Orchestration of Microservices

bernd.ruecker@camunda.com

With thoughts from http://flowing.io
@berndruecker | @martinschimak
Microservoices?
Microservices

- Independent components
- Independent deployments
- Decoupling between components
- Dedicated teams to fight conways law
- Autonomy of technology decisions
- Avoid horizontal team boundaries
- New DevOps paradigms
Increasing complexity of relationships
Event-Driven Microservices

Microservice A

Event

event name payload no recipient

Microservice B
Event-Driven Microservices

Microservice A

Event

Does not know about B

Does not know about A

Microservice B
Isn’t that SOA?
It is not SOA

„smart endpoints and dumb pipes“
Disclaimer: I do not advertise event-driven microservices as silver bullet!

Only use it when appropriate.
Handle with care!
Distributed systems
You need some safe harbors and strategies to sail the wild ocean
Microservices

Event Driven & Reactive

Domain Driven Design
Let’s do an example!

Assume you want to build a Dash button.
I want to have one item!

I am happy!

place order pay receive shipment
Business Capabilities

Shop: Order placed
Payment: Payment received
Inventory: Goods fetched
Shipping: Goods shipped

Bounded Contexts = Microservices
Business Outputs / Domain Events
Collaboration using an eventflow

- Shop
  - Order placed
- Payment
  - Payment received
- Inventory
  - Goods fetched
- Shipping
  - Goods shipped
Great!
Great?
Let’s zoom in the payment context

Order placed → Payment → Payment received
Let’s zoom in the payment context

Order placed → Payment → Payment received

The payment context has to listen to „order placed“ event
De-coupling?

Whenever a new client requires payment, the payment context has to be touched.

The payment context has to know all possible events that trigger a payment.

- Order placed
- Subscription confirmed
- Service fullfilled
- ...

Payment
Event command transformation

**Event**
Something has happened in the past
0..n recipients

**Command**
Something has to happen in the future
1 recipient
Order context

Order placed \rightarrow Order \rightarrow Retrieve payment \rightarrow Payment
Event vs. command

- Order placed
- Order
- Retrieve payment
- Payment

decide where to do the coupling
Business Capabilities

Shop
- Order placed

Order
- Order completed

Payment
- Payment received

Inventory
- Goods fetched

Shipping
- Goods shipped

Retrieve payment
- Fetch goods
- Ship goods
Quality Criteria: Changability

VIP customers can order with invoice (and pay later)
Changability with pure event-chain

If not VIP customer

Order placed → Payment received → Goods fetched → Goods shipped

If VIP customer

VIP customers can order with invoice (and pay later)

Order billed

Shop | Payment | Inventory | Shipping | Billing
Changability with Event Command Transformation

Change how to issue commands

VIP customers can order with invoice (and pay later)

Shop  Order  Payment  Inventory  Shipping

Order placed  Order completed  Payment received  Goods fetched  Goods shipped
Quick demo

http://github.com/flowing/
Now back to real-life

-----Ursprüngliche Nachricht-----
Von: GitHub [mailto:support@github.com]
Gesendet: Samstag, 8. April 2017 12:44
Betreff: [GitHub] We had a problem billing your account

Greetings flowing,

This email is to inform you that we’ve received the following error trying to bill your credit card:

Declined

We’ll try billing your credit card again over the next two weeks. Please resolve this issue to ensure uninterrupted service.

If you need to update your payment information, please head to https://github.com/organizations/flowing/settings/billing to help you out.

If you feel there’s been a mistake or have any questions, please contact support@github.com.

Thanks,
The GitHub Team
Some things in live might be slow
Long running flows require persistent state
How to implement?
State in entity

class Order {
    String id;
    Customer customer;
    List<OrderItem> items;

    static enum GoodsDeliveryStatus {
        NOTHING_DONE,
        GOODS_RESERVED,
        GOODS_PICKED
    }
    boolean paymentReceived = false;
    GoodsDeliveryStatus deliveryStatus = GoodsDeliveryStatus.NOTHING_DONE;
    boolean shipped = false;

    //...
}
The 7 sins of workflow

http://blog.bernd-ruecker.com/
The 7 sins of workflow

No state machine

http://blog.bernd-ruecker.com/

Homegrown state machine
Think about subsequent requirements

- Monitoring & Operations
- Versioning
- Time & Timeouts
- Visibility
- Domain Language
- Performance & Scaling
Process Manager

