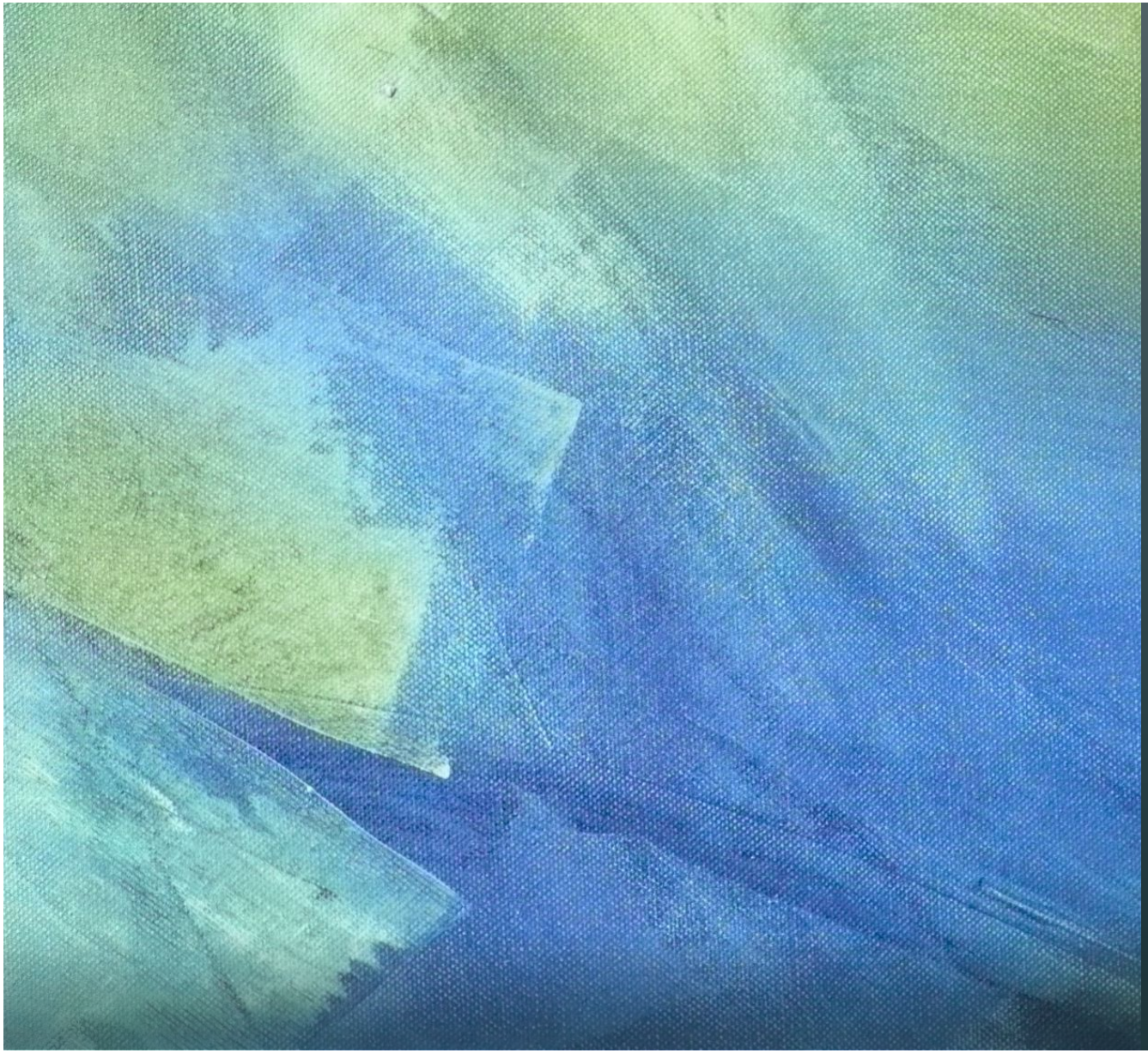


Introduction to Event-driven Architecture

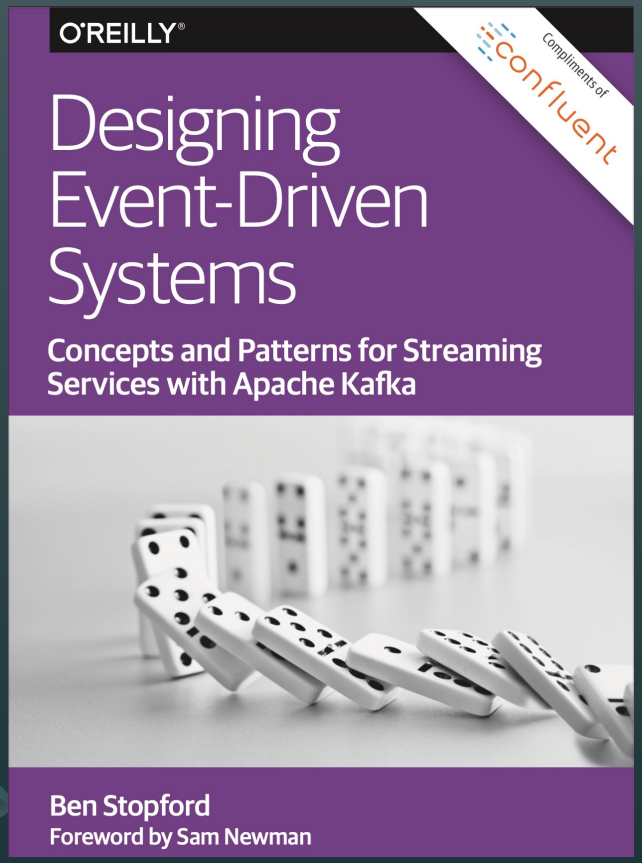
Wiwat Vatanawood

Duangdao Wichadakul

Nuengwong Tuaycharoen

Pittipol Kantavat

References



- Book
  - Ben Stopford, Designing Event-Driven Systems, O'Reilly , 2018

Introduction (1)

