CSC12108

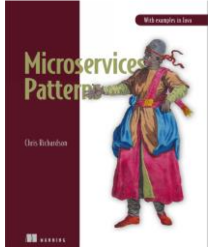
Distributed Application

TOPIC

Communication in Microserivces

Instructor – Msc. PHAM MINH TU

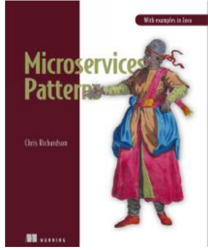
KHOA CÔNG NGHỆ THÔNG TIN
TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN



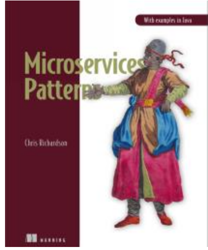
Outline

- ❑Interprocess communication in a microservice architecture
  - ❑Synchronous Remote procedure invocation pattern
  - ❑Asynchronous messaging pattern
- ❑Transaction management in a microservice architecture
- ❑External API patterns
  - ❑Problem in external API
  - ❑The API gateway pattern
  - ❑Implementing an API gateway

Outline



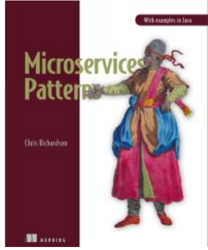
- ❑ **Interprocess communication in a microservice architecture**
  - ❑ **Synchronous Remote procedure invocation pattern**
  - ❑ **Asynchronous messaging pattern**
- ❑ **Transaction management in a microservice architecture**
- ❑ **External API patterns**
  - ❑ **Problem in external API**
  - ❑ **The API gateway pattern**
  - ❑ **Implementing an API gateway**



Interprocess communication in a microservice architecture

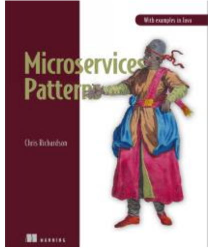
- ❑Communication mechanisms
    - ❑Synchronous request/response-based
        - ❑HTTP-based REST or gRPC
    - ❑Asynchronous
        - ❑Message-based such as AMQP or STOMP
- ❑Messages formats
    - ❑ Text-based formats such as JSON or XML
    - ❑ Binary format such as Avro or Protocol Buffers

Interprocess communication in a microservice architecture



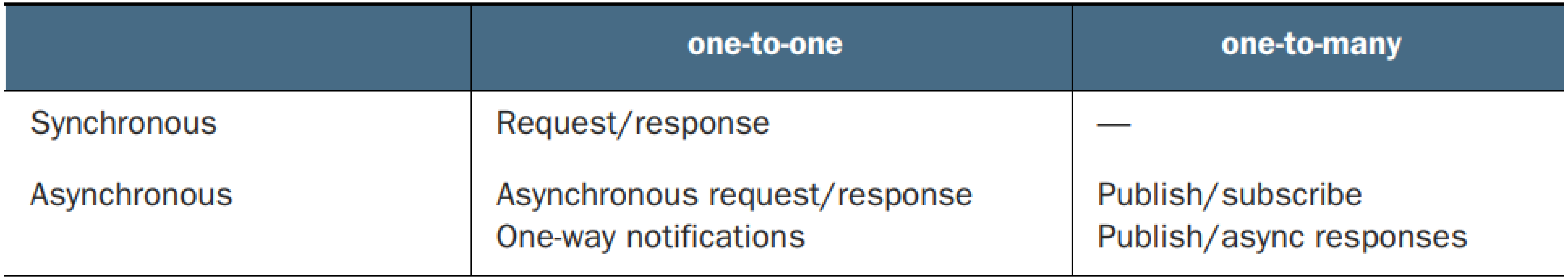
❑Interaction styles

- ❑One-to-one—Each client request is processed by exactly one service
- ❑One-to-many—Each request is processed by multiple services
- ❑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.

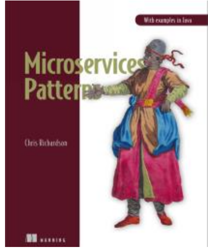


Interprocess communication in a microservice architecture

❑Interaction styles



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

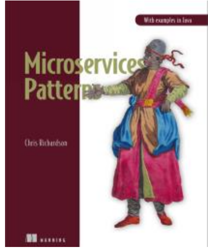


Interprocess communication in a microservice architecture

❑Defining APIs in a microservice architecture

- ❑A service’s API is a contract between the service and its clients
- ❑API definition depends on which IPC mechanism
  - ❑Message
    - ❑ Message channels, the message types, and the message formats
  - ❑Http
    - ❑ URLs, the HTTP verbs, and the request and response formats

Problem?

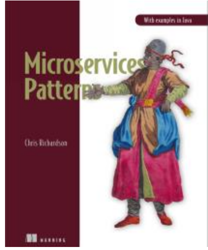


Interprocess communication in a microservice architecture

❑Evolving APIs

❑APIs invariably change over time as new features are added, existing features are changed, and (perhaps) old features are removed

Both old and new versions of a service will be running simultaneously

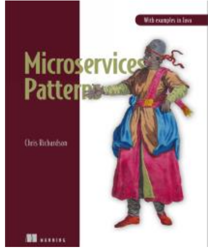


Interprocess communication in a microservice architecture

❑SEMANTIC VERSIONING

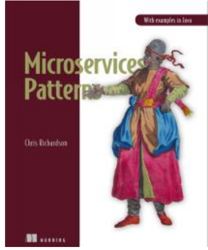
- ❑Version number consist of three parts: **MAJOR.MINOR.PATCH**
  - ❑MAJOR—When you make an incompatible change to the API
  - ❑MINOR—When you make backward-compatible enhancements to the API
    - ❑ Adding optional attributes to request
    - ❑ Adding attributes to a response
    - ❑ Adding new operations
  - ❑PATCH—When you make a backward-compatible bug fix

Example: REST
/v1/….
/v2/…



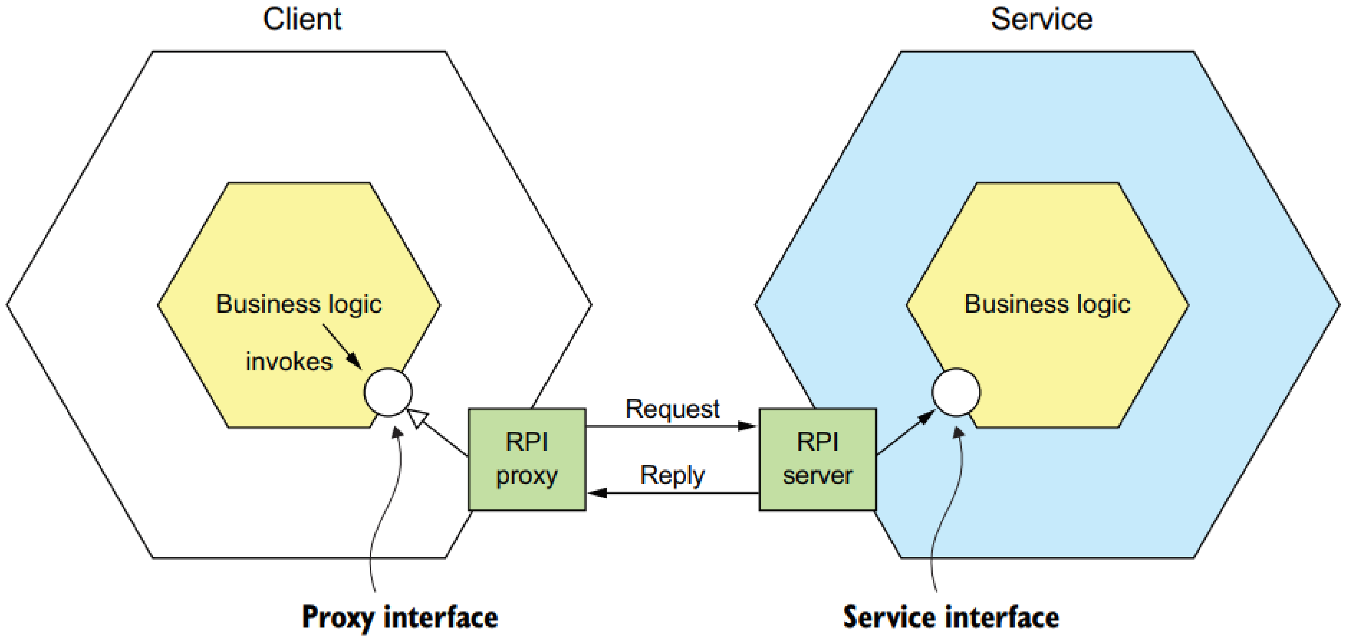
Interprocess communication in a microservice architecture

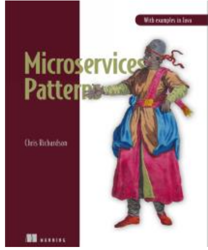
- ❑Message formats
  - ❑Text-based message formats: json and xml
    - ❑XML – XML Schema
    - ❑Json – JSON Shema
  - ❑Binary message formats
    - ❑Protocol Buffers - https://developers.google.com/protocol-buffers/docs/overview
    - ❑Avro - https://avro.apache.org
    - ❑Thrift



Interprocess communication in a microservice architecture

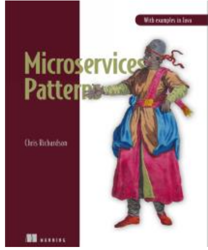
- ❑Communicating using the **synchronous** Remote procedure invocation pattern





Interprocess communication in a microservice architecture

- ❑Synchronous Remote procedure invocation pattern
  - ❑REST
    - ❑is a resource, which typically represents a single business object, such as a Customer or Product
    - ❑uses the HTTP verbs for manipulating resources
      - ❑ GET
      - ❑ POST
      - ❑ PUT
      - ❑ DELETE
      - ❑ …



Interprocess communication in a microservice architecture

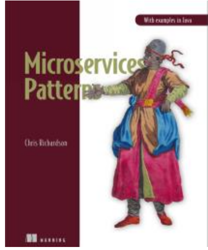
❑REST

- ❑The challenge of fetching multiple resources in a single request
  - ❑For example, imagine that a REST client wanted to retrieve an Order and the Order's Consumer

```
GET /orders/order-id-1345?expand=consumer
```

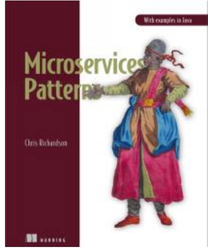
- ❑The challenge of mapping operations to http verbs
  - ❑how to map the operations you want to perform on a business object to an HTTP verb
    - ❑Example: A REST API should use PUT for updates, but there may be multiple ways to update an order, including cancelling it, revising the order

```
POST /orders/{orderId}/cancel
POST /orders/{orderId}/revise
```



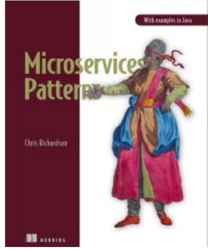
Interprocess communication in a microservice architecture

- ❑REST
  - ❑Benefits
    - ❑Simple and familiar.
    - ❑Test an HTTP API from within a browser using, for example, the Postman plugin, or from the command line using curl (assuming JSON or some other text format is used).
    - ❑It directly supports request/response style communication.
    - ❑HTTP is, of course, firewall friendly.
    - ❑It doesn’t require an intermediate broker, which simplifies the system’s architecture.



Interprocess communication in a microservice architecture

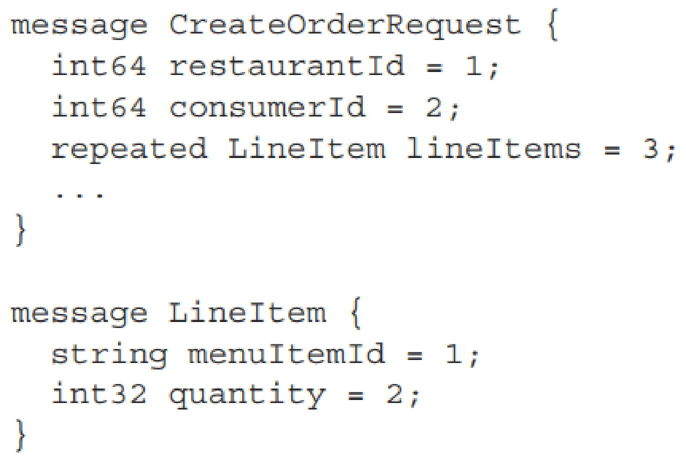
- ❑REST
  - ❑Benefits
    - ❑It only supports the request/response style of communication.
    - ❑Reduced availability. Because the client and service communicate directly without an intermediary to buffer messages, they must both be running for the duration of the exchange.
    - ❑Clients must know the locations (URLs) of the service instances(s), this is a nontrivial problem in a modern application. Clients must use what is known as a service discovery mechanism to locate service instances.
    - ❑Fetching multiple resources in a single request is challenging.
    - ❑It’s sometimes difficult to map multiple update operations to HTTP verbs.



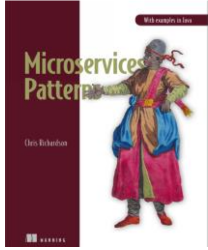
Interprocess communication in a microservice architecture

❑gRPC

- ❑A framework for writing cross-language clients and servers
- ❑Is a binary message-based protocol
- ❑Use the Protocol Buffer compiler to generate client-side stubs and server-side skeletons. The compiler can generate code for a variety of languages, including Java, C#, NodeJS, and GoLang.



