DOMAIN-SPECIFIC SERVICE DECOMPOSITION WITH MICROSERVICE API PATTERNS

Keynote, International Conference on Microservices 2019
Dortmund, Germany
February 19, 2019
Prof. Dr. Olaf Zimmermann (ZIO)
Certified Distinguished (Chief/Lead) IT Architect
Institute für Software, HSR FHO
ozimmerm@hsr.ch
Abstract

Service orientation is a key enabler for cloud-native application development. Microservices have emerged as a state-of-the-art implementation approach for realizations of the Service-Oriented Architecture (SOA) style, promoting modern software engineering and deployment practices such as containerization, continuous delivery, and DevOps.

Designing (micro-)services interfaces to be expressive, responsive and evolvable is challenging. For instance, deciding for suited service granularities is a complex task resolving many conflicting forces; one size does not fit all. Domain-Driven Design (DDD) can be applied to find initial service boundaries and cuts. However, service designers seek concrete, actionable guidance going beyond high-level advice such as “turn each bounded context into a microservice”. Interface signatures and message representations need particular attention as their structures influence the service quality characteristics.

This presentation first recapitulates prevalent SOA principles, microservices tenets and DDD patterns. It then reports on the ongoing compilation of complementary microservices API patterns and proposes a set of pattern-based, tool-supported API refactorings for service decomposition. Finally, the presentation highlights related research and development challenges.
Architecture of this Talk (“Micropresentations”)

- Mythbusting
- SOA 101 & Microservices Tenets
- Service Analysis & Design (Modeling)
- Introduction to Domain-Driven Design
- Service Granularity and Loose Coupling
- Microservice API Patterns (MAP)
- Real-World Service Examples (Case Studies)
- Architectural Refactoring (to Microservices)

Legend:
- Experience
- Literature Analysis
- Patterns
- Research Pbs/Qs
- Opinions
Sample Project: Financial Services Provider (for Retail Banks)

- Supports – and partially automates – core banking business processes
  - More than 1000 of business services, each providing a single operation
  - One database repository, logically partitioned

Reference: IBM, ACM OOPSLA 2004
## Exemplary Service Operations in Core Banking

<table>
<thead>
<tr>
<th>Fine (business)</th>
<th>Coarse (business)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Fine (technical)</strong></td>
<td>“Hello world” of core banking: int getAccountBalance (CustomerId)</td>
</tr>
<tr>
<td><strong>Coarse (technical)</strong></td>
<td>Single domain entity, but complex payload (search/filter capability): CustomerDTOSet searchCustomers (WildcardedCustomerName, CustomerSegment, Region)</td>
</tr>
</tbody>
</table>

### Business granularity:
- Functional scope, domain model coverage

### Technical granularity:
- Structure of message representations a.k.a. Data Transfer Object (DTOs)

---

© Olaf Zimmermann, 2019.
Sample Project: Order Management *Application* (Telecommunications)

**Multi-Channel Order Management SOA in the Telecommunications Industry (in production since Q1/2005) [OOPSLA 2005]**

- **Functional domain**
  - Order entry management
  - Two business processes: new customer, relocation
  - Main SOA drivers: deeper automation grade, share services between domains

- **Service design**
  - Top-down from requirement and bottom-up from existing wholesaler systems
  - Recurring architectural decisions:
    - Protocol choices
    - Transactionality
    - Security policies
    - Interface granularity

Reference: IBM, ECOWS 2007

**Interface granularity (WSDL contract design)? Message- or transport layer encryption?**

**Transaction boundaries inside process? Which BPM/workflow engine to use?**

**Message exchange pattern? Transport protocol?**
Exemplary Services in Order Management (Telecommunications)

- Endpoints play different roles in microservices architectures – and their operations fulfill certain responsibilities:
  - Pre- and postconditions
  - Conversational state
  - Data consistency vs. currentness

Impact on scalability and changeability?
What is Service-Oriented Architecture (SOA)?

