Event-Sourcing Microservices on the JVM at the Norwegian Tax Authority
Logging events

- 2018/05/02 14:30 credited 100 to account
- 2018/05/02 18:15 credited 50 to account
- 2018/05/05 10:00 debitted 200 from account
- 2018/05/06 12:00 credited 150 to account
State auditing

credit 100
credit 50
debit 200
credit 150

Account
100

Audit
0
100
150
-50
Event sourcing

- credit 100
- credit 50
- debit 200
- credit 150

<table>
<thead>
<tr>
<th>Events</th>
</tr>
</thead>
<tbody>
<tr>
<td>credited 100</td>
</tr>
<tr>
<td>credited 50</td>
</tr>
<tr>
<td>debited 200</td>
</tr>
<tr>
<td>credited 150</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Snapshot</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
</tr>
</tbody>
</table>
Event sourcing: resetting snapshots

credit 100
credit 50
debit 200
credit 150

credited 100
credited 50
debited 200
credited 150

Events

Snapshot
100
Event sourcing: events and commands

credit 100
credit 50
debit 200
credit 150

credit 100
credited 100
credited 50
overrun
credited 150

Snapshot
300
Command query [responsibility] segregation (CQ[R]S)

- **Commands** do not query state
- **Queries** do not change state
Who is paying taxes in Norway?

- Passport
- Foreigner id
- Citizen id
- Relate
- Own
- Employ
- Own
- Locally registered
- International
“Partsregister”: tracking taxable entities in Norway

This is a schematic view only.
The promise of event-sourcing and our experience

- Event-sourcing allows you to easily change snapshot representation unless you did not sufficiently future-proof event capture
- Event-sourcing makes snapshots redundant by replaying events unless the event-processing code changes
- Event-sourcing implies full auditability of your application unless an error happens during command-to-event processing
- Event-sourcing offers an easy way of debugging applications unless events are trivial compared to command input
- Event-sourcing is an easy gateway to share-nothing architecture but only if you could shard your data in the first place

Disclaimer: our approach could be described as a combination of event sourcing and “command sourcing” with limited capability to scaling writes. But for us, this solution works great!
### Why “command sourcing”?

<table>
<thead>
<tr>
<th>fnr</th>
<th>Name</th>
<th>City</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>1105995521418</td>
<td>Some Man</td>
<td>Oslo</td>
<td>1503XXXXX185719...</td>
</tr>
<tr>
<td>1702842193749</td>
<td>Some Woman</td>
<td>Drammen</td>
<td>9456 A529184...</td>
</tr>
<tr>
<td>1105995521494</td>
<td></td>
<td></td>
<td>000000Drammen 0000XXXXX000000...</td>
</tr>
</tbody>
</table>

There are multiple rows for the same person due to different sources. The presented file formats are simplified for didactical reasons.

The presented file formats are simplified for didactical reasons.
Persist events for mistakes that need explicit correction

The presented file formats are simplified for didactical reasons.
Event-dependent state and sequence numbers

{ "fnr": "11059955214", "name": "Some Man", "city": "Oslo" }  
sequence: 1

{ "fnr": "17028421937", "name": "Some Woman", "city": "Drammen" }  
sequence: 2

{ "fnr": "11059955214", "name": "Some Man", "city": "Drammen" }  
sequence: 3

/part/9950174/1  
/part/9950174/2  
/part/9950174/3
Using sequence numbers for dealing with eventual consistency

{ "fnr": "11059955214", "name": "Some Man", "city": "Oslo" }

X-Sequence: 2

/rel/7573509

/part/9950174/2

/last-event: 3

event store

/last-event: 2

{ "owner": "9950174" }
Using sequence numbers for dealing with eventual consistency

BAD REQUEST:
{
    "sequence": "2"
}

X-Sequence: 3

/part/9950174/3

/event store

last-event: 2

last-event: 3

/rel/7573509
Observing event-processing of distributed services
Publishing thin change feeds to expose application state

- Event store

- Information icon

- RSS feed icon

Sequence 1:
- 9950174
  - 11059955214
    - Some Man
    - Oslo
  - /part/9950174/1

Sequence 2:
- 294851
  - 17028421937
    - Some Woman
    - Drammen
  - /part/294851/2

Sequence 3:
- 9950174
  - 11059955214
    - Some Man
    - Drammen
  - /part/9950174/3
Revisioning aggregates for idempotency

11059955214
Some Man
Oslo

17028421937
Some Woman
Drammen

11059955214
Some Man
Drammen

/part/9950174/1
/part/9950174/1/1

/part/294851/2
/part/294851/2/1

/part/9950174/3/1
/part/9950174/3[/2]

/part/294851/2/1
/part/294851/2/1

/part/9950174/3/1
/part/9950174/3[/2]

/part/9950174/3/1
/part/9950174/3[/2]
Publishing revisions in a feed

9950174 sequence 1 revision 1

294851 sequence 2 revision 1

9950174 sequence 3 revision 1

9950174 sequence 3 revision 2
Using the event store as a single source of truth

Advantages of "command sourcing":
1. Self-healing state after any bug fix without any user management.
2. Only command-to-event mapping is domain-specific code.
3. Minimal probability to misinterpret events after updates.

