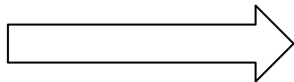
What it is and how it looks like

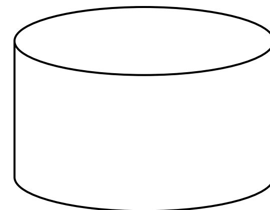
Traditional Approach

```
class Product {
    val id = 42
    fun update(...) {
        this.name = "Coca-Cola"
        this.price = 1.99
    }
}

repository.save(product)
```



ID	Name	Price
42	**Coca-Cola**	**1.99**
43	Water	0.99



Pros and Cons

+

- Easy and familiar concept
- Simple querying of data
- A lot of experience with this model
- Good framework support

-

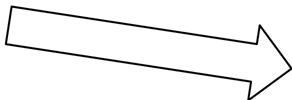
- No historical data

Event Sourcing Approach

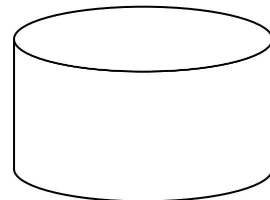
```
class Product {
    val id = 42
    fun update(...) {
        ...
        return ProductUpdatedEvent(42, "Coca-Cola", 1.99)
    }
}
```

All state changes are published as events! No change without an event!

```
eventBus.publish(event)
eventStore.save(event)
```

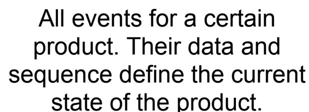


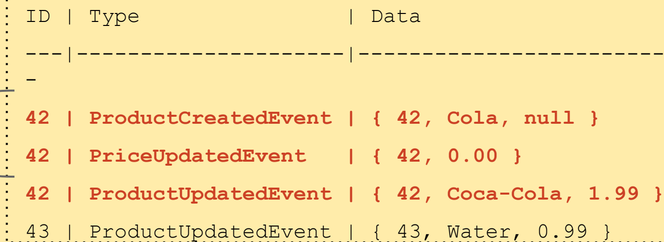
```
ID | Type                 | Data
---|---------------------|---------------------------
-
42 | ProductCreatedEvent | { 42, Cola, null }
42 | PriceUpdatedEvent   | { 42, 0.00 }
42 | ProductUpdatedEvent | { 42, Coca-Cola, 1.99 }
43 | ProductUpdatedEvent | { 43, Water, 0.99 }
```



Event Sourcing Approach

- Only events are saved, no state! *
- To get the current state, we must “replay” all events (for the according ID)





* There’s an exception to this rule: We can create a dedicated read model for better performance. See slide 19.

Pros and Cons

+

- Historical data (== log of all events)
- Simple database schema

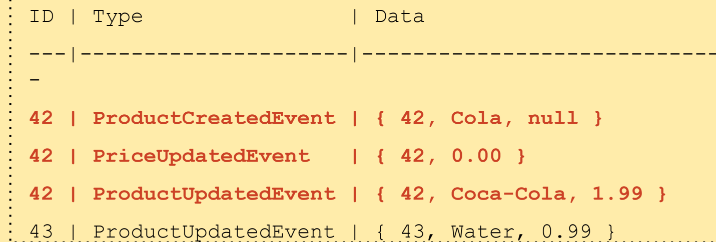
-

- Complicated programming model
- Tough reading/querying of data
- Less framework support

What means “replaying”?

- We don’t save and load a single state/set of data (== *traditional approach*)
- We save and load a list of events
- We apply one event after another to finally come to the current state

```
val events = eventStore.findAllById(42)
val product = Product().applyAll(events)
```



So replaying means...

- Loading a list of events from a local database
- Applying those events one after another
- Changing the internal state of a domain object each time
- It’s the default way of storing data
- Everytime we need an object we replay events (so we do it all the time!)
- Events contain _all_ data (not just an ID)

And replaying doesn't mean...

