DIMENSIONS OF SUCCESSFUL WEB API DESIGN AND EVOLUTION: CONTEXT, CONTRACTS, COMPONENTS

20th International Conference on Web Engineering (ICWE)
Online, June 11, 2020

Prof. Dr. Olaf Zimmermann (ZIO)
Distinguished (Chief/Lead) IT Architect
Institute for Software, HSR FHO OST
ozimmerm@hsr.ch
Blog: https://ozimmer.ch/blog
Happy Birthday ICWE (and HTTP 1.0 turns 25!)

- **At the time of the first conference edition:**
  - WWW 1.0: "e-business" CGI, servlets etc.
  - Amazon was an online book store
  - Shop software was a market, some POX APIs

  - Web 2.0: Wikis, blogs, mashups (what happened to them?)
  - Twitter had started, Amazon now selling many things

- **Since 2010:**
  - Cloud and containers all over the place; AWS leading the cloud market
  - Many social networks, games; public APIs a must, hypermedia in the media

- **This year's conference themes:**
  - Performance, Testing, Machine Learning, Open Data, Sentiment Analysis, Emotion Detection, Location-Awareness, and more
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
1. **Context matters**
   - One size does not fit all
   - Strategic and tactic Domain-Driven Design (DDD) to the remedy
   - *Context Mapper DDD DSL and supporting tools available*

2. **Contracts rule**
   - A unified interfaces is great for browsers, but not enough for application integration
   - Protocol choice depends (on context, on requirements)
   - *Microservice Domain-Specific Language (MDSL) and tools released*

3. **Components contain (cost and risk)**
   - Web API designs do not have to be reinvented on every project
   - Much focus on infrastructure design so far, what about API endpoints, service contracts, message representation elements?
   - *Microservice API Patterns (MAP) structure the solution space*
Part 1: Context
Debunking Myths 1: Technology First, One Size Fits All

Context

**Digitalization**: Development of open source software or commercial project.

Myths

a) Information Technology (IT) must be at the heart of innovation; users can be educated about their wants and needs later.
b) A single software design can serve all user contexts and requirements.

Rectification

Observe/listen to users to carve out the "business" value of the new software. Establish an ubiquitous, application domain-driven language. Apply proven engineering methods and empirical validation techniques. No cargo cults either please… context matters!
What is Context? It depends.

- **Latin roots:** *contextere:* "weave together", *contextus:* "tight coupling" (!)

- "A frame that surrounds [...] event and provides resources for its appropriate interpretation"

- **Dimensions in Software and Web Engineering:**
  - Location (for instance, of a mobile phone or robot)
  - P. Kruchten's *project octopus* and decision making context (incl. ethics, compliance)
  - Trigger and precondition for patterns usage
  - User experience ("operating range")
  - Developer experience (papers by G. Murphy et al.)
  - System context in systems of systems (interfaces)
  - Modeling context
    - Conway's Law vs. The Matrix": universal data model

---

**PL:** *Published Language*  **D:** *Downstream, U:* *Upstream*  **ACL:** *Anti-Corruption Layer*
Methods and Practices – Old and New (?)

Business Requirements Specifications

Agile practices

Domain model, code, tests, etc.

As a business analyst (specializing on a particular business or technical domain),

*I would like to* describe the problem domain and its subdomains in a natural, yet precise and ubiquitous language (i.e., domain concepts, their properties and relations)

*so that* project sponsor, team and other stakeholders can develop and share a common understanding about these concepts and their intricacies in the given domain – in line with Agile values and principles.

---

UserStory PaperArchiving {
   As a "Researcher"
   I want to create a "PaperItem" with its "title", "authors", "venue" in a "PaperCollection"
   so that "other researchers can find and cite the referenced paper easily, and my h-index goes up."
}
Towards an "open and lean architecting framework" (ECSA SAGRA 16):

- Agile practices such as user stories and definition of done (done-deciding)
- **SMART** nonfunctional requirements a.k.a. desired qualities:
  - Specific, Measurable, Agreed upon, Realistic, Time bound
- CRC cards to specify components and their collaborations
- **Y-statements** to capture architectural decisions and their rationale:
  
  *In the context of <use case uc and/or component co>, ... facing <non-functional concern c>,

  .. we decided for <option 01> and neglected <options 02 to on>,

  ... to achieve <quality q>,

  ... accepting downside <consequence c>.
Domain-Driven Design (DDD): Domain Model in the Center

- **Strategic: Bounded Contexts**
  - Model boundaries and their relations (Web APIs)
  - Abstraction of team or (sub-) system
    - E.g. payment, product catalog, shipping

- **Tactic DDD: Aggregates**
  - Object clusters as storage units, consistency invariants
    - E.g. order and its items

- **Event storming to find the domain model elements**
  - Huge momentum right now
  - Similar techniques have been around since 1990s

Figure reference: [https://leanpub.com/theanatomyofdomain-drivendesign](https://leanpub.com/theanatomyofdomain-drivendesign)
Eclipse plugin (v5.12), VSC Extension (new!) or Web IDE/GitPod (new!)

