1

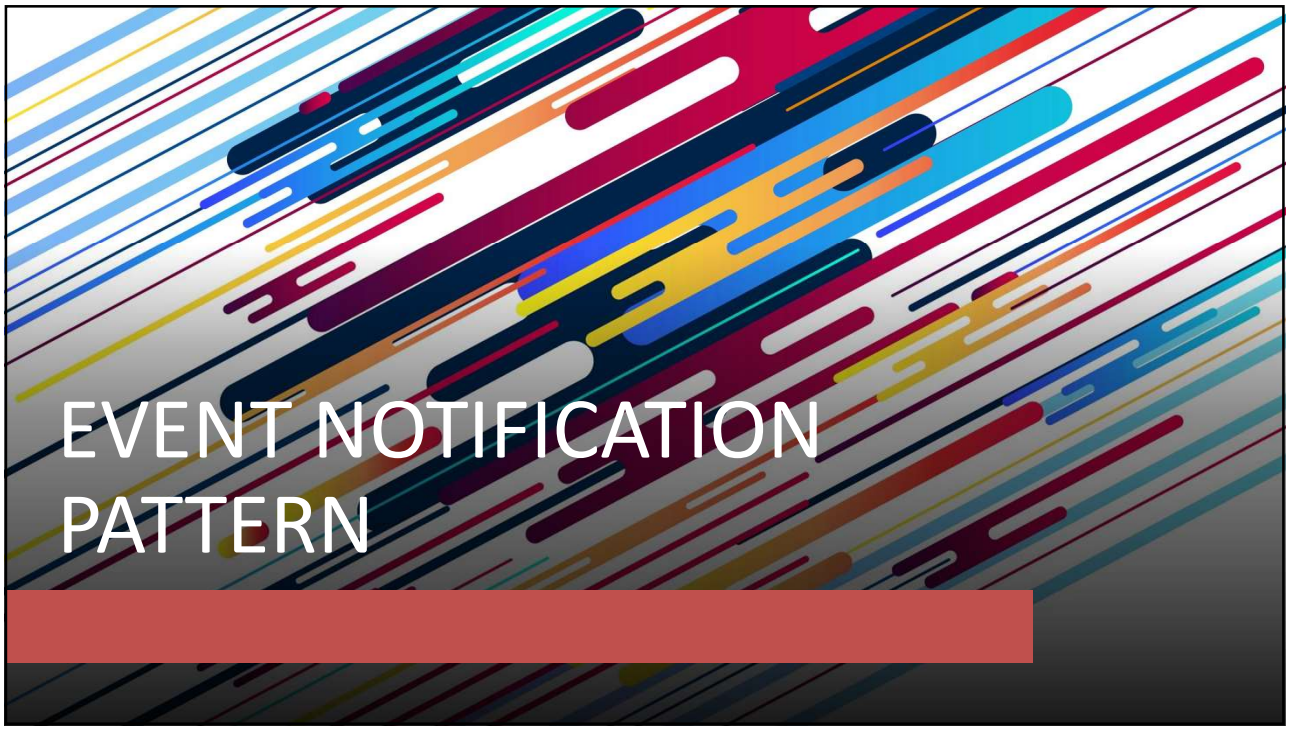
Content

- Event Patterns
  - Event Notification
  - Event-Carried State Transfer
  - Event Sourcing
  - CQRS

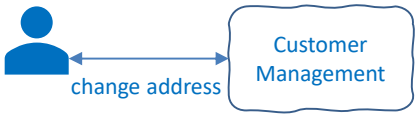
2

Collaboration/Integration

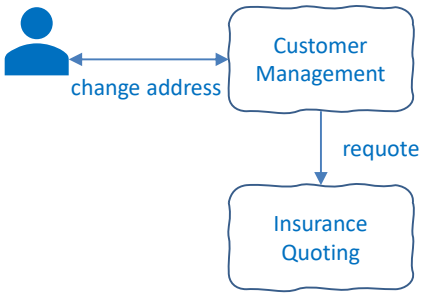
- Distributed Architectures
  - Examples: MSA or SOA
- Collaboration
  - Synchronous
    - Request/response
  - Asynchronous
    - Fire-and-forget
    - Multicast
    - Push notification
    - Publish/Subscribe



