CSC12108

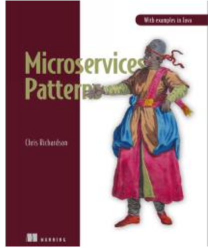
Distributed Application

TOPIC

Designing and Developing business logic in Microserivces

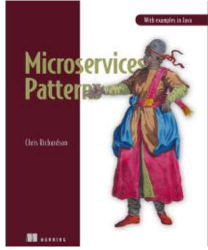
Instructor – Msc. PHAM MINH TU

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



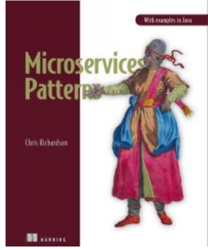
Outline

- ❑Designing business logic in a microservice architecture
  - ❑Business logic organization **patterns**
  - ❑Designing a domain model using the **DDD aggregate pattern**
- ❑Developing business logic with event sourcing
  - ❑Overview of event sourcing
  - ❑Implementing an event store
  - ❑Using sagas and event sourcing together



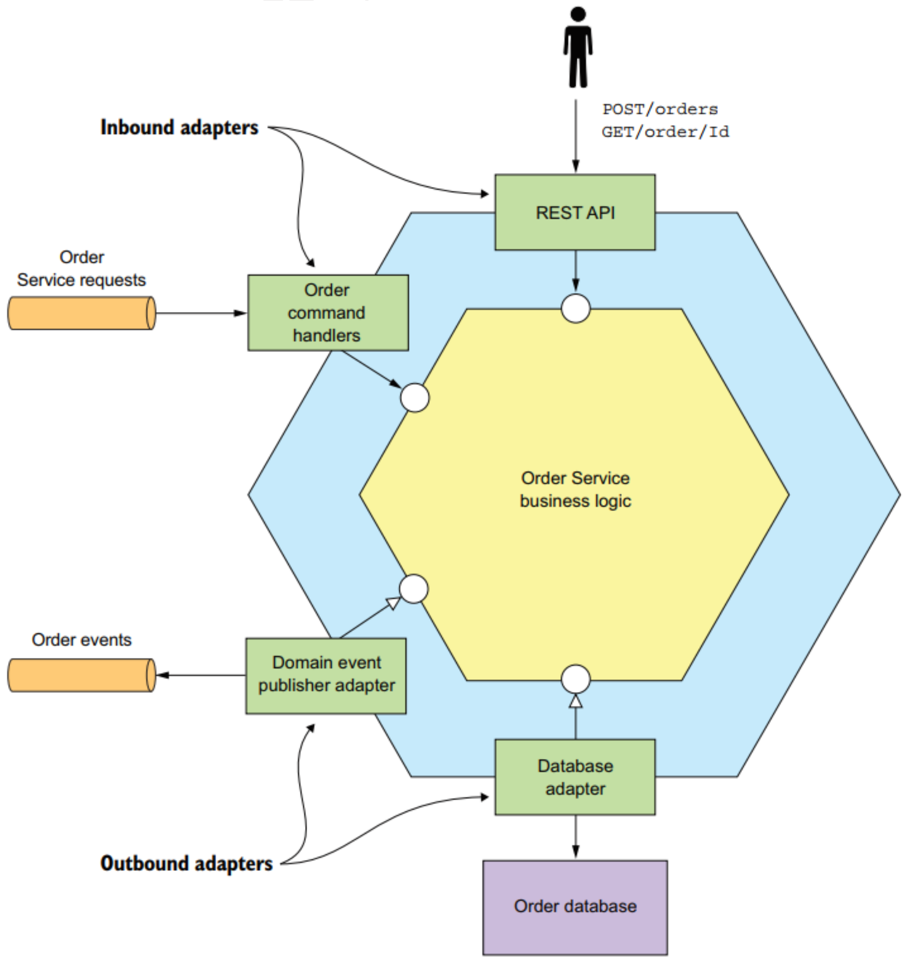
Outline

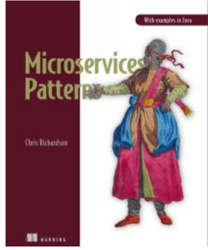
- ❑Designing business logic in a microservice architecture
  - ❑Business logic organization **patterns**
  - ❑Designing a domain model using the **DDD aggregate pattern**
- ❑Developing business logic with event sourcing
  - ❑Overview of event sourcing
  - ❑Implementing an event store
  - ❑Using sagas and event sourcing together



Designing business logic in a microservice architecture

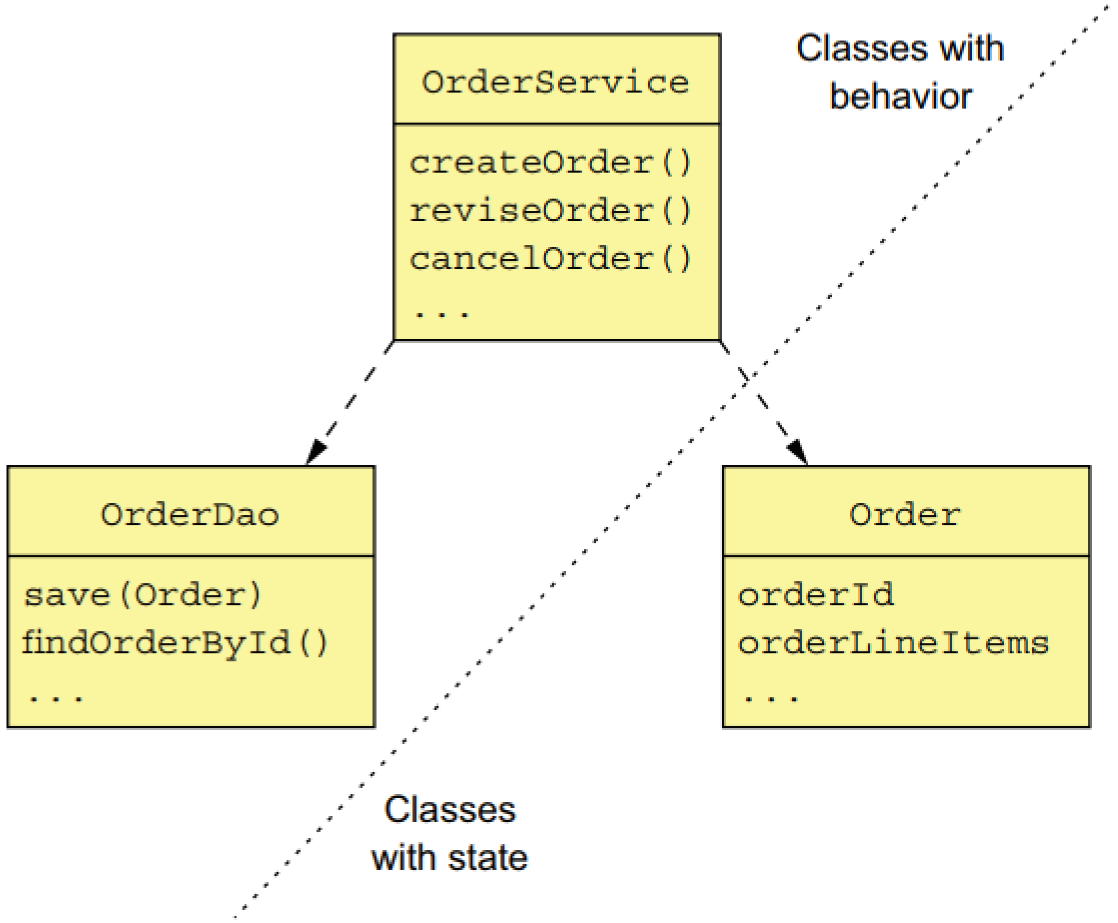
- ❑Business logic organization **patterns**
  - ❑The architecture of a typical service
    - ❑An inbound adapter handles requests from clients and invokes the business logic.
    - ❑An outbound adapter, which is invoked by the business logic, invokes other services and applications
  - ❑Two patterns
    - ❑**Transaction script pattern**
    - ❑**Object-oriented Domain model pattern**