- Actually sending events over the message bus (== Kinesis, ActiveMQ, ...)
- No other system is involved - it's just locally!
- Replaying doesn't cause any side effects - just state is updated
- There's no "replay mode" or "fallback flag"
- We don't do it in a specially exception edge case - we do it all the time!

Consequences

Applying an event changes data, but doesn’t cause side effects.

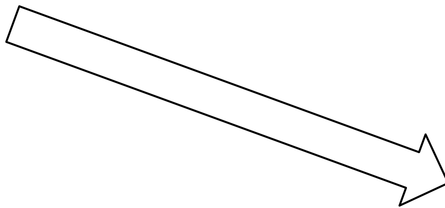
Executing a command may cause side effects and lead to events.

Let’s see some code (1/2)

```
val events = eventStore.findAllById(42)
val product = Product().applyAll(events)

// Now that we have an object with the current state,
// let’s do some business operation on it!

val command = UpdateProductCommand(“Pepsi” 2,49)
val newEvents = product.execute(command)
eventBus.publish(newEvents)
eventStore.save(newEvents)
```



```
ID | Type                 | Data
---|----------------------|----------------------------
-
42 | ProductCreatedEvent | { 42, Cola, null }
42 | PriceUpdatedEvent   | { 42, 0.00 }
42 | ProductUpdatedEvent | { 42, Coca-Cola, 1.99 }
42 | ProductUpdatedEvent | { 42, Pepsi, 2.49 }
43 | ProductUpdatedEvent | { 43, Water, 0.99 }
```

Let's see some code (2/2)

```
class Product {

    fun execute(UpdateProductCommand) {
        // Business logic, calculating stuff, log messages
        // and finally return an event
        return ProductUpdatedEvent(...)
    }

    fun apply(ProductUpdatedEvent) {
        // Apply data! Don't do anything else!
        this.name = event.name
        this price = event.price
    }
}
```

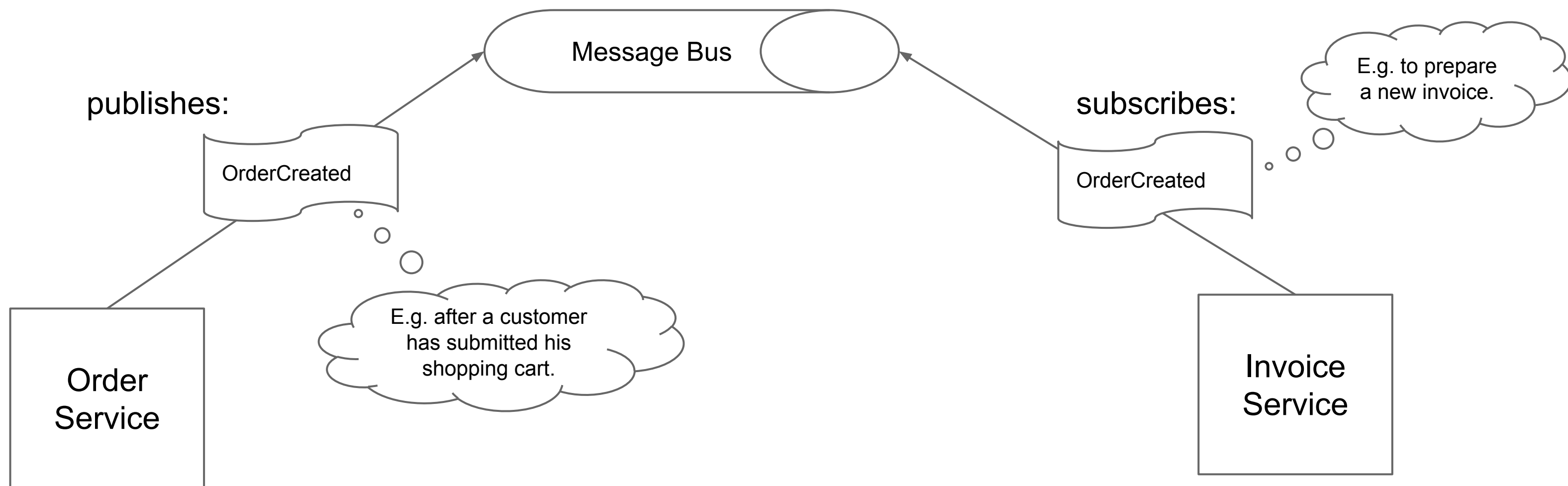
Important: Don't change state here! State only changes in reaction of an event!

```
ID | Type                | Data
---|---------------------|---------------------------
-
42 | ProductCreatedEvent | { 42, Cola, null }
42 | PriceUpdatedEvent   | { 42, 0.00 }
42 | ProductUpdatedEvent | { 42, Coca-Cola, 1.99 }
42 | ProductUpdatedEvent | { 42, Pepsi, 2.49 }
43 | ProductUpdatedEvent | { 43, Water, 0.99 }
```