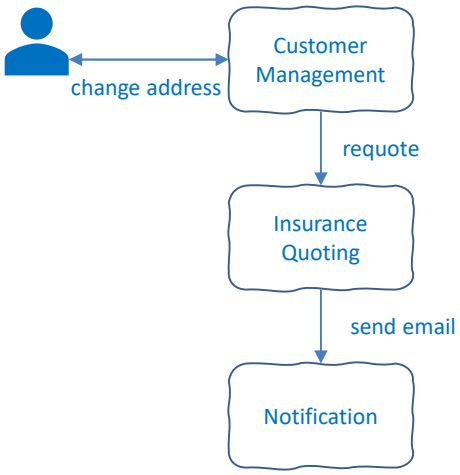
Collaboration Patterns: Commands/Events



Collaboration Patterns: Commands/Events

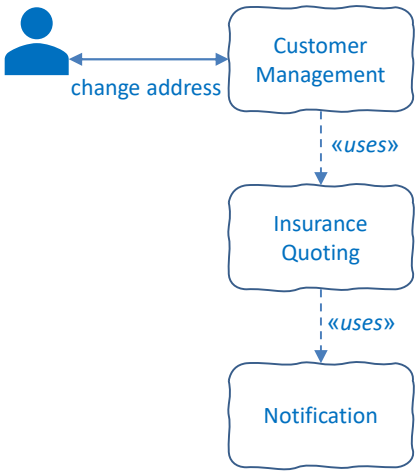


Collaboration Patterns: Commands/Events



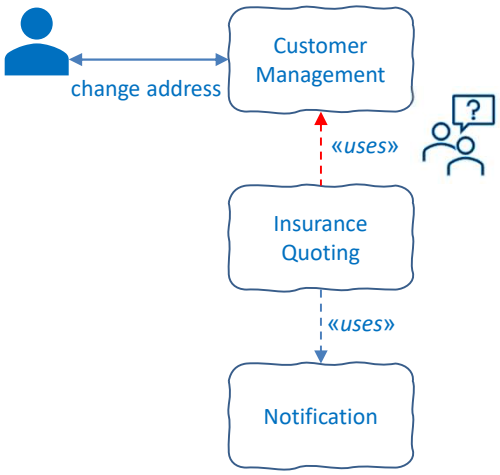
7

Collaboration Patterns: Commands/Events



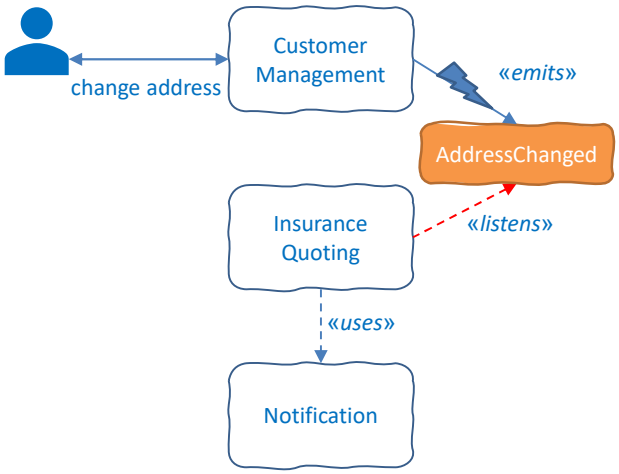
8

Collaboration Patterns: Commands/Events



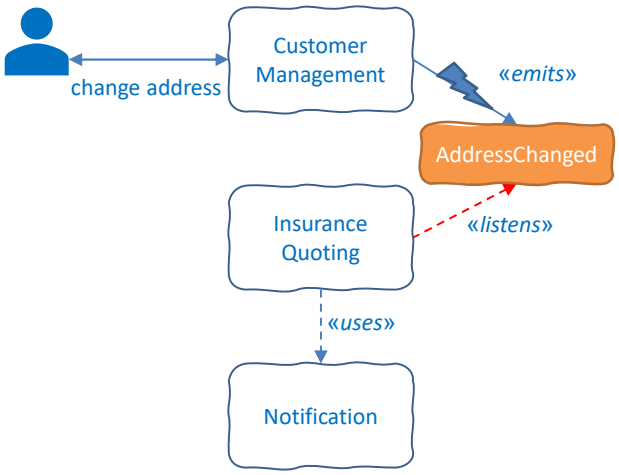
9

Collaboration Patterns: Commands/Events



10

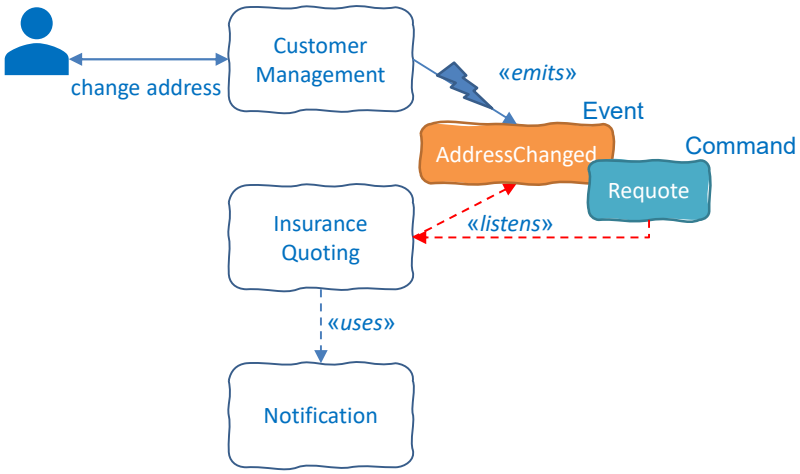
Collaboration Patterns: Commands/Events



Event Notification Pattern

11

Collaboration Patterns: Commands/Events

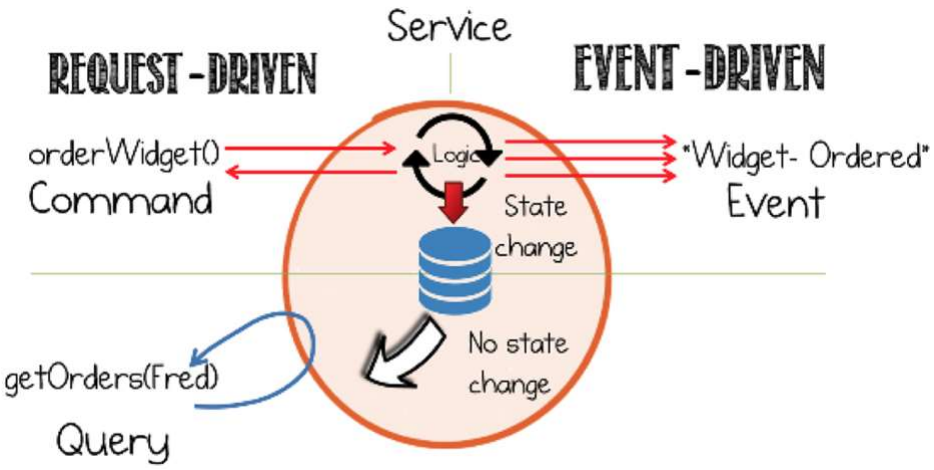


12

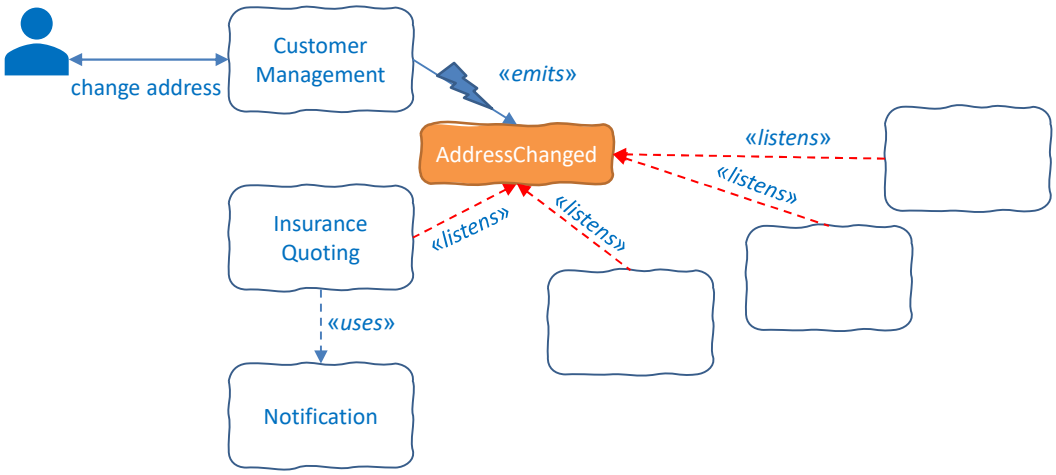
Differences between commands, events, and queries