Strategic DDD: Each bounded context has its own domain model (aggregates etc.)

ContextMap {
    contains ReferenceManagementFrontend
    contains ReferenceManagementService
}

    implementationTechnology "HTTP"
    exposedAggregates PaperArchiveFacade
}

Relations: PL: Published Language, CF: Conformist (and more)

Tactic DDD (of Reference Management Service): Stereotyped OOAD classes
Rapid OOAD/DDD with Context Mapper (and MDSL)

**User Story**
- in Context Mapper (manual)
- Use cases also possible

**Analysis Model**
- Subdomains, Entities

**Design Model**
- Feature BCs with Aggregates
- System BCs
- JHipster JDL

**Service Contract**
- MDSL
- Open API

**Clients, Servers**
- JHipster
- Spring Boot

**Step-by-step instructions:** [https://ozimmer.ch/practices/2020/06/10/ICWEKeynoteAndDemo.html](https://ozimmer.ch/practices/2020/06/10/ICWEKeynoteAndDemo.html)
Context


Myths

a) SOA and microservices are competing styles. Micro means really small.
b) Any (micro-)service must expose an HTTP resource API.
c) The REST principle of a unified interface (GET, POST, PUT, etc.) is suited and sufficient to express rich domain model semantics.

Rectification: Contracts rule (polyglot integration, data type precision)!

a) Microservices are an implementation approach to SOA. Service size varies.
b) IDEAL, FROSTT, CCP Two-Tier; event sourcing and streaming, Kafka
c) DDD, MDSL, operationId in Open API Specification (OAS)
"Napkin Sketch" of SOA Realizations (Adopted from G. Hohpe)

Our focus: Microservices!

Middleware less popular, often custom build (term also used in deployment and clustering context)

Service Registry

Optional (then and now)

Service Endpoint

Discovery

Register

Our focus: Message

(data) contracts

Conversation

Orchestration

Rules

Transform

Document

Application

Microservices!
Microservices architectures evolved from previous incarnations of Service-Oriented Architectures (SOAs) to promote agility and elasticity

- Independently deployable, scalable and changeable services, each having a single responsibility
- Modeling business capabilities

Often deployed in lightweight containers

- Encapsulating their own state, and communicating via message-based remote APIs (HTTP, queueing), IDEALLY in a loosely coupled fashion
- Facilitating polyglot programming and persistence
- Leveraging DevOps practices including decentralized continuous delivery and end-to-end monitoring (for business agility and domain observability)

Cloud-native Application Architectures are API-centric

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

http://www.cloudcomputingpatterns.org
You can view the Web as an asynchronous connector technology.

- Resource takes place of queue in this view:

From Biz and Dev to Ops: Bad Smells and Refactorings

- independent deployability
  - multiple services in one container
  - no API gateway

- horizontal scalability
  - endpoint-based service interactions

- isolation of failures
  - wobbly service interactions
    - add circuit breaker
    - use timeouts
    - add bulkhead

- decentralisation
  - ESB misuse
    - rightsise ESB
    - split database
  - shared persistence
    - add data manager
      - merge services
      - split teams by service
  - single-layer teams

- **Wikipedia** lists 23+ IDLs
  - OAS is one of them
  - Bound to HTTP (AsyncAPI?)
Contracts in Microservice Domain-Specific Language (MDSL)

- **Data contract**
  - Compact, technology-neutral
  - Inspired by JSON, regex

- **Endpoints and operations**
  - Elaborate, terminology from MAP domain model
  - Abstraction of REST resource
  - Abstraction of WS-* concepts

- **API client, provider, gateway; governance (SLA, version, ...)**

How does this notation compare to OpenAPI/JSON Schema (and other IDLs, WSDL/XSD)?

```plaintext
data type PaperItemDTO {
    "title":D<string>, "authors":D<string>,
    "venue":D<string>, "paperItemId":PaperItemKey }

data type PaperItemKey {
    "doi":D<string> }

data type createPaperItemParameter {
    "who":D<string>, "what":D<string>, "where":D<string> }

"id": Role<type> triplet: Role=(Meta-)Data, Link, ID

endpoint type PaperArchiveFacade
serves as INFORMATIONHOLDERRESOURCE
exposes
operation createPaperItem
    with responsibility STATE_CREATION_OPERATION
    expecting
    payload createPaperItemParameter
    delivering
    payload PaperItemDTO

INFORMATIONHOLDER is a MAP decorator (role stereotype)
```
## MDSL and OAS in Comparison