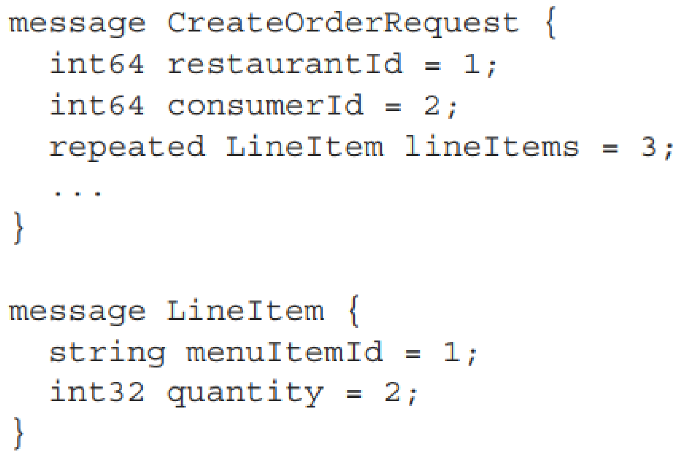
```
message CreateOrderRequest {
  int64 restaurantId = 1;
  int64 consumerId = 2;
  repeated LineItem lineItems = 3;
  ...
}

message LineItem {
  string menuItemId = 1;
  int32 quantity = 2;
}
```



Interprocess communication in a microservice architecture

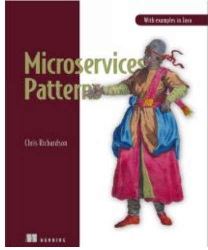
❑gRPC

- ❑A framework for writing cross-language clients and servers
- ❑Is a binary message-based protocol
- ❑Use the Protocol Buffer compiler to generate client-side stubs and server-side skeletons. The compiler can generate code for a variety of languages, including Java, C#, NodeJS, and GoLang.



```
message CreateOrderRequest {
  int64 restaurantId = 1;
  int64 consumerId = 2;
  repeated LineItem lineItems = 3;
  ...
}

message LineItem {
  string menuItemId = 1;
  int32 quantity = 2;
}
```

Interprocess communication in a microservice architecture

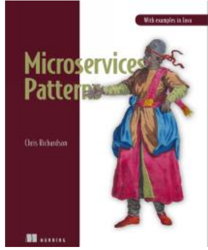


❑gRPC

❑Benefits

- ❑It’s straightforward to design an API that has a rich set of update operations.
- ❑It has an efficient, compact IPC mechanism, especially when exchanging large messages.
- ❑Bidirectional streaming enables both RPI and messaging styles of communication.
- ❑It enables interoperability between clients and services written in a wide range of languages

Interprocess communication in a microservice architecture



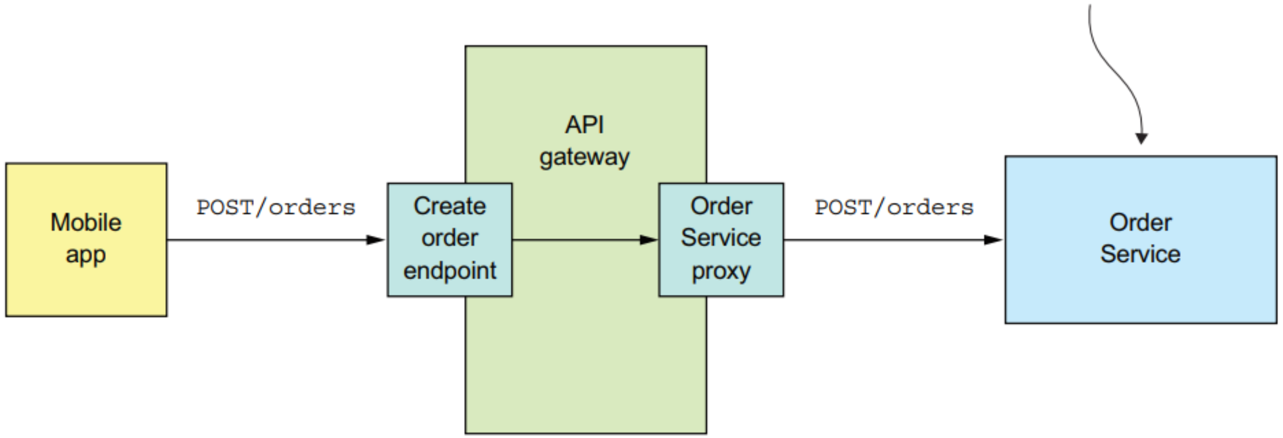
- ❑gRPC
  - ❑**Drawbacks**
    - ❑It takes more work for JavaScript clients to consume gRPC-based API than REST/JSON-based APIs.
    - ❑Older firewalls might not support HTTP/2

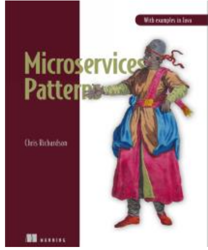
Interprocess communication in a microservice architecture

- ❑Synchronous Remote procedure invocation pattern
  - ❑Handling partial failure using the Circuit breaker pattern
    - ❑Because the client and the service are separate processes, a service may not be able to respond in a timely way to a client's request. The service could be down because of a failure or for maintenance. Or the service might be overloaded and responding extremely slowly to requests.

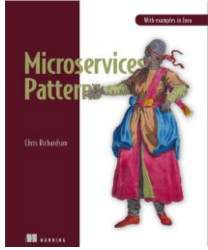
Solutions

Unresponsive remote service

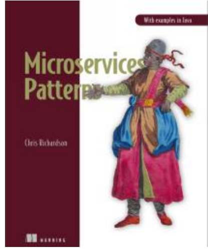




Interprocess communication in a microservice architecture



- ❑ **Synchronous Remote procedure invocation pattern**
    - ❑ **Handling partial failure using the Circuit breaker pattern**
        - ❑ Developing robust RPI proxies
            - ❑ 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 from a client to a service—Impose an upper bound on the number of outstanding requests that a client can make to a particular service. If the limit has been reached, it’s probably pointless to make additional requests, and those attempts should fail immediately
            - ❑ Circuit breaker pattern—Track the number of successful and failed requests, and if the error rate exceeds some threshold, trip the circuit breaker so that further attempts fail immediately. A large number of requests failing suggests that the service is unavailable and that sending more requests is pointless. After a timeout period, the client should try again, and, if successful, close the circuit breaker

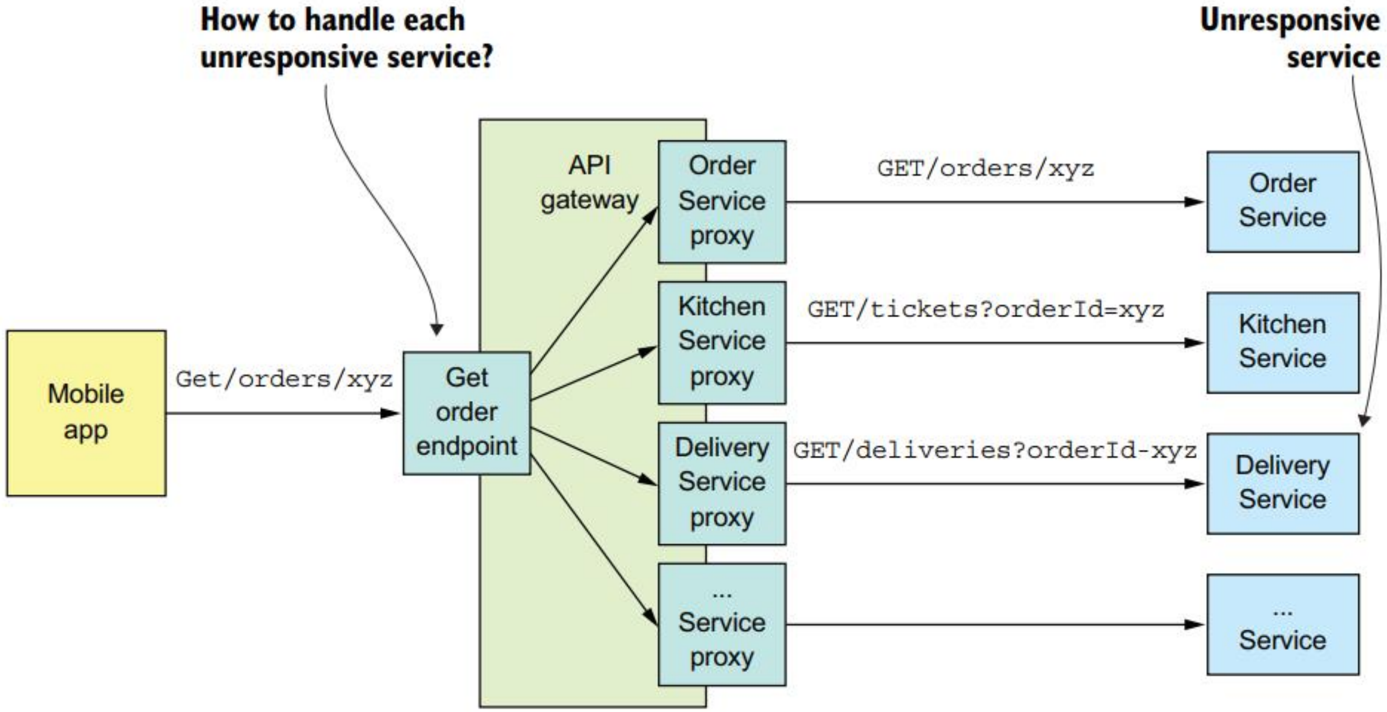


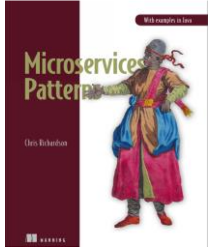
Microservices Patterns

Chris Richardson

Interprocess communication in a microservice architecture

- ❑Synchronous Remote procedure invocation pattern
  - ❑Handling partial failure using the Circuit breaker pattern
    - ❑Recovering from an unavailable service

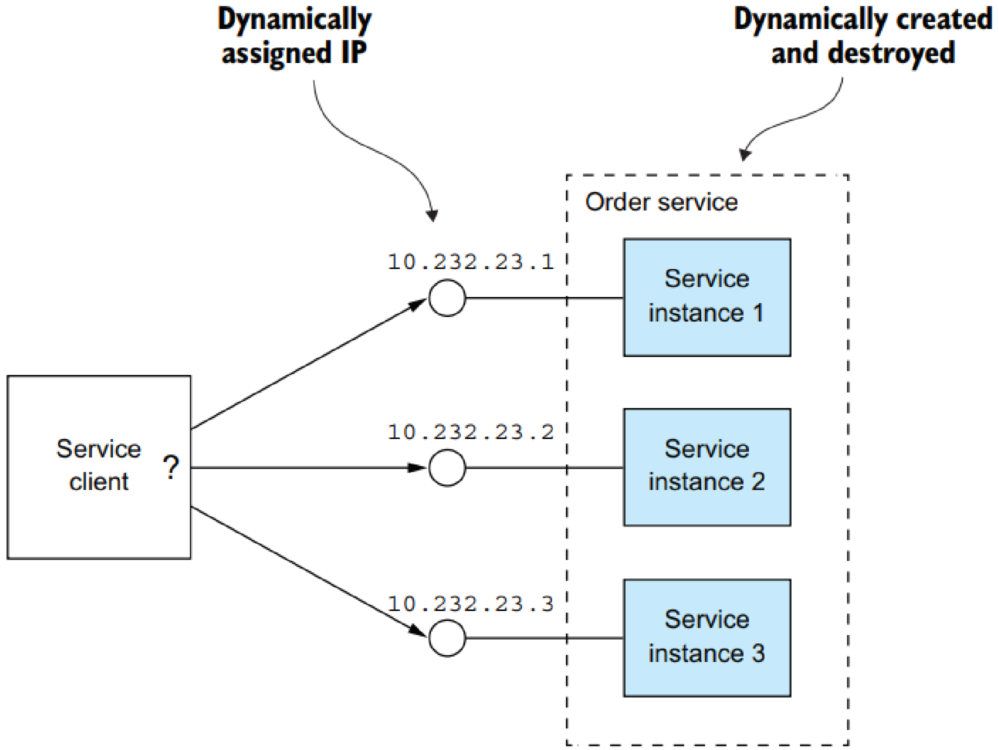


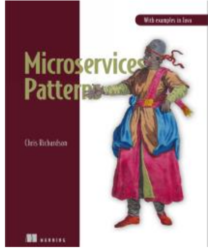


Interprocess communication in a microservice architecture

- □ Synchronous Remote procedure invocation pattern
- □ Handling partial failure using the Circuit breaker pattern
  - □ Using service discovery

Problem?

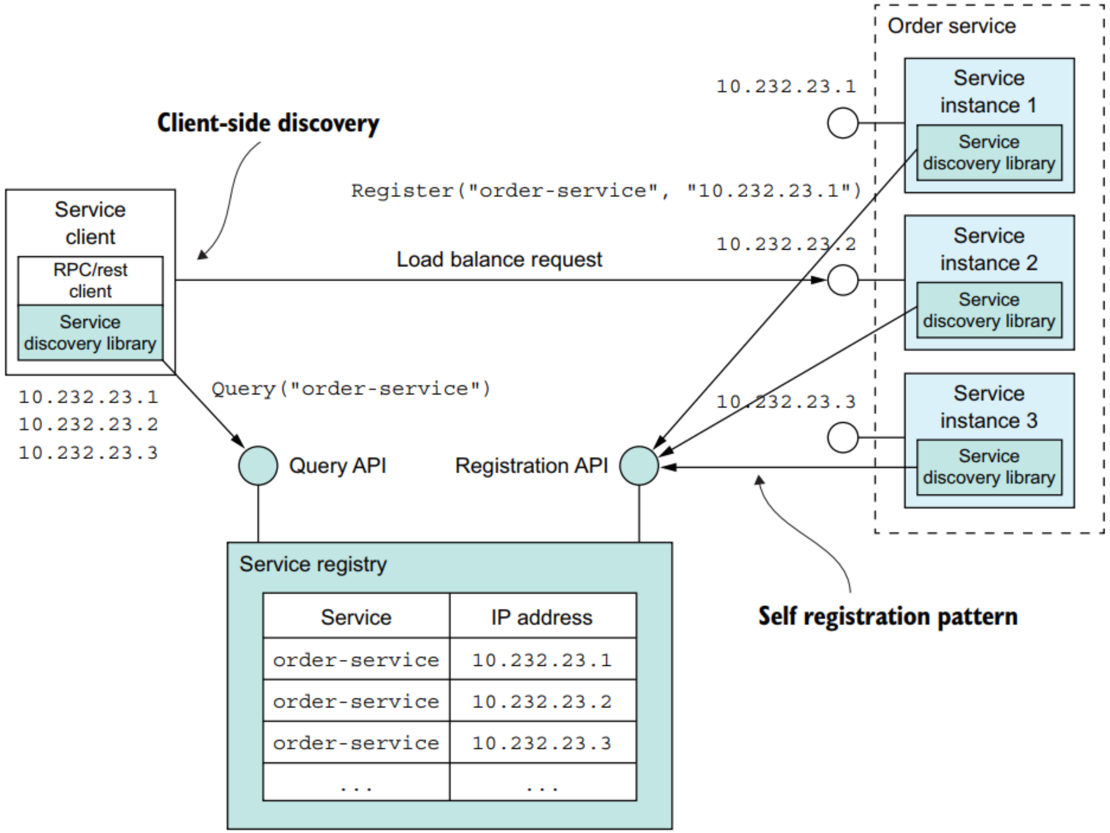




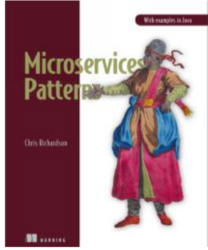
Interprocess communication in a microservice architecture

- ❑Synchronous Remote procedure invocation pattern
  - ❑Handling partial failure using the Circuit breaker pattern
    - ❑Using service discovery

Solution 1



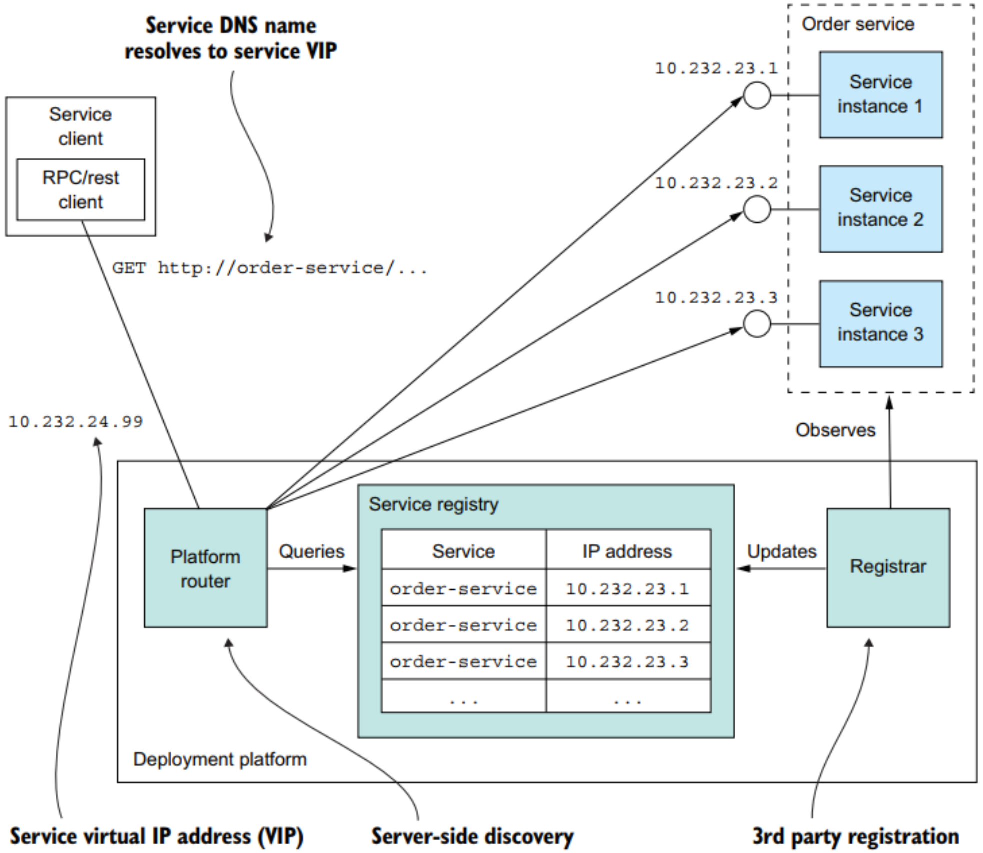
Service	IP address
order-service	10.232.23.1
order-service	10.232.23.2
order-service	10.232.23.3
...	...