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

A visual summary of commands, events, and queries

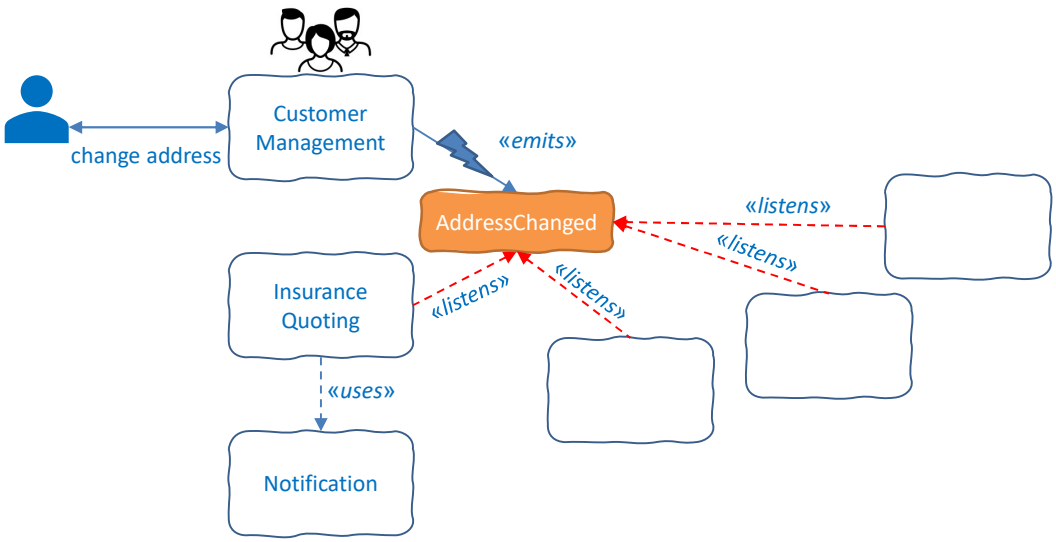


Collaboration Patterns: Commands/Events



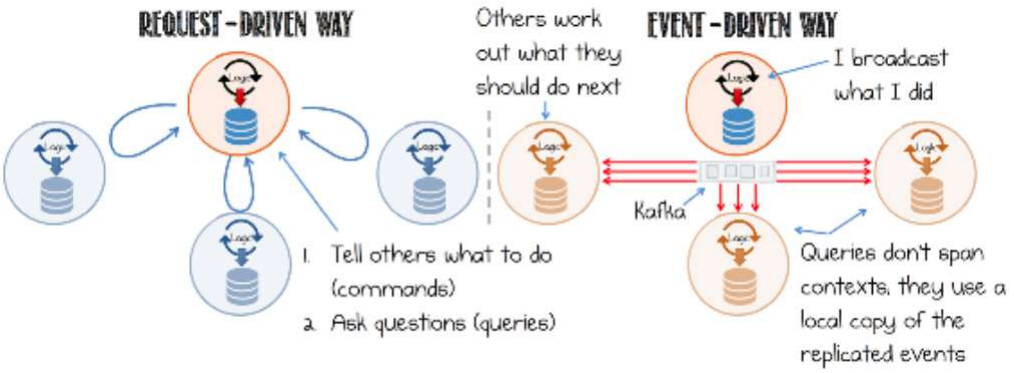
15

Collaboration Patterns: Commands/Events

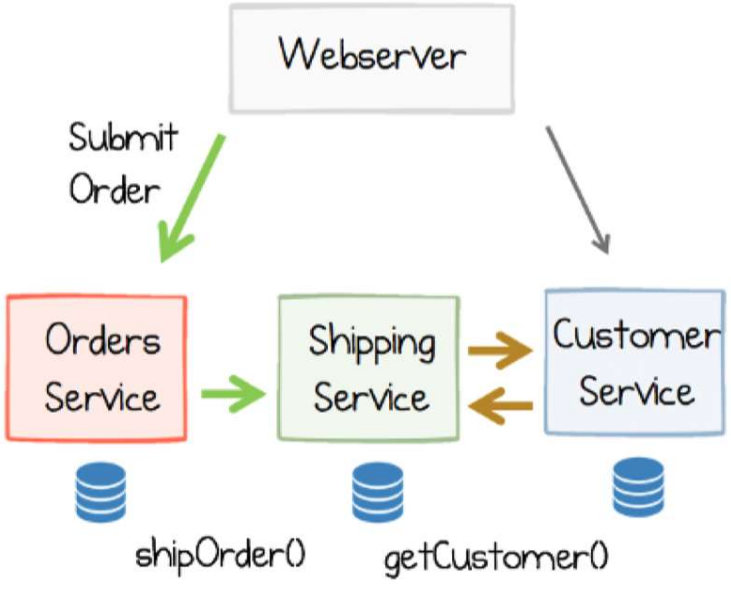


16

Request-response vs Event-driven

