Complex event flows in distributed systems

O'Reilly Software Architecture Conference
16th of October 2017

mail@bernd-ruecker.com
martin.schimak@plexiti.com

With thoughts from http://flowing.io
@berndruecker | @martinschimak
Bernd Rücker
Consultant & Dev. Advocate
10+ years workflow
Co-founder Camunda

http://bernd-ruecker.com/

Martin Schimak
Consultant & Trainer
15+ years domain (de-)coding
DDD, TDD/BDD, EDA

http://plexiti.com/

http://flowing.io/
Example
Distributed systems
Event-Driven

- Does not know recipient
- Does not know sender

Order Placed

Order: Fact that happened in the past, Immutable fact, 0..n recipients

Does know event type...

Customer status changed

Decentral data management

Smart endpoints and dumb pipes
End-to-end capabilities using event flows?
End-to-end capabilities using event flows?

- **Checkout**
- **Payment**
- **Inventory**
- **Shipping**

**Order placed** → **Payment received** → **Goods fetched**

- Please fetch the goods before waiting for payment
- Some customers can pay via invoice
- ...
Some requirements don’t fit anywhere... What are we missing?
Let’s make our core capability transparent – as a dedicated service

**Event**: Fact, happened in the past, immutable, 0..n recipients

**Command**: Intent, 1 recipient.
Commanding is important!

**Event**: Fact, happened in the past, immutable, 0..n recepients

**Command**: Intent, 1 recipient.

Does not know sender

Just knows command type...

Dangerous?
Some things in life take time
A customer can update an expired credit card within two weeks before we cancel his order.
Payment requires state handling

Order
Persist state with Entity, Actor, ...

State machine

Pass around state as „routing slip“

DIY

How to implement?
Persistent thing
(Entity, Actor, ...)

State machine

Routing slip

CADENCE
ING
AWS Step Functions
Camunda
zeebe
conductor
Activiti
State machines solve some hard developer problems

- Handling of time & timeouts
- Retry
- Message correlation & deduplication
- Compensation
- Versioning
- Visibility & Reporting
- Monitoring & Operations
- Performance & scalability
Distributed systems
State machines solve some hard developer problems

- Handling of time & timeouts
- Retry
- Message correlation & deduplication
- Compensation
- Versioning
- Visibility & Reporting
- Monitoring & Operations
- Performance & scalability

Performance & scalability
Do you prefer coded or graphical DSLs?

```json
da: {  "name": "retrieve_payment",  "tasks": [    {      "name": "Retrieve Payment",      "taskReferenceName": "payment",      "type": "SIMPLE",    ...  ]}
```

* BPMN - ISO notation for modeling and executing long-running processes and flows

```javascript
Bpmn.createProcess("order").executable()
//...
.sendTask().name("Retrieve payment").camundaClass(RetrievePaymentAdapter.class)
.receiveTask().name("Payment received").message("PaymentReceivedEvent")
//...
```

---

* Netflix Conductor (JSON)

* Camunda (BPMN * and Java)
Do you need a timeout?
Do you need a compensation?*
BPM?
Flows done right

Developer friendly

Lightweight & composable

BizDevops
public class AllInOne {
    public static void main(String[] args) {
        StandaloneInMemProcessEngineConfiguration conf = new StandaloneInMemProcessEngineConfiguration();
        ProcessEngine engine = conf.buildProcessEngine();

        engine.getRepositoryService().createDeployment()  
          .addModelInstance("flow.bpmn", Bpmn.createProcess("flow").executable() //
          .startEvent()  
          .serviceTask().name("Step1").camundaClass(SysoutDelegate.class)  
          .serviceTask().name("Step2").camundaClass(SysoutDelegate.class)  
          .endEvent().camundaExecutionListenerClass("end", SysoutDelegate.class)  
          .done()  
          ).deploy();

