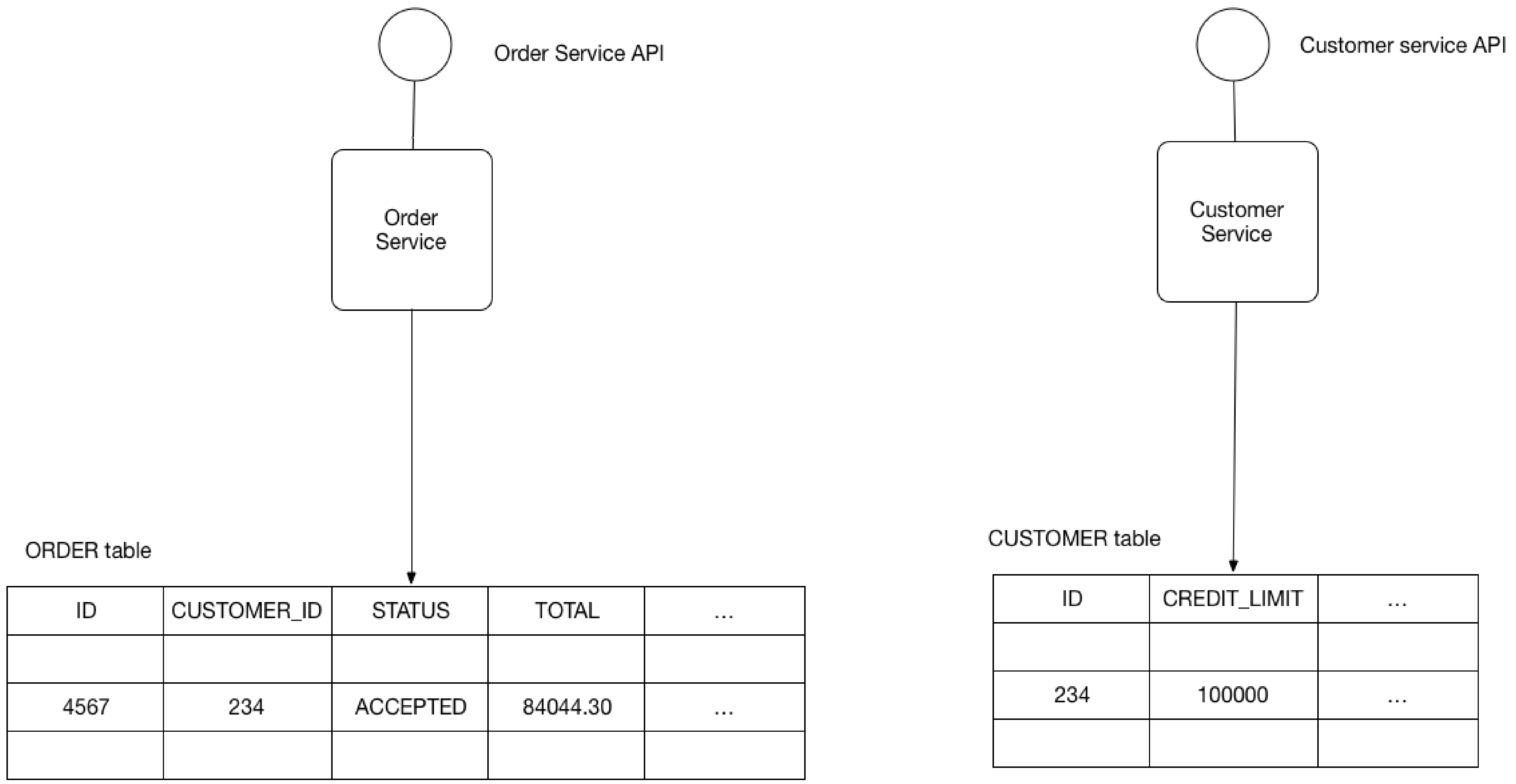
Pattern: Database per service

Context

Let’s imagine you are developing an online store application using the Microservice architecture pattern.

Most services need to persist data in some kind of database.

For example, the Order Service stores information about orders and the Customer Service stores information about customers.