- Service-based architectures, like microservices, are commonly built with synchronous request-response protocols.
- As the number of services grows gradually, the web of synchronous interactions grows with them.
- Previously benign availability issues start to trigger far more widespread outages.

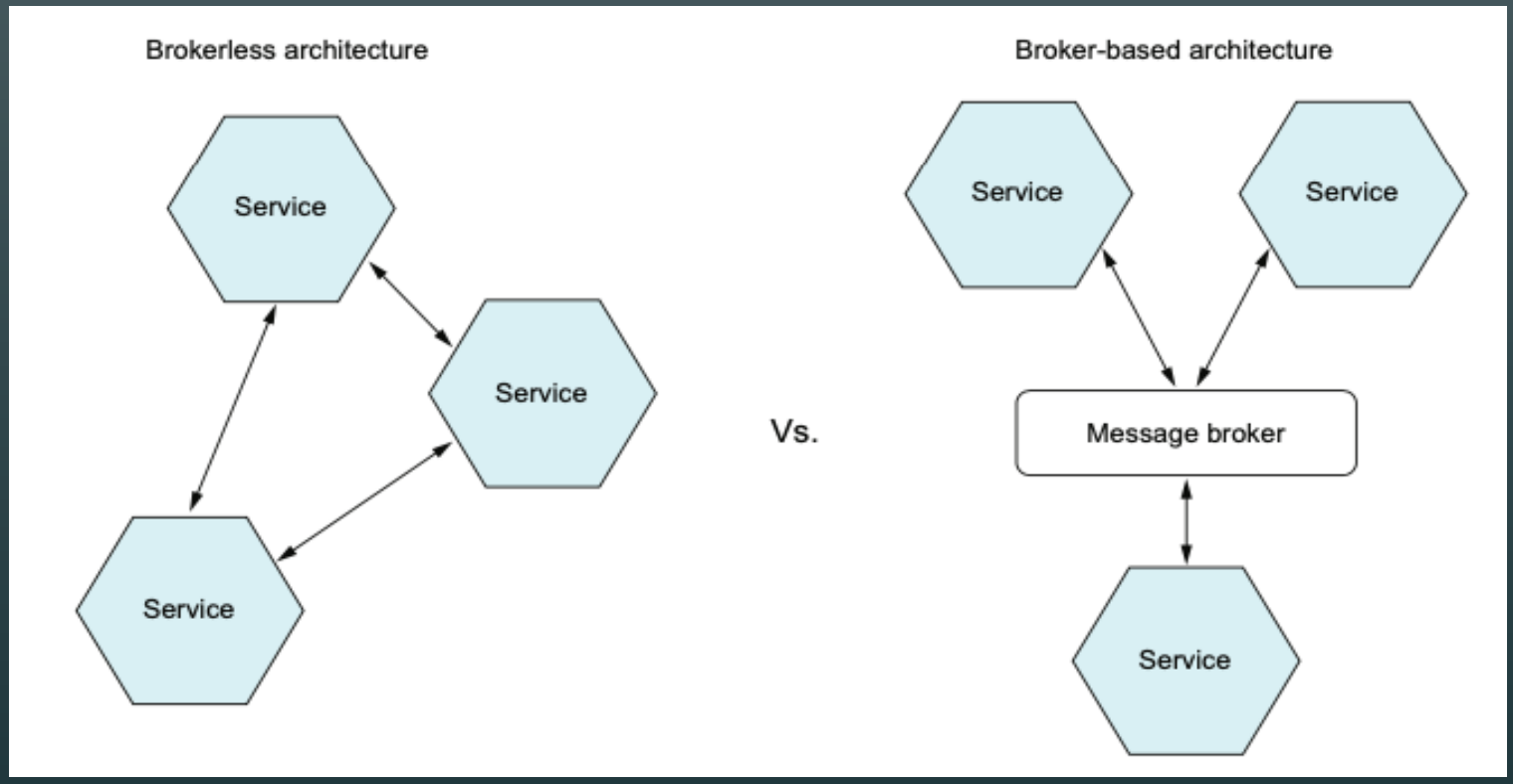
Introduction (2)

