Microservices Contd.

BITS Pilani
Pilani Campus

Akanksha Bharadwaj
Asst. Professor, CSIS Department

innovate achieve lead

BITS Pilani
Pilani Campus

SE ZG583, Scalable Services
Lecture No. 6

BITS Pilani

Pilani Campus

Database related patterns for Microservices

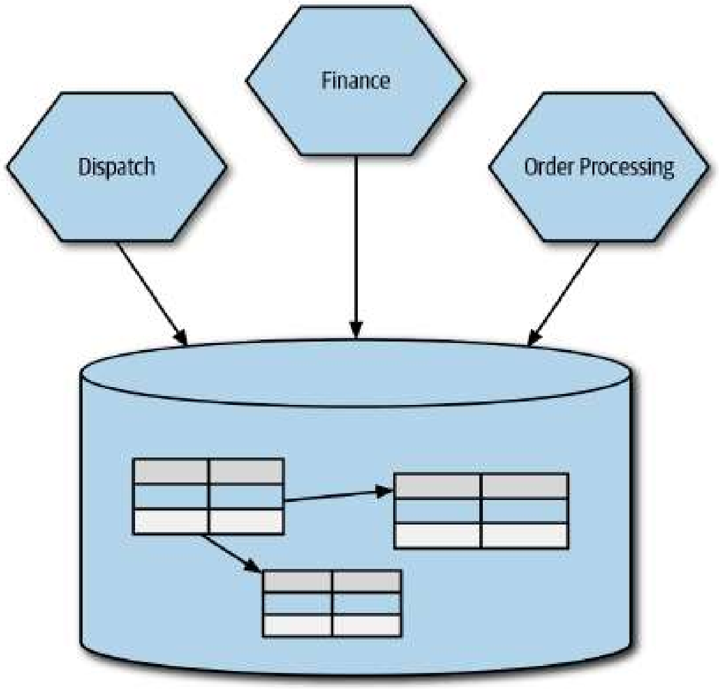
Shared Database pattern

Problems:

- Shared Vs hidden data
- Control on data

When to use:

- Read-only static reference data
- During transition

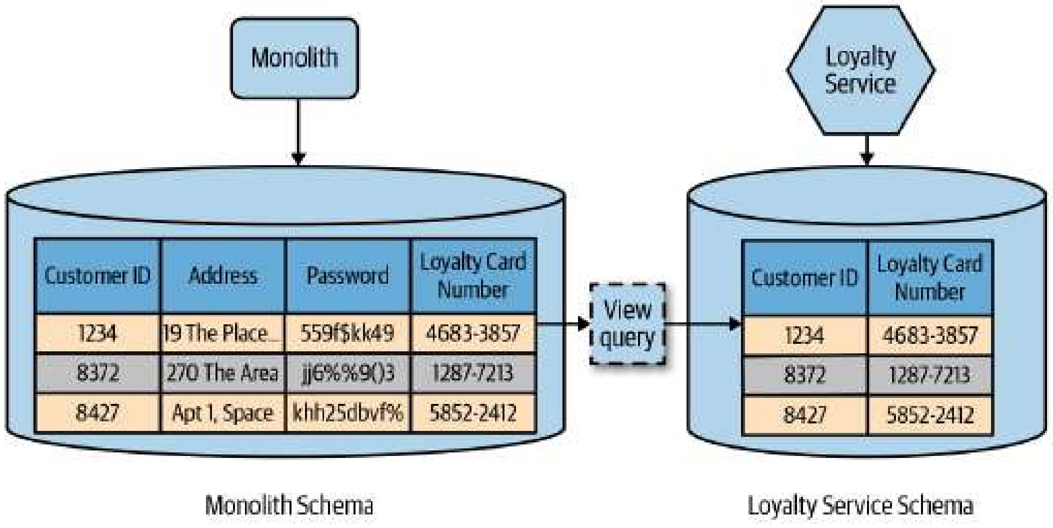


Database View Pattern

- For a single source of data for multiple services, a view can be used to mitigate the concerns regarding coupling

Limitations:

- Views are read-only.

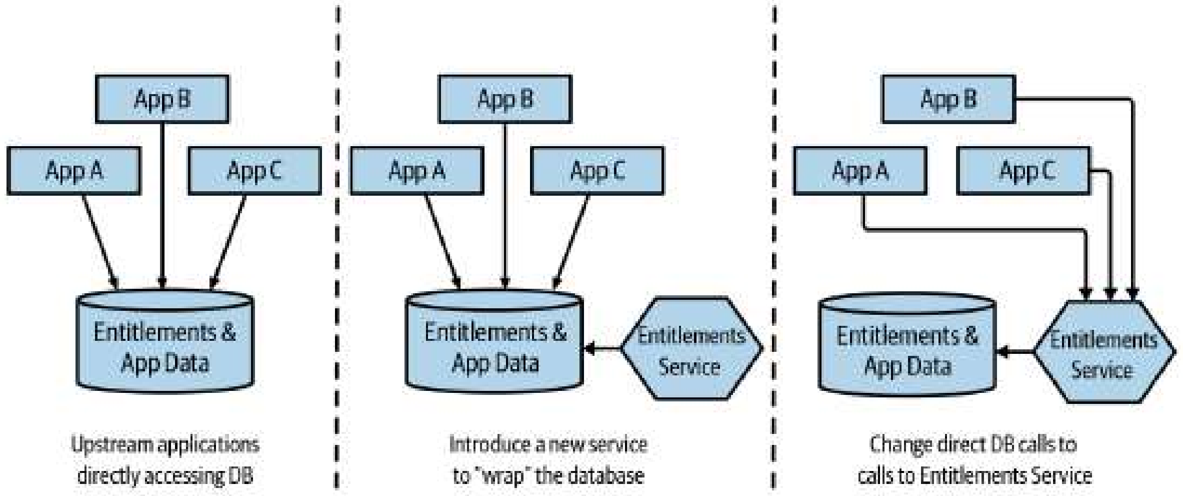


Customer ID	Address	Password	Loyalty Card Number
1234	19 The Place...	559f$kk49	4683-3857
8372	270 The Area	jj6%%9()3	1287-7213
8427	Apt 1, Space	khh25dbvf%	5852-2412