ORDER table

ID	CUSTOMER_ID	STATUS	TOTAL	...
4567	234	ACCEPTED	84044.30	...

CUSTOMER table

ID	CREDIT_LIMIT	...
234	100000	...

Problem

What’s the database architecture in a microservices application?

Forces

Services must be loosely coupled so that they can be developed, deployed and scaled independently

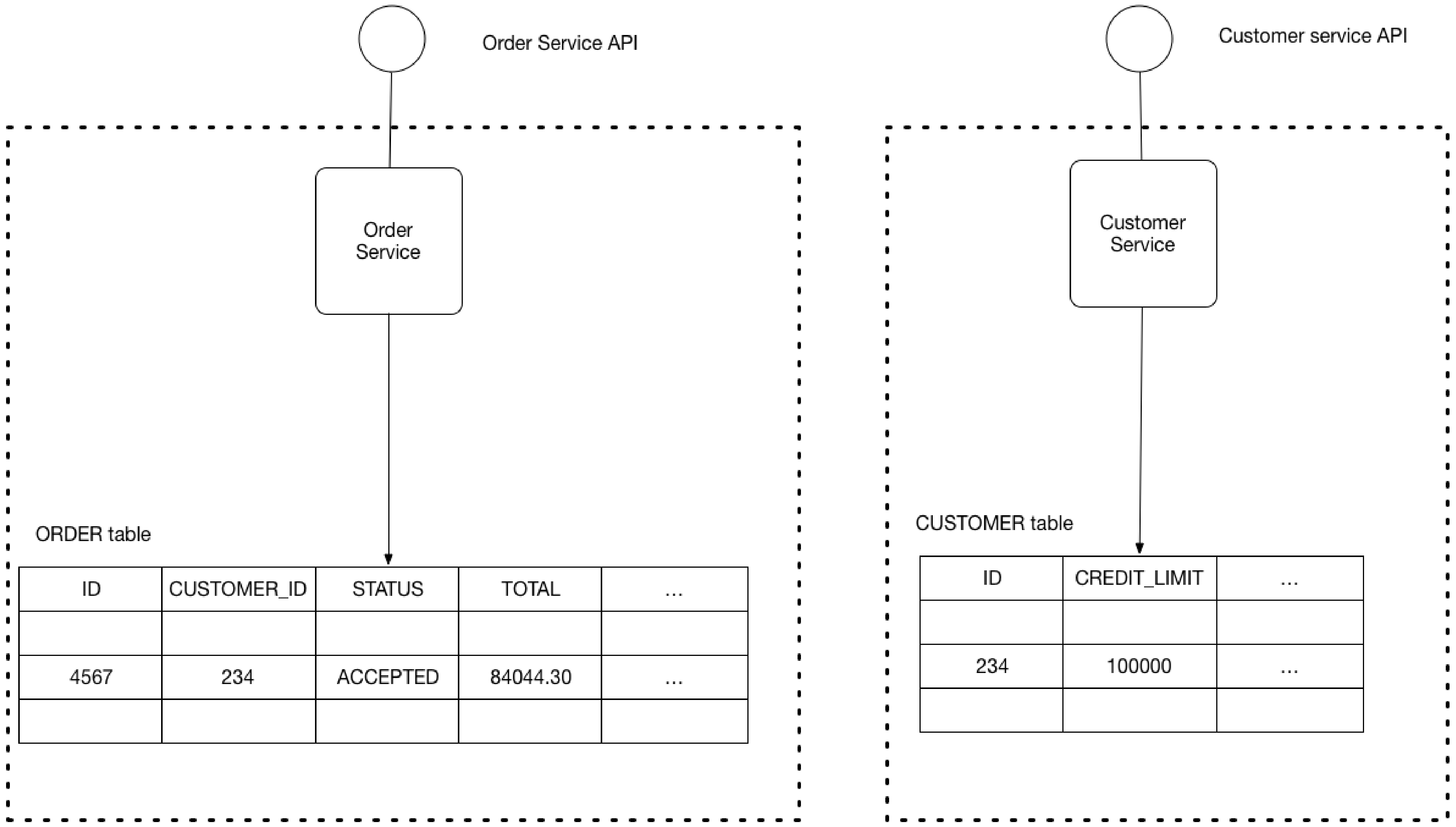
Some business transactions must enforce invariants that span multiple services. For example, the Place Order use case must verify that a new Order will not exceed the customer’s credit limit. Other business transactions, must update data owned by multiple services.