- Splitting software into asynchronous flows allows us to compartmentalize the different problems we need to solve and embrace a world that is itself inherently asynchronous.
- In this chapter, we focus on composing services not through chains of commands and queries but rather through streams of events.

Synchronous vs Asynchronous communication

- Synchronous:
  - The client expects a timely response from the service and might even block while it waits. (for example, REST)
- Asynchronous:
  - A service client makes a request to a service by sending a message
  - The client doesn't block waiting for a reply.
  - Instead, the client is written assuming that the reply won't be received immediately.

Brokerless vs Broker-based architecture (Asynchronous)



Commands, Events, and Queries

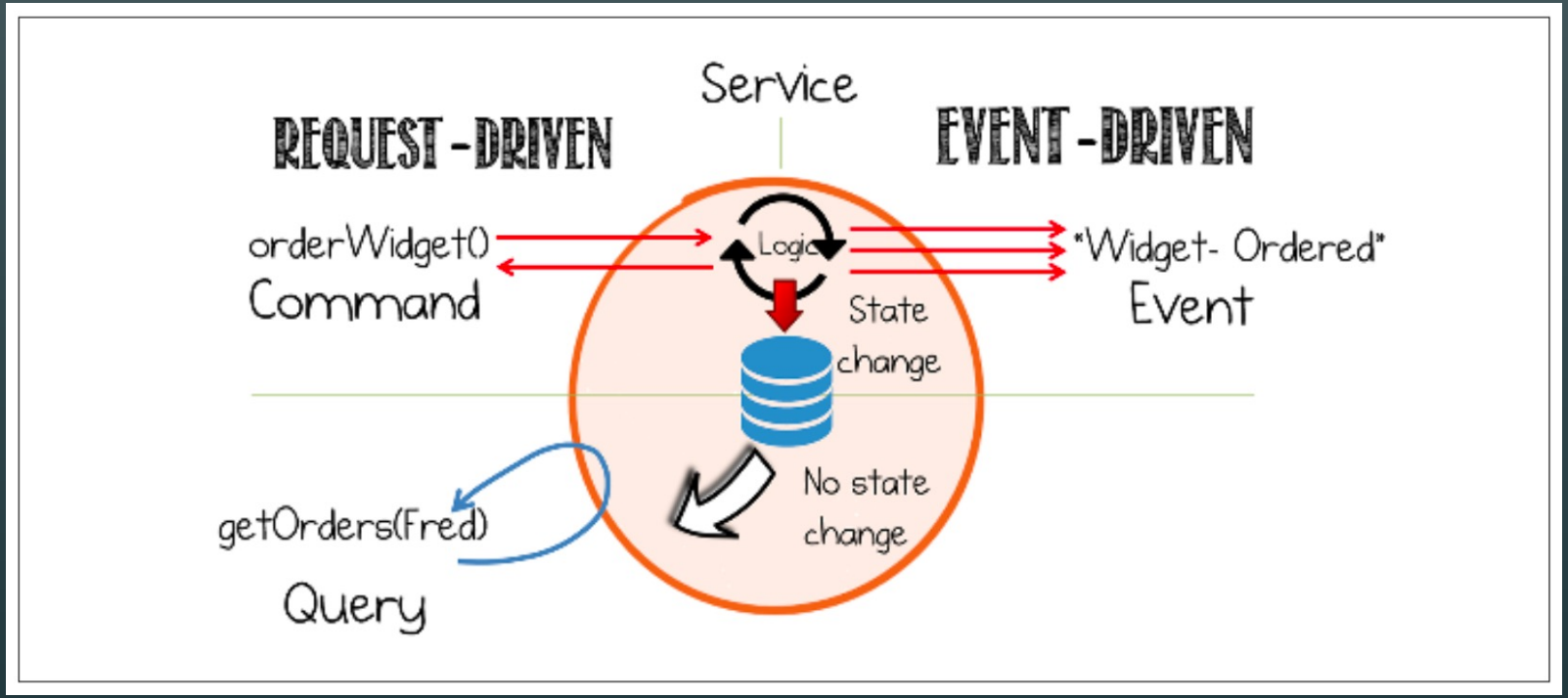
- Three distinct ways that programs can interact over a network.
  - **Commands** are actions—requests for some operation to be performed by another service, something that will change the state of the system.
  - **Events** are both a fact and a notification. They represent something that happened in the real world but include no expectation of any future action. They travel in only one direction and expect no response.
  - **Queries** are a request to look something up. Unlike events or commands, queries are free of side effects; they leave the state of the system unchanged.

