Patterns and Practices for Real-world Event-driven Microservices

Rachel Reese
@rachelreese

Jet.com
@JetTechnology
Real-world
Taking on Amazon!

Launched July 22
- Featured: app store & android store
- Over 20k orders per day
- Over 7 million SKUs
- 600k first-time buyers
- 500k mobile downloads

Start your cart. Build your savings.

Build a Smart Cart

We’re hiring!
http://jet.com/about-us/working-at-jet
<table>
<thead>
<tr>
<th>Azure</th>
<th>Web sites</th>
<th>Cloud services</th>
<th>VMS</th>
<th>Service bus queues</th>
<th>Services bus topics</th>
<th>Blob storage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table storage</td>
<td>Queues</td>
<td>Hadoop</td>
<td>DNS</td>
<td>Active directory</td>
<td>SQL Azure</td>
<td>R</td>
</tr>
<tr>
<td>F#</td>
<td>Paket</td>
<td>FSharp.Data</td>
<td>Chessie</td>
<td>Unquote</td>
<td>SQLProvider</td>
<td>Python</td>
</tr>
<tr>
<td>Deedle</td>
<td>FAK E</td>
<td>FSharp.Async</td>
<td>React</td>
<td>Node</td>
<td>Angular</td>
<td>SAS</td>
</tr>
<tr>
<td>Storm</td>
<td>Elastic Search</td>
<td>EventStore</td>
<td>Microservices</td>
<td>Consul</td>
<td>Kafka</td>
<td>PDW</td>
</tr>
<tr>
<td>Splunk</td>
<td>Redis</td>
<td>SQL</td>
<td>Puppet</td>
<td>Jenkin s</td>
<td>Apache Hive</td>
<td>Apache Tez</td>
</tr>
</tbody>
</table>
Microservices
Microservices

An application of the single responsibility principle at the service level.

“A class should have one, and only one, reason to change.”

Has an input, produces an output.

Benefits

- Easy scalability
- Independent releasability
- More even distribution of complexity
Event-driven
The focus of your application code is to react to events (single or a stream).
Events

Any significant change in state that has happened in your domain

- Past tense
- Immutable
- Contains only relevant data to transaction

All events should be represented as verbs in the past tense such as CustomerRelocated, CargoShipped, or InventoryLossageRecorded.

For those who speak French, it should be passé composé, they are things that have completed in the past.

Greg Young
“Events” as the notification messages

Event Emitter

Event channel

Event Consumer
Does anyone here use Rx?

Event-driven is Reactive

Events should be an Observable

Microservices should be Observables
Event-sourced

- Event-sourced is about how you model the domain.
- An append-only sequence of events as data store.
- Keep track of all state changes.
- Can REPLAY these event streams.

Compare to relational model which captures only the latest state change. These sets are then related to each other.
# Event Stream