Integrating External Systems

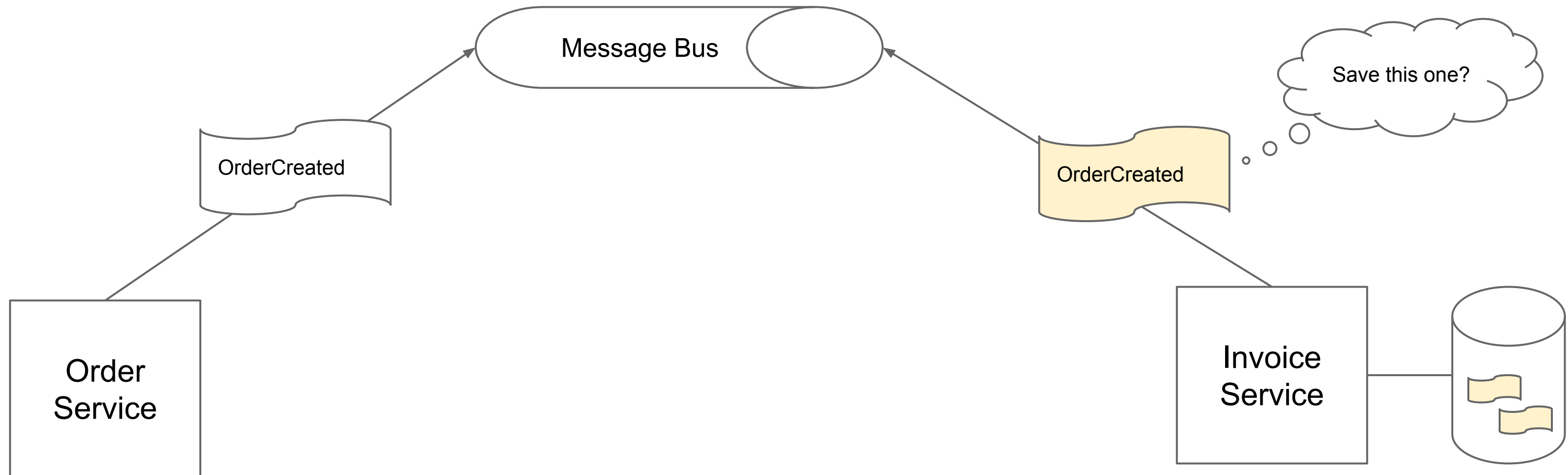
Integrating External Systems (1/3)

- Systems often communicate via asynchronous messages
- Usually those messages are events (rather then commands)



Integrating External Systems (2/3)

So should we save those events?