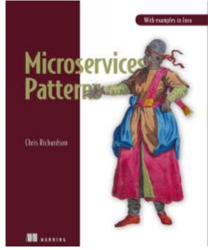




Designing business logic in a microservice architecture

- ❑Business logic organization patterns
  - ❑Transaction script pattern
    - ❑The classes that implement behavior are separate from those that store state
    - ❑To works well for simple business logic
    - ❑Not to be a good way to implement complex business logic





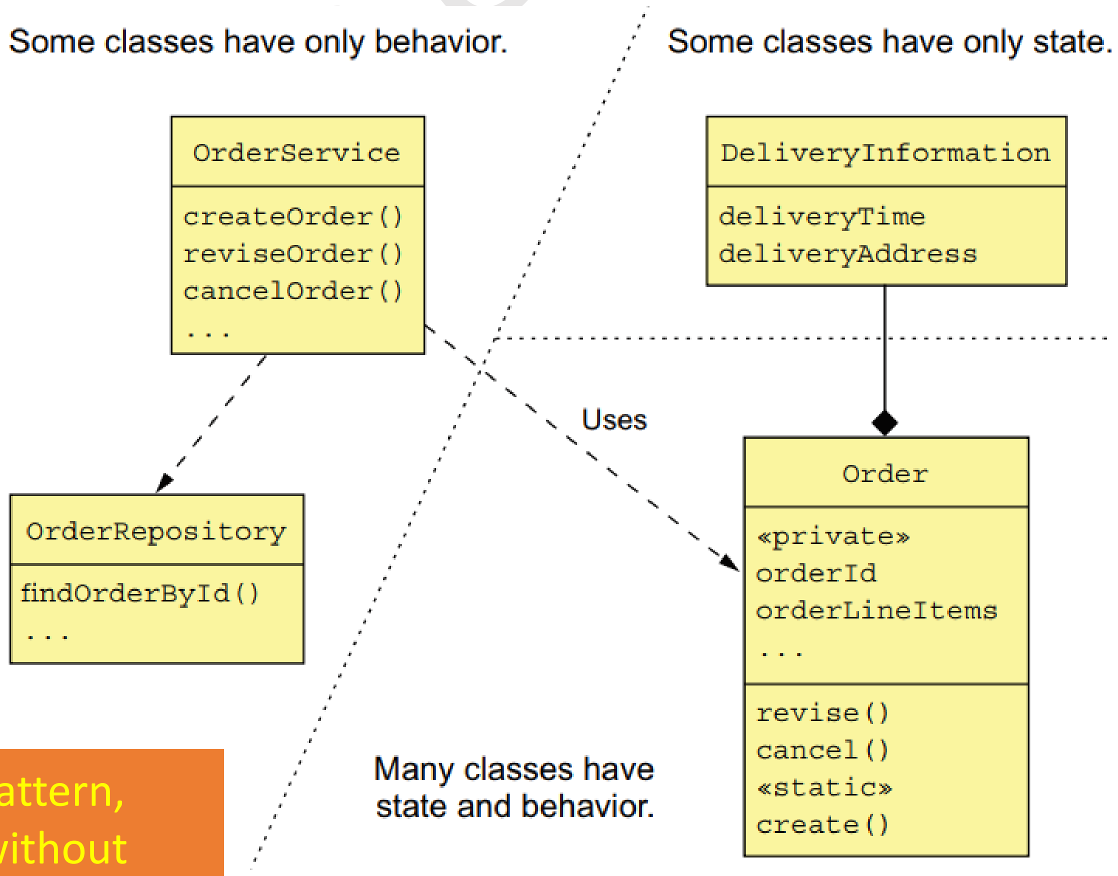
Designing business logic in a microservice architecture

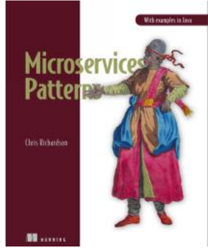
❑Business logic organization patterns

❑Domain model pattern

- ❑In such a design some classes have only either state or behavior, but many contain both
- ❑The design is easy to understand and maintain
- ❑Design is easier to test
- ❑Design is easier to extend

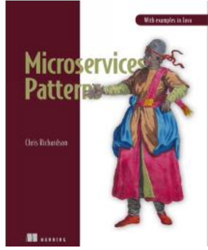
Strategy pattern and the **Template** method pattern, that define ways of extending a component without modifying the code.





Designing business logic in a microservice architecture

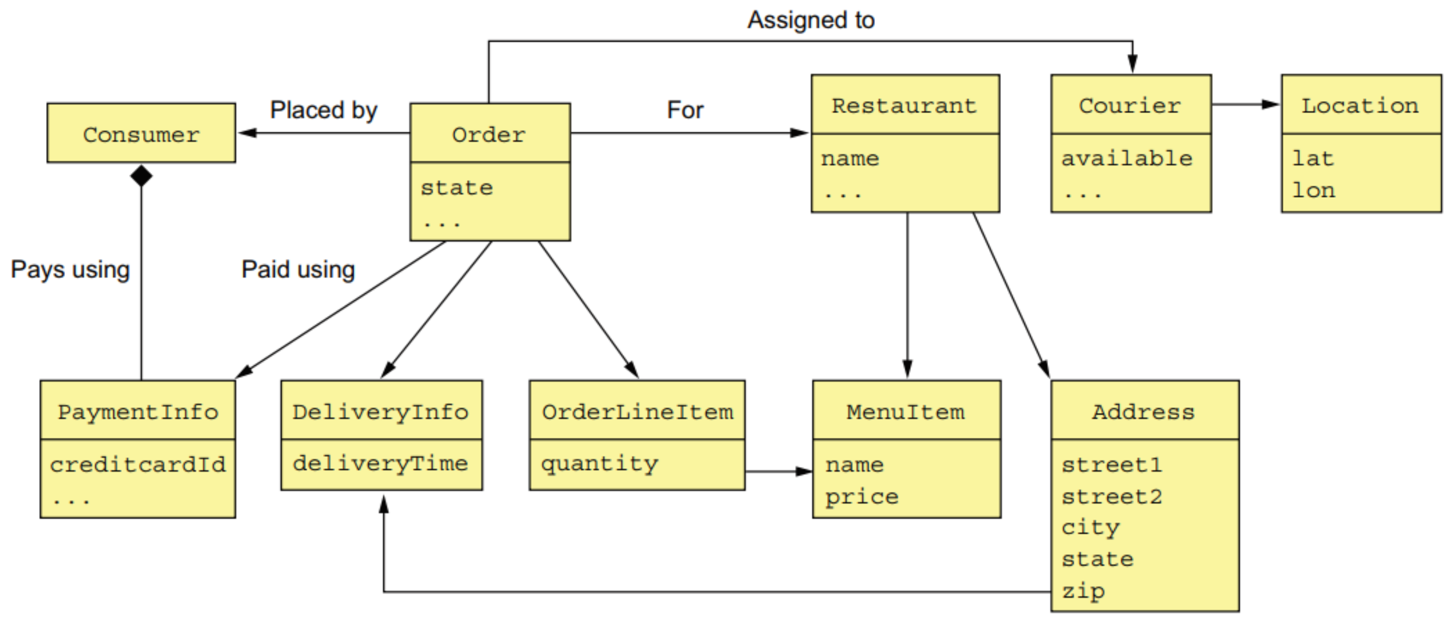
- ❑Business logic organization patterns
  - ❑Domain-driven design
    - ❑DDD is a refinement of OOD and is an approach for developing complex business logic
    - ❑DDD also has some tactical patterns that are building blocks for domain models
      - ❑ **Entity:** An object that has a **persistent** identity. Two entities whose attributes have the **same values** are still different objects
      - ❑ **Value object:** An object that is a collection of values. Two value objects whose attributes **have the same values** can be used interchangeably
      - ❑ **Factory**: An object or method that **implements object creation logic** that's too complex to be done directly by a constructor.
      - ❑ **Repository**: An object that provides **access to persistent entities** and **encapsulates** the mechanism for accessing the database
      - ❑ **Service**: An object that **implements** business logic that doesn't belong in an entity or a value object.