Differences between commands, events, and queries

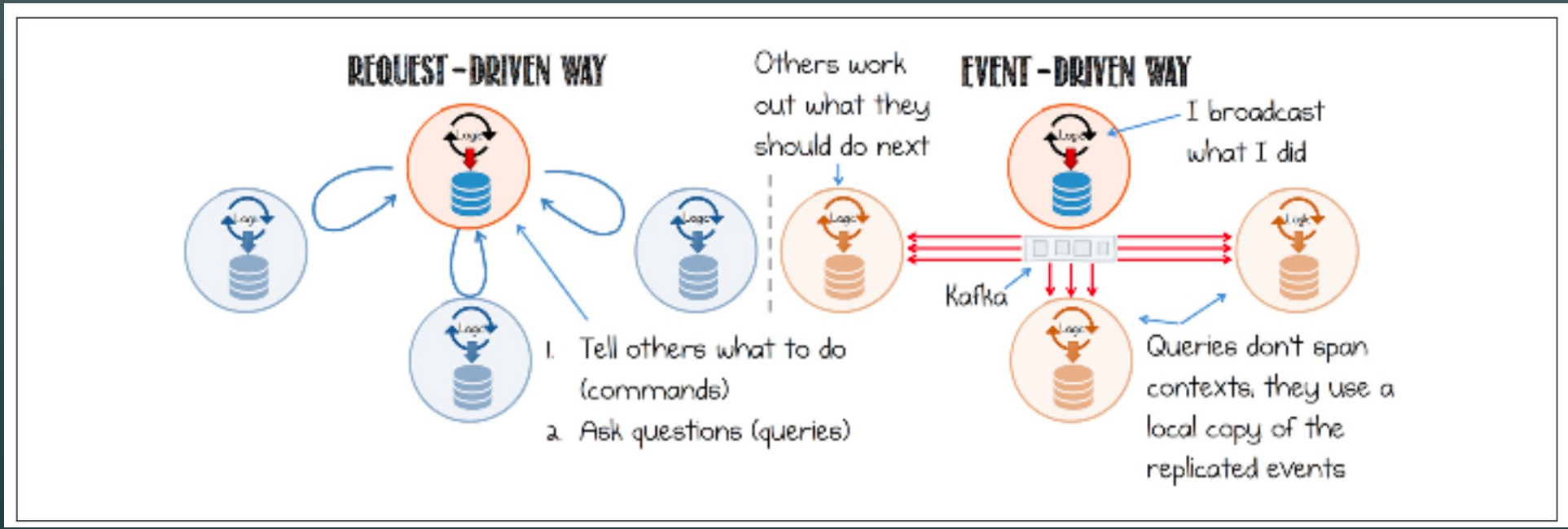
	Behavior/state change	Includes a response
Command	Requested to happen	Maybe
Event	Just happened	Never
Query	None	Always

- The beauty of events is they wear two hats:
  - A notification hat that triggers services into action.
  - A replication hat that copies data from one service to another.
- From a services perspective, events lead to less coupling than commands and queries.

