A competitive food retail architecture with microservices

A four year journey towards microservices / Java Forum Stuttgart 05.07.2018

Ansgar Brauner & Sebastian Gauder
@a_brauner & @rattakresch
Details REWE GROUP

Turnover
>54 bn

Employees
>330,000

Shops
>15,000

Industries
Food Retail,
Tourism,
DIY

History
90 years
Our history (Spoiler alert)

- **2014**: Takeover of monolith (2 teams)
- **2015**: Squad concept & microservices
- **2016**: Introduction of docker / swarm, App Launch μService only (~20 teams), splitting into eCom, fulfillment and big data platform (~30 teams)
- **2017**: Beta launch rewe.de marketplace (~50 teams)
- **2018**: Go live rewe.de marketplace (~50 teams), launch food fulfillment center 2.0
Our history

- **2014**: 1 service, 2 teams
- **2015**: 40 services, 15 teams
- **2016**: 100 services, 28 teams
- **2017**: 200 services, 51 teams
Organization and Architecture at Rewe Digital

- **Domain**: Consists of 1 Subdomain, and consists of N Bounded Contexts. Fulfills requirements of 1 Platform.
- **Subdomain**: Consists of 1 Team, and consists of N Services. Is modelled as 1 Bounded Context.
- **Platform**: Consists of 1 Tribe and builds 1 Subdomain.
- **Tribe**: Consists of 1 Squad and builds 1 Platform.
- **Squad**: Consists of 1 Team and builds 1 Subdomain.
- **Team**: Consists of 1 Service and builds 1 Bounded Context.
How to scale monoliths?
Design Goals

- Decentralization
- Vertical Boundaries
Conway’s law

“All organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization’s communication structure.”

Melvin Conway (1967)
How to organize teams to build a scalable system?

Functional and vertical teams

<table>
<thead>
<tr>
<th>Team A</th>
<th>Team B</th>
<th>Team C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frontend</td>
<td>Frontend</td>
<td>Frontend</td>
</tr>
<tr>
<td>QA</td>
<td>QA</td>
<td>QA</td>
</tr>
<tr>
<td>Analytics</td>
<td>Analytics</td>
<td>Analytics</td>
</tr>
<tr>
<td>Middleware</td>
<td>Middleware</td>
<td>Middleware</td>
</tr>
<tr>
<td>Backend</td>
<td>Backend</td>
<td>Backend</td>
</tr>
<tr>
<td>Ops</td>
<td>Ops</td>
<td>Ops</td>
</tr>
</tbody>
</table>
How to determine boundaries?
Customer journey defines subdomains
How to guide developers?
Guarding rails for developers

Design Goals
- Vertical boundaries
- Decentralization

Architectural principles
- Collection of 9 basic ‘laws’
- Autonomy, Automation and Communication

Guides
- Practical manual for common tasks (RFC 2119) MUST
- e.g. Eventing, REST, Authentication SHOULD
- COULD
autonomy
Autonomy principles

Deploy independently
Ensure that the services can and are deployed by the teams themselves.

Isolate failure
Make the services as resilient as possible.

Hide implementation Details
Different verticals must not share state. Verticals hide their implementation details.

Encapsulate Data Storage
For any data resource exactly one service is responsible. If possible, the data supply should proceed asynchronously in the background.
Challenges

Autonomous Teams
- challenge for product owners as they have to rethink their features
  - they might have to reduce the scope
  - split the original feature into smaller ones for each team / bounded context

Data-integration pattern
- Change your service behavior depending on the data you have to display
- Enable others to deliver functionality later

Evolving Implementations
- Start with a fitting but simple solution and get better by every team implementing it
Autonomy opens your tech stacks
automation
Automation principles

Be Scalable
Services are scaled horizontally

Embrace a Culture of Automation
Test, deployment and operations are completely automated

Be Highly Observable
Use of semantic monitoring to see if the whole platform works correctly.
communication
Communication principles

Follow REST Principles
The API of a service follows the RESTful paradigm (REST maturity level 2)

Standardize Service communication
Inter-service communication is standardized and if possible asynchronous
Problems in HTTP/REST-only architectures

- Timeout
- Fallbacks
- Circuit Breakers
- Eventing
Communication in microservice worlds
Having data is better than fetching data

Asynchronous > Synchronous
- Have as much needed data locally as possible -> resilience at request time
- Duplication of data is accepted and wanted
- Problem of eventual consistency
- Only use synchronous communication if the process is vulnerable to race conditions
Eventing with Apache Kafka

- **Eventing != Messaging**
  - Publish events that already **happened**
  - One owning service per queue/topic
  - Autonomy at request time

- **Complete entities - not deltas**
  - Transactional completeness
  - Allows log compaction

- **Re-writing and Re-reading**
  - needed in case of additional data
  - useful in case of data loss
Lessons learned with microservice APIs

Eventing is an API as well
- Events have to behave like APIs and avoid breaking changes.

Writing APIs is hard
- Teams tend to write APIs for special clients (e.g. mobile apps)
- New clients often don’t match and require API changes

Breaking changes require a strict procedure
- Could be solved by new endpoints, new topics
How to access?
Usage of API gateways as entry points

How to access Microservices?
Dynamic frontend composition
How UIs in microservice architectures should be composed
Dynamic UI composition with DUC
Where does my UI come from?
Thank you

Questions ?
A competitive food retail architecture with microservices

A three four year journey towards microservices / Meetup 21.03.

Sebastian Gauder / @rattakresch