<table>
<thead>
<tr>
<th>Criterion</th>
<th>MDSL</th>
<th>OAS (f.k.a. Swagger)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Concrete syntax</td>
<td>DSL (made with Xtext)</td>
<td>YAML, JSON</td>
</tr>
<tr>
<td>Abstract syntax</td>
<td>SOA concepts, MAP domain model</td>
<td>HTTP concepts</td>
</tr>
<tr>
<td>Main use cases</td>
<td>Agile modeling, SOAD, contract first</td>
<td>Code first, testing, stub generation</td>
</tr>
<tr>
<td>Bindings</td>
<td>HTTP, Java; more possible and planned (gRPC, GSL, MOM/EIP)</td>
<td>HTTP only (sibling language AsyncAPI)</td>
</tr>
<tr>
<td>Size of specification</td>
<td>19 pages (v3.3.2; note: less complete!)</td>
<td>95 pages (v3.0.3)</td>
</tr>
<tr>
<td>Size of &quot;Publication Management&quot; Demo contract (blog post)</td>
<td>1,568 bytes, 132 words (1,321 characters)</td>
<td>3,263 bytes, 199 words (2,159 characters)</td>
</tr>
<tr>
<td>Tools</td>
<td>Few (editor, OAS generator), see <a href="#">here</a></td>
<td>Many, see <a href="#">here</a></td>
</tr>
<tr>
<td>Maturity</td>
<td>since 2018; just open sourced (ZIO)</td>
<td>since 2011 (SmartBear)</td>
</tr>
<tr>
<td>License</td>
<td>Apache License 2.0</td>
<td>Apache License 2.0</td>
</tr>
</tbody>
</table>
Editor, API linter, OpenAPI generator; tutorial available on GitHub

Design Goals

A contract language for (micro-)service API design should/must:

- Support agile modeling practices, for instance in API design workshops:
  - Value and promote readability over parsing efficiency (in language design)
  - Support partial specifications as first-class language concepts to be refined iteratively (see above example `{ID, D}`)

Getting Started

- Presentations featuring Context Mapper, MAP and MDSL can be found here.
- The GitHub Pages for MDSL provide a tutorial and language reference information.
- As a contributor, please consult the readme file of the dsl-core project for getting started information and prerequisites.

- Top-down from requirements (for instance, user stories for integration scenarios)
- Bottom up from existing systems (represented, for instance, as DDD-style context maps)

Design Principles
Part 3: Components (and Patterns)
Context

Web/cloud application development (any project, actually).

Myth

Our performance scalability requirements are more advanced than everybody else's (ok, except for those of the Internet giants). There is no point in reusing existing solutions; we have to invent our own library, framework, protocol, etc.

Rectification

Back to context and requirements (DDD!): Identify candidate services and make a conscious and candid buy-rent-build decision per service. If "build" is required, apply patterns when doing so. Components (captured as patterns, in frameworks) contain (cost and risk)!
Decomposition Heuristics that do not suffice

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

Simple if-then-else rules of thumb
- 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, including “products not projects”

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

Context matters here too – one size does not fit all, (also pointed out in a keynote at Agile Australia)
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

Reference: JUGS presentation, Bern/CH, Jan 9, 2020
Sample request message (note: PUTs and POSTs would look different)

Response message structure

[@-]

---

Sample request message:

```curl
curl -X GET "http://localhost:8080/customers/rgpp0wkpec" -H "accept: */*
```

Response message structure:

```json
{
"_links": [
   {
      "deprecation": "string",
      "href": "string",
      "hreflang": "string",
      "media": "string",
      "rel": "string",
      "templated": true,
      "title": "string",
      "type": "string"
   },
   "birthday": "2019-02-12T09:10:07.370Z",
   "city": "string",
   "customerId": "string",
   "email": "string",
   "firstname": "string",
   "lastname": "string",
   "moveHistory": [
      {
         "city": "string",
         "postalCode": "string",
         "streetAddress": "string"
      }
   ],
   "phoneNumber": "string",
   "postalCode": "string",
   "streetAddress": "string"
}
```

Embed nested entity data? or Link to separate resource?

Sender

Command Message

Receiver

```javascript
C = getLastTradePrice("DIS");
```

Microservices API Patterns (MAP): Overview

Responsibility

- Endpoint Roles
  - Processing Resource
  - Information Holder Resource

Structure

- Representation Elements
  - Atomic Parameter
  - Atomic Parameter List

Quality

- Quality Management and Governance
  - API Key
  - Rate Limit

Evolution

- Version Identifier
- Semantic Versioning

- Two In Production
- Aggressive Obsolescence
- Experimental Preview

- Transactional Data Holder
- Master Data Holder
- Static Data Holder

- Annotated Parameter Collection
- Context Representation
- Pagination

Reference Management

- Embedded Entity
- Linked Information Holder

(not yet) EuroPLoP 2020

Problem: How can you avoid exchanging multiple messages when receivers require insights from multiple related information elements?