A visual summary of commands, events, and queries

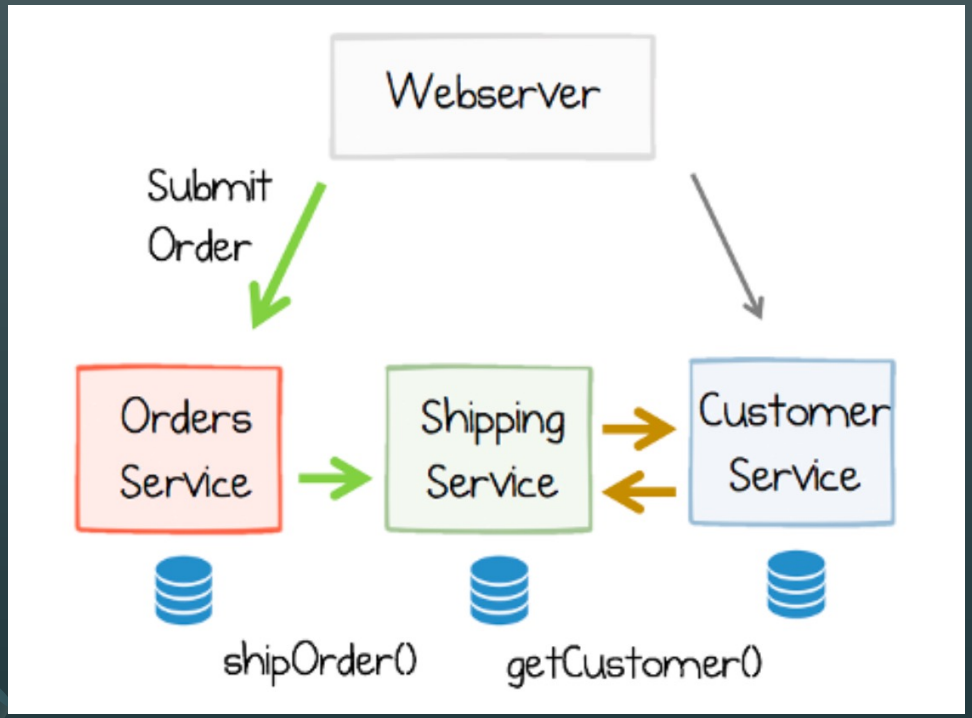


Request-response vs Event-driven approach



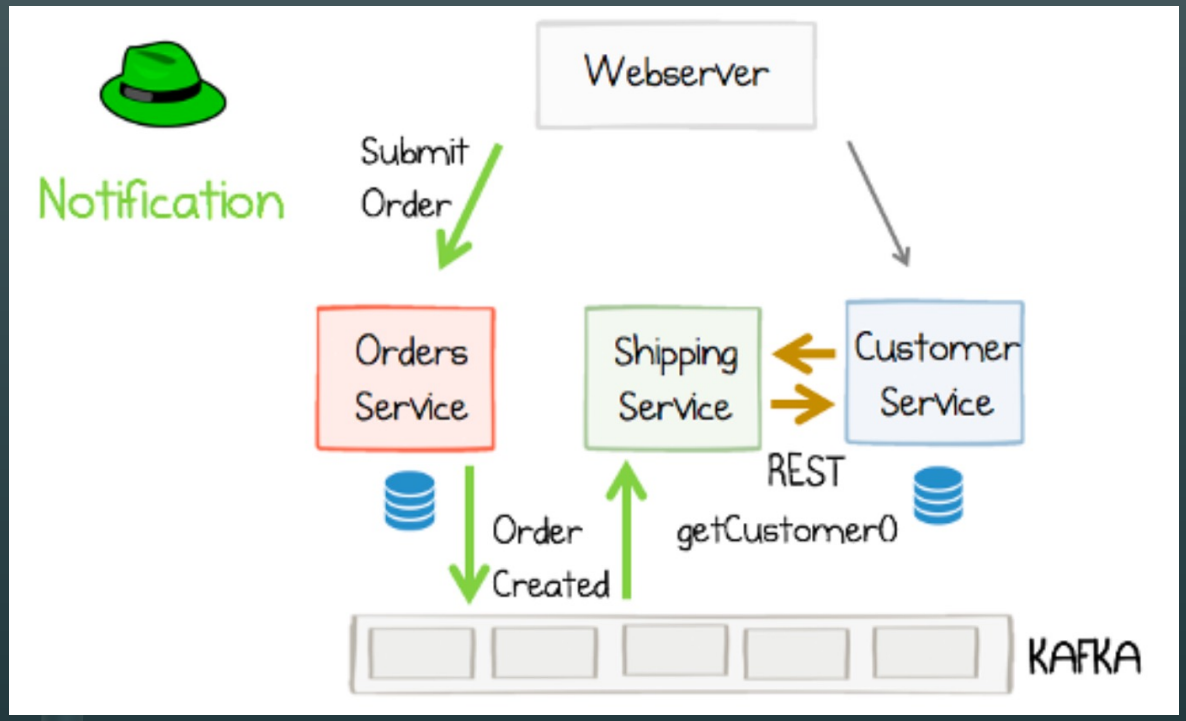
Comparison between the request-response and event-driven approaches demonstrating how event-driven approaches provide less coupling

A request-driven order management system



- The user clicks Buy, and an order is sent to the orders service.
- Three things then happen
  - The shipping service is notified.
  - It looks up the address to send the iPad to.
  - It starts the shipping process.