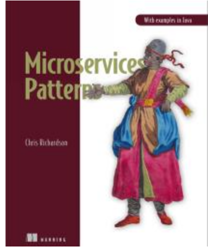


Designing business logic in a microservice architecture

- ❑Business logic organization **patterns**
  - ❑Domain-driven design

A traditional domain model is a web of interconnected classes. It doesn't explicitly specify the boundaries of business objects, such as **Consumer** and **Order**



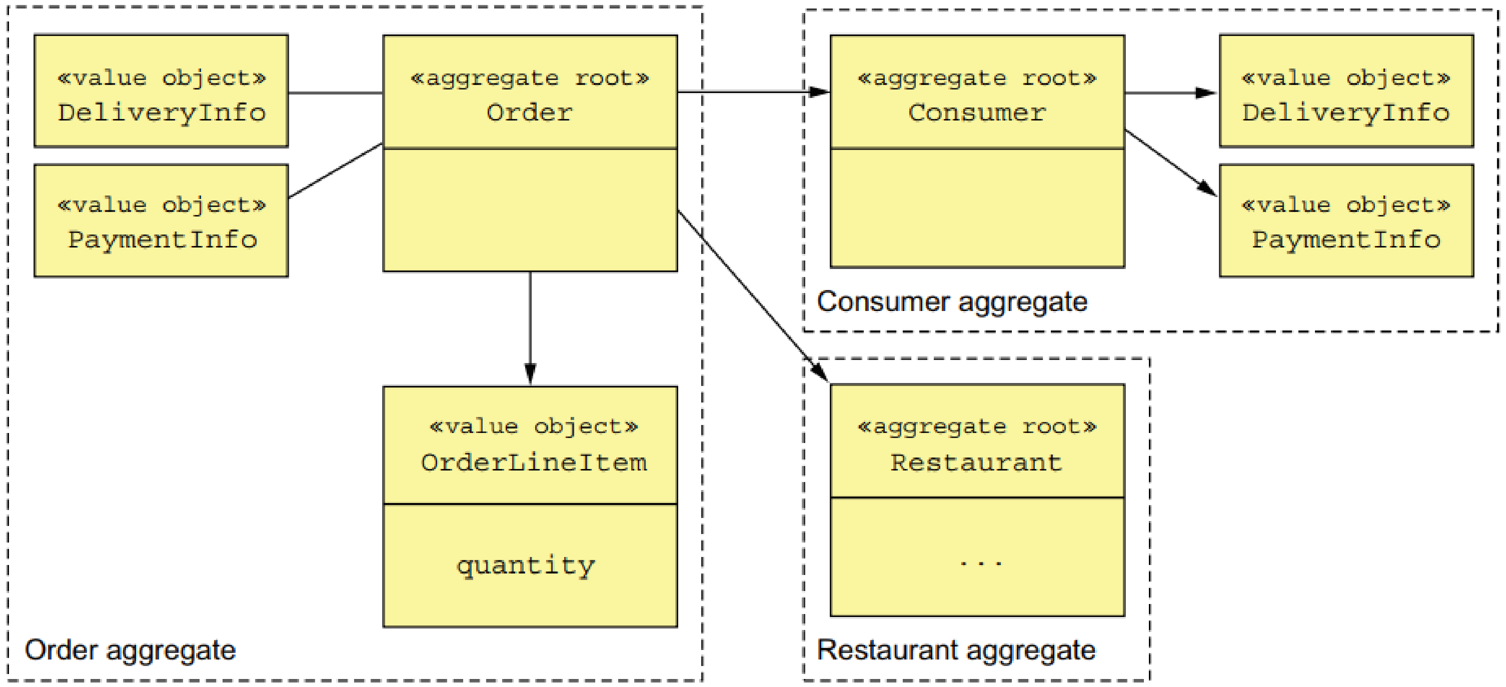


Designing business logic in a microservice architecture

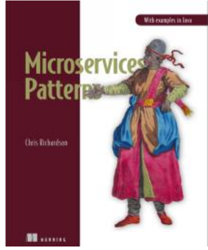
❑Business logic organization patterns

❑Domain-driven design

An **aggregate** is a cluster of domain objects within a **boundary** that can be treated as a **unit**. It consists of a **root entity** and possibly one or more **other entities** and value objects



«value object» DeliveryInfo
«value object» PaymentInfo
«aggregate root» Order
«value object» OrderLineItem
quantity
Order aggregate
«aggregate root» Consumer
«value object» DeliveryInfo
«value object» PaymentInfo
Consumer aggregate
«aggregate root» Restaurant
...
Restaurant aggregate



Designing business logic in a microservice architecture

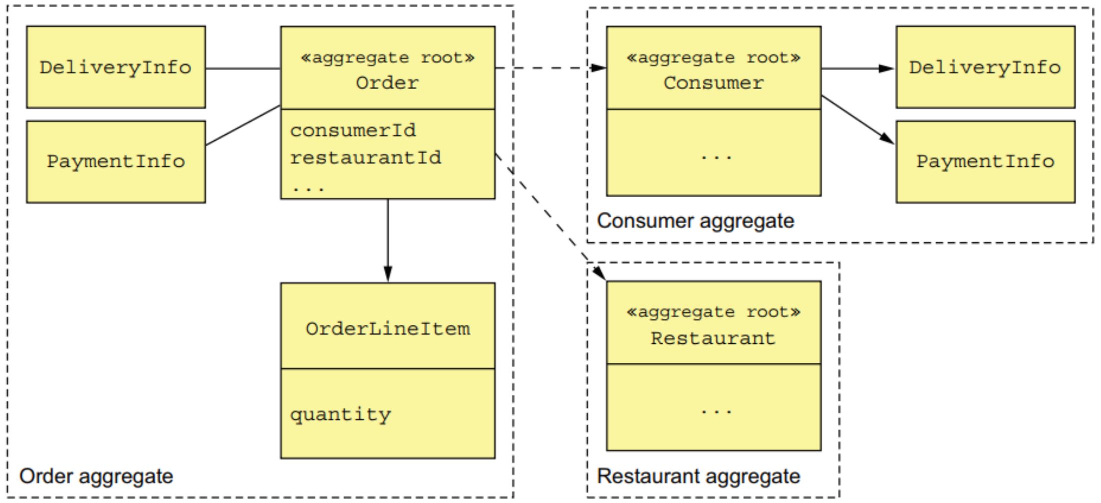
- ❑Business logic organization patterns
  - ❑Domain-driven design

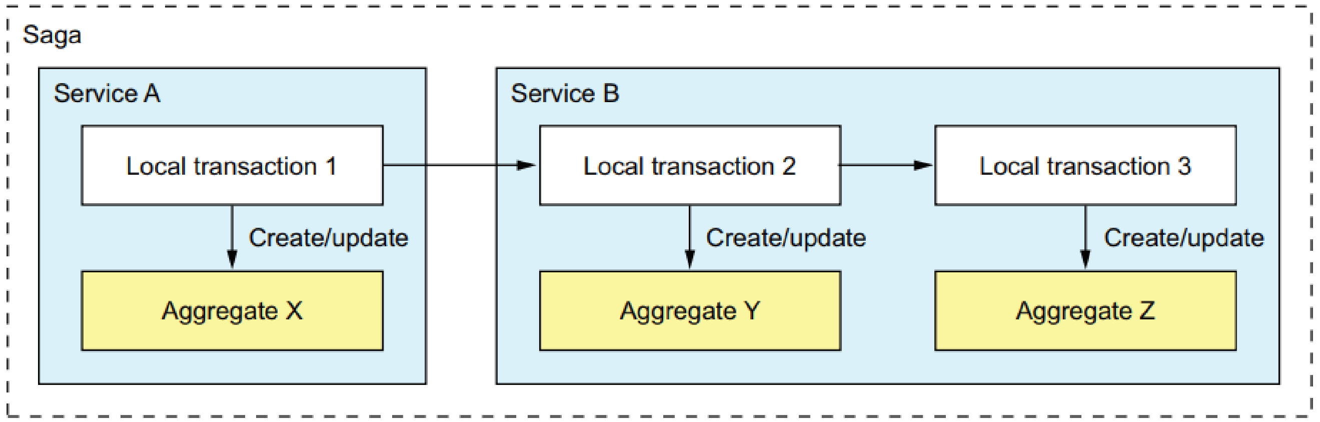
How to Identify aggregates?