Some business transactions need to query data that is owned by multiple services. For example, the View Available Credit use case must query the Customer to find the creditLimit and Orders to calculate the total amount of the open orders.

Some queries must join data that is owned by multiple services. For example, finding customers in a particular region and their recent orders requires a join between customers and orders.

Different services have different data storage requirements. For some services, a relational database is the best choice. Other services might need a NoSQL database such as MongoDB, which is good at storing complex, unstructured data, or Neo4J, which is designed to efficiently store and query graph data.

Solution : Keep each microservice’s persistent data private to that service and accessible only via its API.



ORDER table

ID	CUSTOMER_ID	STATUS	TOTAL	…
4567	234	ACCEPTED	84044.30	…

CUSTOMER table

ID	CREDIT_LIMIT	…
234	100000	…

There are a few different ways to keep a service’s persistent data private. You do not need to provision a database server for each service. For example, if you are using a relational database then the options are:

Private-tables-per-service – each service owns a set of tables that must only be accessed by that service

Schema-per-service – each service has a database schema that’s private to that service

Database-server-per-service – each service has it’s own database server.

Resulting context

Using a database per service has the following benefits:

Helps ensure that the services are loosely coupled. Changes to one service’s database does not impact any other services.

Each service can use the type of database that is best suited to its needs. For example, a service that does text searches could use ElasticSearch. A service that manipulates a social graph could use Neo4j.

Using a database per service has the following drawbacks:

Implementing business transactions that span multiple services is not straightforward. Distributed transactions are best avoide. Moreover, many modern (NoSQL) databases don’t support them.

The best solution is to use the Saga pattern. Services publish events when they update data. Other services subscribe to events and update their data in response.

Implementing queries that join data that is now in multiple databases is challenging.

There are various solutions:

API Composition - the application performs the join rather than the database. For example, a service (or the API gateway) could retrieve a customer and their orders by first retrieving the customer from the customer service and then querying the order service to return the customer’s most recent orders.

Command Query Responsibility Segregation (CQRS) - maintain one or more materialized views that contain data from multiple services. The views are kept by services that subscribe to events that each services publishes when it updates its data.

For example, the online store could implement a query that finds customers in a particular region and their recent orders by maintaining a view that joins customers and orders. The view is updated by a service that subscribes to customer and order events.

Pattern: Saga

Context

You have applied the Database per Service pattern.

Each service has its own database. Some business transactions, however, span multiple service so you need a mechanism to ensure data consistency across services.

For example, lets imagine that you are building an e-commerce store where customers have a credit limit. The application must ensure that a new order will not exceed the customer’s credit limit.

Since Orders and Customers are in different databases the application cannot simply use a local ACID transaction.

Problem

How to maintain data consistency across services?