Customer ID	Loyalty Card Number
1234	4683-3857
8372	1287-7213
8427	5852-2412

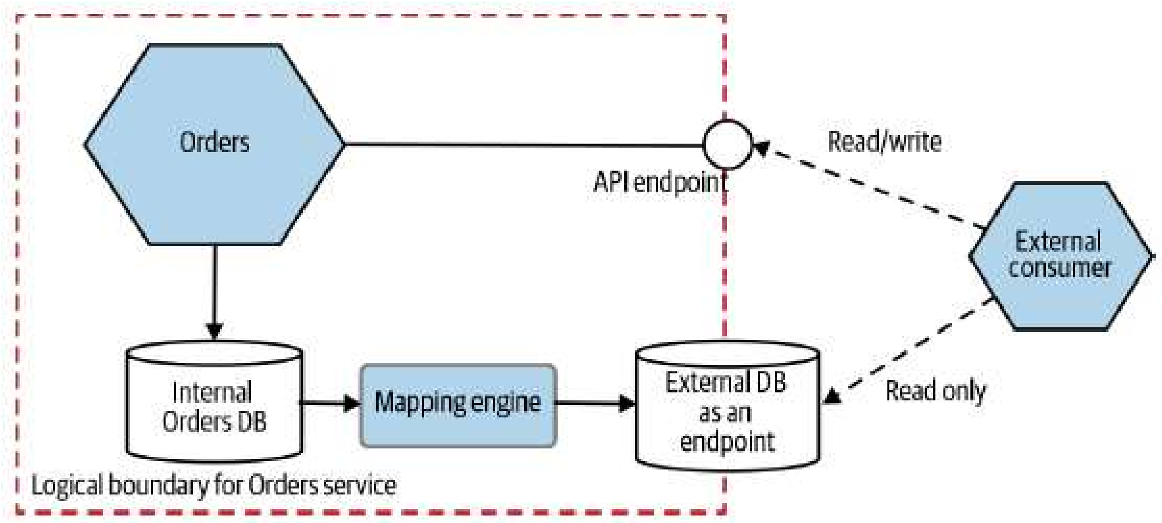
Database Wrapping Service Pattern

- Here we hide the database behind a service that acts as a thin wrapper



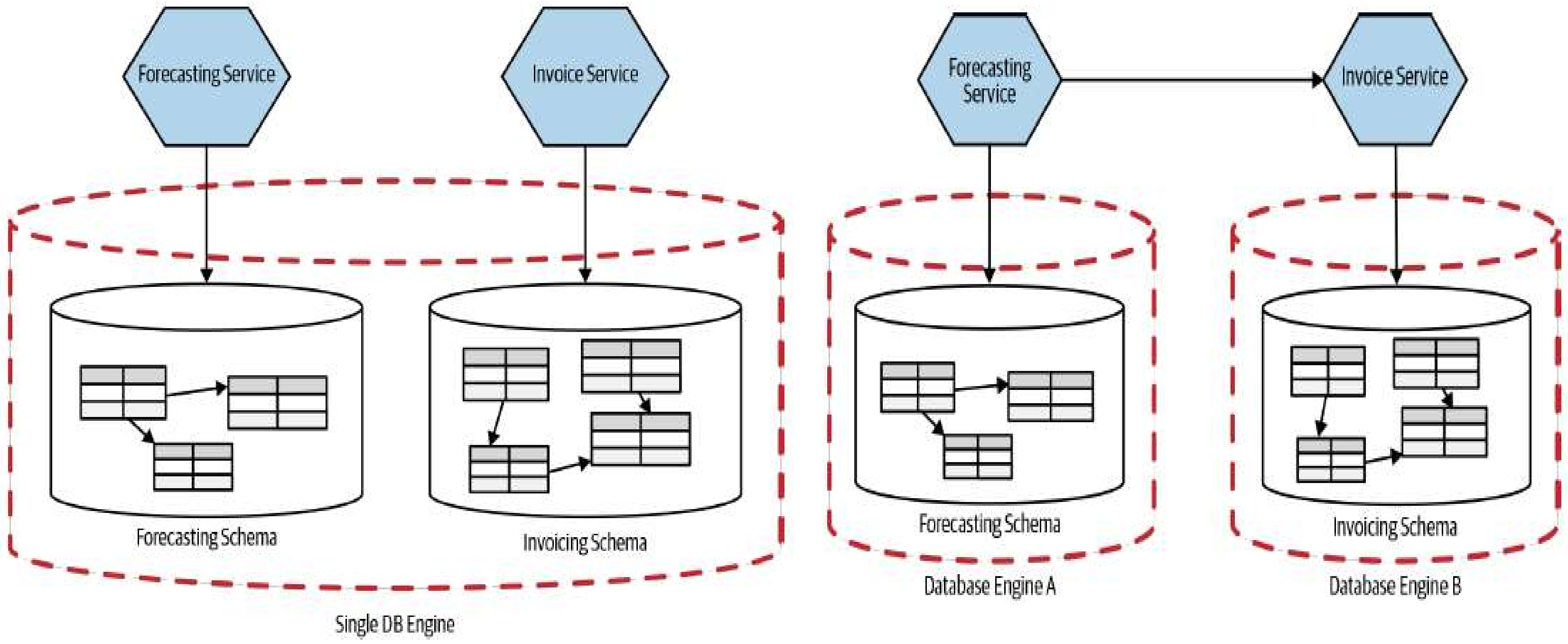
Database-as-a-Service Pattern

Can be used when clients just need a database to query.