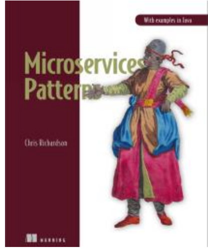


Interprocess communication in a microservice architecture

- ❑Synchronous Remote procedure invocation pattern
  - ❑Handling partial failure using the Circuit breaker pattern
    - ❑Using service discovery

Solution 2





Interprocess communication in a microservice architecture

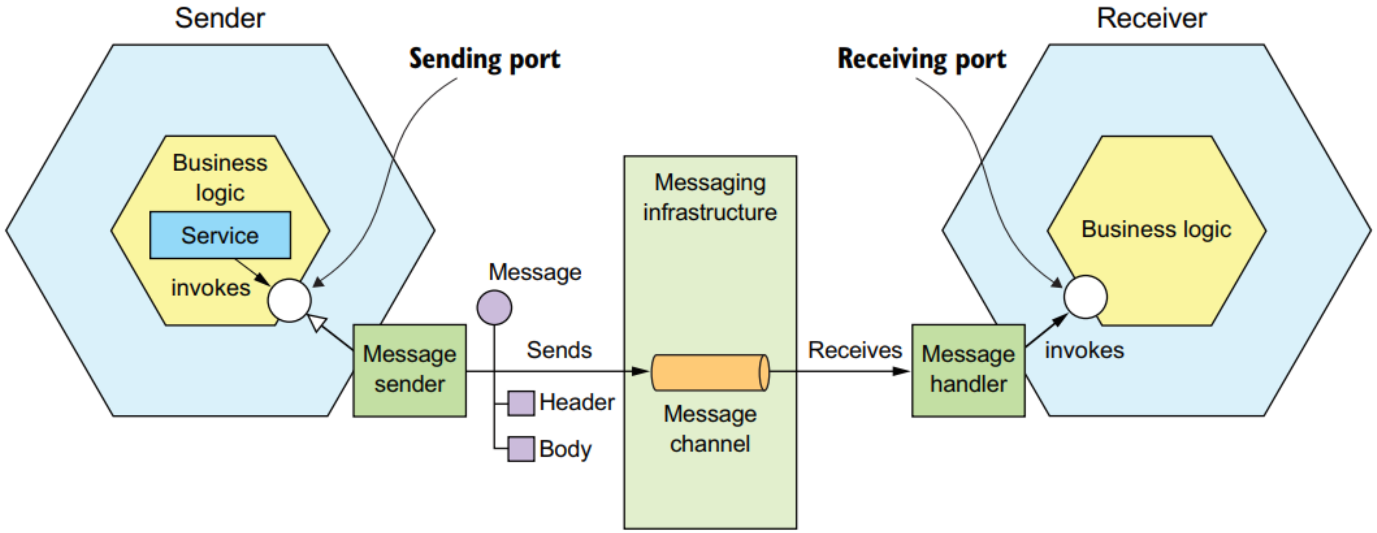
Communicating using the Asynchronous messaging

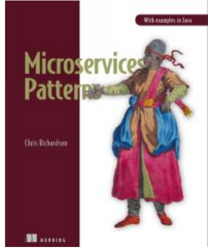
Messages

- A message consists of a header and a message body
  - The **header** is a collection of **name-value pairs**, metadata that describes the data being sent, name-value pairs provided by the message's sender
  - The message **body** is the **data** being sent, in either text or binary format.
    - **Document**—A generic message that contains only data. The receiver decides how to interpret it. The reply to a command is an example of a document message.
    - **Command**—A message that's the equivalent of an RPC request. It specifies the operation to invoke and its parameters.
    - **Event**—A message indicating that something notable has occurred in the sender. An event is often a domain event, which represents a state change of a domain object such as an Order, or a Customer.

Interprocess communication in a microservice architecture

- ❑Communicating using the Asynchronous messaging
  - ❑Message channels





Interprocess communication in a microservice architecture

❑Communicating using the Asynchronous messaging

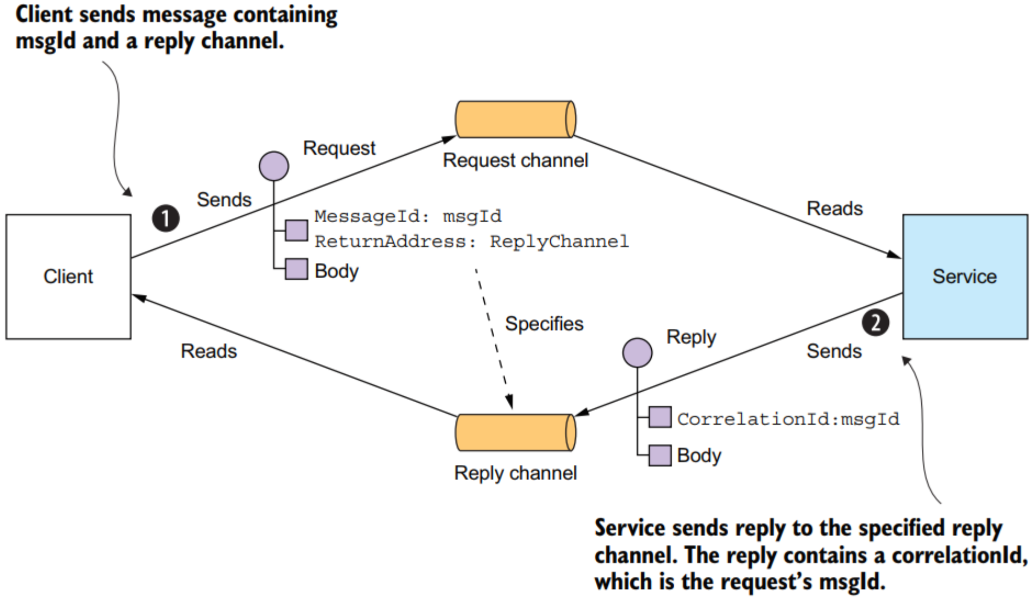
❑Message channels

- ❑Point-to-point
  - ❑ To delivers a message to exactly one of the consumers that is reading from the channel. Services use point-to-point channels for the one-to-one interaction styles described earlier
- ❑Publish-subscribe
  - ❑ To delivers each message to all of the attached consumers. Services use publish-subscribe channels for the one-to-many interaction styles

Interprocess communication in a microservice architecture

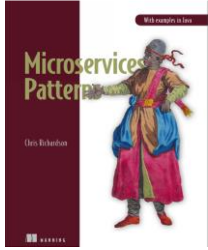
❑Communicating using the Asynchronous messaging

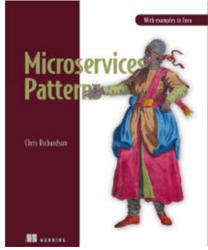
❑Implementing the interaction styles using messaging



Client sends message containing msgId and a reply channel.

Service sends reply to the specified reply channel. The reply contains a correlationId, which is the request's msgId.





Interprocess communication in a microservice architecture

❑Communicating using the Asynchronous messaging

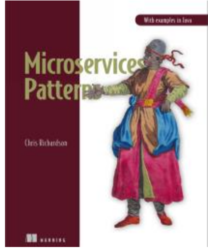
❑Implementing one-way notifications

- ❑Implementing one-way notifications is straightforward using asynchronous messaging. The client sends a message, typically a command message, to a point-to-point channel owned by the service. The service subscribes to the channel and processes the message. It doesn’t send back a reply.

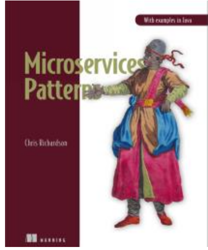
❑Implementing publish/subscribe

- ❑Messaging has built-in support for the publish/subscribe style of interaction. A client publishes a message to a publish-subscribe channel that is read by multiple consumers.

Interprocess communication in a microservice architecture

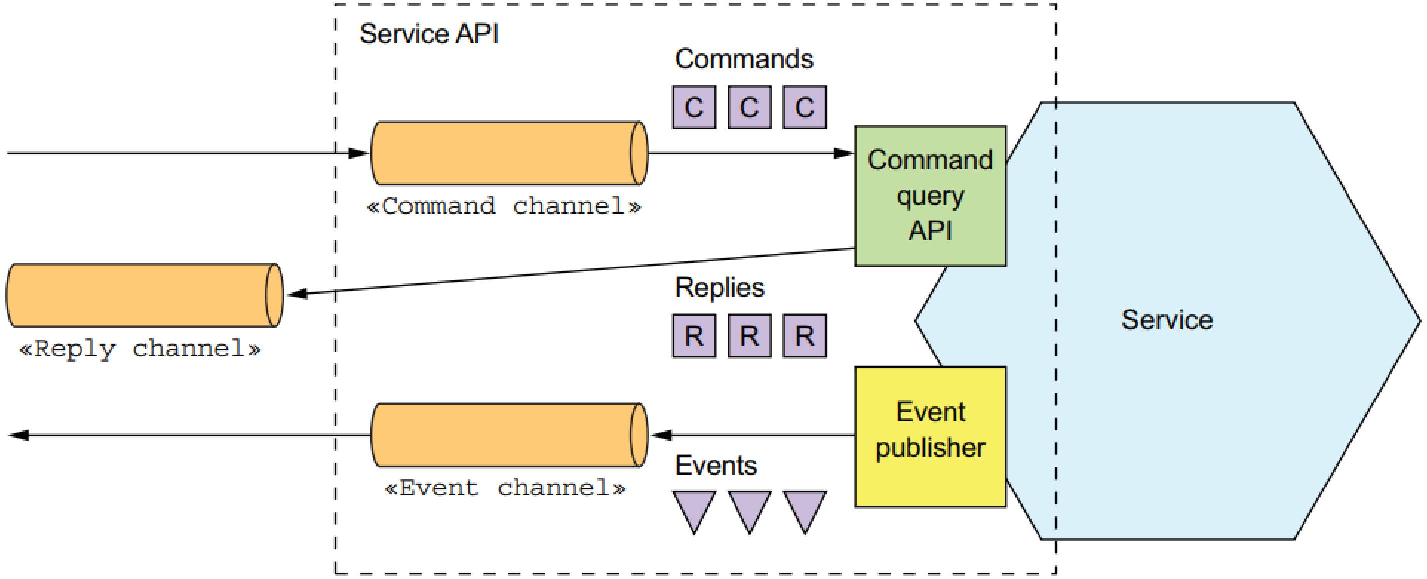


- □Communicating using the **Asynchronous** messaging
  - □Implementing publish/async responses
    - □The publish/async responses interaction style is a higher-level style of interaction that's implemented by combining elements of publish/subscribe and request/response. A client publishes a message that specifies a reply channel header to a publish-subscribe channel. A consumer writes a reply message containing a correlation id to the reply channel. The client gathers the responses by using the correlation id to match the reply messages with the request.

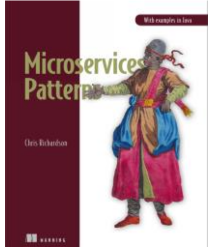


Interprocess communication in a microservice architecture

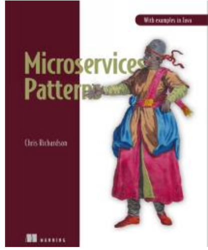
- ❑Communicating using the **Asynchronous** messaging
  - ❑Creating an API specification for a messaging-based service API



Interprocess communication in a microservice architecture



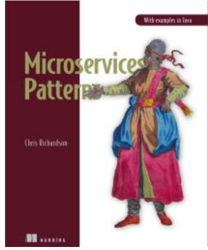
- ❑ Communicating using the Asynchronous messaging
  - ❑ Creating an API specification for a messaging-based service API
    - ❑ Documenting published events
      - ❑ A service can also publish events using a publish/subscribe interaction style. The specification of this style of API consists of the event channel and the types and formats of the event messages that are published by the service to the channel



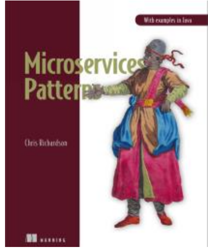
Interprocess communication in a microservice architecture

- □ Communicating using the Asynchronous messaging
  - □ Using a message broker
    - □ Brokerless messaging
      - □ Services can exchange messages directly (ZeroMQ)
      - □ It’s both a specification and a set of libraries for different languages
      - □ It supports a variety of transports, including TCP, UNIX-style domain sockets, and multicast

Interprocess communication in a microservice architecture

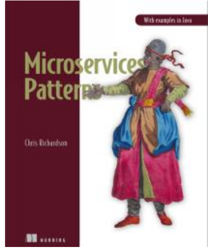


- □ **Communicating using the Asynchronous messaging**
  - □ **Using a message broker**
    - □ **Brokerless messaging**
      - □ **Benefits:**
        - □ Allows lighter network traffic and better latency, because messages go directly from the sender to the receiver, instead of having to go from the sender to the message broker and from there to the receiver
        - □ Eliminates the possibility of the message broker being a performance bottleneck or a single point of failure
        - □ Features less operational complexity, because there is no message broker to set up and maintain
      - □ **Drawbacks:**
        - □ Services need to know about each other’s locations and must therefore use one of the discovery mechanisms.
        - □ It offers reduced availability, because both the sender and receiver of a message must be available while the message is being exchanged.
        - □ Implementing mechanisms, such as guaranteed delivery, is more challenging