Rule #1: Reference only the aggregate root

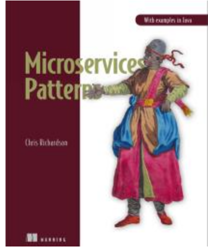
Rule #2: Inter-aggregate references must use primary keys

Rule #3: One transaction creates or updates one aggregate



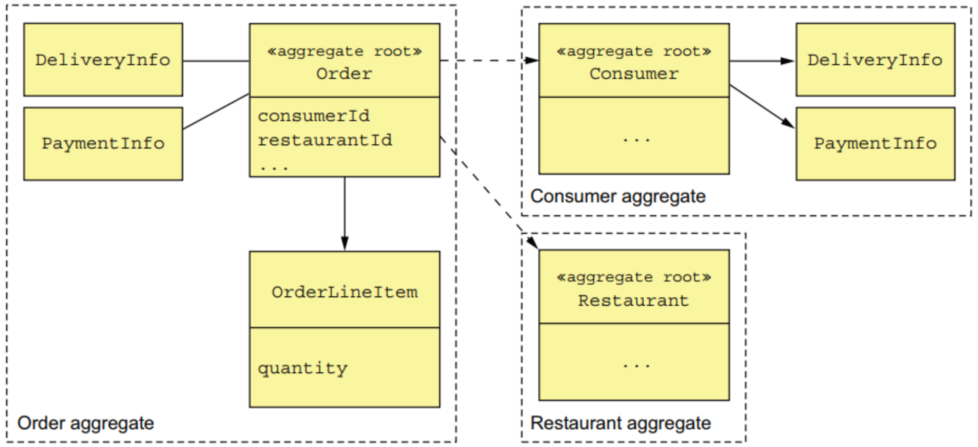


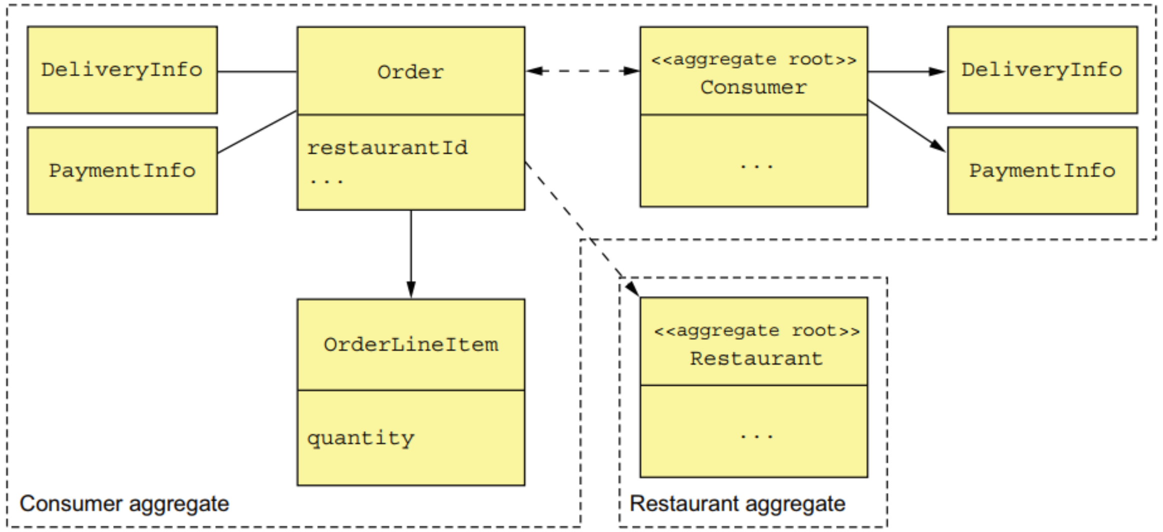
Designing business logic in a microservice architecture



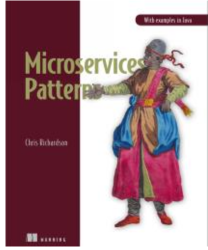
- ❑Business logic organization patterns
  - ❑Domain-driven design

Aggregate granularity





Designing business logic in a microservice architecture

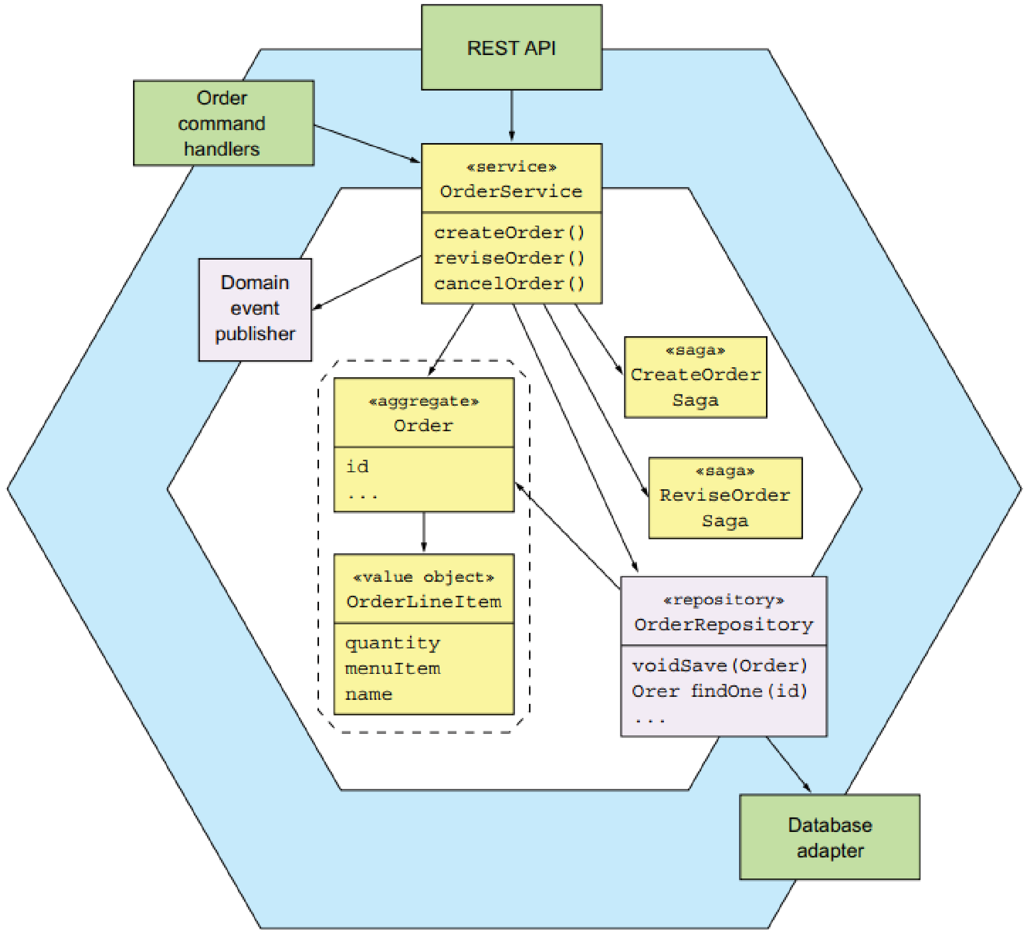


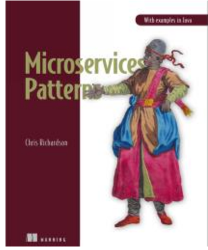
❑Business logic organization patterns

❑Domain-driven design

❑Designing business logic with aggregates

- ❑The business logic consists of the Order aggregate, the OrderService service class, the OrderRepository, and one or more sagas. The OrderService invokes the OrderRepository to save and load Orders.
- ❑For simple requests that are local to the service, the service updates an `Order` aggregate. If an update request spans multiple services, the `OrderService` will also create a saga.

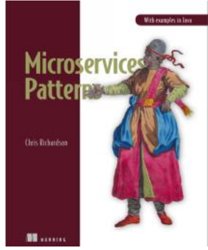




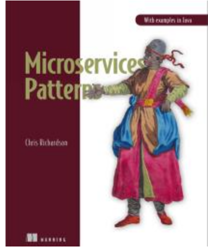
Designing business logic in a microservice architecture

- ❑Publishing domain events
  - ❑What is a domain event?
    - ❑A domain event is a class with a name formed using a past-participle verb. It has properties that meaningfully convey the event. Each property is either a primitive value or a value object.
  - ❑Why publish change events?
    - ❑Domain events are useful because other parties—users, other applications, or other components within the same application—are often interested in knowing about an aggregate's state changes

Designing business logic in a microservice architecture



- ❑Publishing domain events
  - ❑Example scenarios
    - ❑Notifying a different component of the same application in order, for example, to send a WebSocket message to a user’s browser or update a text database such as ElasticSearch
    - ❑Sending notifications—text messages or emails—to users informing them that their order has shipped
    - ❑Monitoring domain events to verify that the application is behaving correctly.
    - ❑Analyzing events to model user behavior.