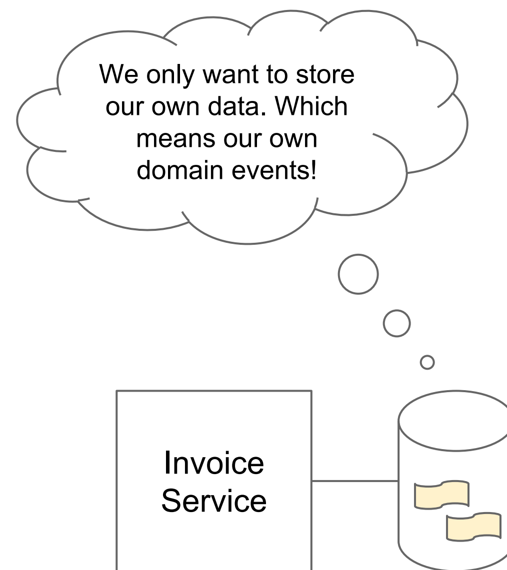
Integrating External Systems (3/3)

So should we save those events?

- No, better not.

- We cannot control the format of external events, which would become part of our persistence.
- External events don't necessarily apply to our domain directly. Usually we run business logic on them to validate and transform the to our context.

So no, don't save messages from external systems!

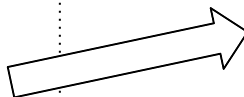


Advanced Topics

Advanced Topics: Snapshots

- Applying a lot of events (hundreds? thousands?) can be inefficient
- To solve this, old events are merged together
- Historical data will be lost…
- …but the number of events is reduced

```
ID | Type                  | Data
---|----------------------|----------------------------
-
42 | ProductCreatedEvent | { 42, Cola, null }
42 | PriceUpdatedEvent   | { 42, 0.00 }
42 | ProductUpdatedEvent | { 42, Coca-Cola, 1.99 }
42 | ProductUpdatedEvent | { 42, Pepsi, 2.49 }
43 | ProductUpdatedEvent | { 43, Water, 0.99 }
```

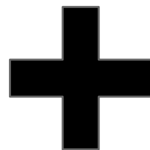


```
ID | Type                  | Data
---|----------------------|----------------------------
-
42 | ProductUpdatedEvent | { 42, Pepsi, 2.49 }
43 | ProductUpdatedEvent | { 43, Water, 0.99 }
```

Advanced Topics: Read Model

- Applying a lot of events (hundreds? thousands?) can be inefficient
- And even if there are just a few events, querying the data is difficult
- To solve this, we can create a write model for queries (CQRS)
- Usually, this is a common relational database where data is duplicated

```
ID | Type                  | Data
---|---------------------|----------------------------
-
42 | ProductCreatedEvent | { 42, Cola, null }
42 | PriceUpdatedEvent   | { 42, 0.00 }
42 | ProductUpdatedEvent | { 42, Coca-Cola, 1.99 }
42 | ProductUpdatedEvent | { 42, Pepsi, 2.49 }
43 | ProductUpdatedEvent | { 43, Water, 0.99 }
```



```
ID | Name        | Price
---|------------|----------
42 | Coca-Cola  | 1.99
43 | Water      | 0.99
```

Demo

https://github.com/bringmeister/event-sourcing-with-kotlin

Resources

- http://microservices.io/patterns/data/event-sourcing.html
  - A short description of the pattern including some sample code and additional resources. This is a good starting point to get a first impression of event sourcing. The page also shows related patterns as well as pros and cons.
- http://engineering.pivotal.io/post/event-source-kafka-rabbit-jpa
  - *A demo from Pivotal showing a small example using DDD, event sourcing, commands and a nice CQRS implementation. You will find the source code on GitHub.*
- http://www.baeldung.com/axon-cqrs-event-sourcing
  - A brief example of DDD and event sourcing with the AXON framework.
- https://www.maibornwolff.de/blog/event-sourcing-part1
  - A discussion on event sourcing in combination with CQRS.
- https://ookami86.github.io/event-sourcing-in-practice/#making-eventsourcing-work
  - A presentation on a lot of different aspects of Event Sourcing.