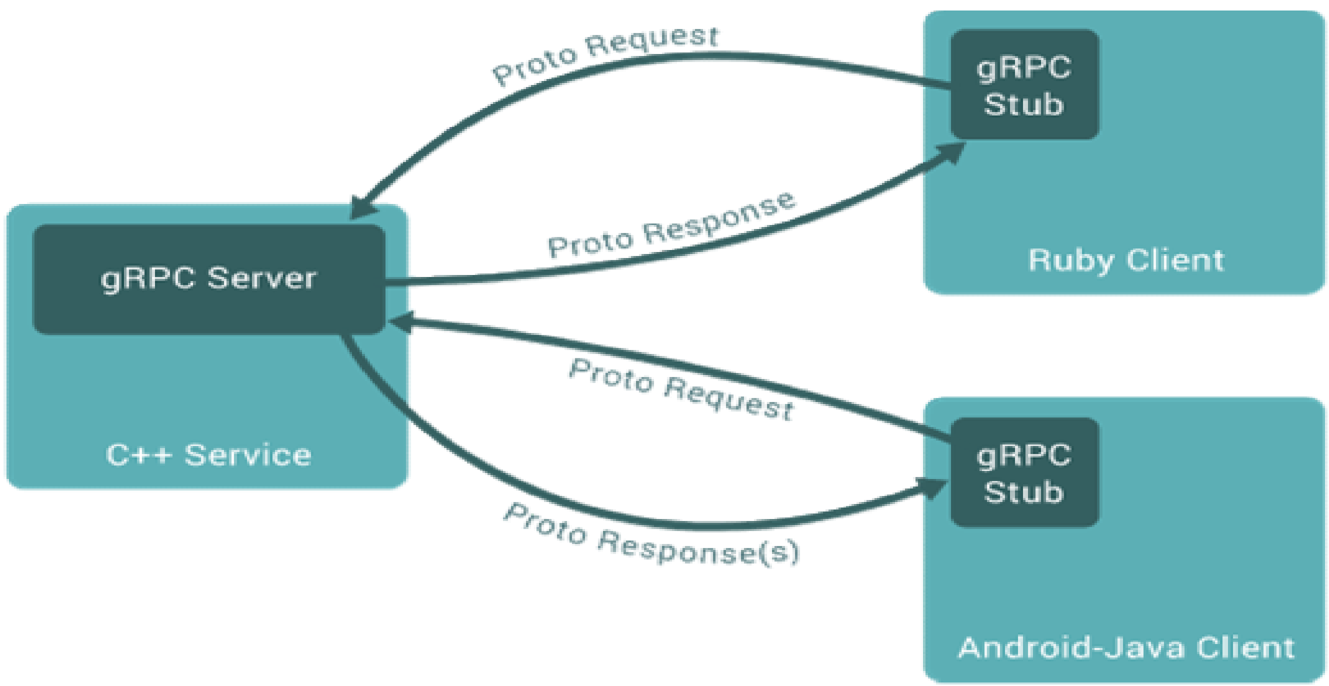
- It is a software architectural style that was created to guide the design and development of the architecture for the World Wide Web.

Guiding Principles of REST

- Resource-Oriented Design
- Stateless
- Cacheable
- Uniform interface
- API Versioning
- Service Discovery & Load Balancing

gRPC: Introduction

gRPC (Google Remote Procedure Call) is a high-performance, language-agnostic RPC (Remote Procedure Call) framework that is well-suited for microservices-based architectures



gRPC

- gRPC clients and servers can run and talk to each other in a variety of environments - from servers inside Google to your own desktop - and can be written in any of gRPC's supported languages.

- So, for example, you can easily create a gRPC server in Java with clients in Go, Python, or Ruby.

- In addition, the latest Google APIs will have gRPC versions of their interfaces, letting you easily build Google functionality into your applications

Why use gRPC in Microservices?

1. **High Performance & Efficiency**

- Uses Protocol Buffers (Protobuf), a compact, binary format that is faster and smaller than JSON over HTTP.
- Uses HTTP/2 for multiplexed connections, reducing latency and improving efficiency.