Designing business logic in a microservice architecture

❑Publishing domain events

❑Generating and publishing domain events

```
public class Ticket {

   public List<DomainEvent> accept(ZonedDateTime readyBy) {
    ...
    this.acceptTime = ZonedDateTime.now();
    this.readyBy = readyBy;
    return singletonList(new TicketAcceptedEvent(readyBy));
   }
}
```

Updates the Ticket

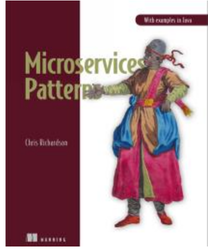
Returns an event

```
public class KitchenService {

   @Autowired
   private TicketRepository ticketRepository;

   @Autowired
   private DomainEventPublisher domainEventPublisher;

public void accept(long ticketId, ZonedDateTime readyBy) {
  Ticket ticket =
        ticketRepository.findById(ticketId)
          .orElseThrow(() ->
                    new TicketNotFoundException(ticketId));
  List<DomainEvent> events = ticket.accept(readyBy);
  domainEventPublisher.publish(Ticket.class, orderId, events);
}
```



Designing business logic in a microservice architecture

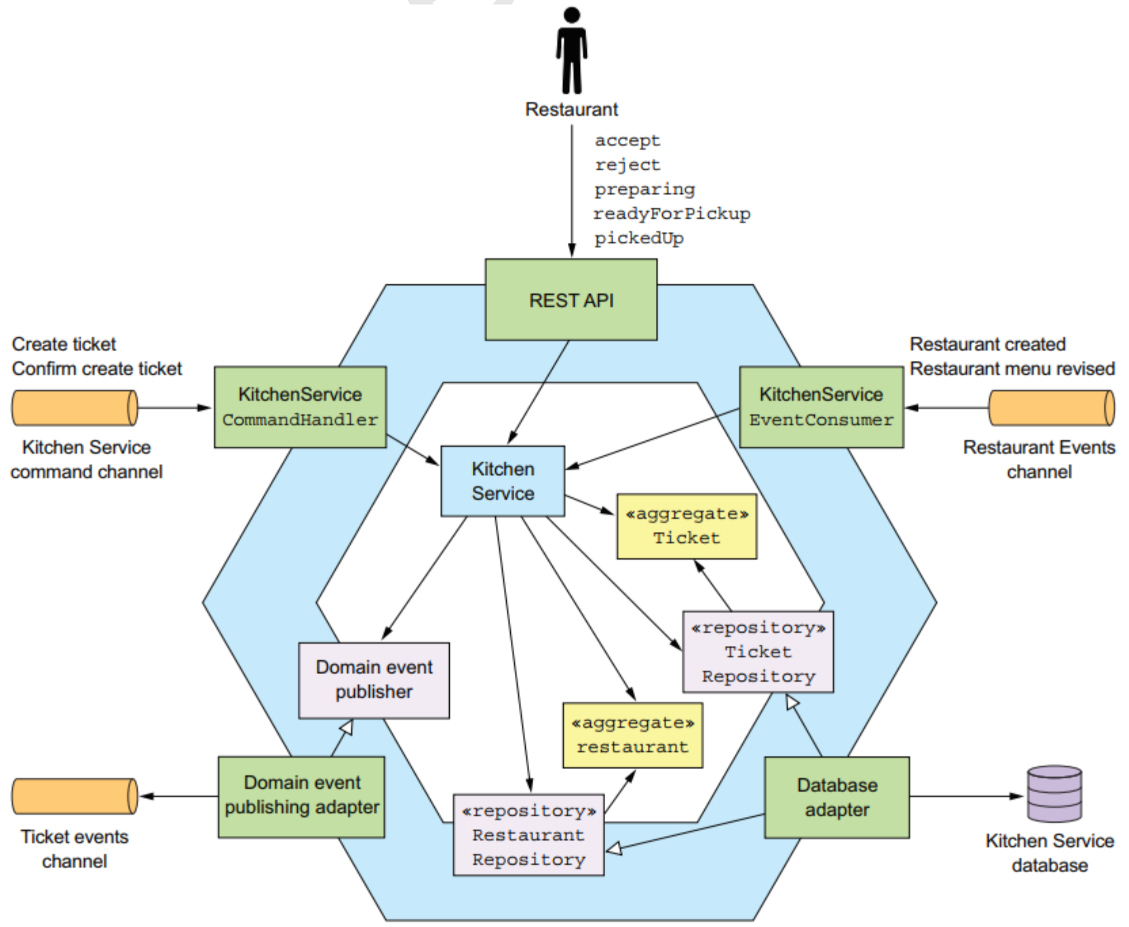
- ❑Publishing domain events
  - ❑Consuming domain events
    - ❑Domain events are ultimately published as messages to a message broker, such as Apache Kafka. A consumer could use the broker's client API directly.

Designing business logic in a microservice architecture

❑Examples

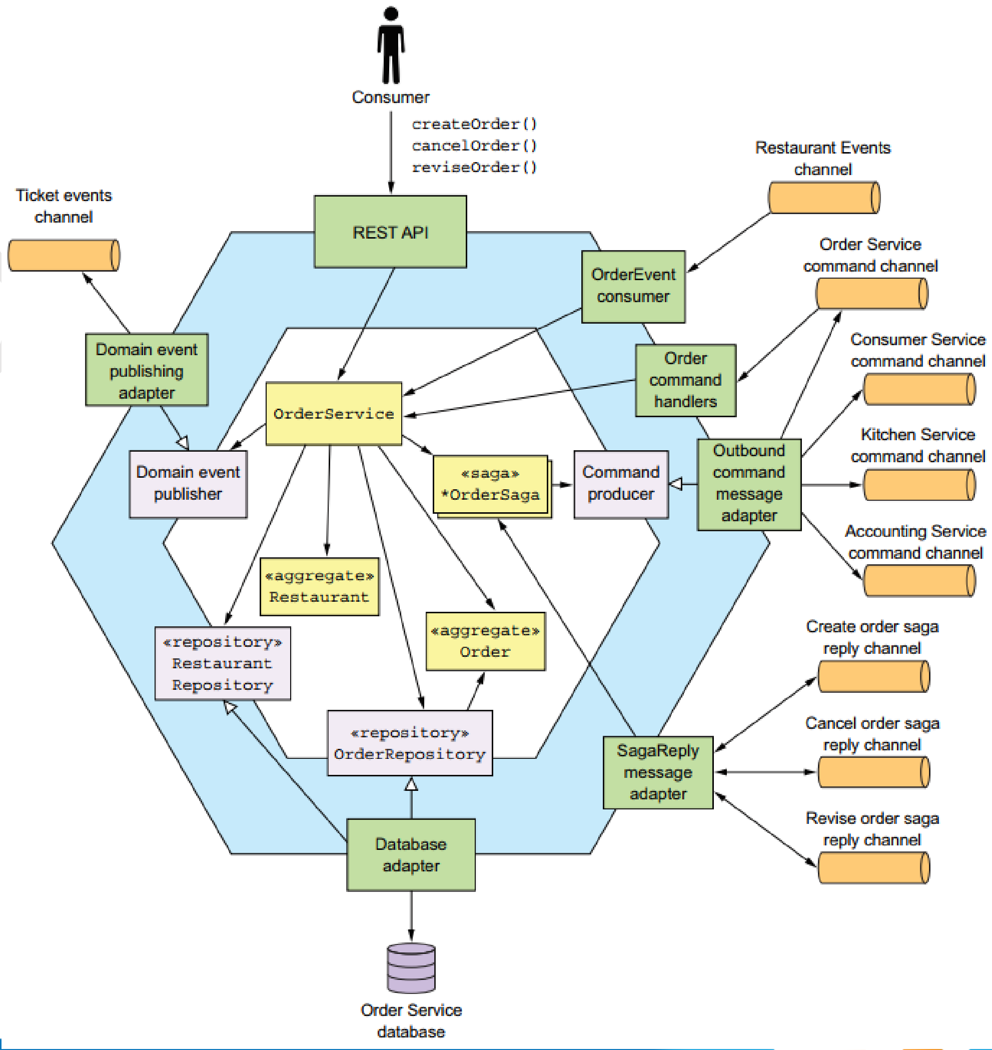
❑Kitchen Service business logic

- ❑ REST API—The REST API invoked by the user interface used by workers at the restaurant. It invokes KitchenService to create and update Tickets
- ❑ KitchenServiceCommandHandler—The asynchronous request/response-based API that's invoked by sagas. It invokes KitchenService to create and update Tickets.
- ❑ KitchenServiceEventConsumer—Subscribes to events published by Restaurant Service. It invokes KitchenService to create and update Restaurants.
- ❑ DB adapter—Implements the TicketRepository and the RestaurantRepository interfaces and accesses the database.
- ❑ DomainEventPublishingAdapter—Implements the DomainEventPublisher interface and publishes Ticket domain events.