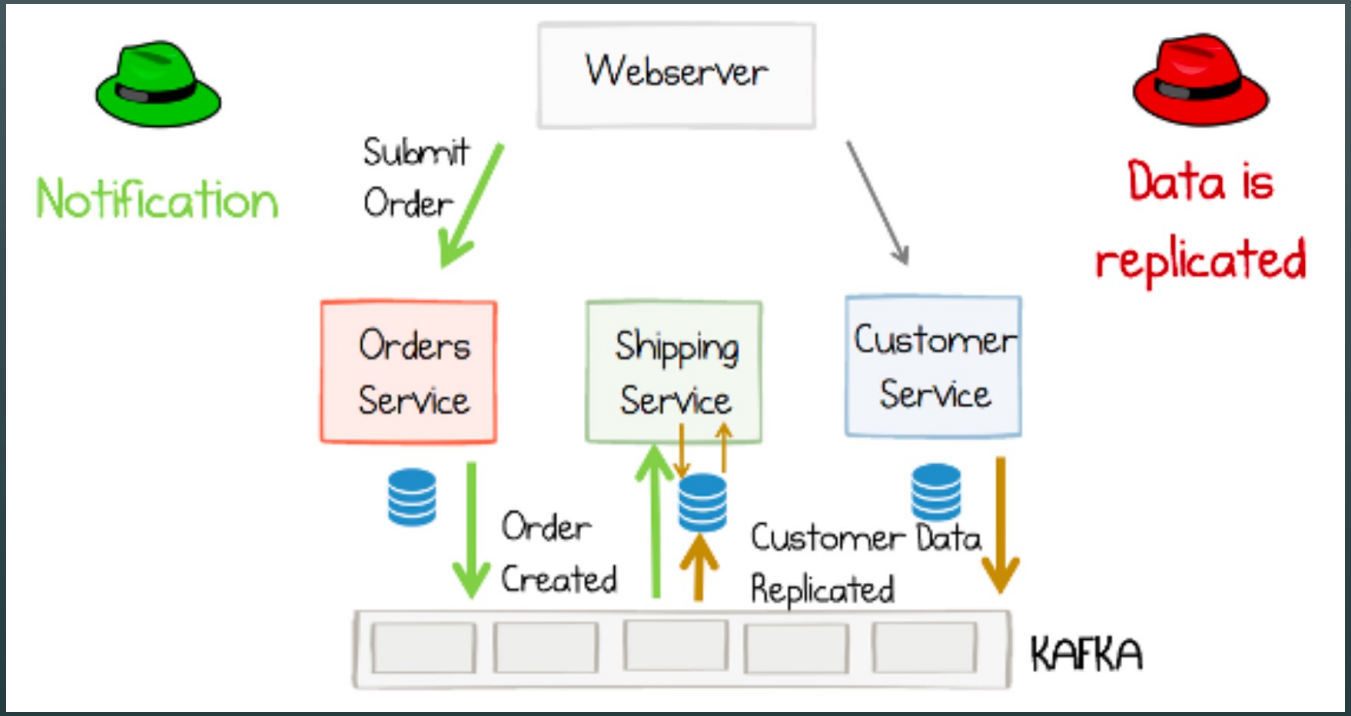
An event-driven order management system (1)



- Using Events for Notification
- The orders service and the shipping service communicate via events rather than calling each other directly.
- The orders service has no knowledge that the shipping service exists.

*In this configuration the events are used only as a means of notification: the orders service notifies the shipping service via Kafka

An event-driven order management system (2)



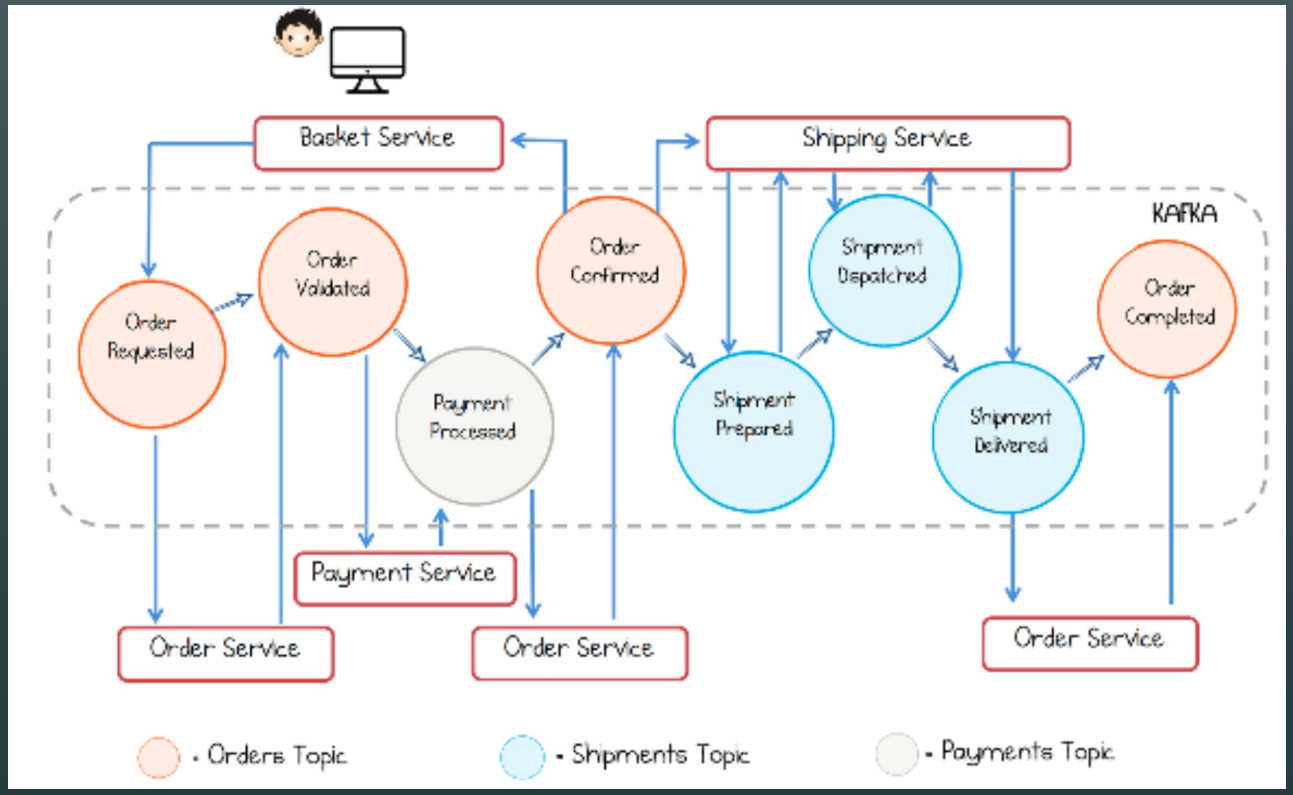
- Using Events to Provide State Transfer
- The event stream replicate customer data from the customer service to the shipping service, where it can be queried locally