2. Strongly Typed Contracts

- Enforces a strict schema using .proto files, ensuring consistency between services.
- Auto-generates client and server code in multiple languages, reducing manual coding errors.

3. Supports Streaming

Unlike REST, gRPC supports real-time communication using:

- Unary RPC (Single request-response, like REST)
- Server Streaming (Server sends multiple responses for a single request)
- Client Streaming (Client sends multiple requests and gets a single response)
- Bi-directional Streaming (Both client and server send multiple messages in real-time)

4. Language-Agnostic

- gRPC supports multiple programming languages (**Java, Python, Go, C++, etc.**), making it ideal for polyglot microservices environments.

5. Built-in Authentication & Security

- Supports **TLS encryption** for secure communication.
- Works with authentication mechanisms like **OAuth 2.0 and JWT**.

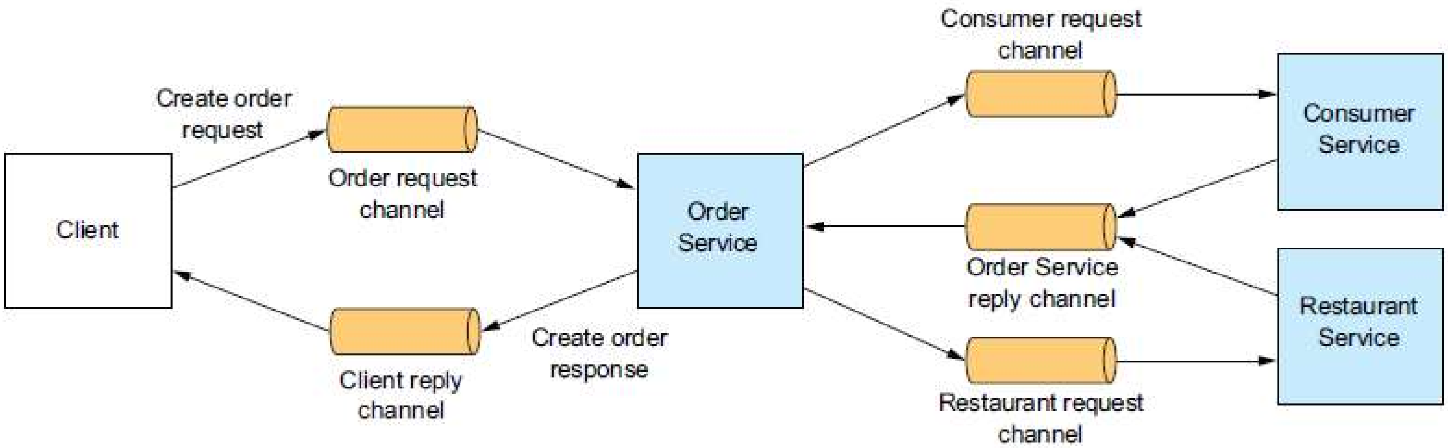
Asynchronous communication

- It means communication which happens ‘out of sync’ or in other words; not in real-time.

- For example, an email to a colleague would be classed as asynchronous communication

Asynchronous Communication Example

- Services communicating by exchanging messages over messaging channels.