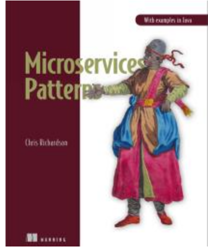


Designing business logic in a microservice architecture

- Examples
  - Order Service business logic
    - REST API—The REST API invoked by the user interface used by consumers. It invokes OrderService to create and update Orders.
    - OrderEventConsumer—Subscribes to events published by Restaurant Service. It invokes OrderService to create and update its replica of Restaurants.
    - OrderCommandHandlers—The asynchronous request/response-based API that's invoked by sagas. It invokes OrderService to update Orders.
    - SagaReplyAdapter—Subscribes to the saga reply channels and invokes the sagas.
    - DB adapter—Implements the OrderRepository interface and accesses the Order Service database
    - DomainEventPublishingAdapter—Implements the DomainEventPublisher interface and publishes Order domain events
    - OutboundCommandMessageAdapter—Implements the CommandPublisher interface and sends command messages to saga participants

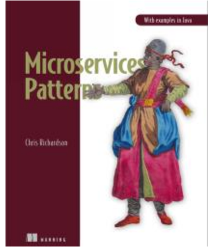


Consumer
createOrder()
cancelOrder()
reviseOrder()
Restaurant Events channel
Ticket events channel
REST API
OrderEvent consumer
Order Service command channel
Domain event publishing adapter
Order command handlers
Consumer Service command channel
OrderService
Domain event publisher
«saga» *OrderSaga
Command producer
Outbound command message adapter
Kitchen Service command channel
Accounting Service command channel
«aggregate» Restaurant
«aggregate» Order
«repository» Restaurant Repository
Create order saga reply channel
«repository» OrderRepository
SagaReply message adapter
Cancel order saga reply channel
Revise order saga reply channel
Database adapter
Order Service database



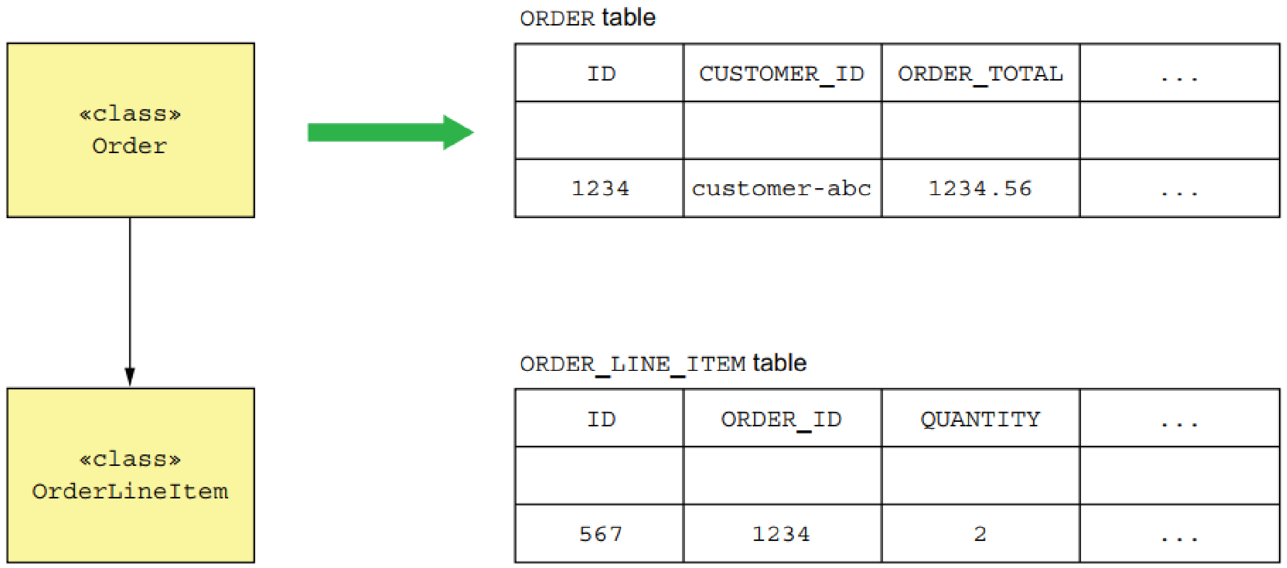
Outline

- ❑Designing business logic in a microservice architecture
  - ❑Business logic organization patterns
  - ❑Designing a domain model using the DDD aggregate pattern
- ❑Developing business logic with event sourcing
  - ❑Overview of event sourcing
  - ❑Implementing an event store



Developing business logic using event sourcing

- ❑The trouble with traditional persistence
  - ❑Drawbacks and limitations
    - ❑Object-Relational impedance mismatch.
    - ❑Lack of aggregate history.
    - ❑Implementing audit logging is tedious and error prone.
    - ❑Event publishing is bolted on to the business logic.

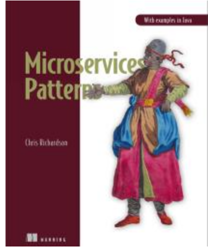


ORDER table

ID	CUSTOMER_ID	ORDER_TOTAL	...
1234	customer-abc	1234.56	...

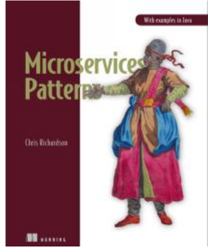
ORDER_LINE_ITEM table

ID	ORDER_ID	QUANTITY	...
567	1234	2	...



Developing business logic using event sourcing

- The trouble with traditional persistence
  - **Drawbacks and limitations**
    - Object-Relational impedance mismatch.
      - There's a fundamental conceptual **mismatch** between the tabular relational schema and the graph structure of a rich domain model with its complex relationships.
    - Lack of **aggregate history**
      - Another limitation of traditional persistence is that it only stores the current state of an aggregate. Once an aggregate has been updated, its previous state is lost
    - Implementing **audit logging** is tedious and error prone
      - Many applications must maintain an audit log that tracks which users have changed an aggregate.
    - Event publishing **is bolted** on to the business logic.
      - Limitation of traditional persistence is that it usually **doesn't support publishing domain events**, domain events are events that are published by an aggregate when its state changes. They're a useful mechanism for synchronizing data and sending notifications in microservice architecture