Solution

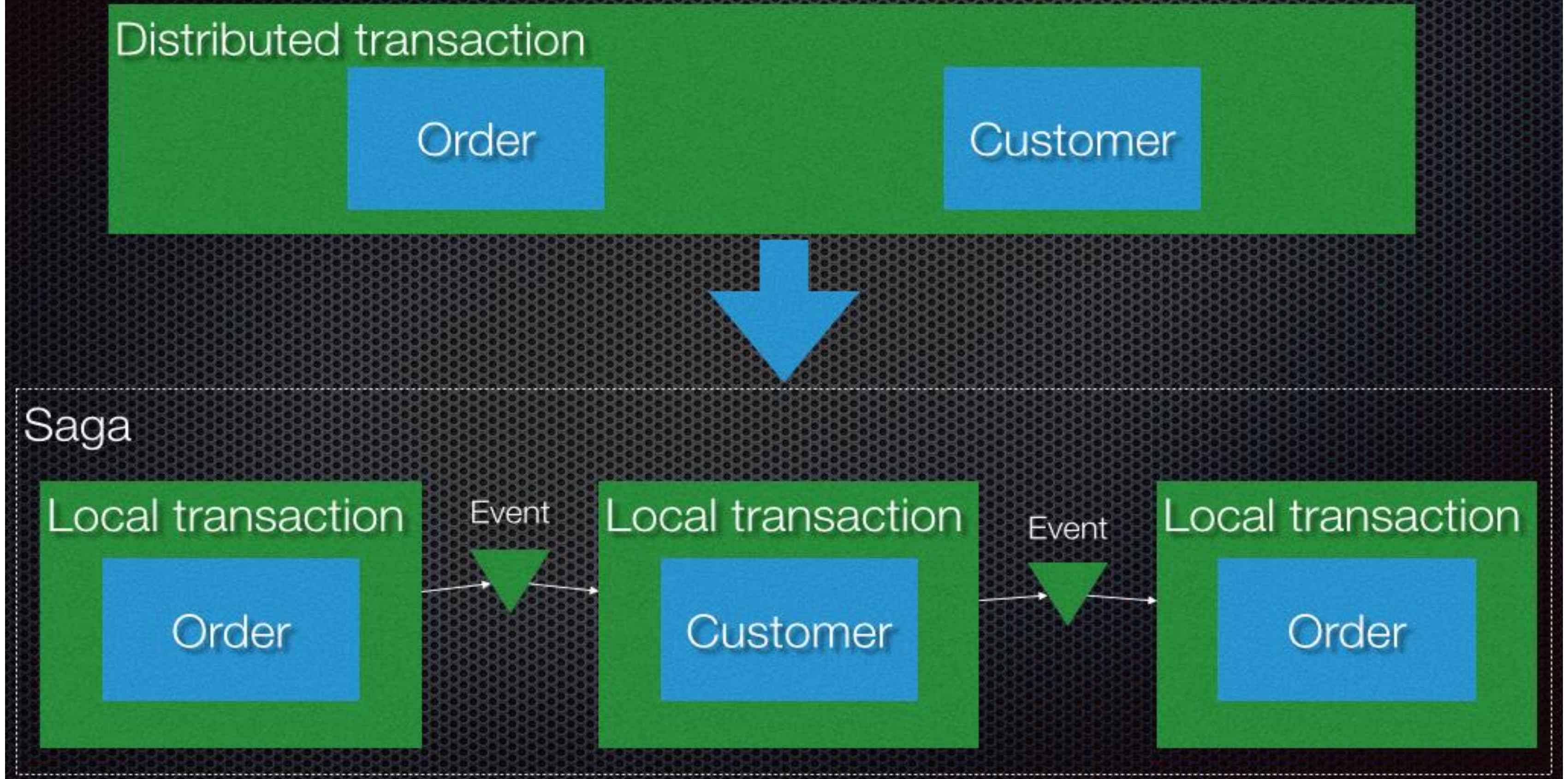
Implement each business transaction that spans multiple services as a saga.

A saga is a sequence of local transactions.

Each local transaction updates the database and publishes a message or event to trigger the next local transaction in the saga.

If a local transaction fails because it violates a business rule then the saga executes a series of compensating transactions that undo the changes that were made by the preceding local transactions.