- Do event command transformation
- Handle state for long running flows
private void createFlow() {
    engine.getRepositoryService().createDeployment()
        .addModelInstance(Bpmn.createExecutableProcess("order")
            .startEvent()
            .serviceTask().name("Retrieve payment")\camundaClass(DoPaymentAdapter.class)
            .serviceTask().name("Fetch goods")\camundaClass(PickGoodsAdapter.class)
            .serviceTask().name("Ship goods")\camundaClass(ShipGoodsAdapter.class)
            .endEvent()\camundaExecutionListenerClass("end", OrderCompletedAdapter.class)
            .done()
        ).deploy();
}
Logic remains in normal code

```java
public class RetrievePaymentAdapter implements JavaDelegate {

    public void execute(ActivityExecution ctx) {
        // prepare message content, you can load variables from persistent flow context
        ctx.getVariable("orderId");
        // ...

        // Send command
        publishCommand("DoPayment", payload);
        // ant tell the engine to wait for the next event
        addEventSubscription("PaymentReceived", ctx);
    }
}
```
Tools provide persistent state, visibility and much more
On top: visibility

Order placed \rightarrow Order \rightarrow Retrieve payment \rightarrow Payment

Order placed \rightarrow Retrieve payment \rightarrow Fetch goods \rightarrow Ship goods \rightarrow Order delivered
Event notification is nice because it implies a low level of coupling, and is pretty simple to set up. **It can become problematic, however, if there really is a logical flow that runs over various event notifications.** The problem is that it can be hard to see such a flow as it's not explicit in any program text. Often the only way to figure out this flow is from monitoring a live system. This can make it hard to debug and modify such a flow. **The danger is that it's very easy to make nicely decoupled systems with event notification, without realizing that you're losing sight of that larger-scale flow,** and thus set yourself up for trouble in future years.
Smells like „BPM“?
The 7 sins of workflow

No engine  Wrong engine  Wrong usage

Homegrown engine  BPM monolith
The end-to-end process?
Don't do a „BPM monolith“ when using Microservices!
Local flows in the bounded contexts

Order

- Order placed
- Retrieve payment
- Fetch goods
- Ship goods
- Order delivered

Payment

- Charge credit card
- Payment received
- Credit card failed
- Inform customer that new credit card is required
- Wait for new credit card details
- 7 days
- Payment failed

"Local orchestration and overall choreography"
Multiple engines

Order engine

Payment engine

... engine

Inventory

Shipping engine
The 7 sins of workflow

1. No engine
2. Wrong engine
3. Zero-code suites
4. Wrong usage
5. BPM monolith
6. Homegrown engine
Death by properties panel

Please enter your complex code here. (Without IDE support of course!)
Bernd Rücker
Consultant & Evangelist
10+ years workflow

http://bernd-ruecker.com/
bernd.ruecker@camunda.com

Co-founder Camunda
http://camunda.org/
KEEP CALM AND LOVE JAVA
Define flows programatically or graphically

```java
private void createFlow() {
    engine.getRepositoryService().createDeployment()
        .addModelInstance("order.bpmn", Bpmn.createExecutableProcess("order")
            .startEvent()
            .serviceTask().name("Retrieve payment").camundaClass(DoPaymentAdapter.class)
            .serviceTask().name("Fetch goods").camundaClass(PickGoodsAdapter.class)
            .serviceTask().name("Ship goods").camundaClass(ShipGoodsAdapter.class)
            .endEvent().camundaExecutionListenerClass("end", OrderCompletedAdapter.class)
        ).deploy();
}
```
Productive development and testing

```java
@Test
@Deployment(resources = { "Order.bpmn", "RiskyOrder.dmn" })
public void testHappyPath() {
    when(orderFlow.waitsAtUserTask("UserTask_ApproveOrder")) //
        .thenReturn((task) -> task.complete(Variables.putValue("approved", true)));

    mockRestServer
        .expect(requestTo("https://api.stripe.com")) // normal mocking...
        .thenReturn((messageSubscription) -> messageSubscription.receive());

    Scenario.run(orderFlow).startByKey( //
        "OrderGraphic", //
        Variables.putValue("customerCategory", "unknown").putValue("orderAmount", 500)).execute();

    verify(orderFlow).hasFinished("EndEvent_OrderShipped");
}```
developer friendly
Big challenges for developers ahead!
Error and timeout handling

- Payment retrieval requested
- Charge credit card
- Payment received
- Credit card failed
- Inform customer that new credit card is required
- Wait for new credit card details (7 days)
- Payment failed
Handling complex scenarios
Demo

http://github.com/flowing/
Demo architecture

for demo in single Java VM for simplicity

- Shop
- Order
- Payment
- Inventory
- Shipping
- Monitor

Camunda Webapp on Tomcat

https://github.com/flowing/flowing-retail/
Lightweight state machines or workflow engines are not evil!

They do help you solving some really hard coding problems.

These problems become more and more frequent every day.
Know the Flow! Microservices and Event Choreographies

Key Takeaways

- In a microservices architecture it is not uncommon to encounter services which are long running and stretch across the boundary of individual microservices.
- Event based architectures with corresponding choreographies are one increasingly

https://www.infoq.com/articles/microservice-event-choreographies
Code online: https://github.com/flowing

Slides online: http://bernd-ruecker.com

Feedback: http://bernd-ruecker.com/feedback

With thoughts from http://flowing.io
@berndruecker | @martinschimak
Thank you!