No single definition – “SOA is different things to different people”:

- A set of **services** and operations that a business wants to expose to their customers and partners, or other portions of the organization.
  - *Note: no scope implied, enterprise-wide or application!*
- An architectural style which requires a **service provider**, a **service requestor** (consumer) and a **service contract** (a.k.a. client/server).
  - *Note: this is where the “business-alignment” becomes real!*
- A set of architectural patterns such as **service layer** (with remote facades, data transfer objects), enterprise service bus, service composition (choreography/orchestration), and service registry, promoting principles such as modularity, layering, and **loose coupling** to achieve design goals such as reuse, and flexibility.
  - *Note: not all patterns have to be used all the time!*
- A **programming and deployment model** realized by standards, tools and technologies such as Web services (WSDL/SOAP), RESTful HTTP, or asynchronous message queuing (AMQP etc.)
  - *Note: the “such as” matters (and always has)!*

Based on and adapted from: IBM SOA Solution Stack, IBM developerWorks
"Napkin Sketch" of SOA Realizations (Adopted from G. Hohpe)

Our focus today

Microservices!

No longer popular
(term repurposed for deployment context)
Seven Microservices Tenets (by Viewpoint)

- Business Alignment (e.g. via DDD)
- Independent-X (X = Deployment, Scaling, Change)
- IDEAL Cloud Architectures (e.g. 12-Factor App)
- Polyglot Programming and Persistence
- Decentralization & Automation (CI/CD)
- Service Monitoring (DevOps Way)
- Containerization and Clustering

Legend:
- Architecture
- Analysis, Design & Coding
- Deployment & Runtime

well-known

fairly recent advances

Microservices Tenets: Agile Approach to Service Development and Deployment
Olaf Zimmermann
Computer Science - Research and Development (ShareLink: http://rdcu.be/mJPz...

DOI 10.1007/s00450-016-0337-0

SPECIAL ISSUE PAPER

Page 10
© Olaf Zimmermann, 2019.
Cloud-native application architectures are API-centric

IDEAL: Isolated State, Distribution/Decomposition, Elasticity, Automation, Loose Coupling

http://www.cloudcomputingpatterns.org
Calls to Service Operations are EIP-style Messages

Sample request message (note: PUTs and POSTs would look different)

Response message structure

{[…]} -- some JSON (or other MIME type)

How to find suited granularities and achieve loose coupling?

Context
We have decided to go the SOA and/or microservices way. We use DDD for domain modeling and agile practices for requirements elicitation.

Problems (Industry, Academia)
How to identify an adequate number of API endpoints and operations?

How to design (command/document) message representation structures so that API clients and API providers are loosely coupled and meet their (non-) functional requirements IDEALy?

Which patterns, principles, and practices do you use? Do they work?
Introducing… Microservices API Patterns (MAP)

- **Identification Patterns:**
  - DDD as one practice to find candidate endpoints and operations

- **Foundation Patterns**
  - What type of (sub-)systems and components are integrated?
  - Where should an API be accessible from?
  - How should it be documented?

- **Structure Patterns**
  - What is an adequate number of representation elements for request and response messages?
  - How are these elements structured?
  - How can they be grouped and annotated with usage information?

- **Responsibility Patterns**
  - Which is the architectural role played by each API endpoint and its operations?
  - How do these roles and the resulting responsibilities impact (micro-)service size and granularity?

- **Quality Patterns**
  - How can an API provider achieve a certain level of quality of the offered API, while at the same time using its available resources in a cost-effective way?
  - How can the quality tradeoffs be communicated and accounted for?

- **Evolution Patterns:**
  - Work in progress (EuroPLoP 2019?)

http://microservice-api-patterns.org
Context

An API endpoint and its calls have been identified and specified.

Problem

How can an API provider optimize a response to an API client that should deliver large amounts of data with the same structure?

Forces

- Data set size and data access profile (user needs), especially number of data records required to be available to a consumer
- Variability of data (are all result elements identically structured? how often do data definitions change?)
- Memory available for a request (both on provider and on consumer side)
- Network capabilities (server topology, intermediaries)
- Security and robustness/reliability concerns
**Solution**

- Divide large response data sets into manageable and easy-to-transmit chunks.
- Send only partial results in the first response message and inform the consumer how additional results can be obtained/retrieved incrementally.
- Process some or all partial responses on the consumer side iteratively as needed; agree on a request correlation and intermediate/partial results termination policy on consumer and provider side.