## FIRST BANK OF WIKI
1425 JAMES ST, PO BOX 4000
VICTORIA BC V8X 3X4  1-800-555-5555

JOHN JONES
1643 DUNDAS ST W APT 27
TORONTO ON M6K 1V2

<table>
<thead>
<tr>
<th>Date</th>
<th>Description</th>
<th>Ref.</th>
<th>Withdrawals</th>
<th>Deposits</th>
</tr>
</thead>
<tbody>
<tr>
<td>2003-10-14</td>
<td>Payroll Deposit - HOTEL</td>
<td></td>
<td></td>
<td>694.81</td>
</tr>
<tr>
<td>2003-10-14</td>
<td>Web Bill Payment - MASTERCARD</td>
<td></td>
<td>200.00</td>
<td></td>
</tr>
<tr>
<td>2003-10-16</td>
<td>ATM Withdrawal - INTERAC</td>
<td>3990</td>
<td>21.25</td>
<td></td>
</tr>
<tr>
<td>2003-10-16</td>
<td>Fees - Interac</td>
<td></td>
<td>1.50</td>
<td></td>
</tr>
<tr>
<td>2003-10-20</td>
<td>Interac Purchase - ELECTRONICS</td>
<td>1975</td>
<td>2.99</td>
<td></td>
</tr>
<tr>
<td>2003-10-21</td>
<td>Web Bill Payment - AMEX</td>
<td>3314</td>
<td>300.00</td>
<td></td>
</tr>
<tr>
<td>2003-10-22</td>
<td>ATM Withdrawal - FIRST BANK</td>
<td>0004</td>
<td>100.00</td>
<td></td>
</tr>
<tr>
<td>2003-10-23</td>
<td>Interac Purchase - SUPERMARKET</td>
<td>1559</td>
<td>29.08</td>
<td></td>
</tr>
<tr>
<td>2003-10-24</td>
<td>Interac Refund - ELECTRONICS</td>
<td>1975</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2003-10-27</td>
<td>Telephone Bill Payment - VISA</td>
<td>2475</td>
<td>6.77</td>
<td></td>
</tr>
<tr>
<td>2003-10-28</td>
<td>Payroll Deposit - HOTEL</td>
<td></td>
<td></td>
<td>694.81</td>
</tr>
<tr>
<td>2003-10-30</td>
<td>Web Funds Transfer - FROM SAVINGS</td>
<td>2920</td>
<td>50.00</td>
<td></td>
</tr>
<tr>
<td>2003-11-03</td>
<td>Pre-Authorized Payment - INSURANCE</td>
<td></td>
<td>33.55</td>
<td></td>
</tr>
<tr>
<td>2003-11-03</td>
<td>Cheque No. - 409</td>
<td></td>
<td>100.00</td>
<td></td>
</tr>
<tr>
<td>2003-11-06</td>
<td>Mortgage Payment</td>
<td></td>
<td>716.48</td>
<td></td>
</tr>
<tr>
<td>2003-11-07</td>
<td>Fees - Overdraft</td>
<td></td>
<td>5.00</td>
<td></td>
</tr>
<tr>
<td>2003-11-08</td>
<td>Fees - Monthly</td>
<td></td>
<td>5.00</td>
<td></td>
</tr>
</tbody>
</table>

*** Totals ***
1,515.03  1,442.01
Guidelines
Don’t abstract

This one magical service could write to ALL of the following:

- Event store
- Nservice Bus
- MSMQ
- 0MQ
- SQL Server

…badly.
Be functional!

- Prefer immutability
  • Avoid state changes, side effects, and mutable data

- Use data in → data out transformations
  • Think about mapping inputs to outputs.

- Look at problems recursively
  • Consider successively smaller chunks of the same problem

- Treat functions as unit of work
  • Higher-order functions
Isolate side effects

Submit order microservice
  Updates SQL
  Sends “Thank you for ordering” Email

Insert order to SQL microservice
  Updates SQL

Send thank you email microservice
  Sends “Thank you for ordering” Email
Use a backup service to replay events

Service 1 runs in production as normal

Backup service 1 replays events until up-to-date. Switch over. Instantly live with changes!

Also stage a copy of any data store until stream has completed replaying!
What do our services look like?

Define inputs & outputs

type Input =
  | Product of Product

type Output =
  | ProductPriceNile of Product * decimal
  | ProductPriceCheckFailed of PriceCheckFailed

Define how input transforms to output

let handle (input: Input) =
  async {
    return Some(ProductPriceNile({Sku="343434"; ProductId = 17; ProductDescription = "My amazing product"; CostPer=1.96M}, 3.96M))
  }

Define what to do with output

let interpret id output =
  match output with
  | Some (Output.ProductPriceNile (e, price)) -> async {} // write to event store
  | Some (Output.ProductPriceCheckFailed e) -> async {} // log failure
  | None -> async.Return ()

Read events, handle, & interpret

let consume =
  EventStoreQueue.consume (decodeT Input.Product) handle interpret
  |> catchLogThrow Log
Microservices should not control their own lifecycle.

- Execution runtime
- Deployment
- Configuration
- Restarting
- Versioning
- Scaling
- Availability
- Grouping by subsystem
- Scheduling
- Input-output static analysis
- Dashboard

Think IoC!
Torch YAML files

torchVer: 2.0.0
subSystem: PriceCheck

name: PriceCheck
description: checks prices on nile
ver: 0.0.1

autoStart: always
compile: true
ha: aa  ##active-active. Could be ap for active-passive

scriptPath: PriceCheckNile\PriceCheckNile.fsx
libPath: bin\release\

args: --jsonConfig=PriceCheckNile.json

YAML = “YAML Ain’t Markup Language”

It used to mean “Yet Another Markup Language” but was backronymed to clarify its focus as data-oriented.
Summary

- Don’t abstract
- Be functional
- Isolate side effects
- Use a backup service
- Use consistent formatting
- Use an outside product to control lifecycle
Patterns and Practices for Real-world Event-driven Microservices

Rachel Reese
@rachelreese
Jet.com
@JetTechnology