innovate achieve lead

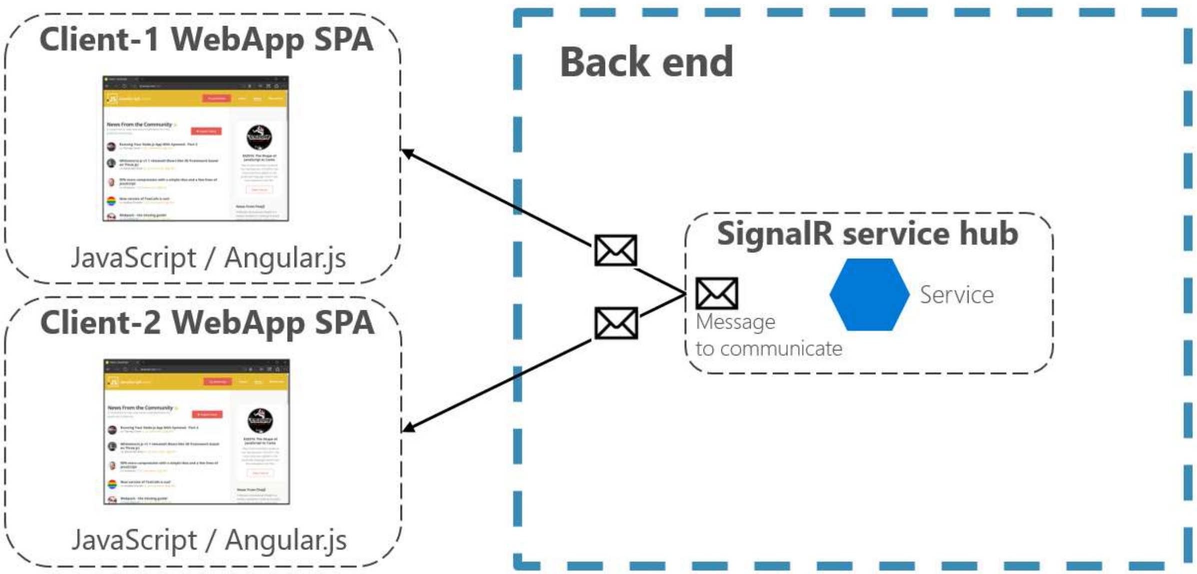
BITS Pilani
Pilani Campus

Communication Styles

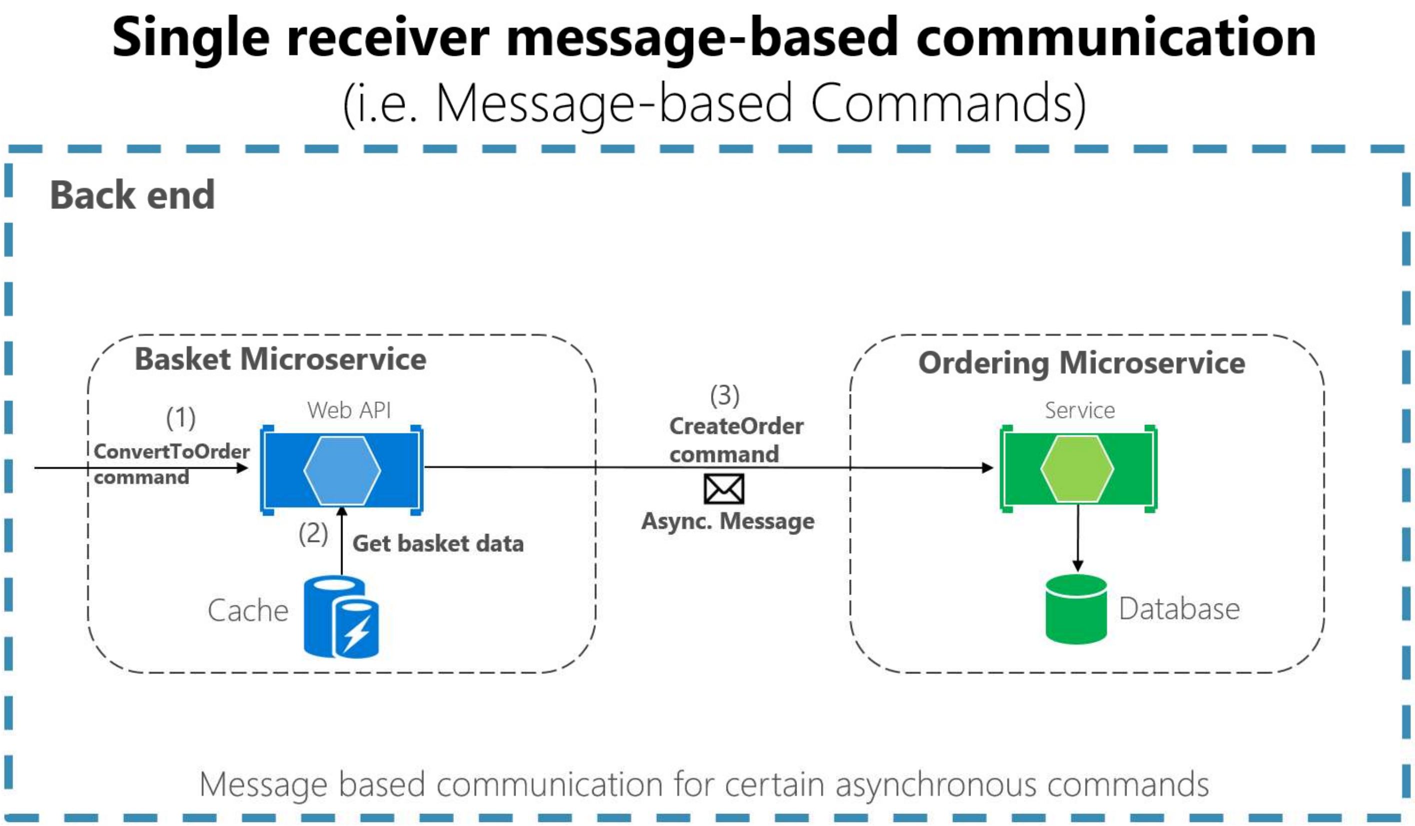
Push and real-time communication based on HTTP