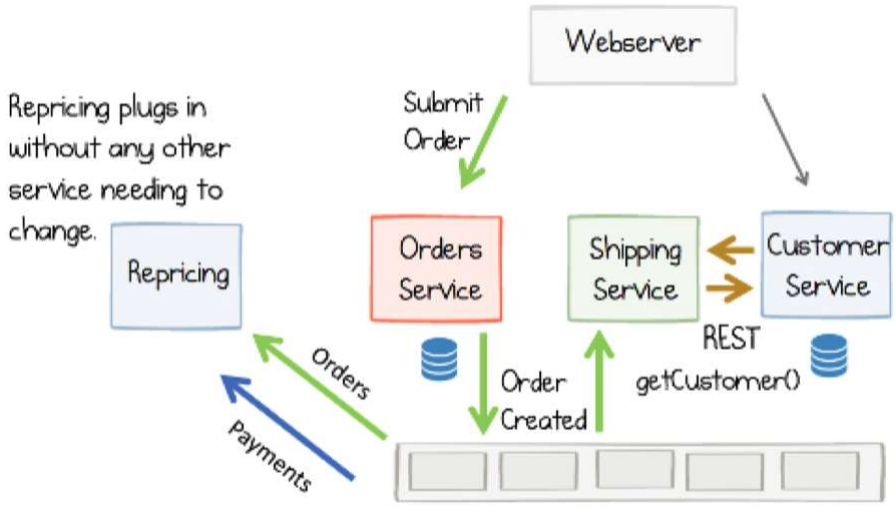


A request-driven order management system

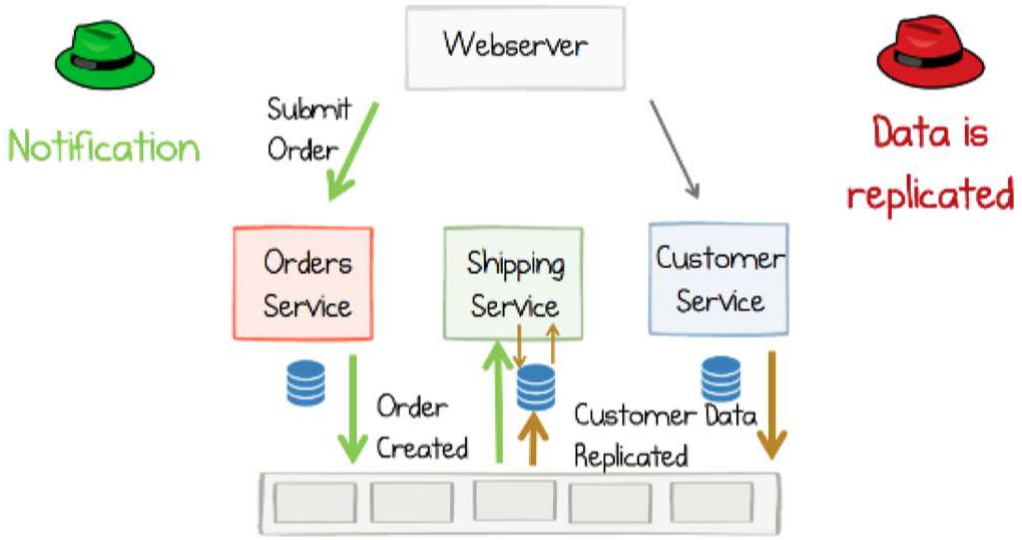


An event-driven order management system

Extensibility



Fully Event-Driven System



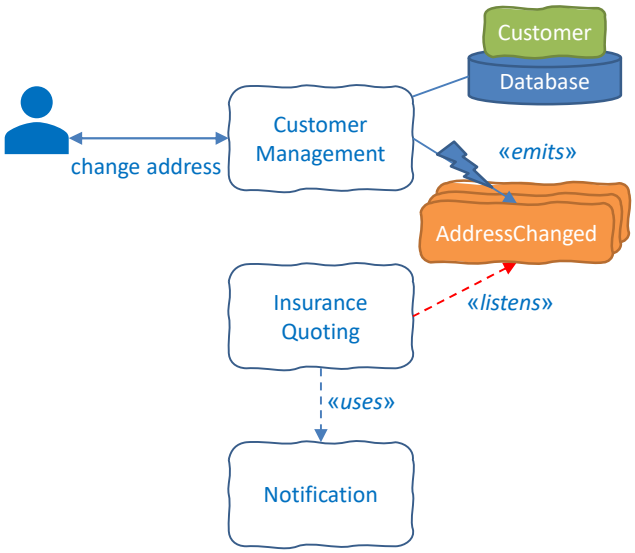
Collaboration Patterns: Commands/Events

> Event Notification
  - Decouple receiver from sender ✓
  - No statement of overall behavior ✗