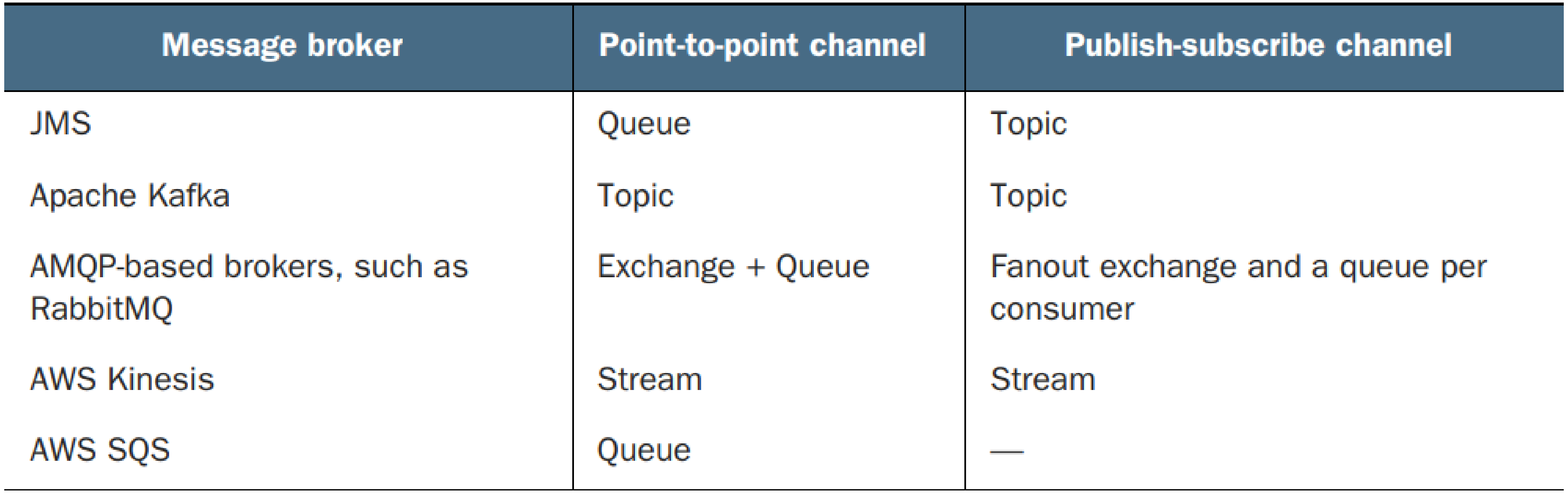
Interprocess communication in a microservice architecture

- □ Communicating using the **Asynchronous** messaging
  - □ Using a message broker
    - □ Broker-based messaging
      - □ A message broker is an intermediary through which all messages flow. A sender writes the message to the message broker, and the message broker delivers it to the receiver (ActiveMQ, RabbitM, Apache Kafka, AWS Kinesis, AWS SQS)

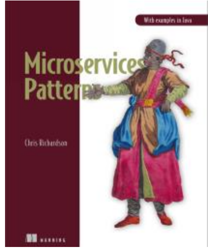


Interprocess communication in a microservice architecture

- ❑Communicating using the **Asynchronous** messaging
  - ❑Using a message broker
    - ❑Implementing message channels using a message broker

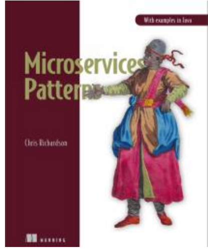


Message broker	Point-to-point channel	Publish-subscribe channel
JMS	Queue	Topic
Apache Kafka	Topic	Topic
AMQP-based brokers, such as RabbitMQ	Exchange + Queue	Fanout exchange and a queue per consumer
AWS Kinesis	Stream	Stream
AWS SQS	Queue	—



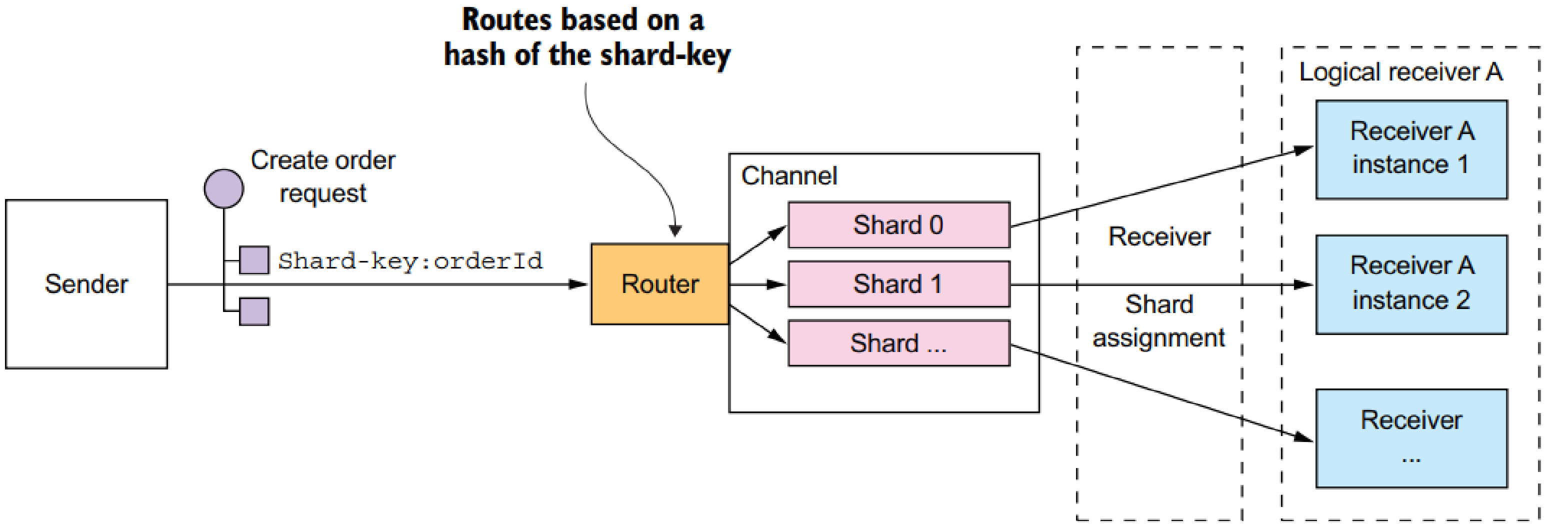
Interprocess communication in a microservice architecture

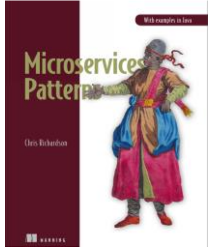
- ❑ **Communicating using the Asynchronous messaging**
  - ❑ **Using a message broker**
    - ❑ Benefit
      - ❑ Loose coupling—A client makes a request by simply sending a message to the appropriate channel. The client is completely unaware of the service instances. It doesn't need to use a discovery mechanism to determine the location of a service instance
      - ❑ Message buffering—The message broker buffers messages until they can be processed.
      - ❑ Flexible communication—Messaging supports all the interaction styles described earlier.
    - ❑ Downsides
      - ❑ Potential performance bottleneck
      - ❑ Potential single point of failure
      - ❑ Additional operational complexity



Interprocess communication in a microservice architecture

- ❑ Communicating using the Asynchronous messaging
  - ❑ Competing receivers and message ordering

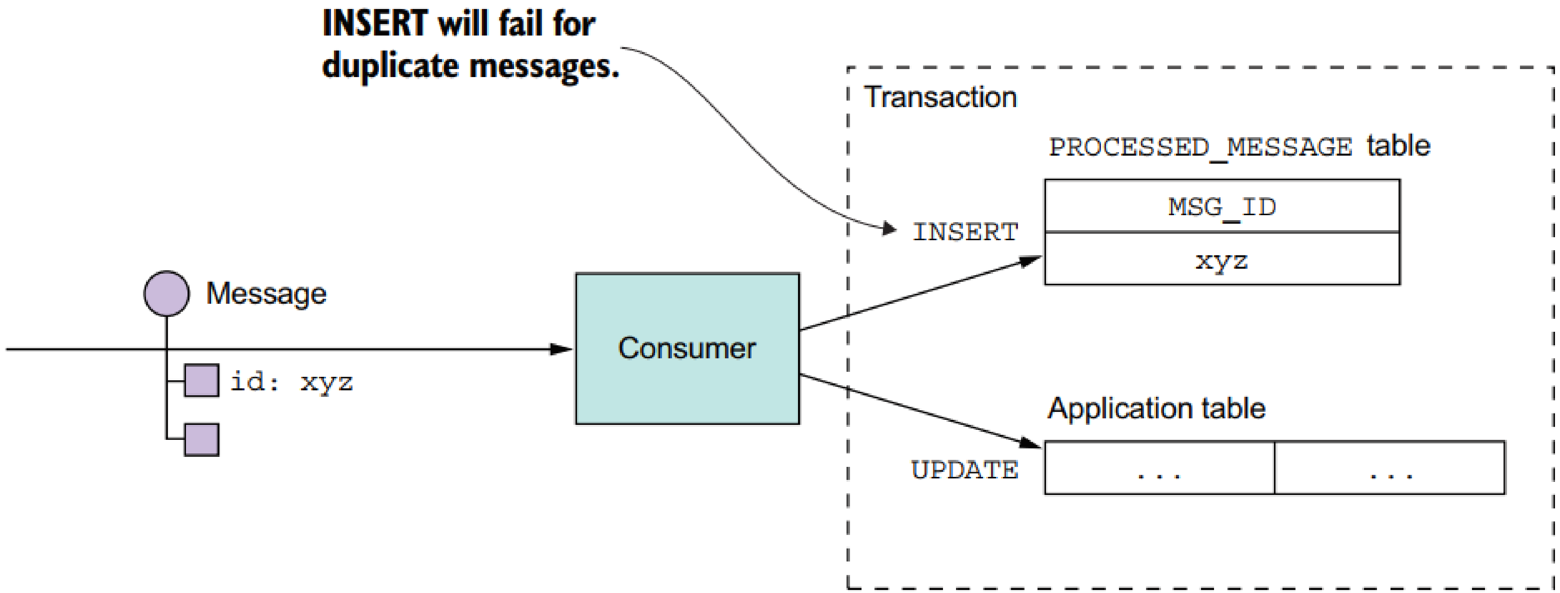




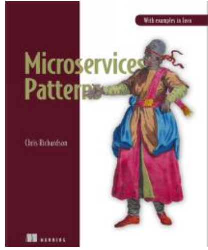
Interprocess communication in a microservice architecture

❑Communicating using the Asynchronous messaging

❑Handling duplicate messages

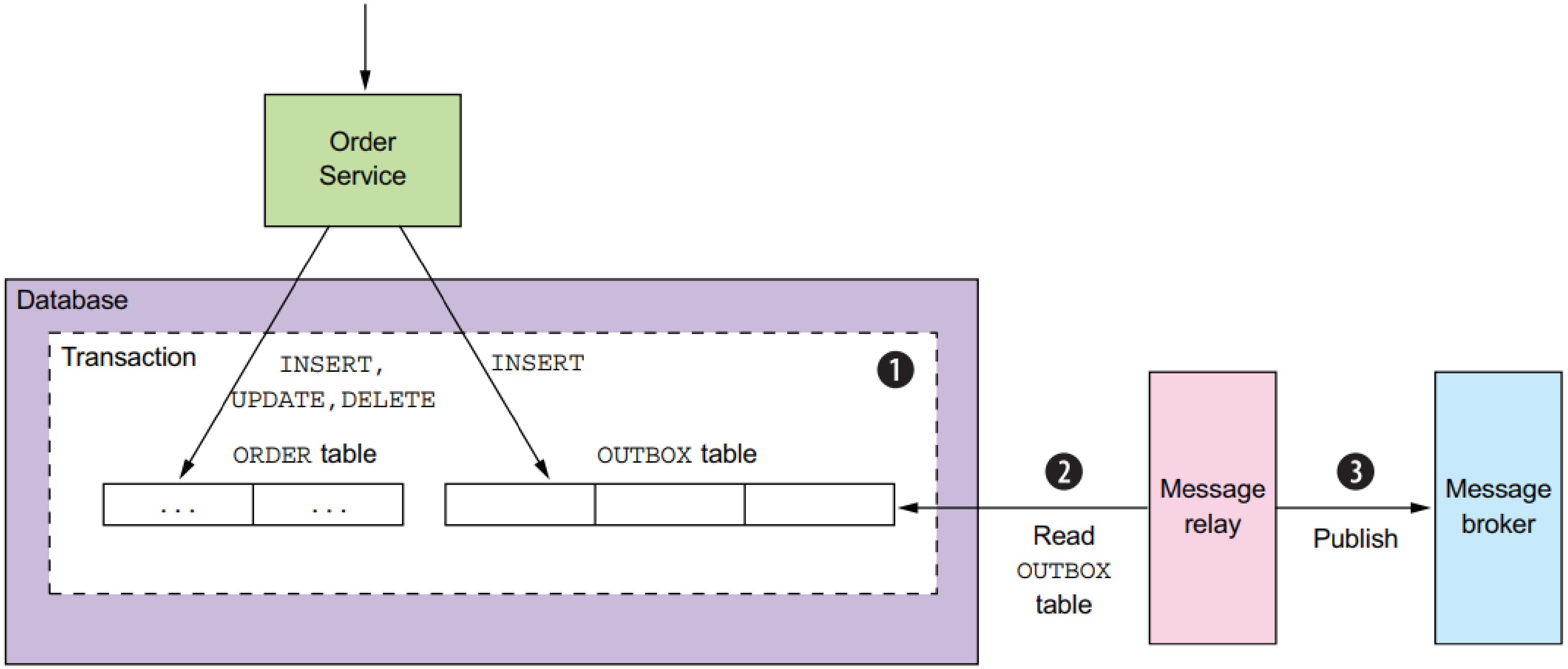


Interprocess communication in a microservice architecture

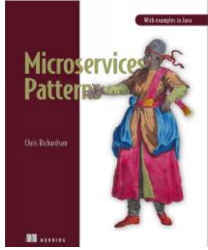


❑Communicating using the Asynchronous messaging

❑Transactional messaging - USING A DATABASE TABLE AS A MESSAGE QUEUE

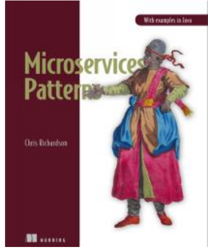


Outline



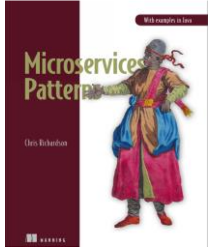
- ❑Interprocess communication in a microservice architecture
  - ❑Synchronous Remote procedure invocation pattern
  - ❑Asynchronous messaging pattern
- ❑**Transaction management in a microservice architecture**
- ❑External API patterns
  - ❑Problem in external API
  - ❑The API gateway pattern
  - ❑Implementing an API gateway

Transaction management in a microservice architecture



The trouble with distributed transactions

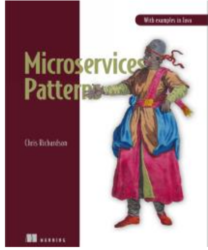
- Distributed transactions aren’t supported by modern message brokers such as RabbitMQ and Apache Kafka.
- Another problem with distributed transactions is that they are a form of synchronous IPC, which reduces availability
- In order for a distributed transaction to commit, all the participating services must be available.
- There is even Eric Brewer’s CAP theorem, which states that a system can only have two of the following three properties:
  - Consistency, availability, and partition tolerance



