TRANSACTIONAL SOLUTION FOR MICROSERVICES

ADAM RŮŽIČKA A ONDŘEJ CHALOUPKA
ABOUT US

ADAM RŮŽIČKA
- Red Hat
- developer of Foreman project
- mainly Ruby
- github: github.com/adamruzicka

ONDŘEJ CHALOUPKA
- Red Hat
- developer at WildFly project
- mainly Java
- working on Narayana transaction manager
- github: github.com/ochaloup
- twitter: @_chalda
AGENDA

- what is transaction management
- microservices and transactions
- introduction to saga pattern
- Long Running Actions for MicroProfile
- DynFlow framework
TRANSACTION

An atomic unit of work where all parts either finish with success or fail.
MONOLITHIC APPLICATION

1. Client call

2. Database call

3. Database call
LET'S CREATE A BOOKING

1. Booking order
2. Create order
3. Fill shipment
4. Confirmation

- Creating order
- Filling shipment
- Sending confirmation
...AND NOW WHAT?

- Creating order
- Filling shipment
- Sending confirmation
A SINGLE UNIT OF WORK

1. Booking order
2. Create order
3. Fill shipment
4. Confirmation

- Creating order
- Filling shipment
- Sending confirmation
ACID TRANSACTIONS TO RESCUE

1. Booking order
2. Create order
3. Fill shipment
4. Confirmation
5. Sending confirmation

- Creating order
- Filling shipment
- Sending confirmation
WORLD OF MICROSERVICES

1. Booking order
2. Create order
3. Fill shipment
4. Confirmation

- Creating order
- Filling shipment
- Sending confirmation
WORLD OF MICROSERVICES

1. Booking order
2. Create order
3. Fill shipment
4. Confirmation

- Creating order
- Filling shipment
- Sending confirmation
FAILURES HAPPENS

1. Booking order
2. Create order
3. Fill shipment
   CONNECTION ERROR
4. Confirmation
   CRASH
Every sufficiently large deployment of microservices contains an ad-hoc, informally-specified, bug-ridden, slow implementation of half of transactions.

Martin Kleppmann: Transactions: myths, surprises and opportunities
...AND NOW WHAT?

- just use ACID transactions
- but:
  - using locks
  - coupling microservices together
DISTRIBUTED XA TRANSACTION

Application → DATABASE INSERT → Database

Application → DATABASE INSERT → Database

Transaction Manager → TRANSACTION MANAGEMENT USING TWO-PHASE COMMIT → Database
ACID AND TWO-PHASE COMMIT

1. Application initiates a transaction.
2. Application inserts data into the database.

Transaction Manager

Database

Table: Order

<table>
<thead>
<tr>
<th>ID</th>
<th>ITEM</th>
<th>QUANTITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>BOOK ABOUT MSA</td>
<td>2</td>
</tr>
</tbody>
</table>
ACID AND TWO-PHASE COMMIT
ACID AND TWO-PHASE COMMIT

Application

START TRANSACTION

1

DATABASE INSERT

2

Transaction Manager

PREPARE

3

Transaction

COMMIT

4

Database

Table: Order

<table>
<thead>
<tr>
<th>ID</th>
<th>ITEM</th>
<th>QUANTITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>BOOK ABOUT MSA</td>
<td>2</td>
</tr>
</tbody>
</table>

Table: Order

<table>
<thead>
<tr>
<th>ID</th>
<th>ITEM</th>
<th>QUANTITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>BOOK ABOUT MSA</td>
<td>2</td>
</tr>
</tbody>
</table>

Table: Order

<table>
<thead>
<tr>
<th>ID</th>
<th>ITEM</th>
<th>QUANTITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>BOOK ABOUT MSA</td>
<td>2</td>
</tr>
</tbody>
</table>
...AND NOW WHAT?