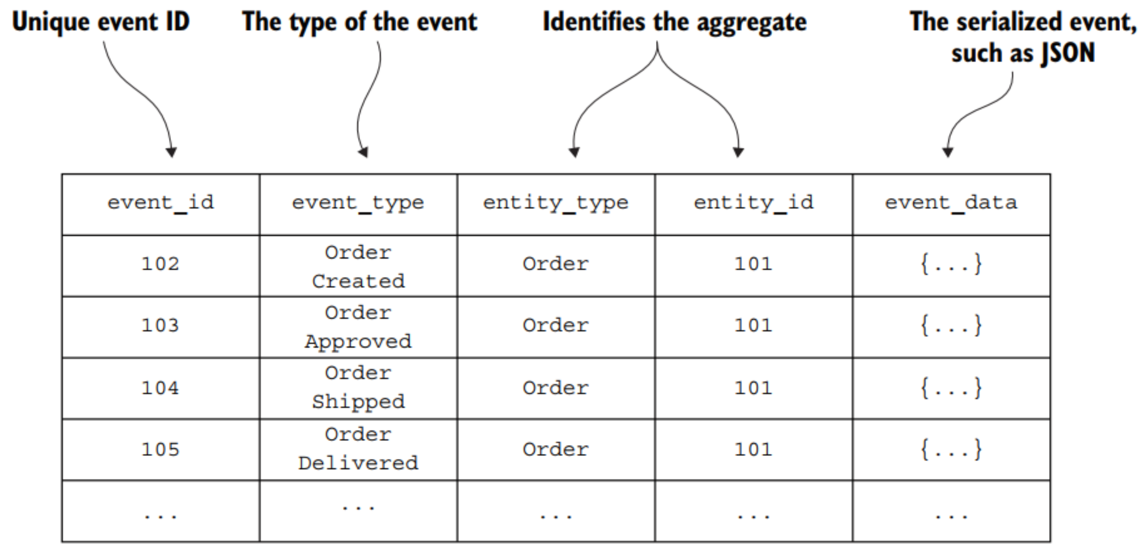
Developing business logic using event sourcing

❑Overview of event sourcing

Event sourcing is an **event-centric technique** for implementing business logic and persisting aggregates.

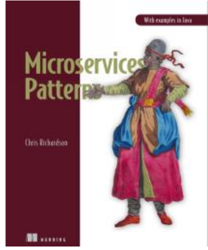
An aggregate is stored in the database as a **series of events**

Each event **represents a state change** of the aggregate



event_id	event_type	entity_type	entity_id	event_data
102	Order Created	Order	101	{...}
103	Order Approved	Order	101	{...}
104	Order Shipped	Order	101	{...}
105	Order Delivered	Order	101	{...}
...	...	...	...	...

EVENTS table

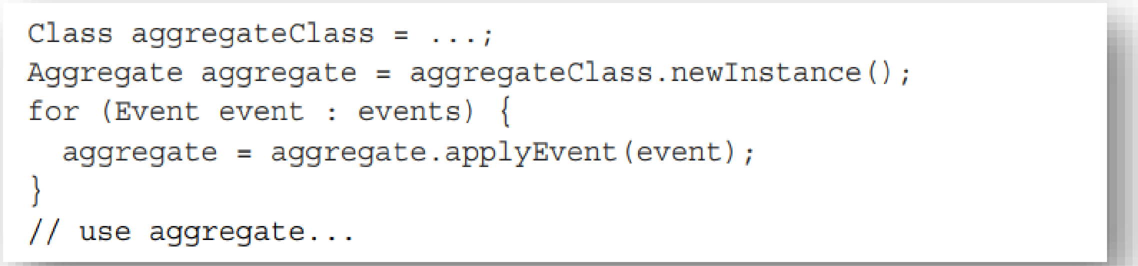


Developing business logic using event sourcing

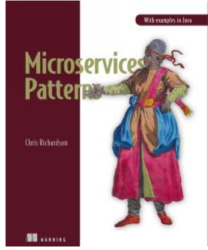
- ❑Overview of event sourcing
  - ❑Event sourcing persists aggregates using events
    - ❑When an application creates or updates an aggregate, it inserts the events emitted by the aggregate into the EVENTS table. An application loads an aggregate from the event store by retrieving its events and replaying them.

1. Load the events for the aggregate.
2. Create an aggregate instance by using its default constructor.
3. Iterate through the events, calling apply()

Eventuate Client framework



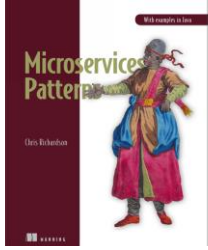
```
Class aggregateClass = ...;
Aggregate aggregate = aggregateClass.newInstance();
for (Event event : events) {
  aggregate = aggregate.applyEvent(event);
}
// use aggregate...
```



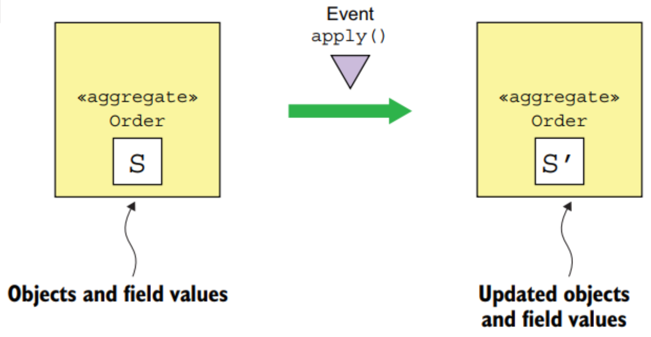
Developing business logic using event sourcing

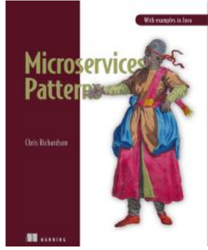
- ❑Overview of event sourcing
  - ❑Events represent state changes
    - ❑Events can either contain **minimal data**, such as just the aggregate ID, or can be enriched to **contain data that's useful** to a typical consumer. For example, the Order Service can publish an OrderCreated event when an order is created. An OrderCreated event may only contain the orderId. Alternatively, the event could contain the complete order so consumers of that event don't have to fetch the data from the Order Service
    - ❑Every state change of an aggregate, including its creation, is represented by a domain event. Whenever the **aggregate's state changes**, it must **emit an event**

Developing business logic using event sourcing



- ❑Overview of event sourcing
  - ❑Events represent state changes
    - ❑An event must **contain the data** that the aggregate needs to perform the **state transition.** The state of an aggregate consists of the values of the fields of the objects that comprise the aggregate
    - ❑ A state change might be as simple as changing the value of the field of an object, such as Order.state. Alternatively, a state change can involve adding or removing objects, such as revising an Order's line items.



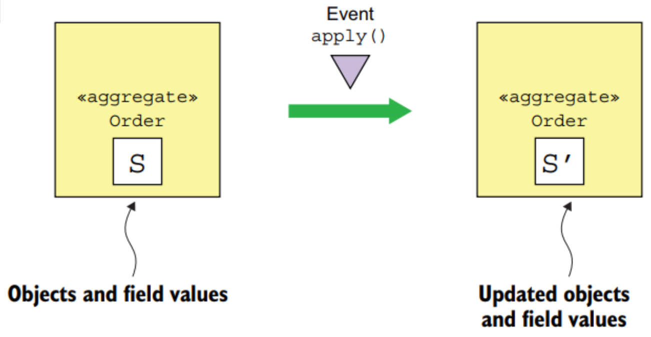


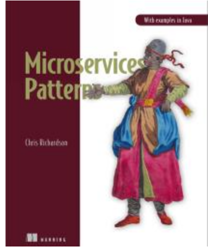
Developing business logic using event sourcing

❑Overview of event sourcing

❑Events represent state changes

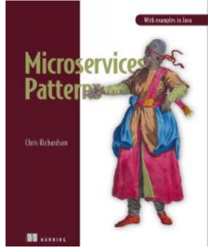
- ❑An event must **contain the data** that the aggregate needs to perform the **state transition.** The state of an aggregate consists of the values of the fields of the objects that comprise the aggregate
- ❑ A state change might be as simple as changing the value of the field of an object, such as Order.state. Alternatively, a state change can involve adding or removing objects, such as revising an Order's line items.