EVENT-CARRIED STATE TRANSFER

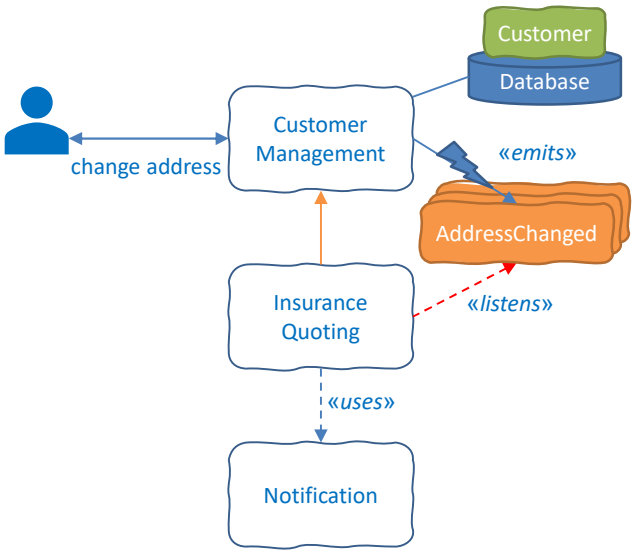
23

Collaboration Patterns



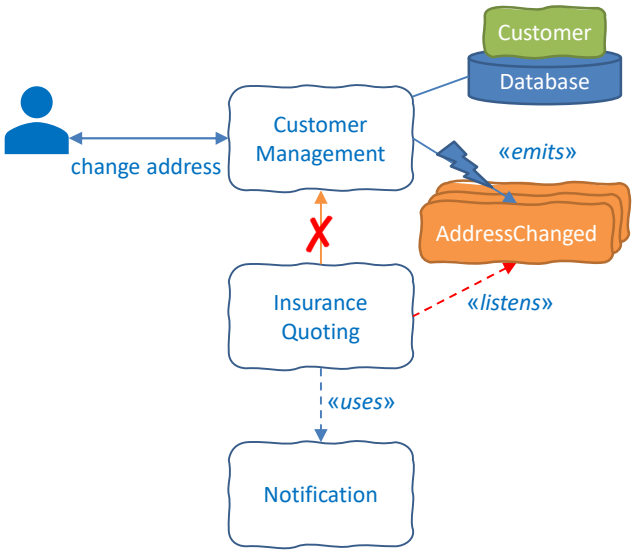
24

Collaboration Patterns



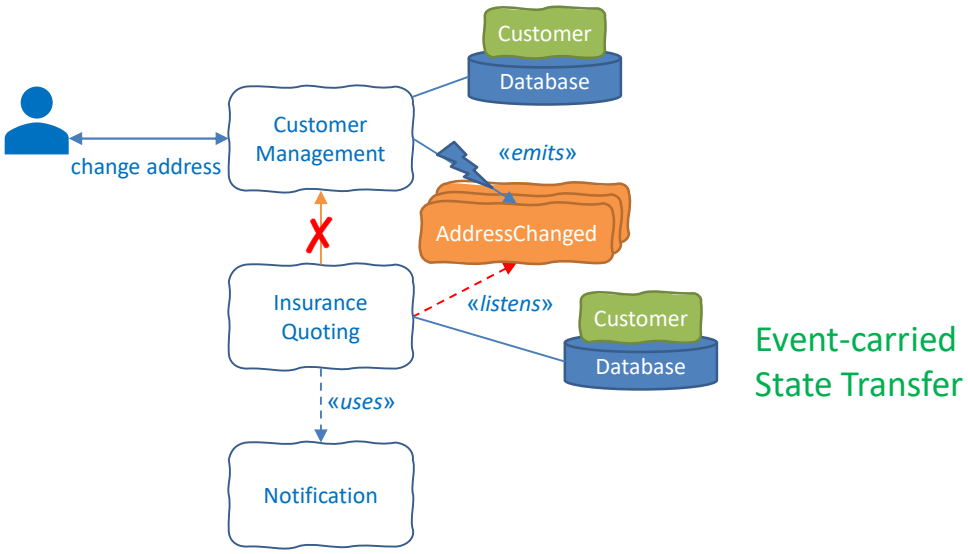
25

Collaboration Patterns



26

Collaboration Patterns



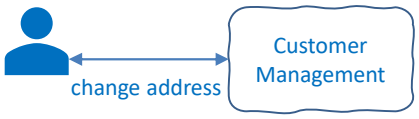
Collaboration Patterns

> Event-Carried State Transfer
  - Decoupling ✓
  - Reduced load on supplier ✓
  - Replicated Data ✗
    - Eventual Consistency ✗