**Variants**

- Cursor-based vs. offset-based

**Consequences**

- E.g. state management required

**Know Uses**

- Public APIs of social networks
Quality-related decision model published at ICSOC 2018

Avoid Unnecessary Data Transfers

<table>
<thead>
<tr>
<th>Decision Criteria</th>
<th>Options</th>
</tr>
</thead>
<tbody>
<tr>
<td>Client Information Needs</td>
<td>1. 📌 Wish List</td>
</tr>
<tr>
<td>Network bottlenecks</td>
<td>2. 📌 Wish Template</td>
</tr>
<tr>
<td>Performance</td>
<td>3. 📌 Conditional Request</td>
</tr>
<tr>
<td>Security</td>
<td>4. 📌 Request Bundle</td>
</tr>
<tr>
<td>Development and Testing Complexity</td>
<td></td>
</tr>
</tbody>
</table>

More problem-pattern mappings (emerging):
- MAP Cheat Sheet: [https://microservice-api-patterns.org/cheatsheet](https://microservice-api-patterns.org/cheatsheet)
- Attribute-Driven Design: [https://microservice-api-patterns.org/patterns/byforce](https://microservice-api-patterns.org/patterns/byforce)
## More Decisions that Recur in (Micro-)Service Design

### ISSUE

<table>
<thead>
<tr>
<th>API clients report interoperability and usability problems</th>
<th>Switch from minimal to full API DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>My clients report performance problems</td>
<td>Add <strong>META</strong>DATA <strong>ELEMENT</strong> to <strong>PARAMETER TREE</strong> to realize an <strong>ANNOTATED PARAMETER COLLECTION</strong></td>
</tr>
<tr>
<td></td>
<td>Switch from <strong>EMBEDDED ENTITIES</strong> to <strong>LINKED INFORMATION HOLDERS</strong></td>
</tr>
<tr>
<td></td>
<td>Reduce transferred data with a <strong>WISH LIST</strong> or a <strong>WISH TEMPLATE</strong></td>
</tr>
<tr>
<td></td>
<td>Consider any other <strong>QUALITY PATTERN</strong> improving data transfer parsimony (e.g., <strong>CONDITIONAL REQUEST, REQUEST BUNDLE</strong>))</td>
</tr>
<tr>
<td></td>
<td>Introduce <strong>PAGINATION</strong></td>
</tr>
<tr>
<td>I need to implement some access control</td>
<td>Introduce <strong>API KEYS</strong> or full-fledged security (CIA/IAM) solution such as OAuth</td>
</tr>
</tbody>
</table>

### Patterns to Consider

- Use **ATOMIC PARAMETER LIST** and/or **ATOMIC PARAMETER LIST** if data is simple
- Use **PARAMETER TREE** and/or **PARAMETER FOREST** if data is complex
- Add **ENTITY ELEMENT** with one or more **EMBEDDED ENTITIES** (following relationships)
- Add **ID ELEMENT**
- Upgrade from **ID ELEMENT** to **LINK ELEMENT** to support HATEOAS and reach REST maturity level 3

---

© Olaf Zimmermann, 2019.
Research Questions

Which existing patterns are particularly suited to analyze and design cloud-native applications and to modernize existing systems (monoliths/megaliths)? How can these patterns be combined with Microservices API Patterns (MAP) and other SOA/microservices design heuristics to yield a service-oriented analysis and design practice?

Which patterns and practices do you apply? What are your experiences?
Insurance scenario, source: [https://contextmapper.github.io/](https://contextmapper.github.io/)
What is Context Mapper?

Context Mapper provides a DSL to create context maps based on Domain-driven Design (DDD) and its strategic patterns. DDD and its bounded contexts further provide an approach for decomposing a domain into multiple bounded contexts. With our Service Cutter integration we illustrate how the Context Mapper DSL (CML) can be used as a foundation for structured service decomposition approaches. Additionally, our context maps can be transformed into PlantUML diagrams.