Push and real-time communication based on HTTP

One-to-many communication



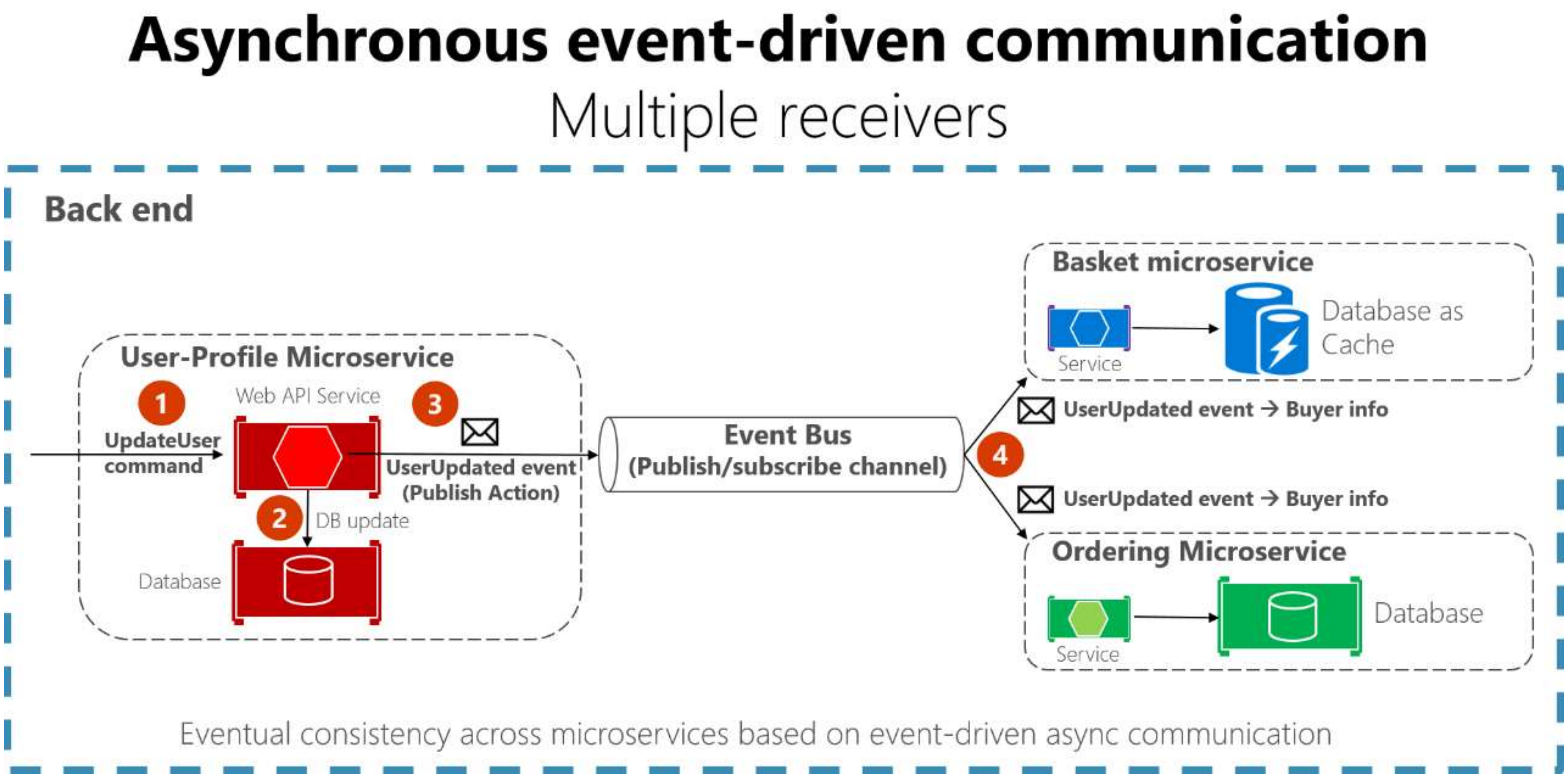
Single-receiver message-based communication