        engine.getRuntimeService().startProcessInstanceByKey("flow", Variables.putValue("someData", "someValue"));
    }  
    public static class SysoutDelegate implements JavaDelegate {
        public void execute(DelegateExecution execution) throws Exception {
            System.out.println("hello from activity " + execution.getCurrentActivityId());
        }
    }
}
public class AllInOne {
    public static void main(String[] args) {
        StandaloneInMemProcessEngineConfiguration conf = new
          ProcessEngine engine = conf.buildProcessEngine();
        engine.getRepositoryService().createDeployment() //
            .addModelInstance("flow.bpmn", Bpmn.createProcess("flow").executable() //
            .startEvent()
            .serviceTask().name("Step1").camundaClass(SysoutDelegate.class)
            .serviceTask().name("Step2").camundaClass(SysoutDelegate.class)
            .endEvent().camundaExecutionListenerClass("end", SysoutDelegate.class)
            .done()
        ).deploy();

        engine.getRuntimeService().startProcessInstanceByKey("flow", Variables.putValue("someData", "someValue"));
    }
}

public static class SysoutDelegate implements JavaDelegate {
    public void execute(DelegateExecution execution) throws Exception {
        System.out.println("hello from activity " + execution.getCurrentActivityId());
    }
}
More complex flows - transparent and directly executable
Focus on long-running behaviour - requiring state
Specifying and testing long-running behaviour

As a customer I expect that my credit, which was withdrawn for a payment exceeding that credit, is restored again in case the remaining payment fails, so that I can use my credit again for later payments.

**Scenario Outline:** Restore account balance when payment ultimately fails

<table>
<thead>
<tr>
<th>Given</th>
<th>And</th>
<th>But</th>
<th>And</th>
<th>When</th>
<th>Then</th>
<th>And</th>
</tr>
</thead>
</table>
| the customer has an account balance of $\langle start \rangle$ | a payment of $\langle amount \rangle$ is required | the customer's credit card already expired | the customer does not update his credit card for 14 days and 1 hours | we walk the customer through the payment retrieval | the payment retrieval $\langle is\_not \rangle$ successful | the customer has a final balance of $\langle final \rangle$

**Examples:**

<table>
<thead>
<tr>
<th>start</th>
<th>amount</th>
<th>is_not</th>
<th>final</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>70</td>
<td>is_not</td>
<td>50</td>
</tr>
<tr>
<td>50</td>
<td>50</td>
<td>is</td>
<td>0</td>
</tr>
<tr>
<td>50</td>
<td>30</td>
<td>is</td>
<td>20</td>
</tr>
</tbody>
</table>
A visual test report for example #1 (Balance = 50, Amount = 70)
A visual test report for example #3 (Balance = 50, Amount = 30)
The binding strategy uses the step only when required

Given("the customer's credit card already expired", () -> {
    when(paymentRetrieval.waitsAtServiceTask("chargeCard")).thenReturn((task) -> {
        task.handleBpmnError("creditCardExpired");
    });
});

Given("the customer does not update his credit card for (\d+) days and (\d+) hours", (days, hours) -> {
    when(paymentRetrieval.waitsAtReceiveTask("updateCard")).thenReturn((task) -> {
        task.defer("P" + days + "DT" + hours + "H", task::receive);
    });
});
Living documentation for long-running behaviour
Flows done right

Developer friendly

Lightweight & composable

BizDevOps
Flows are owned by services
Architecture

Order

Engine A

Order

Payment

Engine B

Human Task Management

Central

Monitoring

Fine grained monitoring & operations (per context)

Coarse grained central monitoring

DevOps

Tec Ops

Biz Ops
Example

Checkout

Order

Payment

Inventory

Shipping

Monitor

https://github.com/flowing/flowing-retail/
Operational visibility in monitoring – possibly end-to-end
Mitigate tight coupling

Consider core capabilities

Leverage state machines

Decentralize
Thank you!
Code online: https://github.com/flowing


With thoughts from http://flowing.io @berndruecker | @martinschimak
Images licensed from iStock
no attribution required

All icons licensed from Noun Project
no attribution required