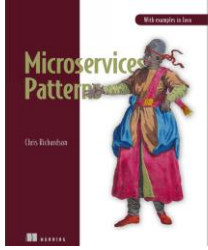
Developing business logic using event sourcing

- ❑Implementing an event store
  - ❑An application that uses event sourcing stores its events in an event store. An event store is a hybrid of a database and a message broker
  - ❑It behaves as a database because it has an API for inserting and retrieving an aggregate’s events by primary key
  - ❑It behaves as a message broker because it has an API for subscribing to events



Developing business logic using event sourcing

- ❑Implementing an event store
  - ❑There are a few different ways to implement an event store
    - ❑Implement own event store and event sourcing framework (RDBMS)
    - ❑Use a special-purpose event store
      - ❑Event Store—A .NET-based open source event store developed by Greg Young, an event sourcing pioneer
      - ❑Lagom—A microservices framework developed by Lightbend, the company formerly known as Typesafe
      - ❑Axon—An open source Java framework for developing event-driven applications that use event sourcing and CQRS
      - ❑Eventuate—http://eventuate.io. There are two versions of Eventuate: Eventuate SaaS, a cloud service, and Eventuate Local, an Apache Kafka/RDBMS-based open source project

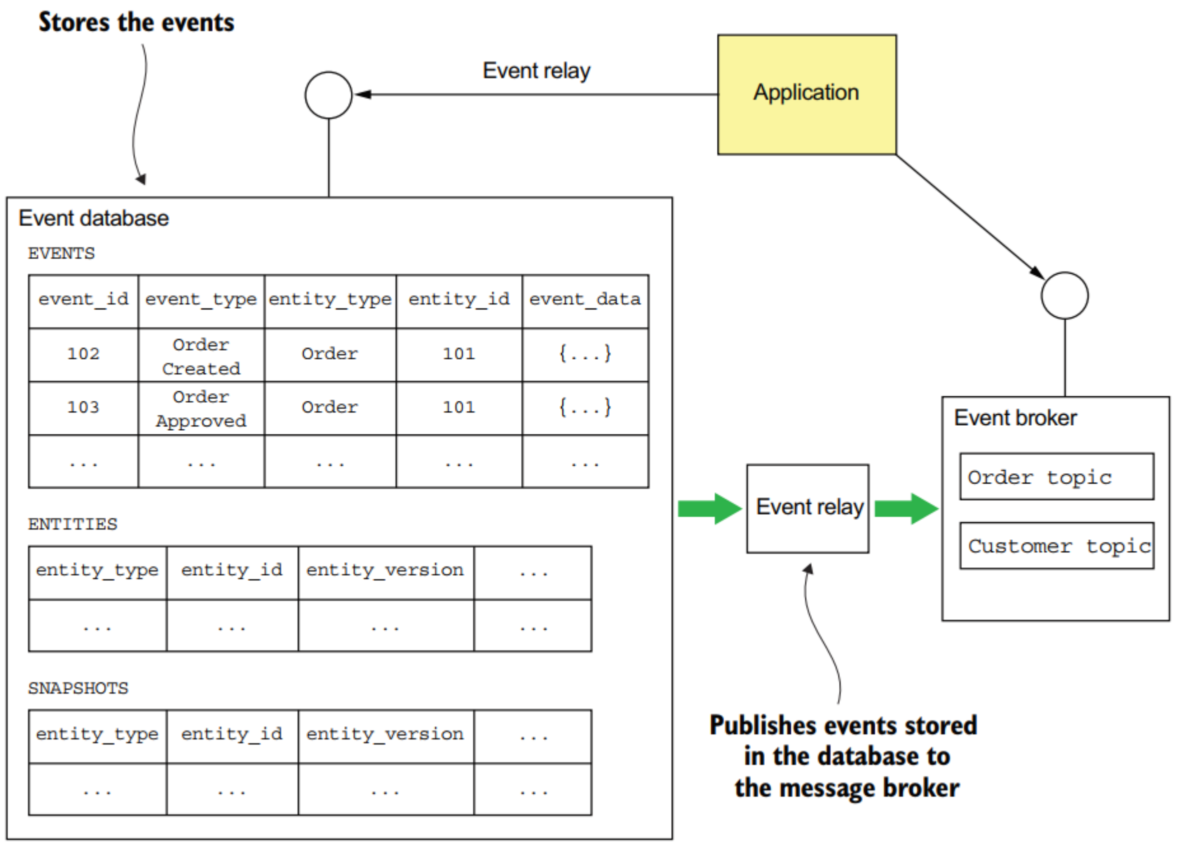


Developing business logic using event sourcing

❑Implementing an event store

❑How the Eventuate Local event store works

- ❑Events are stored in a database, such as MySQL
- ❑Applications insert and retrieve aggregate events by primary key
- ❑Applications consume events from a message broker, such as Apache Kafka
- ❑A transaction log tailing mechanism propagates events from the database to the message broker



EVENTS

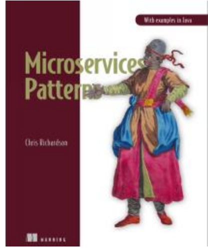
event_id	event_type	entity_type	entity_id	event_data
102	Order Created	Order	101	{...}
103	Order Approved	Order	101	{...}
...	...	...	...	...

ENTITIES

entity_type	entity_id	entity_version	...
...	...	...	...

SNAPSHOTS

entity_type	entity_id	entity_version	...
...	...	...	...



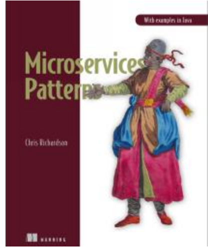
Developing business logic using event sourcing

- ❑Implementing an event store
  - ❑The schema of eventuate local's event database
    - ❑The event database consists of three tables:
      - ❑ Events—Stores the events
      - ❑ Entities—One row per entity
      - ❑ Snapshots—Stores snapshots

```
create table events (
  event_id varchar(1000) PRIMARY KEY,
  event_type varchar(1000),
  event_data varchar(1000) NOT NULL,
  entity_type VARCHAR(1000) NOT NULL,
  entity_id VARCHAR(1000) NOT NULL,
  triggering_event VARCHAR(1000)
);
```

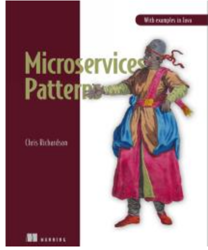
```
create table entities (
  entity_type VARCHAR(1000),
  entity_id VARCHAR(1000),
  entity_version VARCHAR(1000) NOT NULL,
  PRIMARY KEY(entity_type, entity_id)
);
```

```
create table snapshots (
  entity_type VARCHAR(1000),
  entity_id VARCHAR(1000),
  entity_version VARCHAR(1000),
  snapshot_type VARCHAR(1000) NOT NULL,
  snapshot_json VARCHAR(1000) NOT NULL,
  triggering_events VARCHAR(1000),
  PRIMARY KEY(entity_type, entity_id, entity_version)
)
```



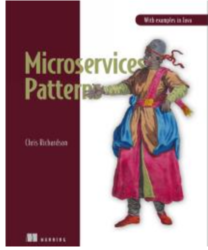
Developing business logic using event sourcing

- ❑Implementing an event store
  - ❑Consuming events by subscribing to eventuate local's event broker
    - ❑Services consume events by subscribing to the event broker, which is implemented using Apache Kafka
    - ❑The event broker has a topic for each aggregate type
    - ❑To consume an aggregate's events, a service subscribes to the aggregate's topic
  - ❑The eventuate local event relay propagates events from the database to the message broker
    - ❑ The event relay propagates events inserted into the event database to the event broker
    - ❑ It uses transaction log tailing whenever possible and polling for other databases



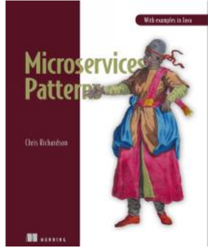
Developing business logic using event sourcing

- ❑Implementing an event store
  - ❑The Eventuate client framework for Java
    - ❑The Eventuate client framework enables developers to write event sourcing-based applications that use the Eventuate Local event store
    - ❑The framework provides the foundation for developing event sourcing-based aggregates, services, and event handlers
    - ❑The framework provides base classes for aggregates, commands, and events. There's also an AggregateRepository class that provides CRUD functionality. And the framework has an API for subscribing to events



Summary

- The procedural **Transaction script pattern** is often a good way to implement simple business logic. But when implementing complex business logic you should consider using the object-oriented Domain model pattern
- A good way to organize a service's business logic is as a collection of **DDD aggregates**. DDD aggregates are useful because they modularize the domain model, eliminate the possibility of object reference between services, and ensure that each ACID transaction is within a service
- **Event sourcing** persists an aggregate as a sequence of events. Each event represents either the creation of the aggregate or a state change.
- Events are stored in an **event store**, a **hybrid of a database and a message broker**. When a service saves an event in an event store, it delivers the event to subscribers.
- **Eventuate Local** is an open source event store based on MySQL and Apache Kafka. Developers use the Eventuate client framework to write aggregates and event handlers



Reference

- https://github.com/eventuate-tram/eventuate-tram-examples-java-spring-todo-list