Using Sagas instead of 2PC



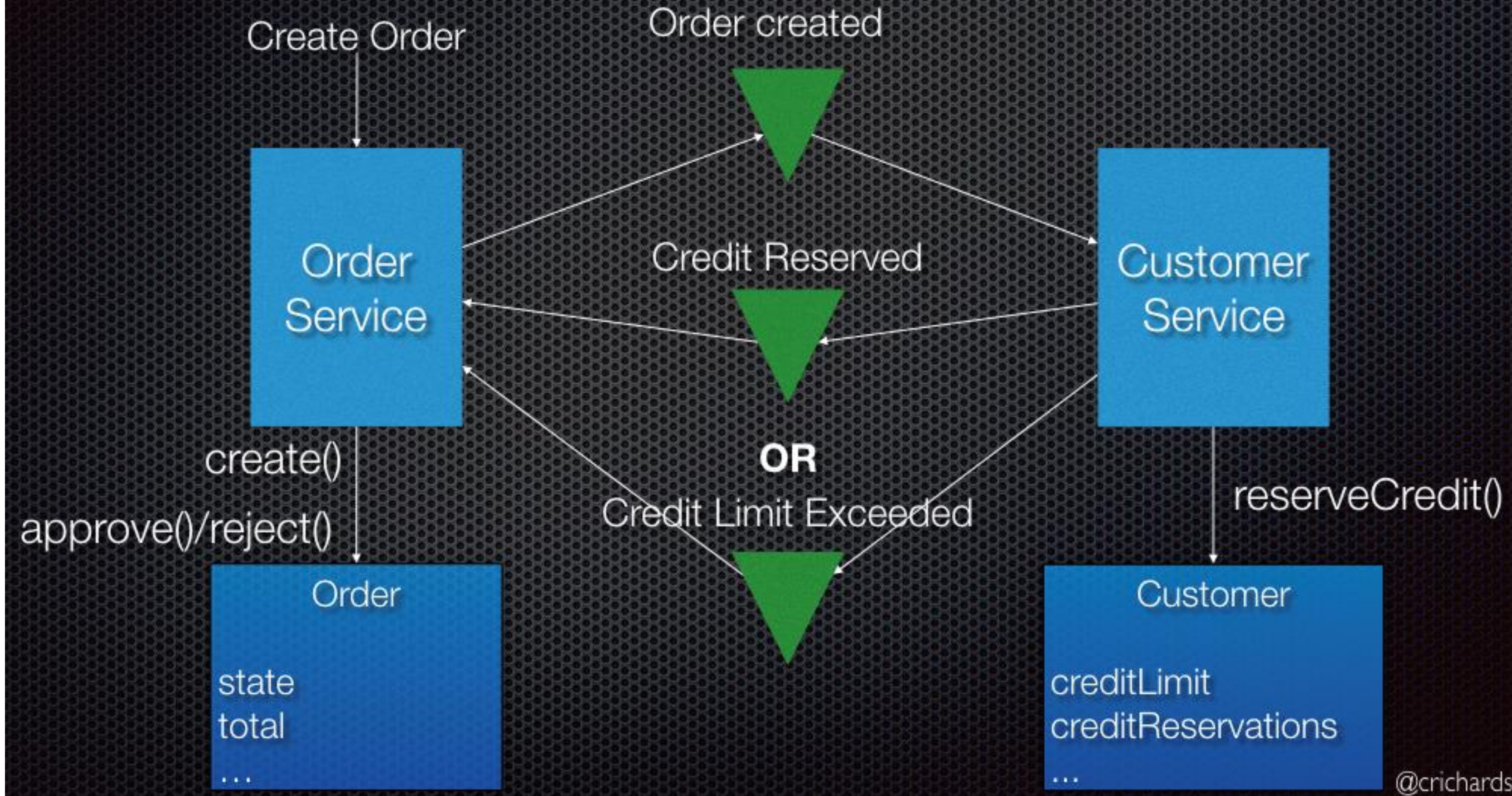
@crichardson

There are two ways of coordination sagas:

Choreography - each local transaction publishes domain events that trigger local transactions in other services (Event Sourcing)

Orchestration - an orchestrator (object) tells the participants what local transactions to execute