Multiple-receivers message-based communication

Use a publish/subscribe mechanism so that your communication from the sender will be available to all the subscriber microservices or to external applications.

Asynchronous event-driven communication



innovate achieve lead

BITS Pilani
Pilani Campus

Choosing the Right Communication Pattern for Microservices

REST API (Synchronous, Request-Response)

Example Scenario: User Authentication & Profile Management

- A user logs into an **e-commerce platform** using email and password. The system validates the credentials and returns a response.
- Similarly, fetching user details (name, address, order history) should be a simple request-response operation.

gRPC (High-Performance Remote Procedure Calls)

Example Scenario: Ride-Sharing App's Driver Location Tracking

- A passenger requests a ride, and the system continuously updates the **driver's real-time location** every few seconds. Since performance and efficiency are crucial, gRPC's low latency and streaming features make it ideal.

Kafka (Event-Driven, Asynchronous Messaging)

Example Scenario: Order Processing in an E-Commerce System

- When a user places an order, multiple microservices need to react to this event:
  Order Service → Saves order details.
  Inventory Service → Updates stock levels.
  Payment Service → Charges the user.
  Notification Service → Sends confirmation emails/SMS.
- Kafka acts as an **event bus** that ensures all these services process the event without direct coupling.

RabbitMQ (Message Queuing, Reliable Delivery)

Example Scenario: Asynchronous Task Processing in a Banking System

- A banking system needs to process transactions in a **fraud detection microservice**. Instead of waiting synchronously, the **Transaction Service** places a message in a queue, and the **Fraud Detection Service** processes it asynchronously.

WebSockets (Persistent, Real-Time Bi-Directional Communication)

Example Scenario: Stock Market Price Updates

- A stock trading platform needs to show **real-time stock price changes** to users without them having to refresh the page.

Comparison

Feature	REST	gRPC	Kafka	RabbitMQ	WebSockets
Type	Request-Response	Remote Procedure Call	Event Streaming	Message Queue	Persistent Connection
Best For	Web APIs, CRUD	Microservices, Low Latency	High-Volume Events	Reliable Async Tasks	Real-Time Updates
Data Format	JSON/XML	Protobuf	Binary	Any	JSON/Binary
Communication	Synchronous	Synchronous/Streaming	Asynchronous	Asynchronous	Full Duplex
Browser Support	✅ Yes	❌ No	❌ No	❌ No	✅ Yes
Throughput	Medium	High	Very High	High	Medium
Latency	High	Low	Medium	Low	Very Low
Persistence	❌ No	❌ No	✅ Yes	✅ Yes	❌ No

References

- Chapter 2 and 3: Microservices Patterns by Chris Richardson
- Chapter 3: Monolith to Microservices by Sam Newman
- Link: https://microservices.io/patterns
- Link: https://docs.microsoft.com/en-us/dotnet/architecture/microservices/architect-microservice-container-applications/communication-in-microservice-architecture
- https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm
- https://grpc.io/docs/what-is-grpc/introduction/