Transaction management in a microservice architecture

❑Using the Saga pattern to maintain data consistency

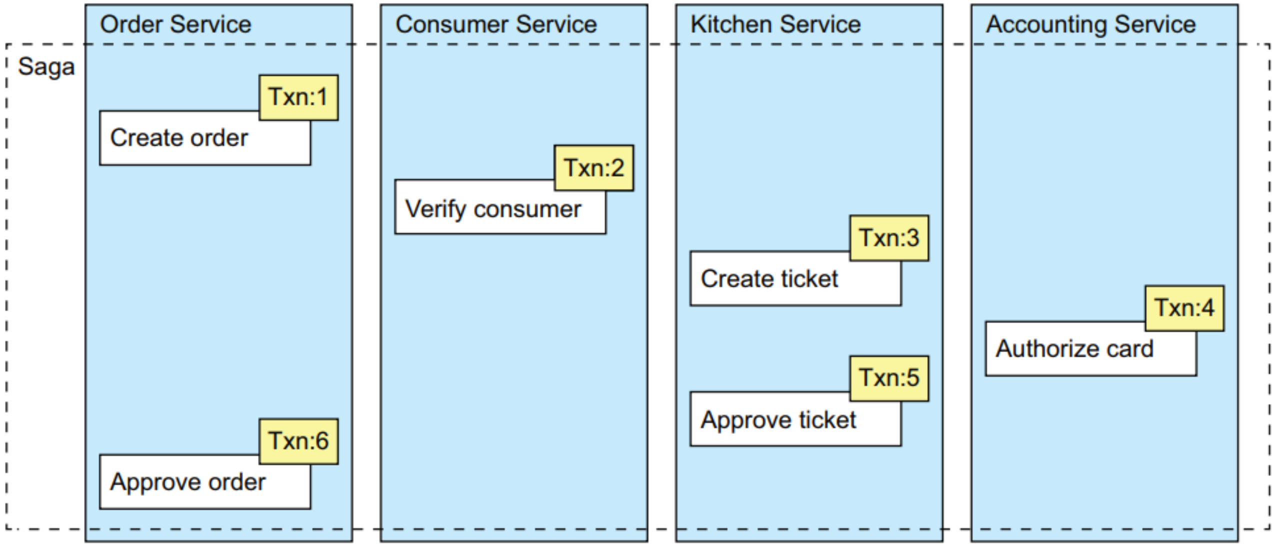
- ❑Sagas are mechanisms to maintain data consistency in a microservice architecture without having to use distributed transactions.
- ❑A saga is a sequence of local transactions.
- ❑Each local transaction updates data within a single service using the familiar ACID transaction frameworks and libraries.
- ❑The system operation initiates the first step of the saga. The completion of a local transaction triggers the execution of the next local transaction.



Transaction management in a microservice architecture

❑An example saga: the create order saga

- ❑The saga's first local transaction is initiated by the external request to create an order. The other five local transactions are each triggered by completion of the previous one.



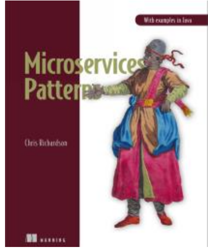
Saga

Order Service
Txn:1 Create order
Txn:6 Approve order

Consumer Service
Txn:2 Verify consumer

Kitchen Service
Txn:3 Create ticket
Txn:5 Approve ticket

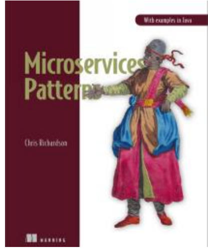
Accounting Service
Txn:4 Authorize card



Transaction management in a microservice architecture

❑An example saga: the create order saga

1. **Order Service** - Create an Order in an APPROVAL_PENDING state.
2. **Consumer Service** - Verify that the consumer can place an order.
3. **Kitchen Service** - Validate order details and create a Ticket in the CREATE _PENDING.
4. **Accounting Service** - Authorize consumer’s credit card.
5. **Kitchen Service** - Change the state of the Ticket to AWAITING_ACCEPTANCE.
6. **Order Service** - Change the state of the Order to APPROVED.

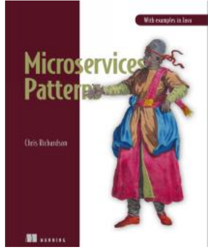


Transaction management in a microservice architecture

❑An example saga: the create order saga

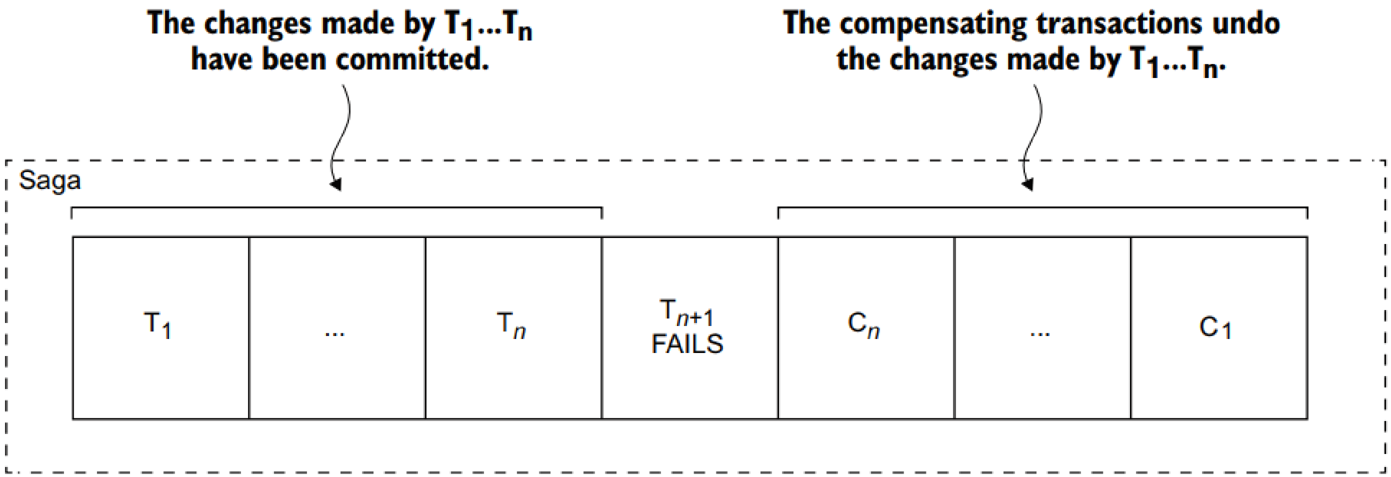
- ❑How the services that participate in a saga communicate using asynchronous messaging?
  - ❑A service publishes a message when a local transaction completes.
  - ❑The message then triggers the next step in the saga.
  - ❑Not only does using messaging ensure the saga participants are loosely coupled, it also guarantees that a saga completes.

if the recipient of a message is temporarily unavailable, the message broker buffers the message until it can be delivered.

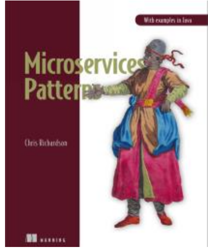


Transaction management in a microservice architecture

- ❑ An example saga: the create order saga
  - ❑ Sagas use compensating transactions to roll back changes
    - ❑ Suppose that the **(n + 1)th** transaction of a saga fails. The effects of the previous **n** transactions must be undone


The changes made by T1...Tn have been committed.
The compensating transactions undo the changes made by T1...Tn.
Saga
T1
...
Tn
Tn+1 FAILS
Cn
...
C1

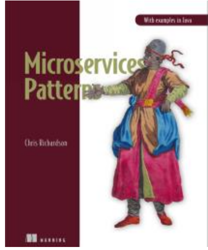
Transaction management in a microservice architecture



❑An example saga: the create order saga

- ❑Sagas use compensating transactions to roll back changes
  1. Order Service - Create an Order in an APPROVAL_PENDING state.
  2. Consumer Service - Verify that the consumer can place an order.
  3. Kitchen Service - Validate order details and create a Ticket in the CREATE_PENDING state.
  4. Accounting Service - Authorize consumer’s credit card, which fails.
  5. Kitchen Service - Change the state of the Ticket to CREATE_REJECTED.
  6. Order Service - Change the state of the Order to REJECTED

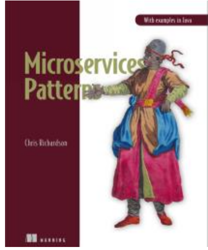
Transaction management in a microservice architecture



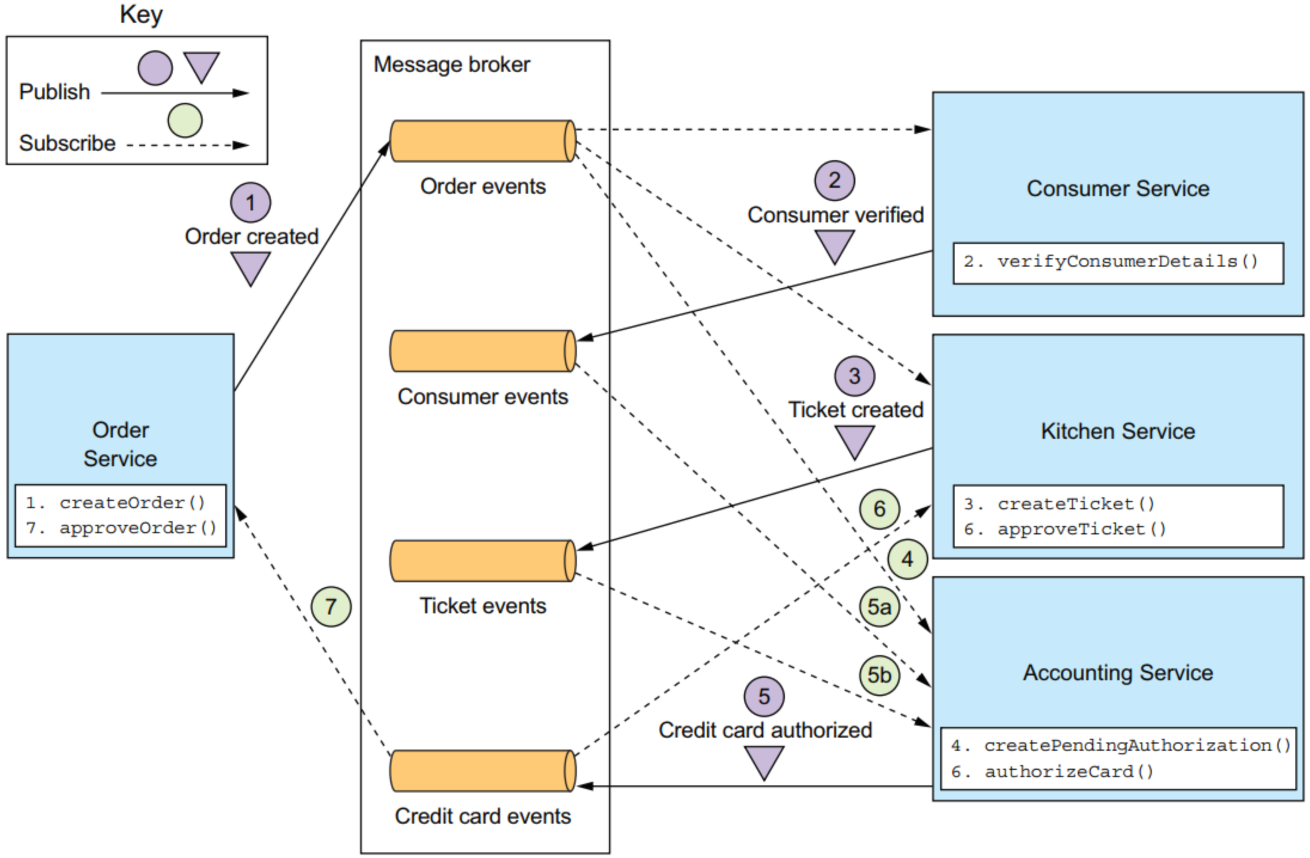
□Coordinating sagas

- □There are a couple of different ways to structure a saga’s coordination logic:
  - □**Choreography** - Distribute the decision making and sequencing among the saga participants. They primarily communicate by exchanging events.
  - □**Orchestration** - Centralize a saga’s coordination logic in a saga orchestrator class. A saga orchestrator sends command messages to saga participants telling them which operations to perform.

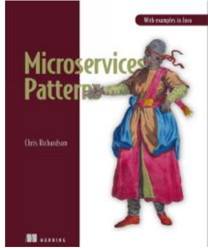
Transaction management in a microservice architecture



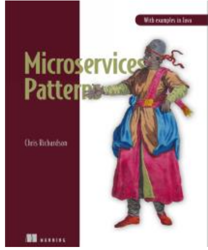
- ❑Coordinating sagas
  - ❑Implementing the create order saga using choreography



Transaction management in a microservice architecture

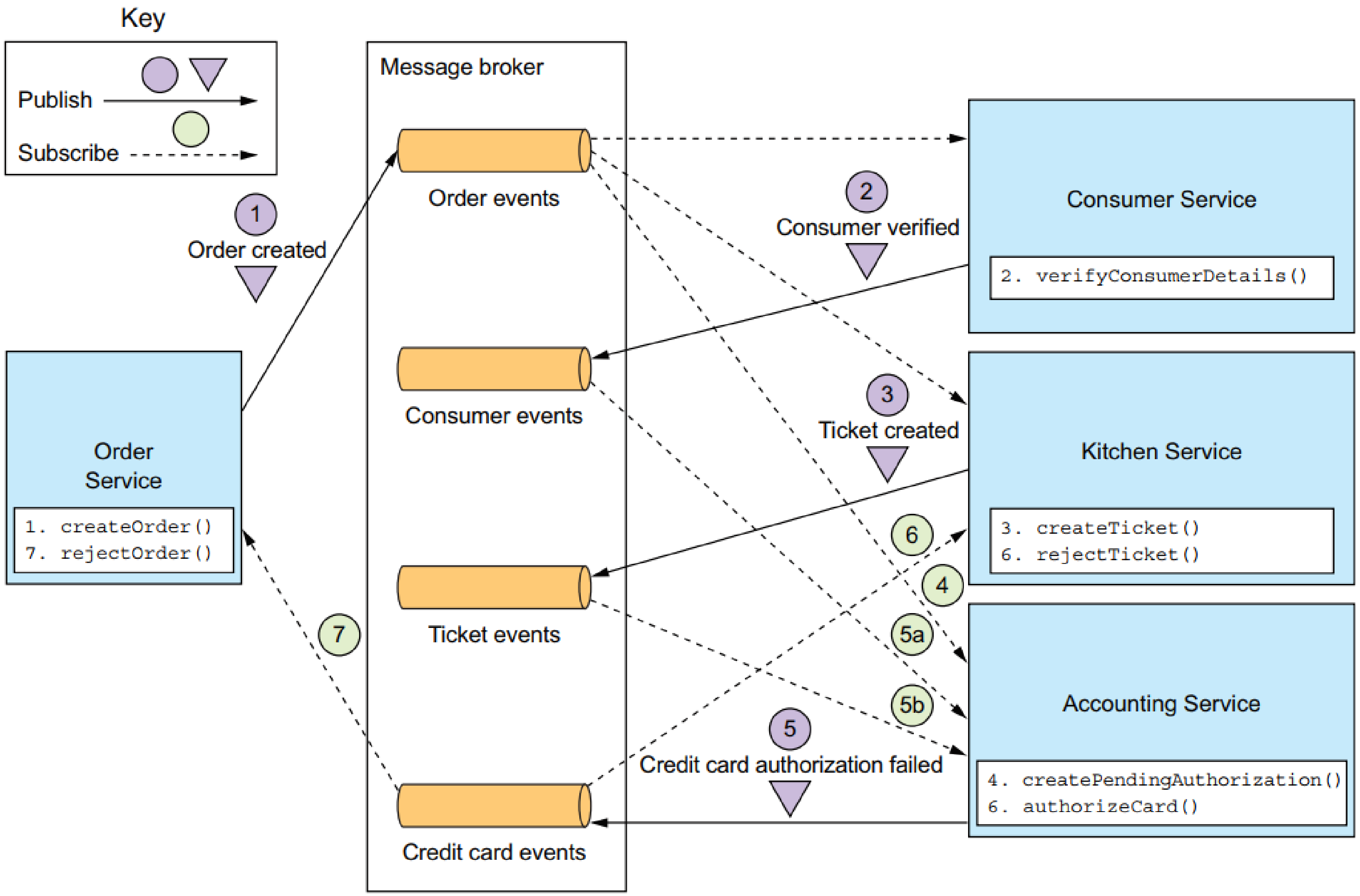


- ❑ **Coordinating sagas**
  - ❑ Implementing the create order saga using choreography
    - ❑ Order Service creates an Order in the APPROVAL_PENDING state and publishes an OrderCreated event.
    - ❑ Consumer Service consumes the OrderCreated event, verifies that the consumer can place the order, and publishes a ConsumerVerified event.
    - ❑ Kitchen Service consumes the OrderCreated event, validates the Order, creates a Ticket in a CREATE_PENDING state, and publishes the TicketCreated event.
    - ❑ Accounting Service consumes the OrderCreated event and creates a CreditCardAuthorization in a PENDING state.
    - ❑ Accounting Service consumes the TicketCreated and ConsumerVerified events, charges the consumer's credit card, and publishes the CreditCardAuthorized event.
    - ❑ Kitchen Service consumes the CreditCardAuthorized event and changes the state of the Ticket to AWAITING_ACCEPTANCE.
    - ❑ Order Service receives the CreditCardAuthorized events, changes the state of the Order to APPROVED, and publishes an OrderApproved event