Splitting Apart the Database

Physical Versus Logical Database Separation

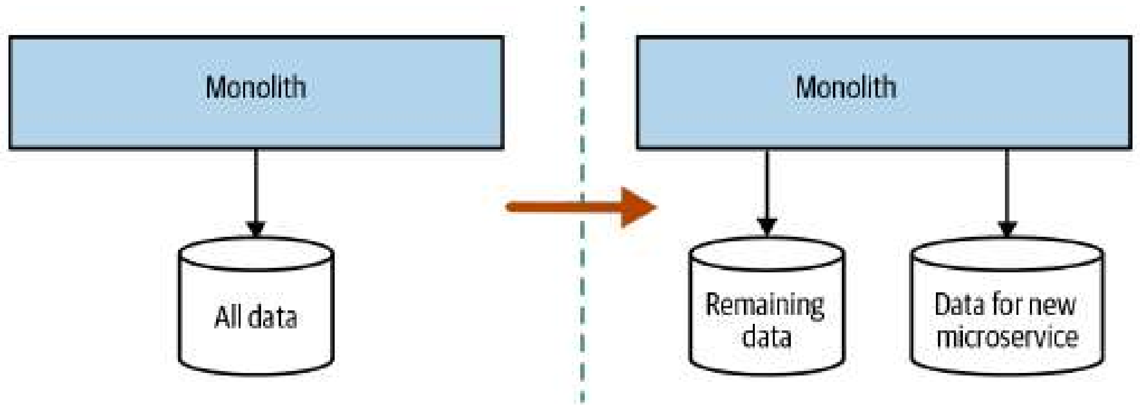


innovate achieve lead

BITS Pilani
Pilani Campus

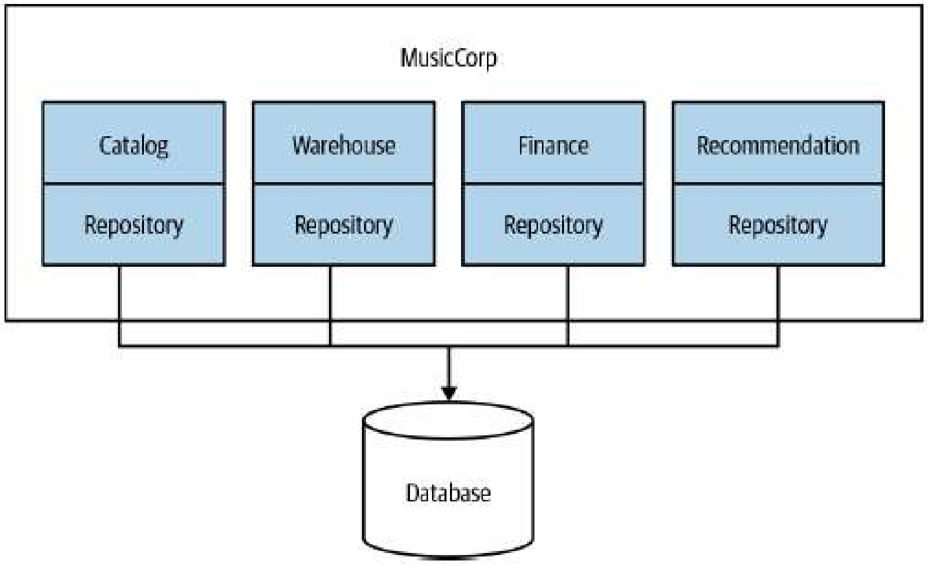
Transition from monolith to microservices