EVENT SOURCING

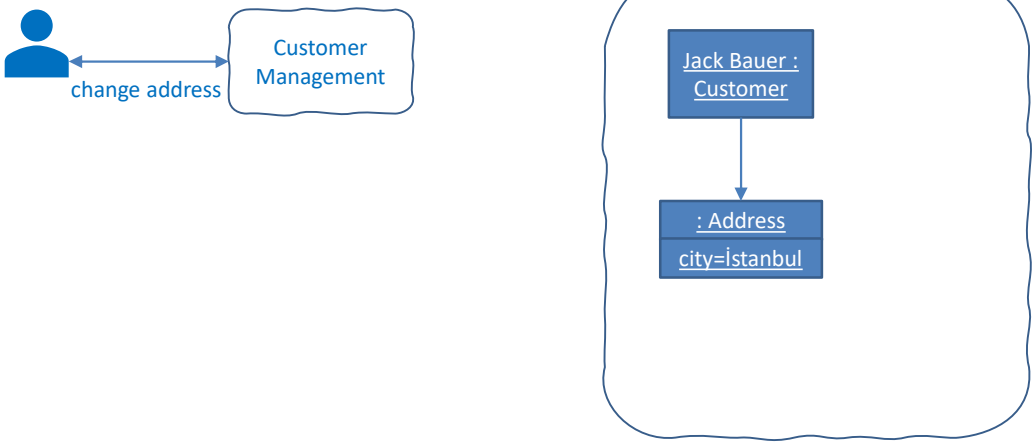
29

Event Sourcing



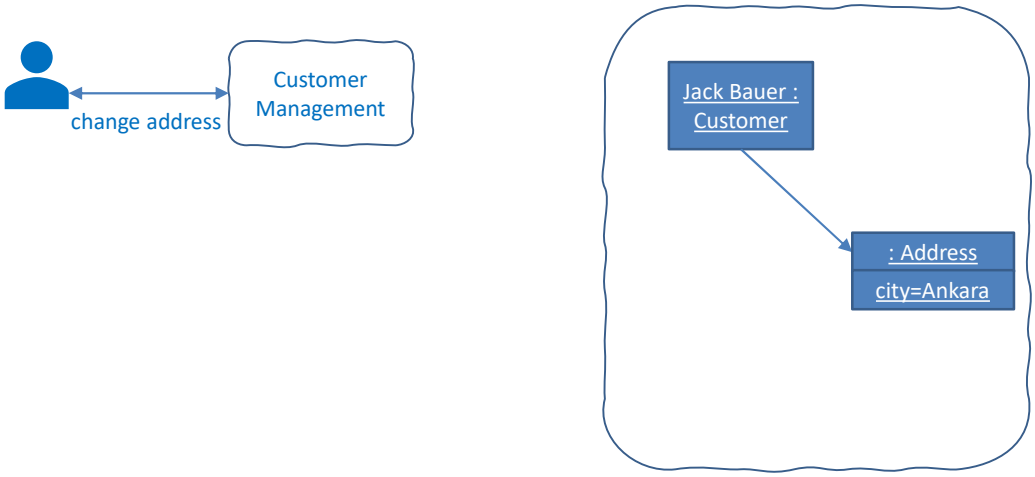
30

Event Sourcing



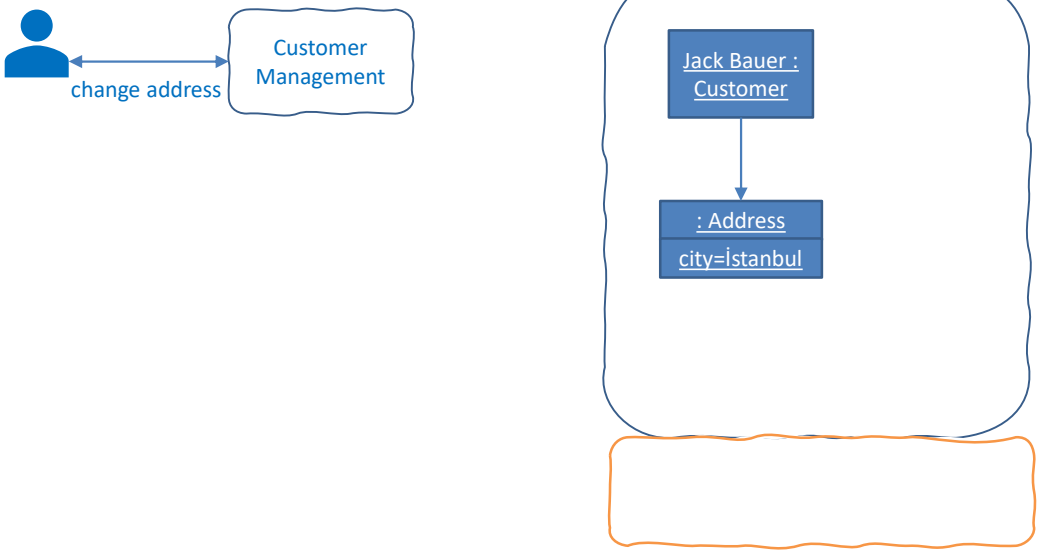
31

Event Sourcing

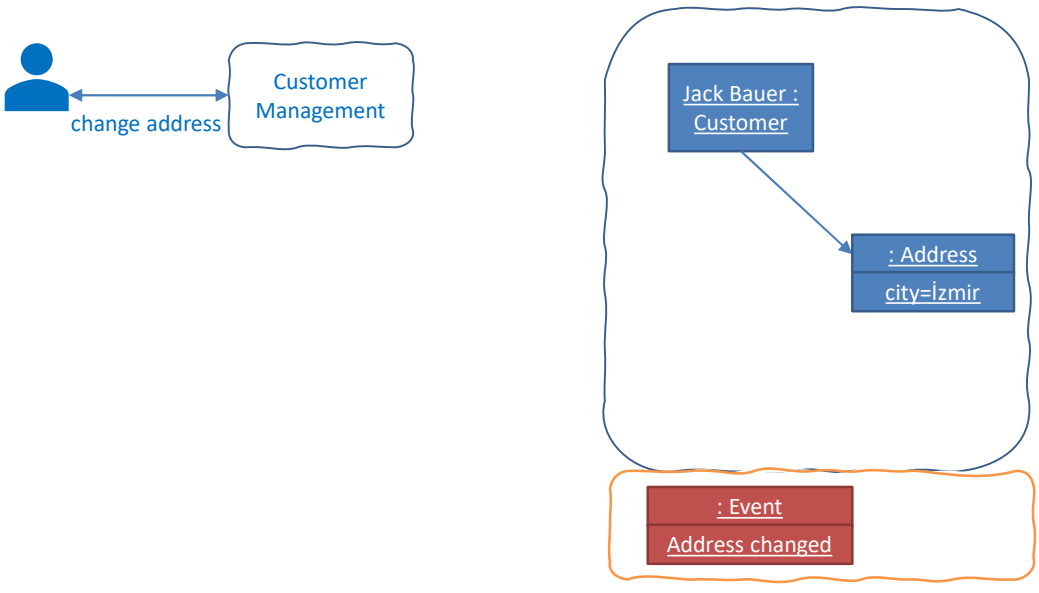


32

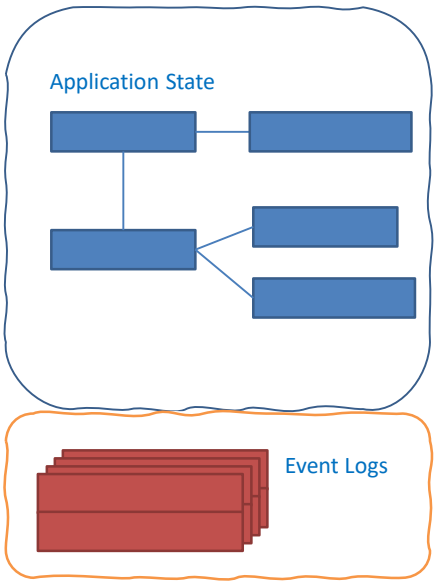
Event Sourcing



Event Sourcing

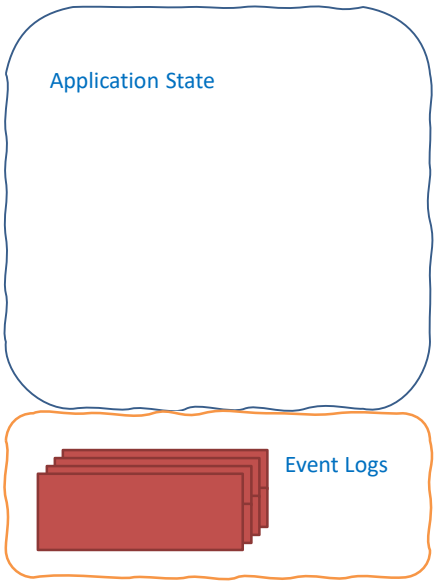


Event Sourcing



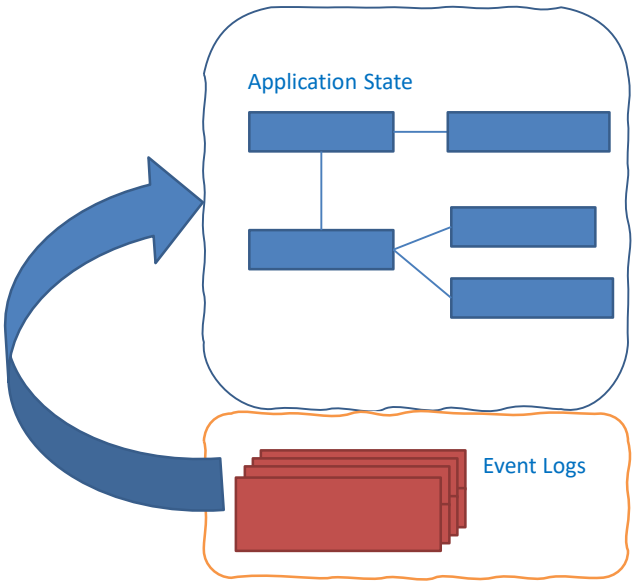
35

Event Sourcing



36

Event Sourcing



Event Sourcing

- > Audit ✓
- > Debugging ✓
- > Historic State ✓
- > Memory Image ✓
- > Unfamiliar ✗
- > Event Schema ✗
- > Identifiers ✗
- > Versioning ✗

39

Introduction to CQRS

- **Definition**
  - CQRS separates the responsibilities of commands (write operations) and queries (read operations) in a system.
- **Key Idea**
  - Use different models for data modifications and data retrieval.
- **Benefits**
  - Scalability
  - Flexibility
  - Enhanced performance for specific workloads

40

Command

- Represents an intention to change the state of the system.
- Commands are responsible for modifying the data or triggering some action in the application.