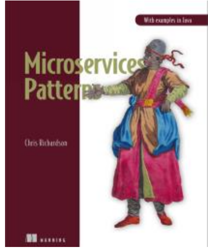


Transaction management in a microservice architecture

- ❑ Coordinating sagas
  - ❑ Implementing the create order saga using choreography
    - ❑ For example, the authorization of the consumer’s credit card might fail
    - ❑ The participants of choreography-based sagas interact using publish/subscribe



Transaction management in a microservice architecture

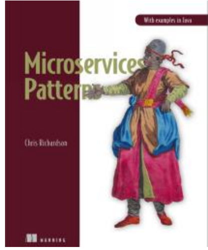


❑Coordinating sagas

❑Implementing the create order saga using choreography

- ❑Benefits
  - ❑Simplicity - publish events when they create, update, or delete business objects.
  - ❑Loose coupling - The participants subscribe to events and don’t have direct knowledge of each other.
- ❑Drawbacks
  - ❑More difficult to understand
  - ❑Cyclic dependencies between the services
  - ❑Risk of tight coupling

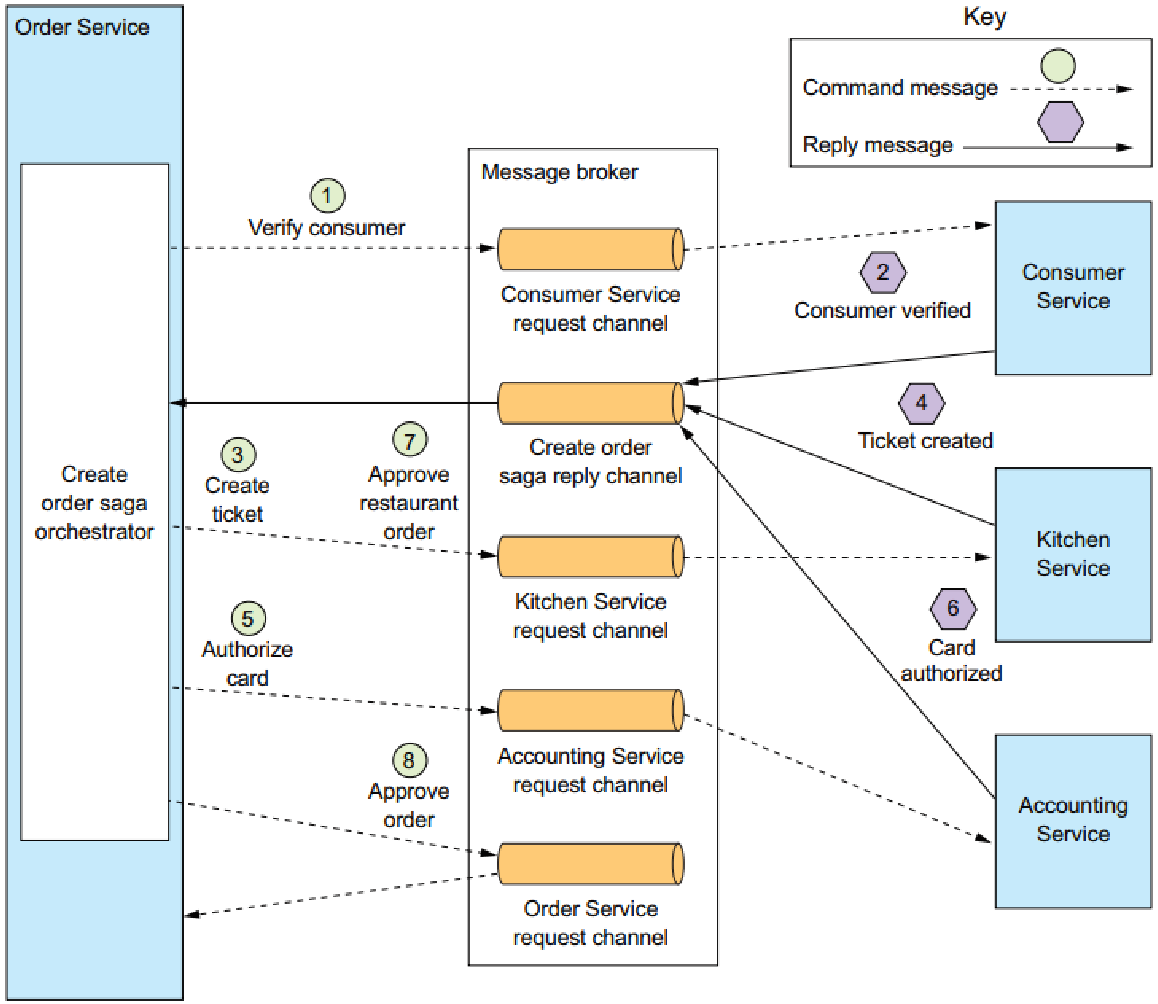
Transaction management in a microservice architecture

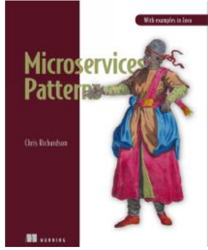


❑Coordinating sagas

❑Implementing the create order saga using orchestration

- ❑The saga is orchestrated by the CreateOrderSaga class, which invokes the saga participants using asynchronous request/response.
- ❑This class keeps track of the process and sends command messages to saga participants.



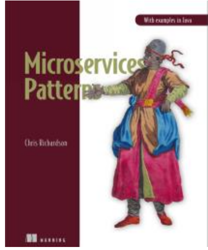


Transaction management in a microservice architecture

Coordinating sagas

Implementing the create order saga using orchestration

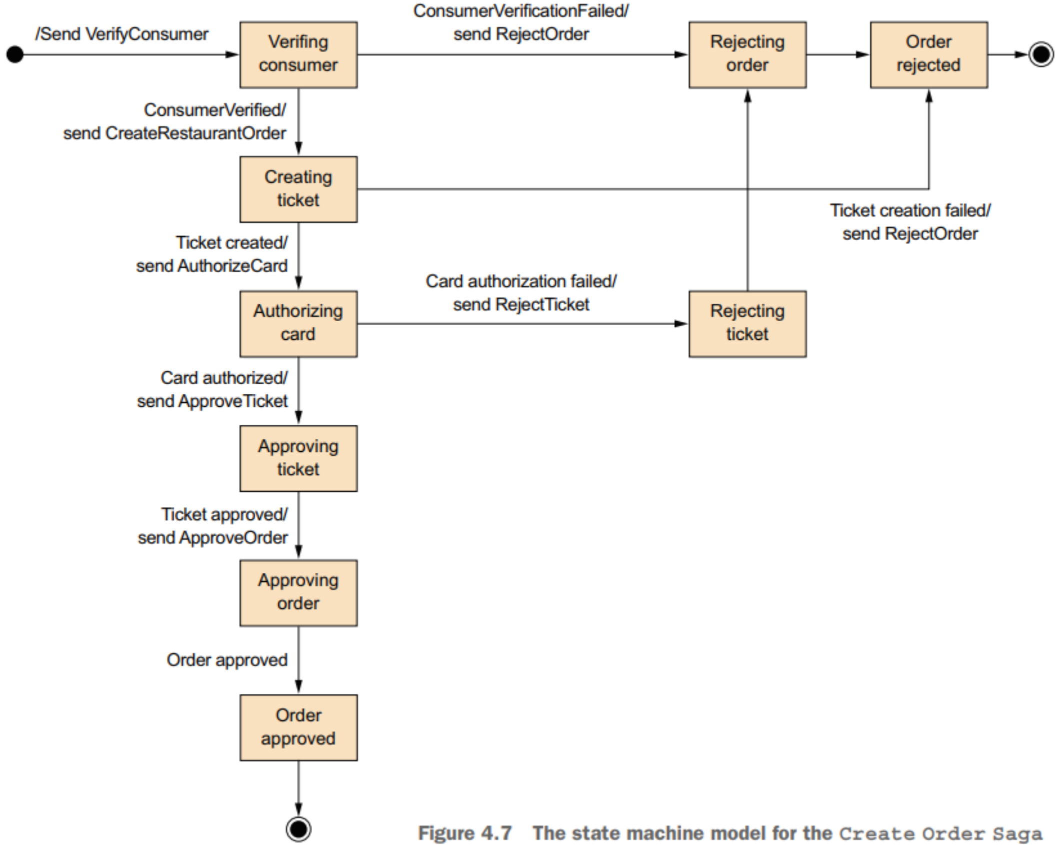
- The saga orchestrator sends a Verify Consumer command to Consumer Service.
- Consumer Service replies with a Consumer Verified message.
- The saga orchestrator sends a Create Ticket command to Kitchen Service.
- Kitchen Service replies with a Ticket Created message.
- The saga orchestrator sends an Authorize Card message to Accounting Service.
- Accounting Service replies with a Card Authorized message.
- The saga orchestrator sends an Approve Ticket command to Kitchen Service.
- The saga orchestrator sends an Approve Order command to Order Service.



Transaction management in a microservice architecture

- Coordinating sagas
  - Implementing the create order saga using orchestration

Modeling saga orchestrators as state machines



/Send VerifyConsumer

Verifing consumer

ConsumerVerificationFailed/ send RejectOrder

Rejecting order

Order rejected

ConsumerVerified/ send CreateRestaurantOrder

Creating ticket

Ticket creation failed/ send RejectOrder

Ticket created/ send AuthorizeCard

Authorizing card

Card authorization failed/ send RejectTicket

Rejecting ticket

Card authorized/ send ApproveTicket

Approving ticket

Ticket approved/ send ApproveOrder

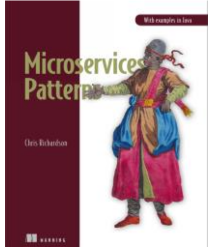
Approving order

Order approved

Order approved

Figure 4.7 The state machine model for the `Create Order Saga`

Transaction management in a microservice architecture

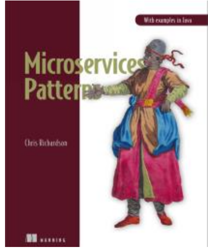


❑Coordinating sagas

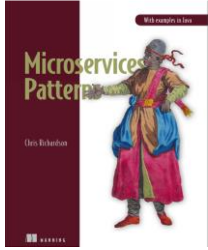
❑Implementing the create order saga using orchestration

- ❑Benefits
  - ❑Simpler dependencies
  - ❑Less coupling
  - ❑Improves separation of concerns and simplifies the business logic
- ❑Drawbacks
  - ❑The risk of centralizing too much business logic in the orchestrator

Outline



- ❑Interprocess communication in a microservice architecture
    - ❑Synchronous Remote procedure invocation pattern
    - ❑Asynchronous messaging pattern
- ❑External API patterns
    - ❑Problem in external API
    - ❑The API gateway pattern
    - ❑Implementing an API gateway



External API patterns

❑API design issues for the mobile client

❑Problem in external API

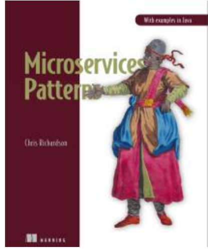
API design issues for the FTGO mobile client

?

The monolithic version of the FTGO application has an API endpoint that returns the order details. The mobile client retrieves the information it needs by making a **single request**

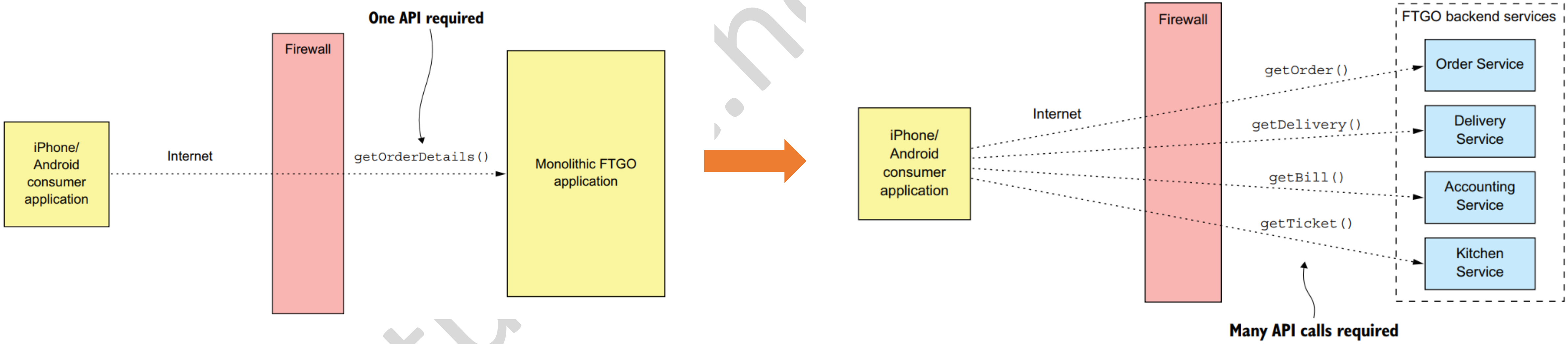
⇔

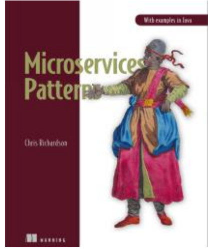
In the microservices version of the FTGO application, the order details are, scattered across **several services**, including the following: *Order Service, Kitchen Service, Delivery Service, Accounting Service*