*In this configuration, the events are used for notification (the orders service notifies the shipping service) as well as for data replication (data is replicated from the customer service to the shipping service, where it can be queried locally).

The Event Collaboration Pattern

- To build fine-grained services using events, a pattern called Event Collaboration is often used.
  - A set of services collaborate around a single business workflow.
  - Each service listens to events, then creates new ones.
- No single service owns the whole process;
- Instead, each service owns a small part—some subset of state transitions—and these plug together through a chain of events.

An example workflow implemented with Event Collaboration



Each circle represents an event. The color of the circle designates the topic it is in. A workflow evolves from Order Requested through to Order Completed.

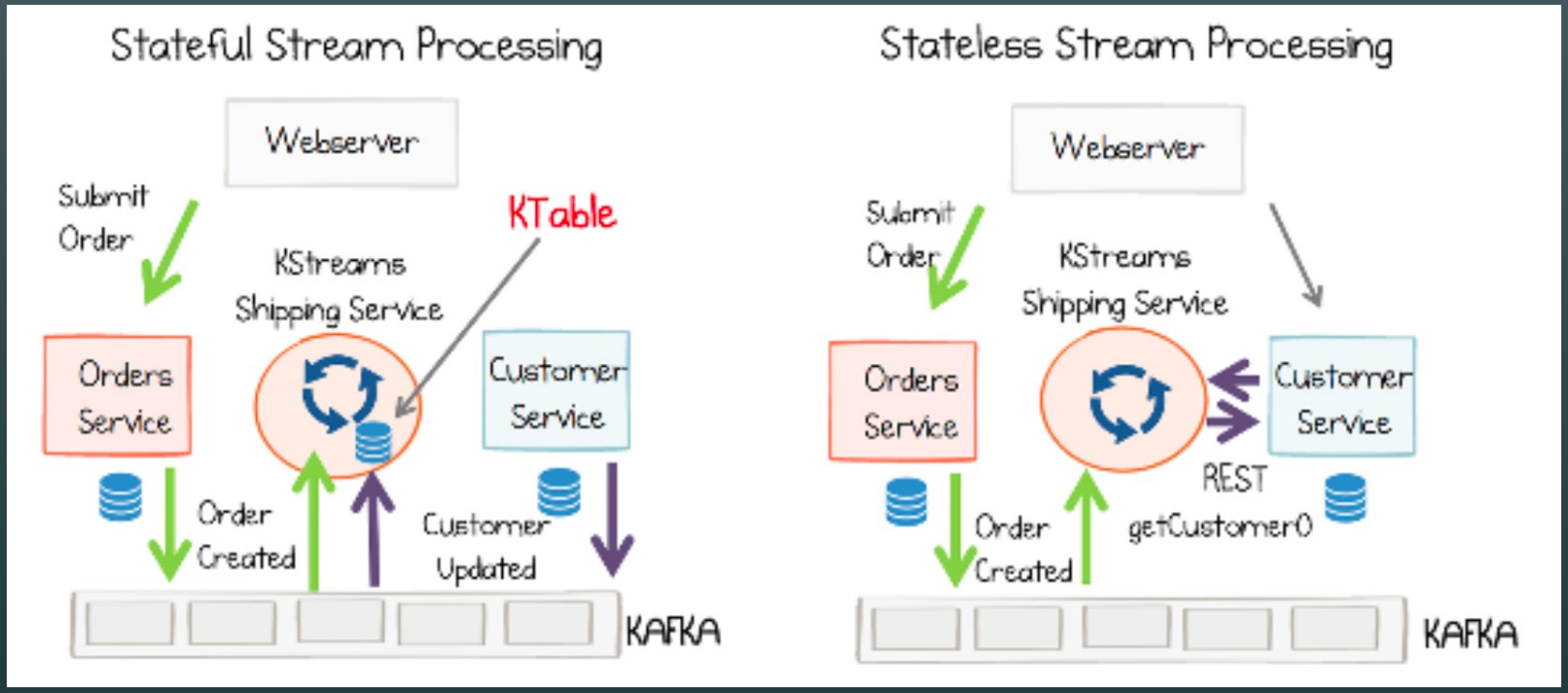
Choreographies vs Orchestration (1)