Downside: command-to-event processing must be stateless to allow reprocessing. Revision-sensitive event observers can often remedy this limitation.
Deleting events and compaction events

Why would you want to delete events?
1. Because you want.
   Storage space is not free after all.
2. Because you should.
   Storing obsolete personal data makes you a target for attackers and is immoral.
3. Because you have to.
   Laws like the GDPR demand physical erasure.
Deleting events with tombstones

Tombstones must not be deleted themselves to allow for propagation to all services. For this reason, it is crucial to choose primary identifiers that do not contain personal data (unlike a fødselsnummer). Ideally, an internal, synthetic identifier is used as a proxy for each personal identifier.
Compacting events with compaction events

event store

11059955214
Some Man
Oslo

17028421937
Some Woman
Drammen

11059955214
Some Man
Drammen

[compaction: 3]

Can be represented by same database entity.
What is out there?

API-wapper for MongoDB. Originates from the .NET space. Java client but Scala-oriented.

Java framework for CQRS. Strict command and event separation. Support for JDBC-integration.

Append-only database. Only recently published.

DIY at Skatteetaten.

Reasons for choice:
1. Performance.
   Streaming has a high overhead for mass processing. Need for microbatching to allow for microservice orchestration.
2. Complexity
   Event sourcing is not yet mainstream. APIs feel often immature. Event stores often aim for distributability at the cost of simplicity.
3. Loose command-to-aggregate mapping
   Many frameworks assume that there exists an obvious mapping of any command to an aggregate.
Events and event stores

class Event {
    long sequence; // 0 if not set
    String uid;
    String id;
    String type; // XML namespace id
    String value; // XML
}

interface EventStore {
    Stream<Event> read(long afterSequence);
    ClosableConsumer<Event> write();
}

EventStore source, target;
try (Stream<Event> stream = source.read(0)) {
    ClosableConsumer<Event> consumer = target.write();
    stream.forEach(consumer);
}
Events and event stores

class SQLEventStore implements EventStore
class InMemoryEventStore implements EventStore
class HttpEventStore implements EventStore

LOCK TABLE events;

INSERT INTO events (sequence, uid, id, type, value)
SELECT seq.NEXTVAL, ?, ?, ?, ?, ?
FROM dual
WHERE ? NOT IN (SELECT uid FROM events)

SELECT /*+ index(events seq) */ */
FROM events
WHERE seq > 0
FETCH FIRST 1000 ROWS ONLY
Aggregates and aggregate stores

```java
interface AggregateStore {
    Optional<String> read(String id, long sequence);
}

interface WriteableAggregateStore extends AggregateStore {
    void write(String id, long sequence, String aggregate);
}
```

```java
EventStore source;
AggregateStore target;
try (Stream<Event> stream = source.read(0)) {
    stream.forEach(event -> {
        String aggregate = target.read(event.id, event.sequence)
            .map(aggregate -> Domain.updateAggregate(aggregate, event.value))
            .orElse(() -> Domain.newAggregate(event.value));
        target.write(event.id, event.sequence, aggregate);
    });
}
```
Aggregates and aggregate stores

class SQLAggregateStore implements WriteableAggregateStore
class InMemoryAggregateStore implements WriteableAggregateStore
class HttpAggregateStore implements AggregateStore

```
INSERT INTO aggregates (sequencee, id, valuee)
VALUES (?, ?, ?)

SELECT s.id, s.value
FROM aggregates s
INNER JOIN (
  SELECT MAX(sequence) ms, id
  FROM aggregates
  WHERE sequence <= ?
  GROUP BY id
) t
ON s.id = t.id
AND s.sequence = t.ms
WHERE id = ?
```
Testing

```
<events>
  <event>
    <id>fnsdjFD94d</id>
    <type>sample-event</type>
    <value>some-event</value>
  </event>
</events>
```

```json
{
  "state": "some-event"
}
```
Test-automation

```yaml
# example.yml

test: example
timeout: 10000
applications:
  - eventstore
  - identity-management
  - folkeregister-export
given:
  - application: eventstore
    POST: ./some-events.xml
when:
  - application: folkeregister-export
    GET: /info/sequence
    text: 10
then:
  - application: folkeregister-export
    GET: /part/947652
    json: ./some-result.json

~$ part-test-cli example.yml

PartTestRunner extends JUnitRunner
```
Concept

Implementation

Operation
The event store as a bottleneck
Scaling reads by event store replication

writer 1

writer 2

event store

event store mirror 1

event store mirror 2

reader 1

reader 2

reader 3

reader 4
Scaling reads by splitting reader responsibility

writer 1

writer 2

event store

reader 1 (aggregator)

reader 2 (aggregator)

reader 1 (API)

reader 2 (API)
Scaling writes via buffers (with priority)

writer 1 -> buffer 1 -> event store -> reader 1

writer 2 -> buffer 2 -> event store -> reader 2