- **Eclipse plugin Based on:**
  - Xtext
  - ANTLR
  - Sculptor (tactic DDD DSL)

- **Author: S. Kapferer**
  - Term project HSR FHO
M. Ploed is one of the “go-to-guys” here (find him on Speaker Deck)

Applies and extends DDD books by E. Evans and V. Vernon
Implementing Domain-Driven Design with RESTful HTTP APIs

- **Mentioned in DDD book by V. Vernon (and blog posts, presentations):**
  - No 1:1 pass-through (interfaces vs. application/domain layer)
  - **Bounded Contexts (BCs)** offered by API provider, one API endpoint and IDE project for each team/system BC (a.k.a. microservice)
  - **Aggregates** supply API resources or (responsibilities of) microservices
  - Services donate top-level (home) resources in BC endpoint as well
  - The Root Entity, the Repository and the Factory in an Aggregate suggest top-level resources; contained entities yield sub-resources
  - Repository lookups as paginated queries (GET with search parameters)

- **Additional rules of thumb (own experience, literature):**
  - Master data and transactional data go to different BCs/aggregates
  - Creation requests to Factories become POSTs
  - Entity modifiers become PUTs or PATCHes
  - Value Objects appear in the custom mime types representing resources
Open Problem: Service Decomposition

Research Questions
How can systems be decomposed into services (in forward engineering)?
How do the applied criteria and heuristics differ from software engineering and software architecture “classics” such as separation of concerns and single responsibility principle?

Which methods and practices do you use? Are they effective and efficient?
Heuristics that do not suffice (IMHO)

- Two-pizza rule (team size)
- Lines of code (in service implementation)
- Size of service implementation in IDE editor

What is wrong with these “metrics” and “best practice” recommendations?

- Simple if-then-else rules
  - E.g. “If your application needs coarse-grained services, implement a SOA; if you require fine ones, go the microservices way” (I did not make this up!)

- Non-technical traits such as “products not projects”
  - Because context matters, as M. Fowler pointed out at Agile Australia 2018
Agility, Consistency, State/Scalability (CAS) Tradeoffs

"Conservative" SOA (Macroservices)

- Big data requirements
  - Sharding, partitioning
  - Strict & eventual consistency

- Audit requirements
  - Incl. backup

ACS Dichotomy

- Data freshness
- Ability to respond to change

Microservices

- State management
  - Quick access, caching?
  - Stickiness in cluster?

Resource Consistency

Modular Monolith

Scalable State Mgmt.
A Software Architect’s Dilemma….

How do I split my system into services?

Step 1: Analyze System
- Entity-relationship model
- Use cases
- System characterizations
- Aggregates (DDD)

Coupling information is extracted from these artifacts.

Step 2: Calculate Coupling
- Data fields, operations and artifacts are nodes.
- Edges are coupled data fields.
- Scoring system calculates edge weights.
- Two different graph clustering algorithms calculate candidate service cuts (=clusters).

Step 3: Visualize Service Cuts
- Priorities are used to reflect the context.
- Published Language (DDD) and use case responsibilities are shown.

Technologies:
Java, Maven, Spring (Core, Boot, Data, Security, MVC), Hibernate, Jersey, JHipster, AngularJS, Bootstrap

https://github.com/ServiceCutter
### Coupling Criteria (CC) in “Service Cutter” (Ref.: ESOCC 2016)

**Cohesiveness**
- **Semantic Proximity**
- **Identity & Lifecycle Commonality**
- **Security Contextuality**
- **Shared Owner**
- **Latency**

**Compatibility**
- **Structural Volatility**
- **Consistency Criticality**
- **Storage Similarity**
- **Content Volatility**
- **Availability Criticality**
- **Security Criticality**

**Constraints**
- **Consistency Constraint**
- **Security Constraint**
- **Predefined Service Constraint**

**Communication**
- **Mutability**
- **Network Traffic Suitability**


- **E.g. Semantic Proximity can be observed if:**
  - Service candidates are accessed within same use case (read/write)
  - Service candidates are associated in OOAD domain model

- **Coupling impact (note that coupling is a relation not a property):**
  - Change management (e.g., interface contract, DDLs)
  - Creation and retirement of instances (service instance lifecycle)
Open Research Problem: Refactoring to Microservices

Research Questions

How to migrate a modular monolith to a services-based cloud application (a.k.a. cloud migration, brownfield service design)?
Can “micro-migration/modernization” steps be called out?