Split the Database First

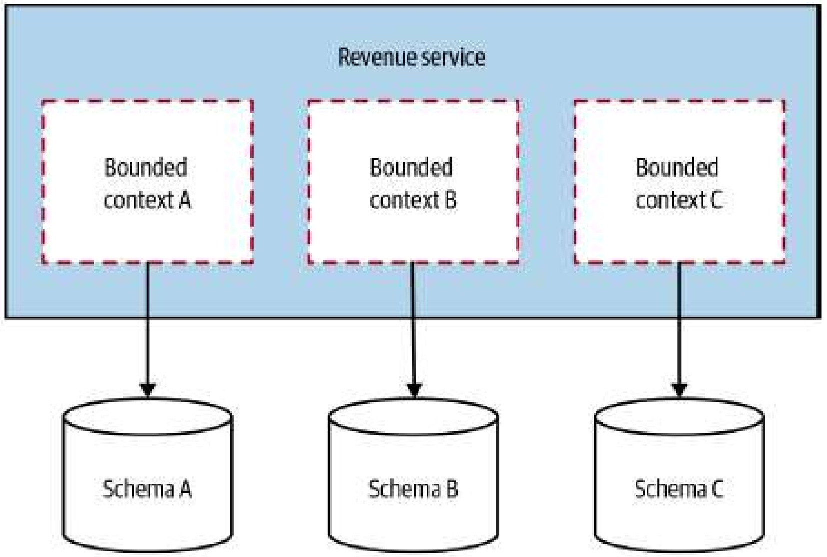


Split the Database First

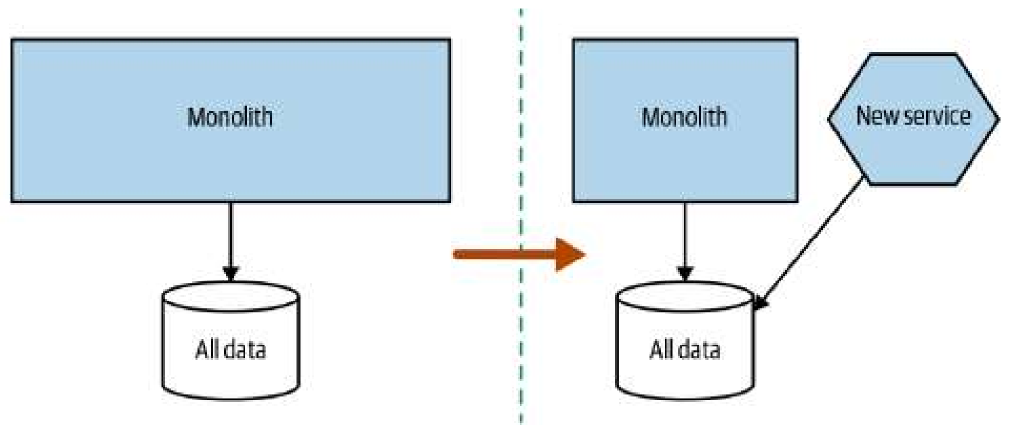
Pattern: Repository per bounded context



Pattern: Database per bounded context

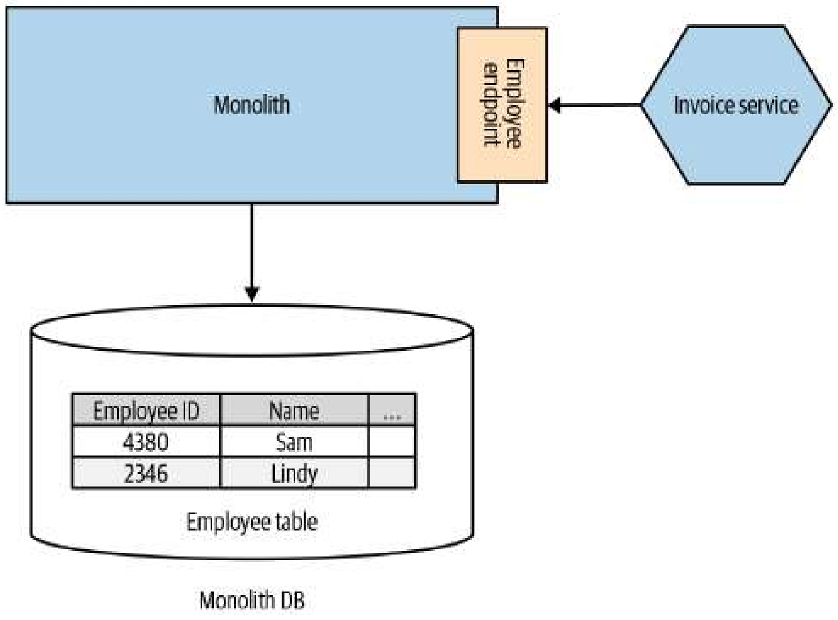


Split the Code First



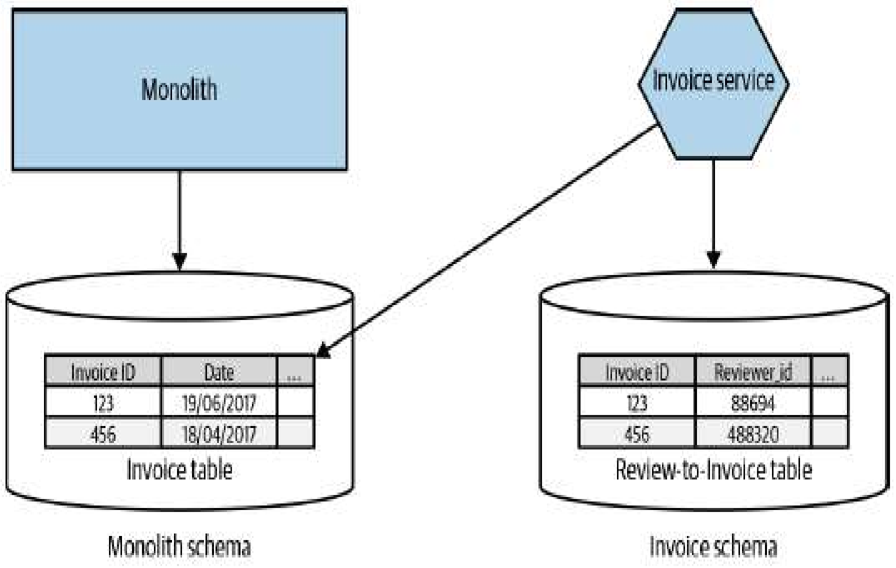
Split the Code First

Pattern: Monolith as data access layer



Employee ID	Name	...
4380	Sam	
2346	Lindy	

Pattern: Multi-schema storage



Invoice ID	Date	...
123	19/06/2017	
456	18/04/2017	

Invoice ID	Reviewer_id	...
123	88694	
456	488320	

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 classes preventing decomposition

BITS Pilani
Pilani Campus

Defining service APIs

Introduction

A service API operation exists for one of two reasons:

- some operations correspond to system operations. They are invoked by external clients and perhaps by other services.
- The other operations exist to support collaboration between services. These operations are only invoked by other services