- Examples of commands include creating a new record, updating existing data, or deleting information.

Query

- Represents a request for information or data retrieval from the system without causing any state changes.
- Queries are read operations that retrieve data from the system without modifying it.
- Examples of queries include fetching a list of items, retrieving details of a specific record, or executing a search operation.

Responsibility Segregation

> CQRS advocates separating the read and write responsibilities into different parts of the application.
> The code for handling commands (write operations) is distinct from the code for handling queries (read operations).
> This separation can simplify the design and maintenance of the application.

Model Separation

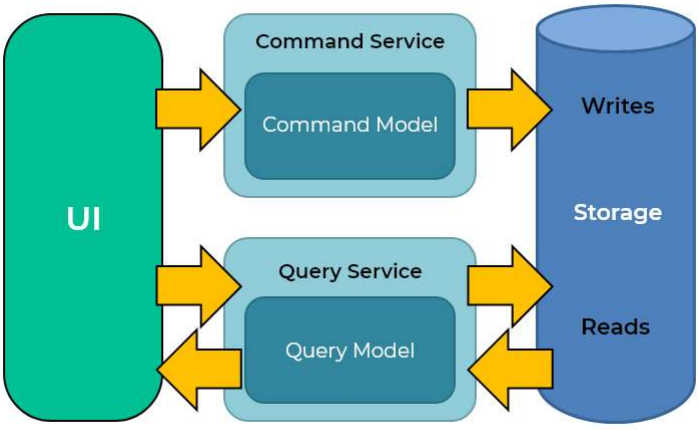
> In addition to separating responsibilities, CQRS often involves using different models for reading and writing.
> The write model is optimized for making changes efficiently, while the read model is optimized for querying and presenting data.