Which techniques and practices do you employ? Are you content with them?
Refactoring are “small behavior-preserving transformations” (M. Fowler 1999)

Code refactorings, e.g. “extract method”
- Operate on Abstract Syntax Tree (AST)
- Based on compiler theory, so automation possible (e.g., in Eclipse Java/C++)

Catalog and commentary: http://refactoring.com/

Architectural refactorings
- Resolve one or more architectural smells, have an impact on quality attributes
  - Architectural smell: suspicion that architecture is no longer adequate (“good enough”) under current requirements and constraints (which may differ from original ones)
- Are carriers of reengineering knowledge (patterns?)
- Can only be partially automated
Architectural refactoring for the cloud: a decision-centric view on cloud migration

Olaf Zimmermann

- Template and cloud refactorings
  - First published @ SummerSoc 2016

### Coupling Smells

<table>
<thead>
<tr>
<th>Smell</th>
<th>Suggested Refactoring(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>API clients and their providers can only be deployed and updated jointly due to a tight coupling</td>
<td>Downsize data contract by adding Linked Information Holders replacing Embedded Entities</td>
</tr>
</tbody>
</table>

### Granularity Smells

<table>
<thead>
<tr>
<th>Smell</th>
<th>Suggested Refactoring(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>God service with many operations that takes long to update, test and deploy</td>
<td>Split Service</td>
</tr>
<tr>
<td>Fat Information Holder violating SRP</td>
<td>Split Information Holder according to data lifetime and incoming dependencies</td>
</tr>
<tr>
<td>Big Ball of Service Mud (doing processing and data access)</td>
<td>Split into Processing Resource and Information Holder Resource (CQRS for API)</td>
</tr>
<tr>
<td>Service proliferation syndrome (unmanageable)</td>
<td>Consolidate different processing responsibility types into single Business Activity Processor</td>
</tr>
</tbody>
</table>

- Microservices refactorings:
  - Future work for MAP

Work in progress!
Open Problem: Service/Data Visualization (Modeling)

Research Questions

What is an intuitive, easy-to-sketch graphical representation for (micro-)services and their endpoints, operations, and message representations?

Which notations and tools do you use?
Do they make communication effective and efficient?
Ports-and-adapters combined with layering (“hexagonioning”):
Use patterns to specify:

- Role and responsibility of API call
- Message representations
- Documentation and governance
Microservices – Summary and Opinions

- **Microservices have many predecessors (evolution not revolution)**
  - Implementation approach and sub-style of SOA
    - More emphasis on autonomy and decentralization (of decisions, of data ownership), less vendor-driven
    - Automation advances and novel target environments

- **One service size does not fit all**
  - Context matters and forces at work
  - Size and granularity are not ends in themselves
    - Goal: achieve “Independent X” – but do not forget BAC and CAP (and ACS)
  - Architecture and architects needed more than ever
    - More options, higher consequences of not making adequate decisions

- **Microservices API Patterns; Context Mapper, Service Cutter**
  - [Public website](#) now available
    - Pattern language, sample implementations, supporting tools

- **Service modeling, identification, decomposition, refactoring problems**
Microservices Publications

- **Zimmermann, O.**: *Microservices Tenets – Agile Approach to Service Development and Deployment*

- **Pardon, G., Pautasso, C., Zimmermann, O.**: *Consistent Disaster Recovery for Microservices: the Backup, Availability, Consistency (BAC) Theorem*

- **Pahl, C., Jamshidi, P., Zimmermann, O.**: *Architectural Principles for Cloud Software*

- **Furda, A., Fidge, C., Zimmermann, O., Kelly, W., Barros, A.**: *Migrating Enterprise Legacy Source Code to Microservices: On Multitenancy, Statefulness, and Data Consistency*