External API patterns

- ❑Problem in external API
    - ❑API design issues for the mobile client

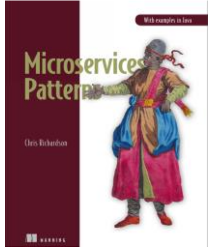




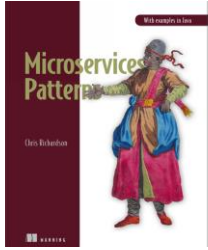
External API patterns

- ❑ **Problem in external API**
  - ❑ **API design issues for the mobile client**
    - ❑ Poor user experience due to the client making multiple requests
    - ❑ Lack of encapsulation requires frontend developers to change their code in lockstep with the backend
    - ❑ Services might use client-unfriendly ipc mechanisms

External API patterns



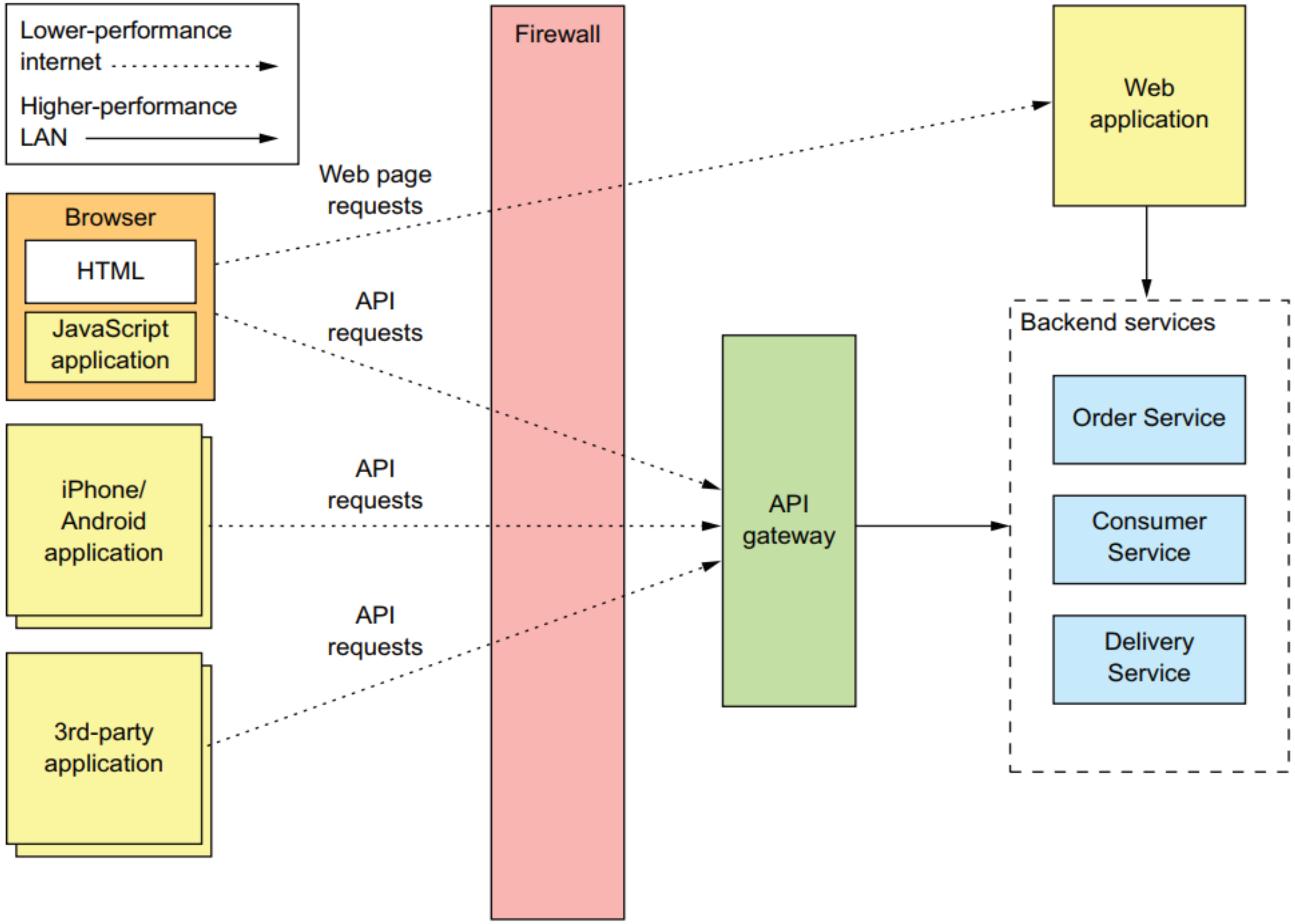
- ❑Problem in external API
  - ❑API design issues for other kinds of clients
    - ❑Api design issues for web applications
    - ❑Api design issues for browser-based javascript applications

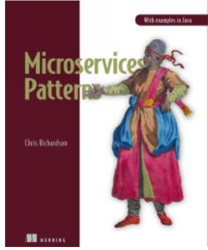


External API patterns

❑The API gateway pattern

- ❑An API gateway is a service that's the entry point into the application from the outside world. It's responsible for **request routing**, **API composition**, and other functions, such as authentication.





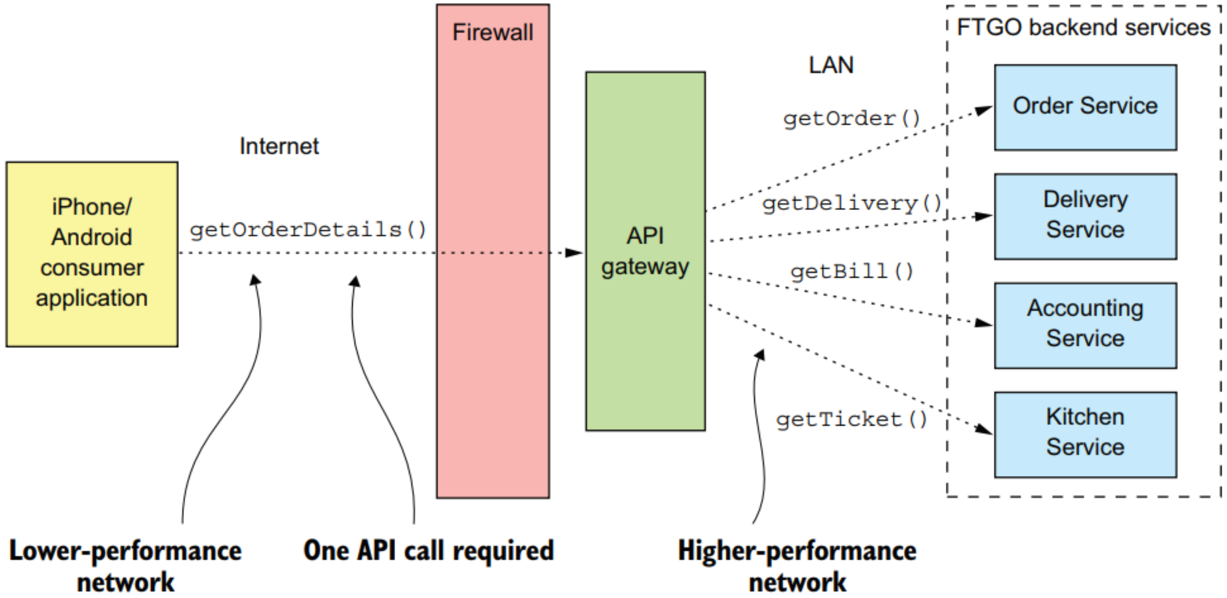
External API patterns

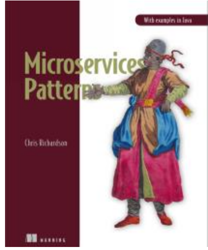
❑The API gateway pattern

An API gateway might also perform protocol translation. It might provide a RESTful API to external clients, even though the application services use a mixture of protocols internally, including REST and gRPC

An API gateway implements some API operations by routing requests to the corresponding service.

When it receives a request, the API gateway consults a routing map that specifies which service to route the request to





External API patterns

❑The API gateway pattern

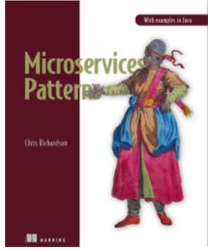
Authentication - Verifying the identity of the client making the request

Rate limiting - Limiting how many requests per second from either a specific client and/or from all clients.

Authorization - Verifying that the client is authorized to perform that particular operation.

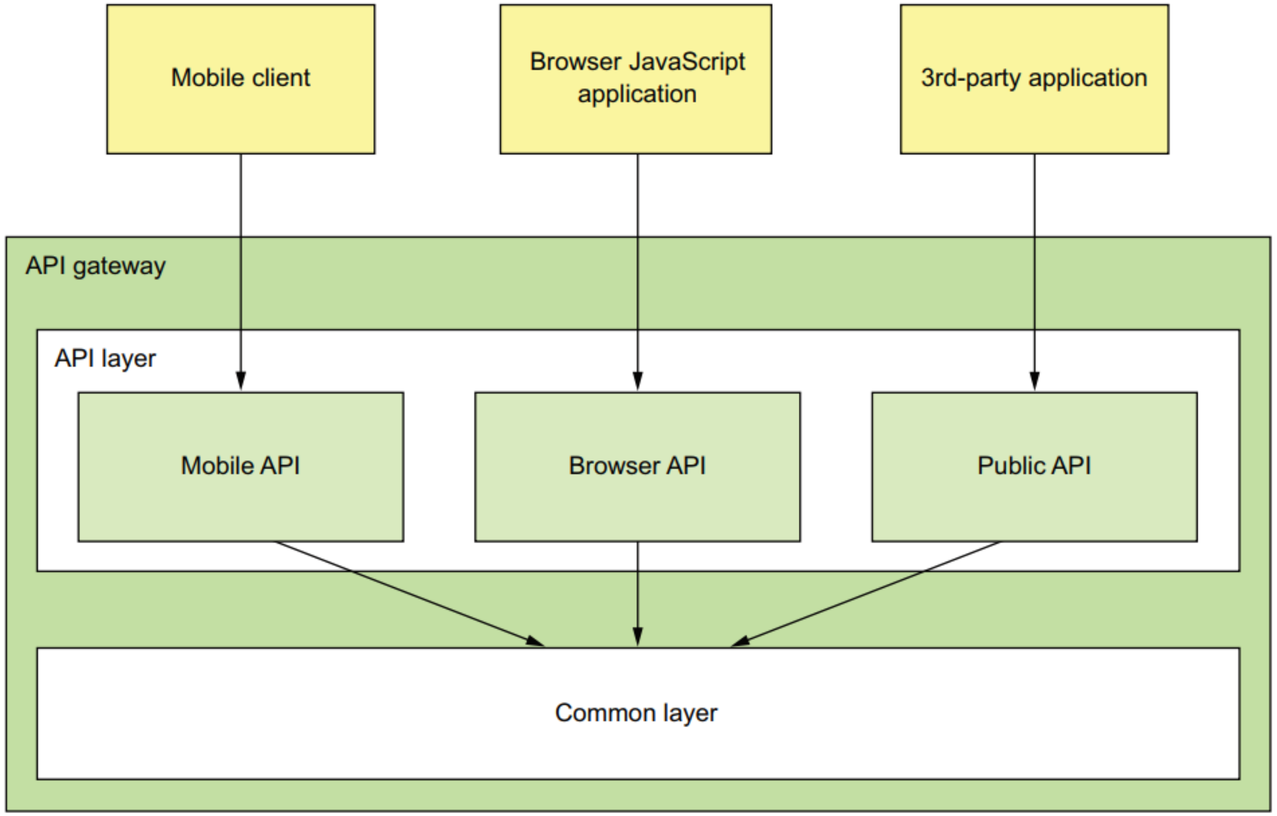
Caching - Cache responses to reduce the number of requests made to the services

Request logging - Log requests



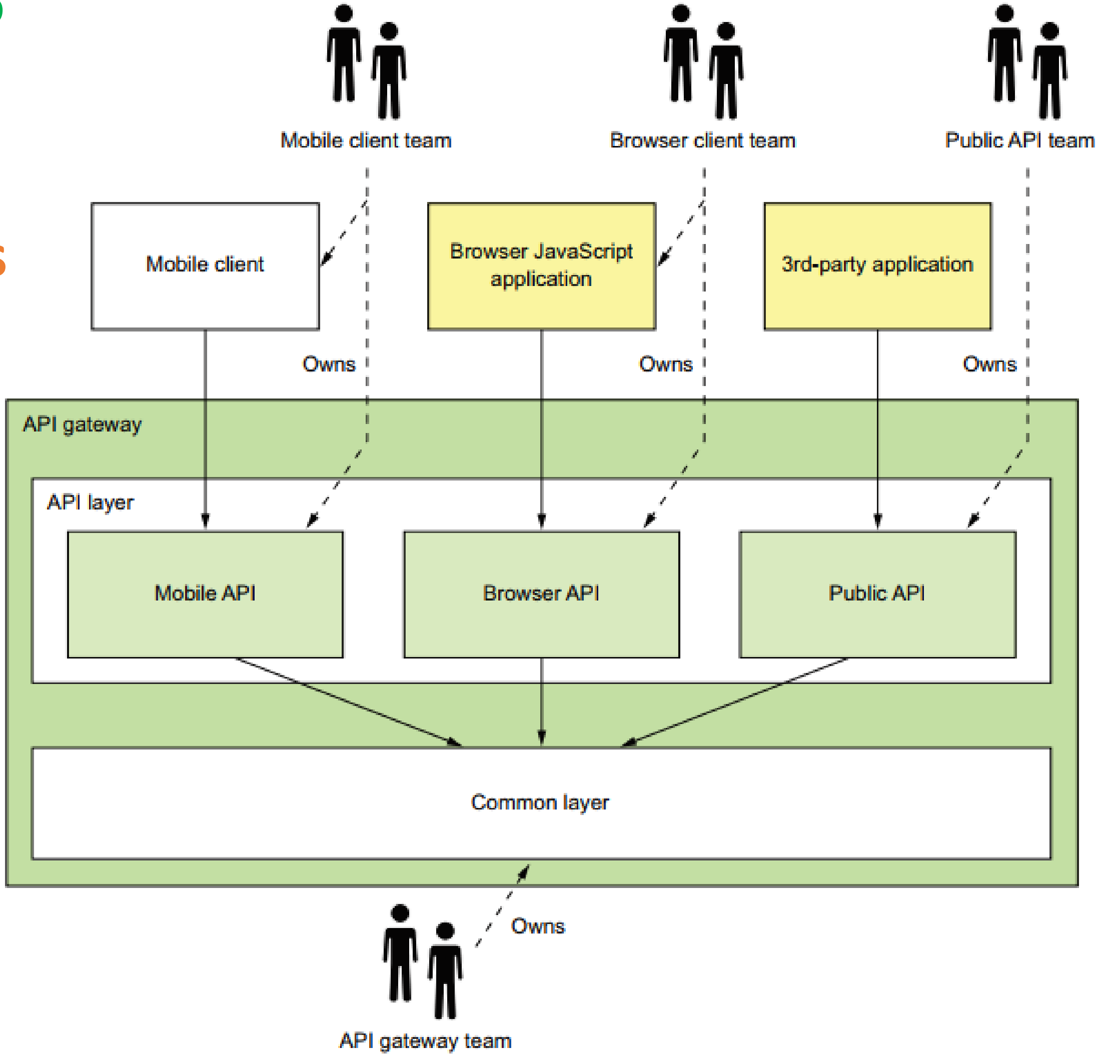
External API patterns

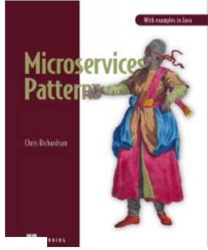
- ❑The API gateway pattern
  - ❑Api gateway architecture