Assigning system operations to services

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

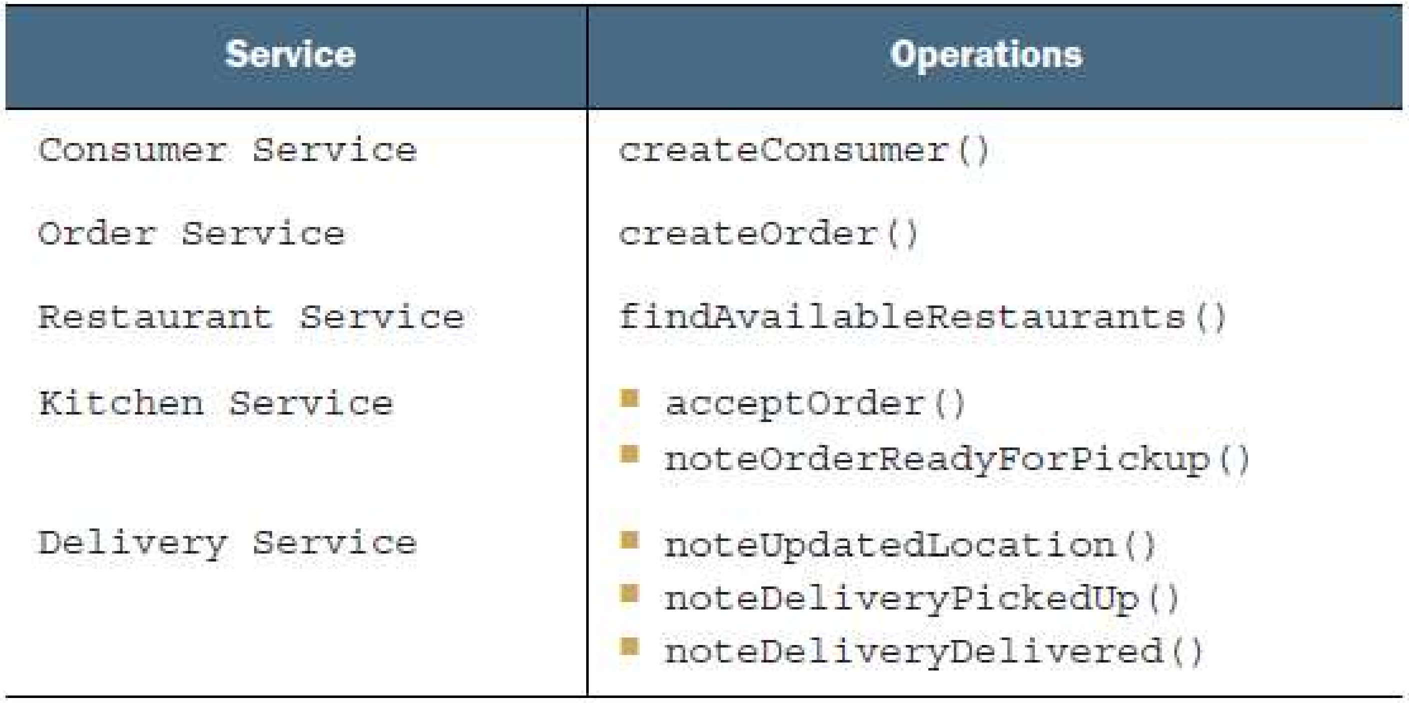
We can either

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

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

Example

- Mapping system operations to services in the FTGO application



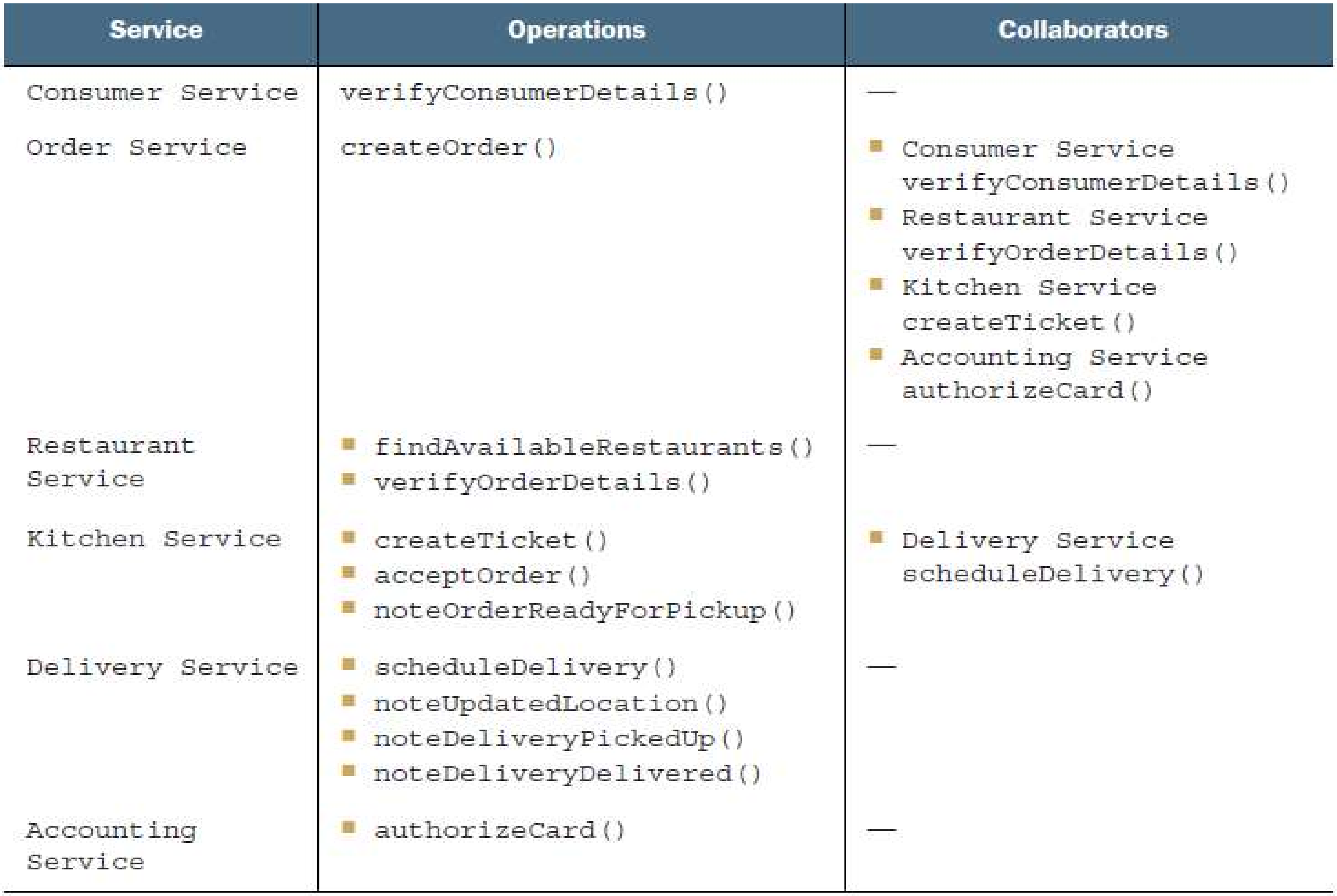
Service	Operations
Consumer Service	createConsumer()
Order Service	createOrder()
Restaurant Service	findAvailableRestaurants()
Kitchen Service	■ acceptOrder()
	■ noteOrderReadyForPickup()
Delivery Service	■ noteUpdatedLocation()
	■ noteDeliveryPickedUp()
	■ noteDeliveryDelivered()

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()	—

Communication Protocols

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.