Option #1: Choreography-based coordination using events



@crichardson

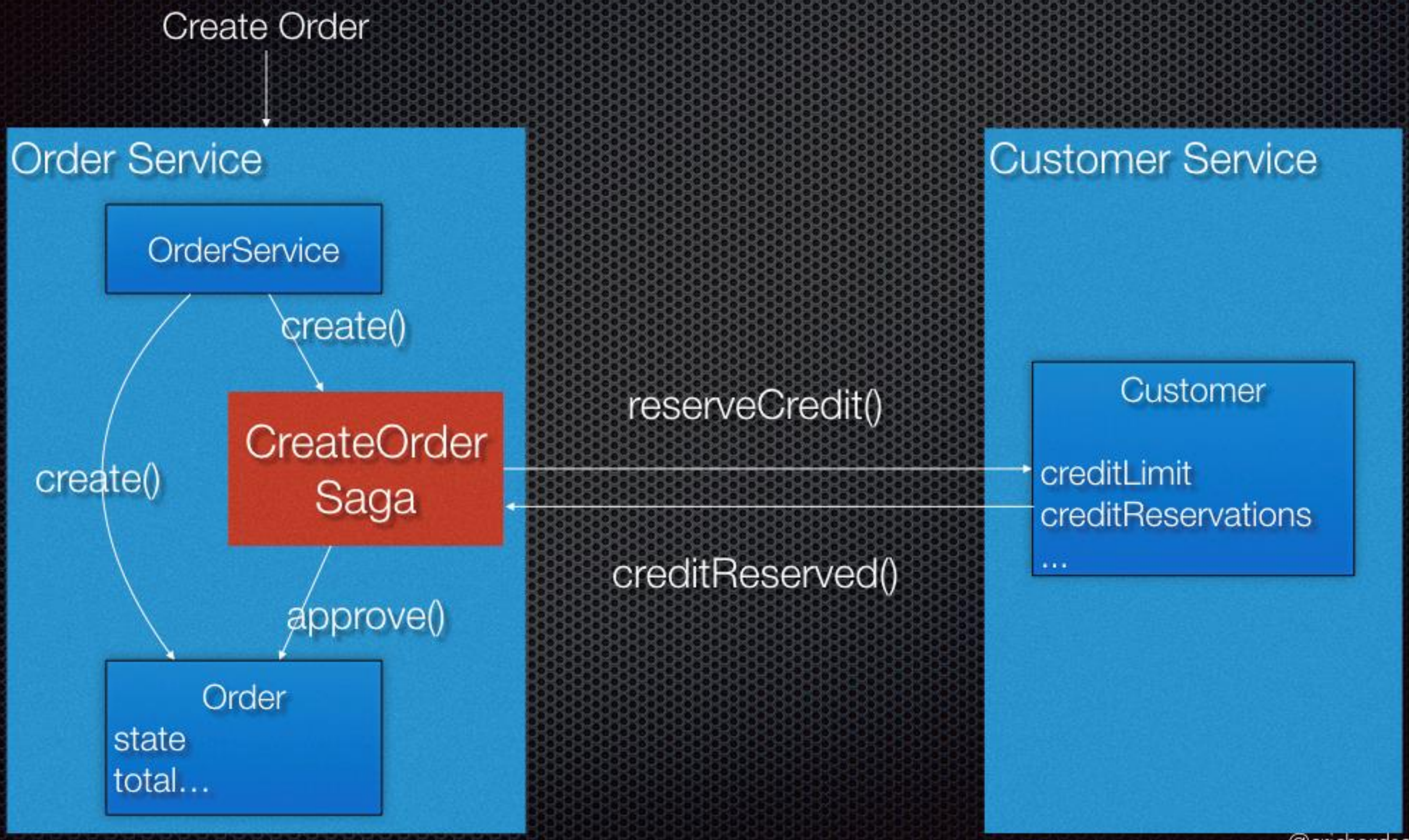
An e-commerce application that uses this approach would create an order using a choreography-based saga that consists of the following steps:

The Order Service creates an Order in a pending state and publishes an OrderCreated event

The Customer Service receives the event attempts to reserve credit for that Order. It publishes either a Credit Reserved event or a CreditLimitExceeded event.

The Order Service receives the event and changes the state of the order to either approved or cancelled

CreateOrderSaga orchestrator



@crichardson

An e-commerce application that uses this approach would create an order using an orchestration-based saga that consists of the following steps:

The Order Service creates an Order in a pending state and creates a CreateOrderSaga

The CreateOrderSaga sends a ReserveCredit command to the Customer Service

The Customer Service attempts to reserve credit for that Order and sends back a reply

The CreateOrderSaga receives the reply and sends either an ApproveOrder or RejectOrder command to the Order Service

The Order Service changes the state of the order to either approved or cancelled

Resulting context

This pattern has the following benefits:

It enables an application to maintain data consistency across multiple services without using distributed transactions

This solution has the following drawbacks:

The programming model is more complex. For example, a developer must design compensating transactions that explicitly undo changes made earlier in a saga.