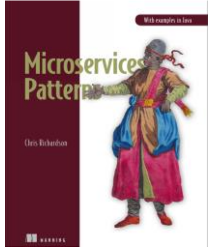


External API patterns

- ❑The API gateway pattern
  - ❑Using the backends for frontends pattern

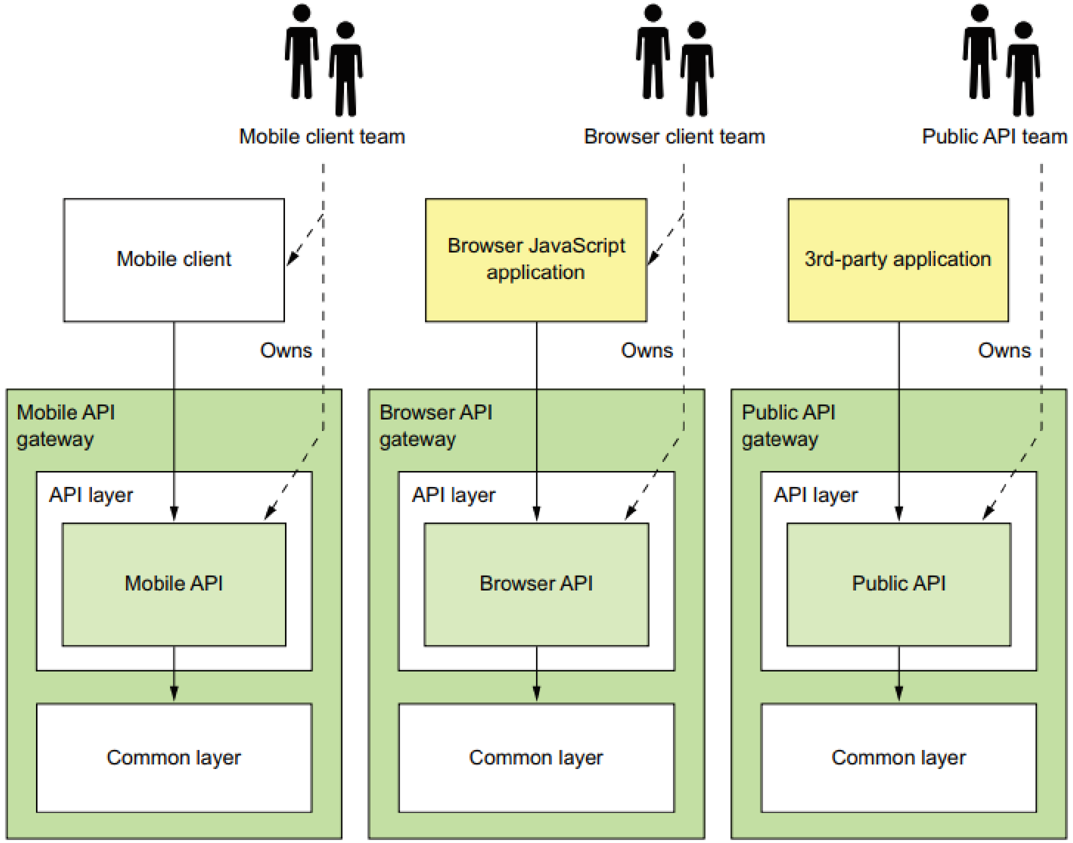


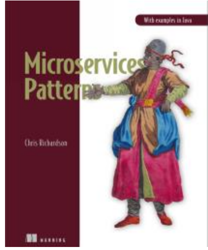




External API patterns

- ❑The API gateway pattern
  - ❑Using the backends for frontends pattern



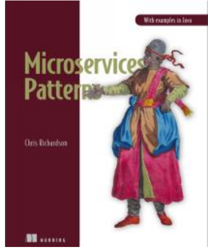


External API patterns

❑The API gateway pattern

❑Benefits

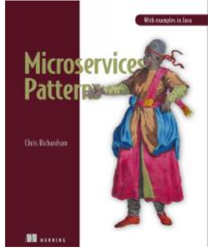
- ❑It encapsulates internal structure of the application.
- ❑The API gateway provides each client with a client-specific API
- ❑It also simplifies the client code.



External API patterns

- ❑The API gateway pattern
  - ❑Drawbacks
    - ❑Development bottleneck
    - ❑Must be developed, deployed, and managed

External API patterns

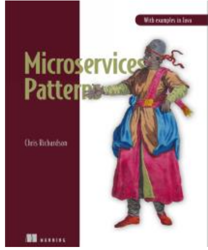


❑The API gateway pattern

❑API gateway design issues

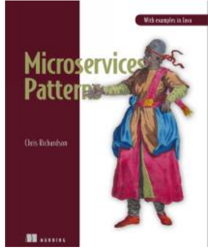
- ❑Performance and scalability
- ❑Use reactive programming abstractions
- ❑Handling partial failures
- ❑Being a good citizen in the architecture

External API patterns



- ❑ **Implementing an API gateway**
  - ❑ **Using an off-the-shelf API gateway product/service**
    - ❑ AWS api gateway
      - ❑ One of the many services provided by Amazon Web Services, is a service for deploying and managing APIs
      - ❑ It doesn’t support API composition, so you’d need to implement API composition in the backend services
      - ❑ It only supports the Serverside discovery pattern
    - ❑ AWS application load balancer
      - ❑ It implements basic routing functionality
      - ❑ It doesn’t implement HTTP method-based routing
      - ❑ the AWS Application Load Balancer doesn’t meet the requirements for an API gateway

External API patterns

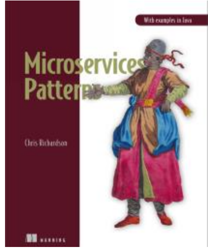


□ Implementing an API gateway

□ Using an off-the-shelf API gateway product/service

- □ Using an api gateway product: Kong or Traefik
  - □ Kong is based on the NGINX HTTP server
  - □ Traefik is written in GoLang
  - □ Kong configure plugins that implement edge functions such as authentication
  - □ Traefik can even integrate with some service registries

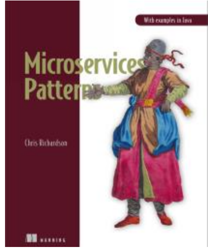
They don't support API composition => need to **develop own API gateway**



External API patterns

- □ Implementing an API gateway
  - □ Developing your own API gateway
    - □ Implementing a mechanism for defining routing rules in order to minimize the complex coding
    - □ Correctly implementing the HTTP proxying behavior, including how HTTP headers are handled
    - □ Frameworks
      - □ Netflix Zuul, an open source project by Netflix
      - □ Spring Cloud Gateway, an open source project from Pivotal

Transaction management in a microservice architecture

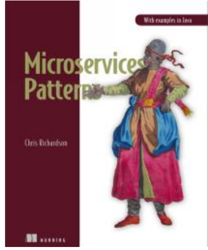


❑The need for distributed transactions in a microservice architecture

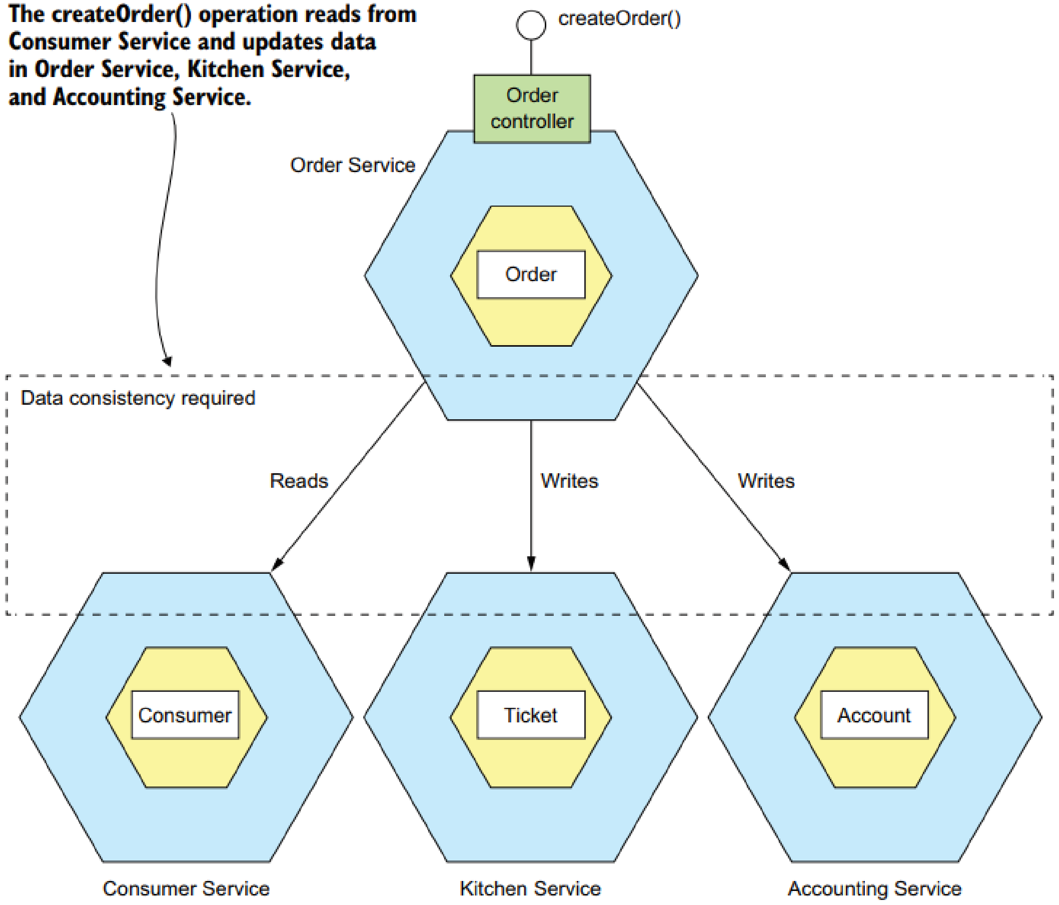
❑The createOrder() operation accesses data in numerous services. It reads data from Consumer Service and updates data in Order Service, Kitchen Service, and Accounting Service.

❑Because each service has its own database, need to use a mechanism to maintain data consistency across those databases.

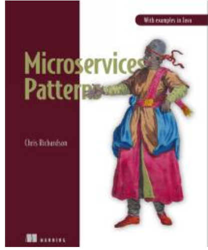
Transaction management in a microservice architecture



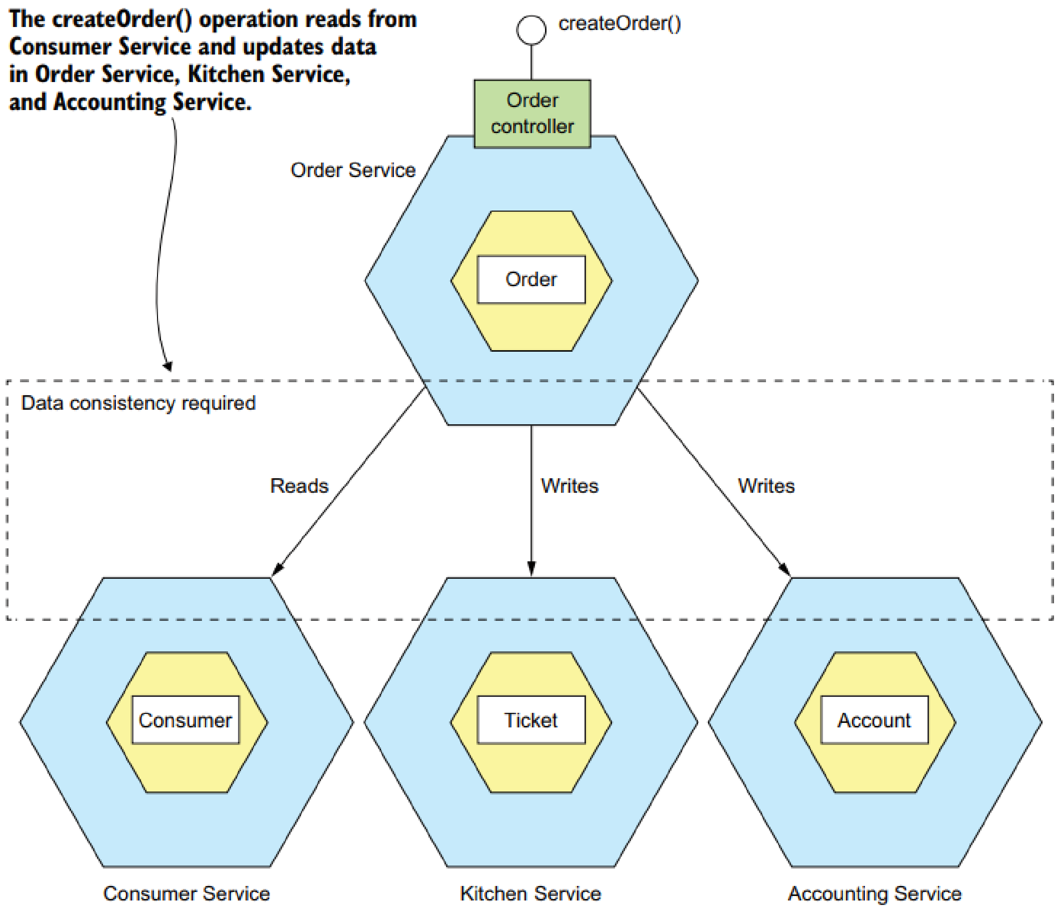
- ❑The trouble with distributed transactions
  - ❑X/Open Distributed Transaction Processing (DTP) Model uses two-phase commit (2PC) to ensure that all participants in a transaction either commit or rollback


The createOrder() operation reads from Consumer Service and updates data in Order Service, Kitchen Service, and Accounting Service.
createOrder()
Order controller
Order Service
Order
Data consistency required
Reads
Writes
Writes
Consumer
Ticket
Account
Consumer Service
Kitchen Service
Accounting Service

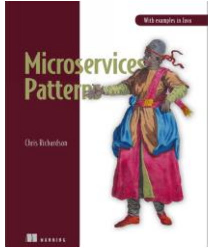
Transaction management in a microservice architecture



- ❑The trouble with distributed transactions
  - ❑X/Open Distributed Transaction Processing (DTP) Model uses two-phase commit (2PC) to ensure that all participants in a transaction either commit or rollback



The createOrder() operation reads from Consumer Service and updates data in Order Service, Kitchen Service, and Accounting Service.



Summary

- The microservice architecture is a distributed architecture, so interprocess communication plays a key role
- There are numerous IPC technologies, each with different trade-offs
- Use the Circuit breaker pattern to avoid making calls to a failing service
- An architecture that uses synchronous protocols must include a service discovery mechanism in order for clients to determine the network location of a service instance.
- API gateway is responsible for request routing, API composition, protocol translation, and implementation of edge functions such as authentication.
- API gateway can use the Backends for frontends pattern, which defines an API gateway for each type of client.
- There are numerous technologies you can use to implement an API gateway, including off-the-shelf API gateway products.