- Choreographies:
  - Each service handles some subset of state transitions, which, when put together, describe the whole business process.
- Orchestration:
  - A single process commands and controls the whole workflow from one place—for example, a process manager that is implemented with request-response.

Choreographies vs Orchestration (2)

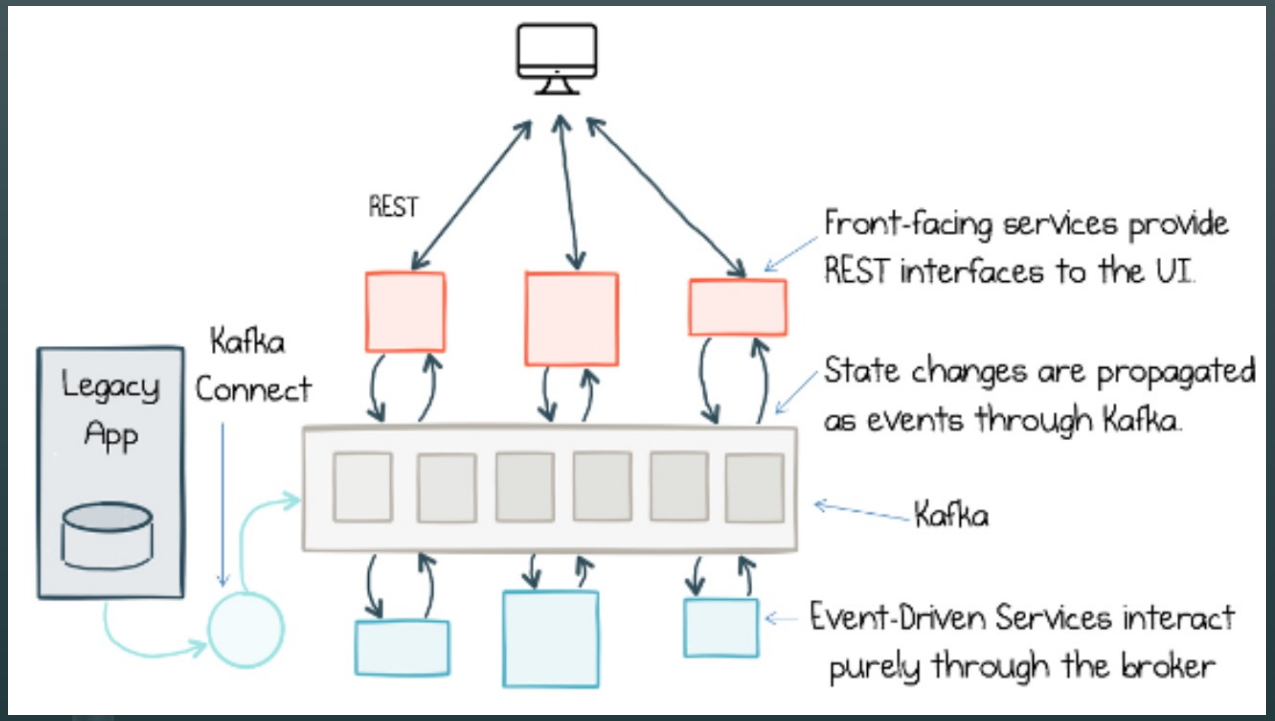
- Choreographed systems have the advantage of pluggable.
  - You can change the way you work, and no other services need to know or care about it.
- By contrast, in an orchestrated system, where a single service dictates the workflow, all changes need to be made in the controller.
  - The advantage of orchestration is that the workflow is written down in one place, making it easy to reason about the system.
  - The downside is that the model is tightly coupled to the controller.

Stateful stream vs Stateless stream processing



Stateful stream processing is like using events for both notification and state transfer (left), while stateless stream processing is like using events for notification (right)

Mixing Request- and Event-Driven Protocols



- Data is imported from a legacy application via the Connect API.
- User-facing services provide REST APIs to the UI.
- State changes are journaled to Kafka as events.
- At the bottom, business processing is performed via Event Collaboration