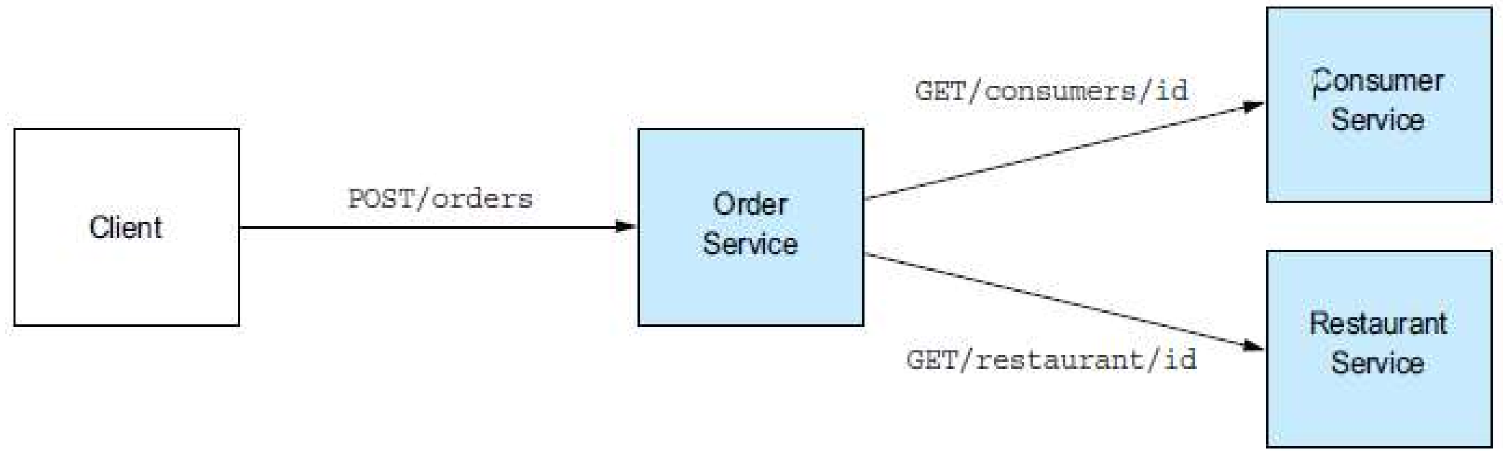
Number of Receivers

- Single receiver
- Multiple receivers

Synchronous communication

- REST is an extremely popular IPC mechanism under this category

Example: FTGO CreateOrder request



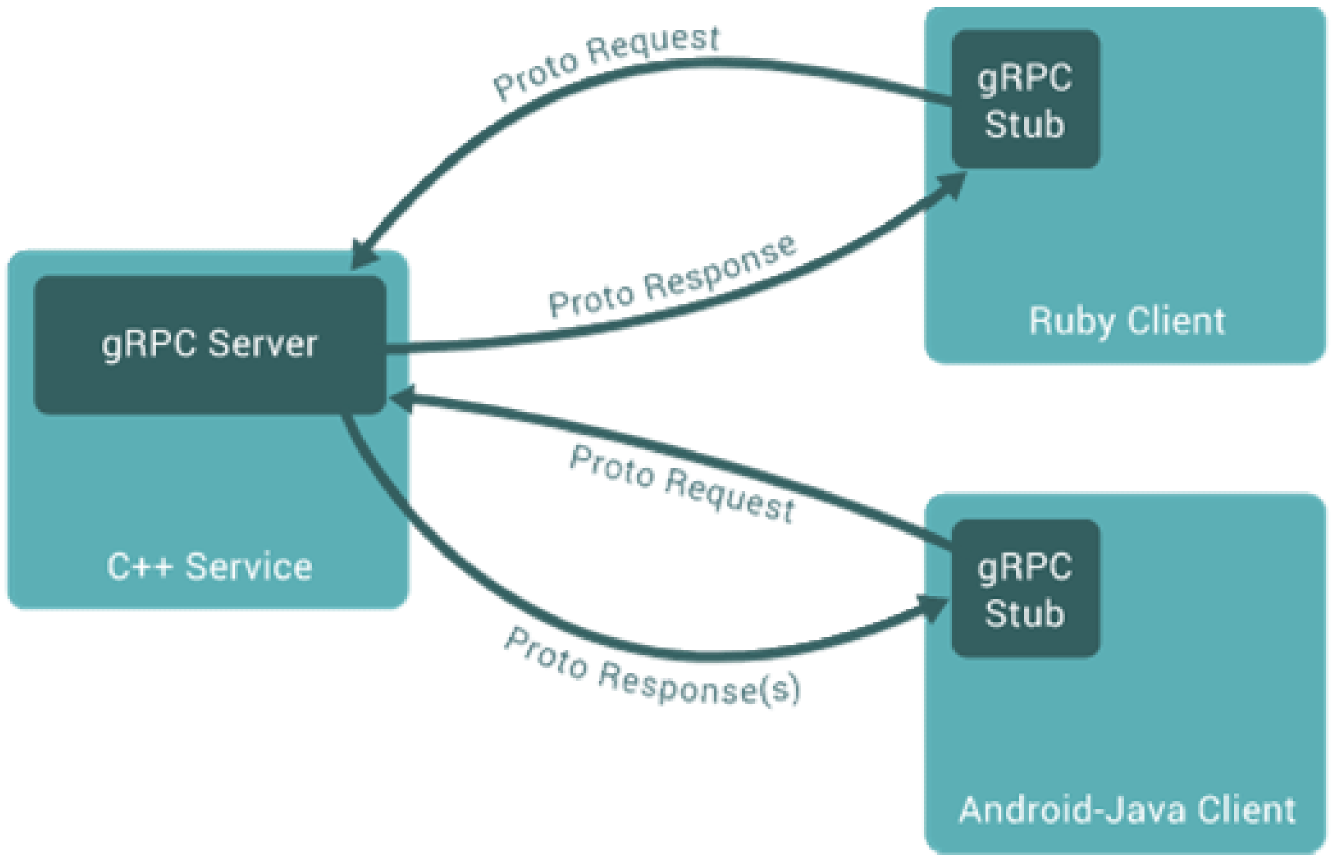
Representational state transfer (REST)

- 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

- Client–server
- Stateless
- Cacheable
- Uniform interface
- Layered system
- Code on demand (optional)

gRPC: Introduction



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

gRPC: Using the API

- On the server side, the server implements the methods declared by the service and runs a gRPC server to handle client calls.

- On the client side, the client has a local object known as *stub* that implements the same methods as the service.

RPC life cycle

Unary RPC

- Once the client calls a stub method, the server is notified that the RPC has been invoked
- The server can then either send back its own initial metadata straight away, or wait for the client's request message.
- Once the server has the client's request message, it does whatever work is necessary to create and populate a response. The response is then returned (if successful) to the client together with status details.
- If the response status is OK, then the client gets the response, which completes the call on the client side

Server streaming RPC

- A server-streaming RPC is similar to a unary RPC, except that the server returns a stream of messages in response to a client's request.
- After sending all its messages, the server's status details and optional trailing metadata are sent to the client.
- This completes processing on the server side. The client completes once it has all the server's messages.

Client streaming RPC

- A client-streaming RPC is similar to a unary RPC, except that the client sends a stream of messages to the server instead of a single message.
- The server responds with a single message along with its status details and optional trailing metadata

Bidirectional streaming RPC

- In a bidirectional streaming RPC, the call is initiated by the client invoking the method. The server can choose to send back its initial metadata or wait for the client to start streaming messages.

- Client- and server-side stream processing is application specific. Since the two streams are independent, the client and server can read and write messages in any order.

Deadlines/Timeouts

- gRPC allows clients to specify how long they are willing to wait for an RPC to complete before the RPC is terminated

- On the server side, the server can query to see if a particular RPC has timed out, or how much time is left to complete the RPC.

RPC termination

- In gRPC, both the client and server make independent and local determinations of the success of the call, and their conclusions may not match.

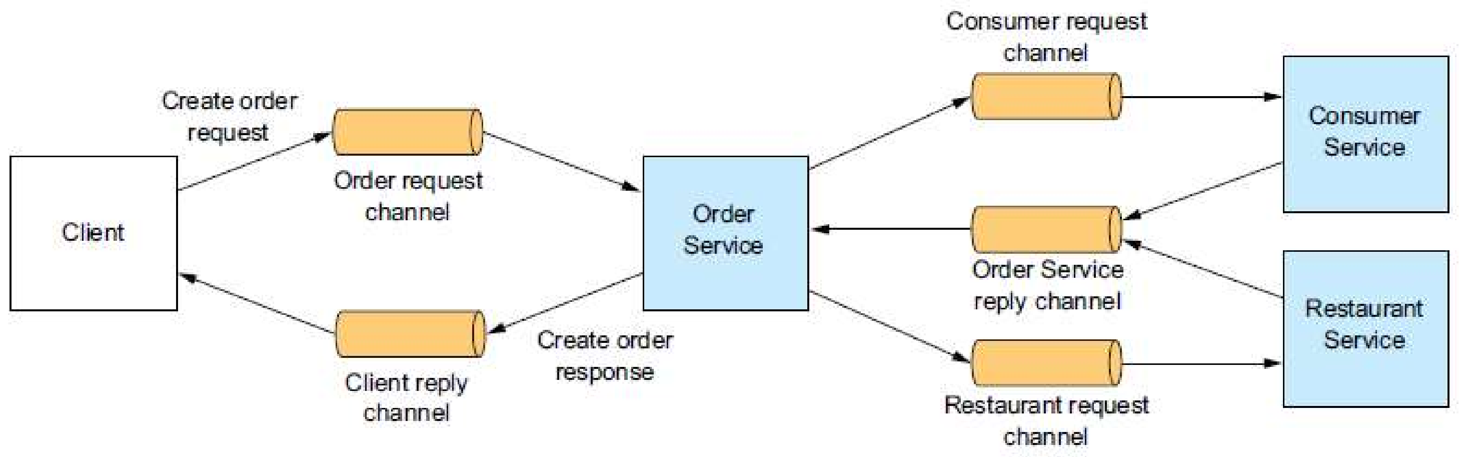
- This means that, for example, you could have an RPC that finishes successfully on the server side but fails on the client side. It's possible for a server to decide to complete before a client has sent all its requests