- rollback to monolithic approach
- but:
  - agility
  - independence
  - scalability
  - easy to understand
  - fault isolation
SAGA PATTERN

a distributed domain transaction
SAGA PATTERN - THE BASIC IDEA

- break overall transaction into smaller steps
- steps can be performed in atomic transactions internally
- saga ensures that either the overall transaction is fully completed or the changes are undone
SAGA PATTERN
SAQA PATTERN

- first published in 1987
SAGA PATTERN

- first published in 1987
- intended for long running transactions in databases
SAGA PATTERN

- first published in 1987
- intended for long running transactions in databases
- good fit for microservices nowadays
- two main approaches to saga
  - orchestration
    - provides a good way of controlling the flow
    - an orchestrator tells participants what local transactions to execute
  - choreography
    - each local transaction publishes events that trigger local transaction in other services
LRA: LONG RUNNING ACTIONS

- Java based
- specification proposal for long running activities under Eclipse MicroProfile umbrella
  - [https://github.com/eclipse/microprofile-lra](https://github.com/eclipse/microprofile-lra)
- defines LRA coordinator
- over HTTP, LRA context is passed in HTTP headers
- definition for REST style endpoints
- implementation in project Narayana.io
DYNFLOW

- workflow engine written in Ruby
- currently in use by the Foreman project
- can do all sorts of stuff out of scope of this talk
  - running independent steps concurrently
  - polling external tasks
  - and much more
- support for Sagas in the form of rescue strategy
LRA VS. DYNFLOW

Long Running Actions

1. Client request
2. Enlist
3. Hotel booking service written in Ruby
4. Enlist
5. Close LRA
6. Confirm

Narayana LRA coordinator

DynFlow

1. Client request
2. Get status / work done
3. Hotel booking service written in Ruby
4. Get status / work done

Flight booking service written in Java
LRA VS DYNFLOW
LONG RUNNING ACTIONS
LONG RUNNING ACTIONS

@LRA
@NestedLRA

@Complete
@Compensate

@Leave
@Status

org.eclipse.microprofile.lra.client.LRAClient

startLRA()
closeLRA()
cancelLRA()
leaveLRA()
getStatus()

getAllLRAs()
getActiveLRAs()
getRecoveringLRAs()
DYNFLOW BUILDING BLOCKS

- **Actions**
  - have three phases - plan, run and finalize
  - can be composed

- **Execution plans**
  - are generated by planning actions
  - in our case a scope for transaction

- **Steps**
  - units of work
ACTION EXAMPLE

class BookHotel < ::Dynflow::Action
  include REST

  def run
    output[:response] = post_rest(input[:url])
  end
end

class BookTrip < ::Dynflow::Action
  def plan
    5.times { plan_action BookHotel, :url => 'http://hotel.california/book' }
  end
end
SAGAS IN DYNFLOW

- For an execution plan we know how all its steps finished
- If we know how to undo every single step, we can undo the entire execution plan
ROLLBACKS IN DYNFLOW

class BookHotel < ::Dynflow::Action
  include ::Dynflow::Action::Revertible
  include REST

  def run
    output[:response] = post_rest(input[:url], :parse_json => true)
  end

  def revert_run
    id = original_output.fetch(:response, {}).id
    post_rest(original_input[:url] + '/#{id}/compensate', :parse_json => false) if id
  end
end

class BookTrip < ::Dynflow::Action
  include ::Dynflow::Action::Revertible

  def plan
    5.times { plan_action BookHotel, :url => 'http://hotel.california/book' }
  end
end
SUMMARY

- Sagas are great solution for transactions in microservice deployments
  - if you're willing to loosen your requirements and go from strict atomicity to eventual consistency
QUESTIONS
• MicroProfile LRA specification: https://github.com/eclipse/microprofile-lra
• Community gitte: https://gitter.im/eclipse/microprofile-lra
• Blog posts: Narayana LRA: implementation of saga transactions, Saga implementations comparison
• Link to LRA demo: https://github.com/ochaloup/devconf2019-lra

• Dynflow: https://github.com/dynflow/dynflow
• Dynflow documentation: https://dynflow.github.io

THANK YOU!

redhat