Asynchronous Communication

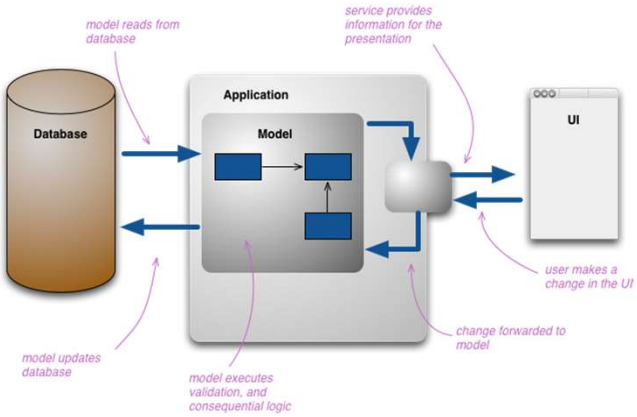
> CQRS systems may use asynchronous communication between the write and read components.

> This can help in achieving better scalability and responsiveness, especially when dealing with a high volume of write operations.

CQRS

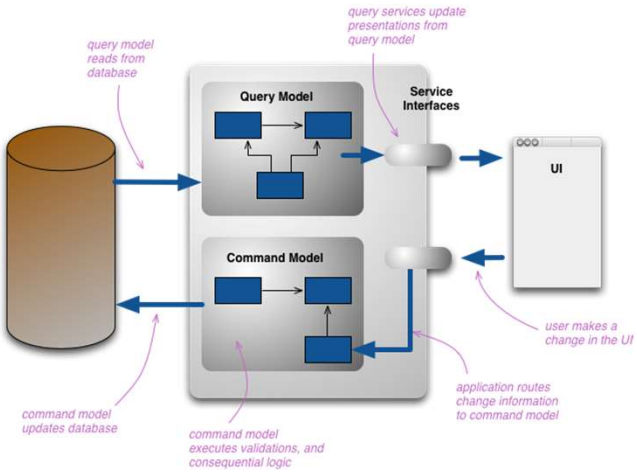


CQRS



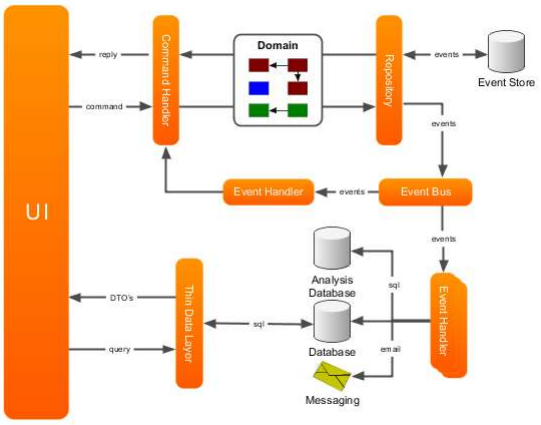
47

CQRS



48

CQRS



Advantages

- Improved performance
  - The command and query sides can be scaled independently, which can improve the overall performance of the system.
- Increased scalability
  - CQRS can make it easier to scale the system to handle more users and data.
- Improved security
  - The command and query sides can be separated by a firewall, which can help to improve the security of the system.

Disadvantages

- Increased complexity
  - CQRS can make the system more complex to design and implement.
- Eventual consistency
  - The query side may not always be up-to-date with the latest events, which can lead to inconsistencies.

When to use it

- CQRS should only be used on specific portions of a system (a Bounded Context in DDD) and not the system as a whole
- Each Bounded Context needs its own decisions on how it should be modeled
- Using CQRS on a domain that doesn't match it will add complexity, thus reducing productivity and increasing risk

Sample Implementation in Python

```
class CreateOrderCommand:
    def__init__(self, customer_id, product_id, quantity):
        self.customer_id = customer_id
        self.product_id = product_id
        self.quantity = quantity

class CreateOrderCommandHandler:
    def handle(self, command):
        # Create an order object with data from the command
        order = Order(command.customer_id, command.product_id, command.quantity)
        # Save the order to your database
        order.save()
        # Publish an event to notify listeners about the created order
        event_bus.publish(OrderCreatedEvent(order))
```


Sample Implementation in Python

```
class GetOrderDetailsQuery:
    def __init__(self, order_id):
        self.order_id = order_id

class GetOrderDetailsHandler:
    def handle(self, query):
        # Fetch the order details from the database or event store
        order = Order.find_by_id(query.order_id)
        # Build the response object with retrieved data
        response = GetOrderDetailsResponse(order.customer_id, order.product_id,
order.quantity)
        return response
```

Sample Implementation in Python

```
class OrderCreatedEvent:
    def __init__(self, order):
        self.order_id = order.id
        self.customer_id = order.customer_id
        self.product_id = order.product_id
        self.quantity = order.quantity

# Use a suitable event store library to persist events
event_store.save(OrderCreatedEvent(order))
```


Sample Implementation in Python

```
# Subscribe to OrderCreatedEvent
@event_bus.subscribe(OrderCreatedEvent)
def on_order_created(event):
    # Update the read model with information from the event
    read_model.add_order(event.order_id, event.customer_id, event.product_id,
event.quantity)
```