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

- Services communicating by exchanging messages over messaging channels.



Real Time Examples

- Email
- Messages via any instant messaging app (e.g. WhatsApp messenger, RingCentral Message, Slack)
- Messaging via project management tools such as Basecamp, Trello, Mondays etc.
- Intranets such as Yammer or Sharepoint.

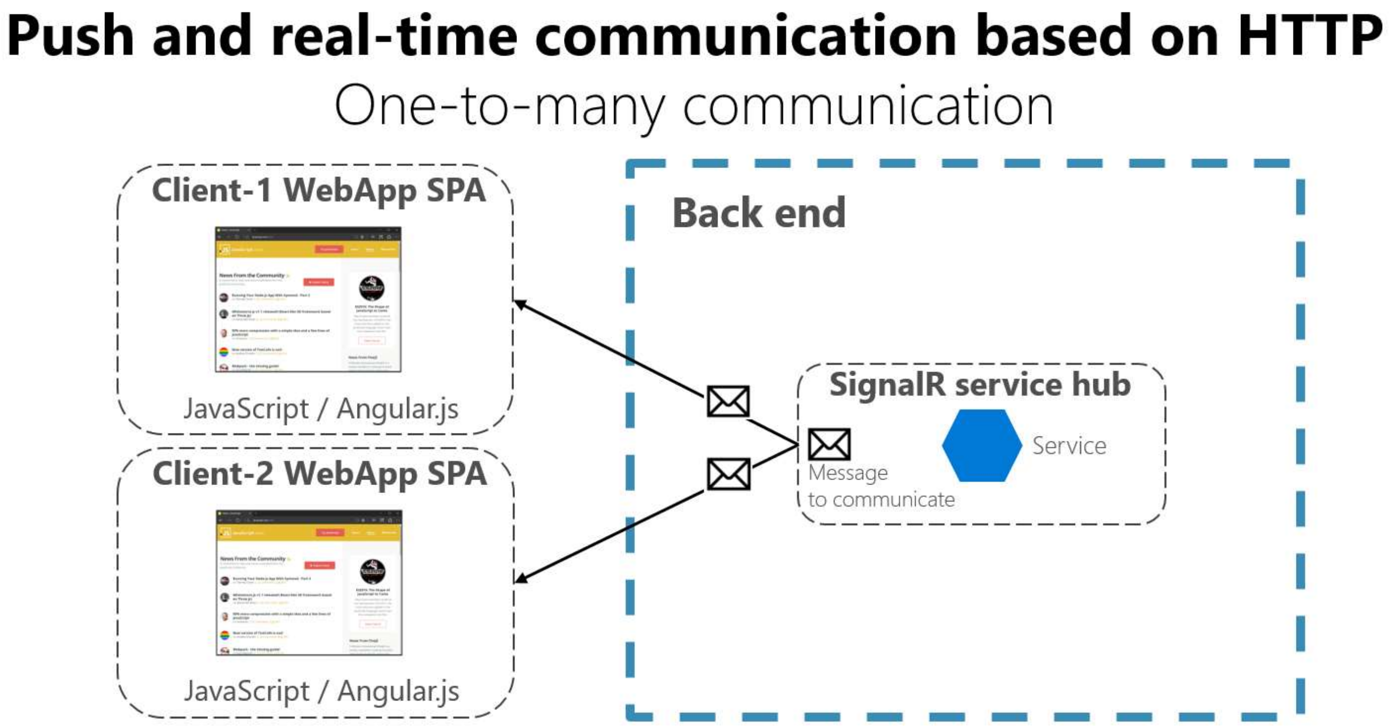
innovate achieve lead

BITS Pilani

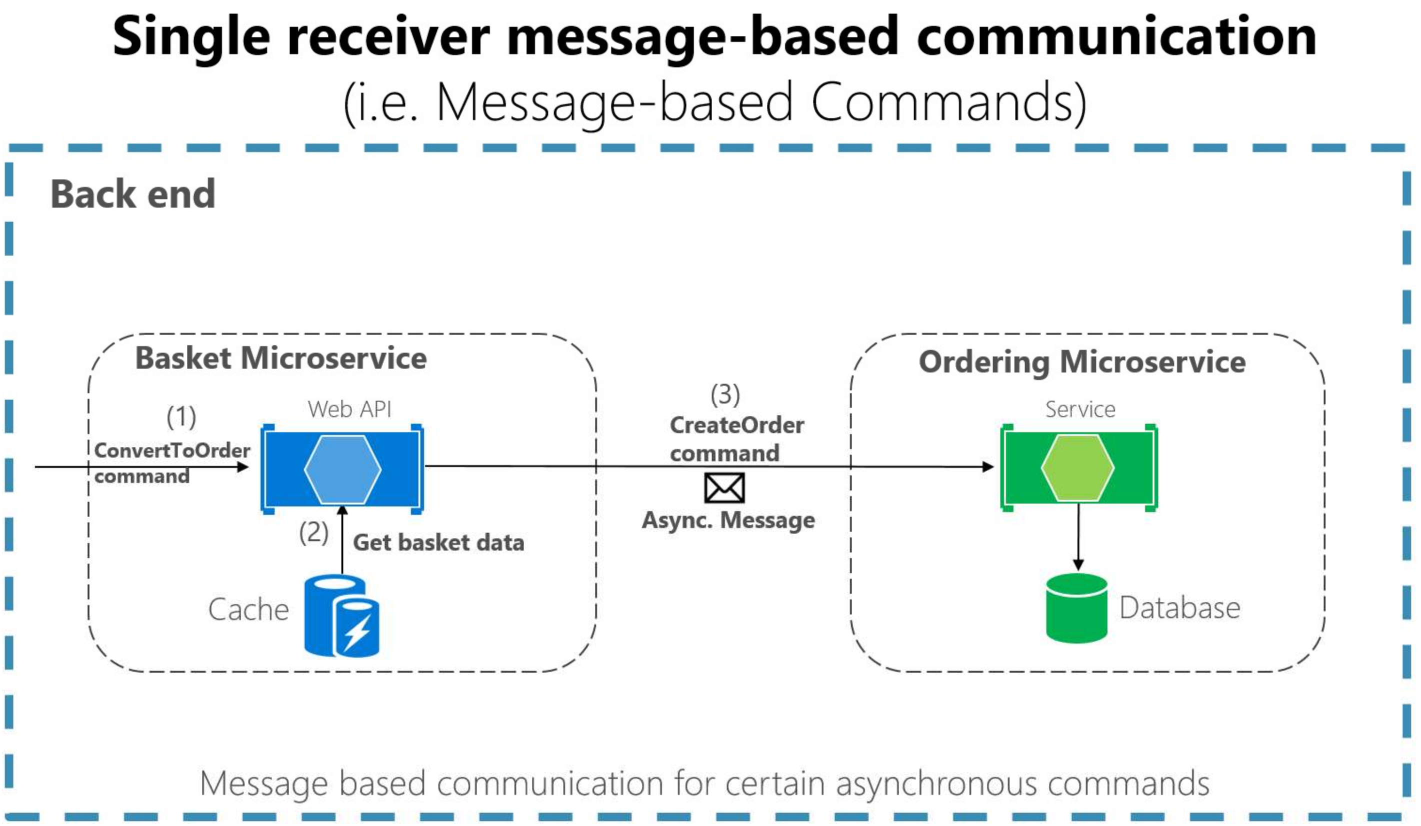
Pilani Campus

Communication Styles

Push and real-time communication based on HTTP



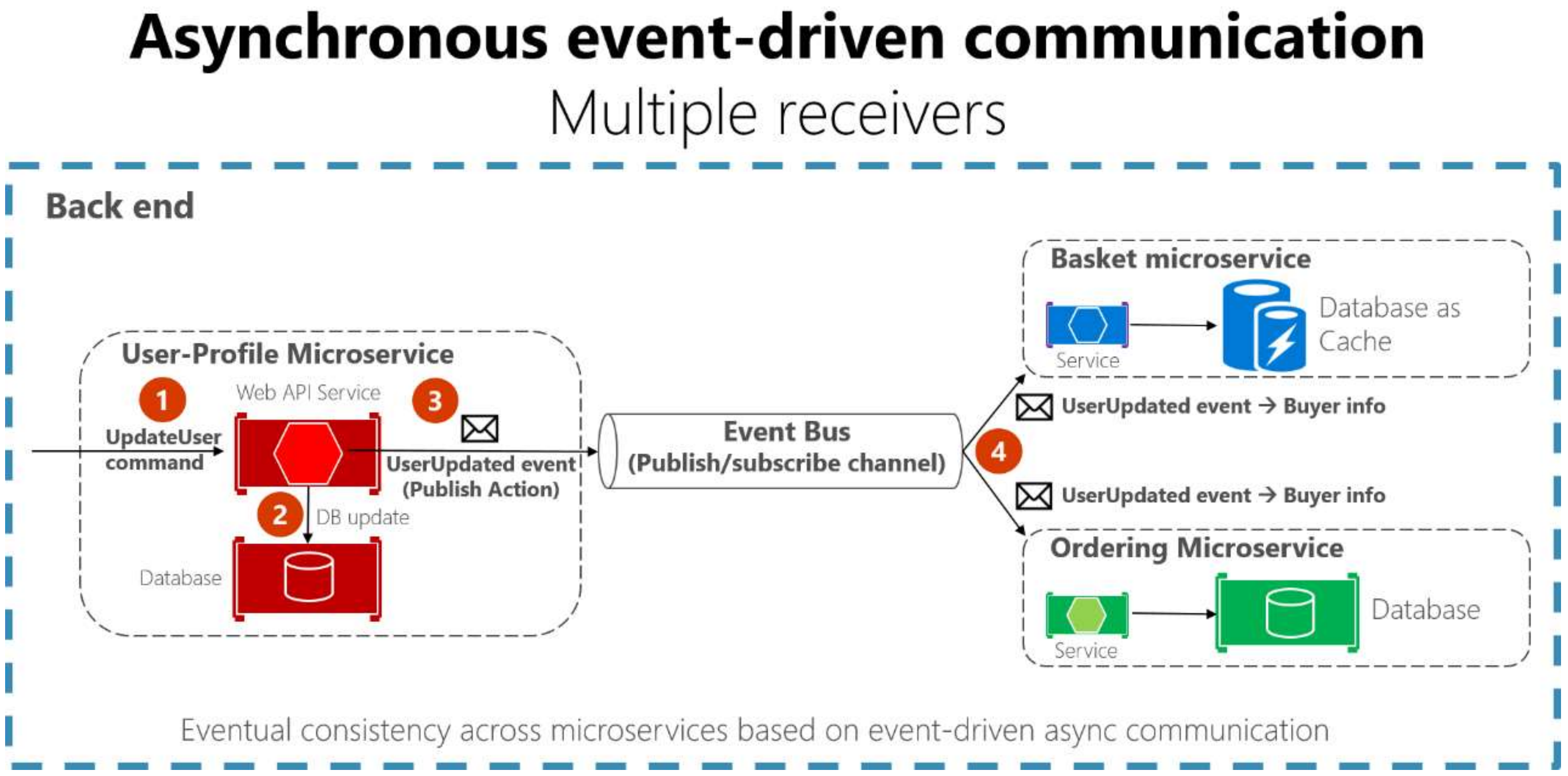
Single-receiver message-based communication



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


Asynchronous event-driven communication
Multiple receivers
Back end
User-Profile Microservice
1
UpdateUser command
Web API Service
3
UserUpdated event (Publish Action)
2
DB update
Database
Event Bus (Publish/subscribe channel)
4
UserUpdated event → Buyer info
UserUpdated event → Buyer info
Basket microservice
Service
Database as Cache
Ordering Microservice
Service
Database
Eventual consistency across microservices based on event-driven async communication

References

- Book: Microservices Patterns by Chris Richardson
- Book: 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/