Solution: For any relationship that the client has to follow, embed a Data Element in the message that contains the data of the target entity (instead of linking to the target entity).

Forces: Performance, scalability; flexibility and modifiability; data quality, freshness, consistency.
Problem: When exposing structured, possibly deeply nested information elements in an API, how can you avoid sending large messages containing lots of data that is not always useful for the message receiver in its entirety?

Forces: Same as for Embedded Entity.

Solution: Add a Link Element to the message that references an API endpoint. Let this API endpoint represent the linked entity; for instance, use an Information Holder Resource for the referenced information element.
Key Messages of this Talk

- It is the API contracts (and their implementations) that make or break projects – not (or not only) middleware and network protocols
  - Frameworks and infrastructures come and go, APIs stay
  - Rich domain semantics call for higher-level design tools

- Microservice API Patterns (MAP) language: **website first**
  - Focus on message representation elements
  - 20+ patterns, sample implementation in public repo, supporting tools

- Microservices Domain-Specific Language (MDSL) **open sourced**
  - Uses MAPs in service contracts (as decorators)
  - Can be generated from DDD bounded contexts

- **Context Mapper** supports strategic Domain-Driven Design (DDD), rapid OOAD/tactic DDD and architectural refactoring
  - Other tools emerging: context discovery, application layer design
You had been tasked to develop a RESTful HTTP API for a master data management system that stores customer records and allows sales staff to analyze customer behavior. The system is implemented in Java and Spring. A backend B2B channel uses message queues (RabbitMQ).

What do you do (now)?

a) I hand over to my software engineers and students because all I manage to do these days is attend meetings and write funding proposals.

b) I annotate the existing Java interfaces with @POST and @GET, as defined in Spring MVC or JAX-RS etc. and let libraries and frameworks finish the job.

c) I install an API gateway product in Kubernetes and hire a sys admin, done.

d) I design a service layer (Remote Facade with Data Transfer Objects) and publish an Open API Specification (f.k.a. Swagger) contract. I worry about message sizes, transaction boundaries, error handling and coupling criteria while implementing the contract. To resolve such issues, I create my own novel solutions. Writing infrastructure code and test cases is fun after all!

e) I leverage Context Mapper, MDSL, MAP for API design and evolution 😊
Research Papers and Practitioner Articles (Selection) ²

Microservices and related topics (since 2016):

  - The MAP website can be found [here](#).

  - More information on Context Mapper can be found [here](#) (presentations, papers).

  - All papers from the MAP project are listed [here](#) (5 so far).


DIMENSIONS OF SUCCESSFUL WEB API DESIGN AND EVOLUTION: CONTEXT, CONTRACTS, COMPONENTS BACKUP CHARTS

20th International Conference on Web Engineering (ICWE)

Online, June 11, 2020

Prof. Dr. Olaf Zimmermann (ZIO)
Distinguished (Chief/Lead) IT Architect
Institute for Software, HSR FHO OST
ozimmerm@hsr.ch
Blog: https://ozimmer.ch/blog
Many design issues, typically recurring

- per system/team, per relationship, per interface

- Data duplication and/or on-demand exchange? Strict/eventual consistency?
- Data and control flow direction? Data formats (norms, transformations)? Frequency of message exchange?
- Client influence on API design and stability/evolution (governance)? API contracts and technologies?

Customer Self-Service

Printing

Debt Collection

Risk Management

System decomposition?
A Strategic DDD Context Map with Relationships

- Insurance scenario, example model from [https://contextmapper.org/](https://contextmapper.org/)

---

**Customer Self-Service Context**

**Printing Context**

**Debt Collection Context**

**Risk Management Context**

**Policy Management Context**

**Customer Management Context**

**Shared Kernel**

**Bounded Context**

A Domain-Specific Language (DSL) for DDD:

- Formal, machine-readable DDD Context Maps via *editors and validators*
- Model/code *generators* to convert models into other representations
- Model transformations for *refactorings* (e.g., “Split Bounded Context”)

Plugin update site: [https://dl.bintray.com/contextmapper/context-mapping-dsl/updates/](https://dl.bintray.com/contextmapper/context-mapping-dsl/updates/)
ContextMap DDDSampleMap {
    contains CargoBookingContext
    contains VoyagePlanningContext
    contains LocationContext

    [U,OHS,PL] LocationContext --> [D] CargoBookingContext
    VoyagePlanningContext [D]<-[U,OHS,PL] LocationContext
}

Bounded Contexts (systems or teams)

DDD relationship patterns (role of endpoint)

Influence/data flow direction: ->, <-> (upstream-downstream or symmetric)

SK: Shared Kernel, PL: Published Language
D: Downstream, U: Upstream
ACL: Anti-Corruption Layer, OHS: Open Host Service
Tool Big Picture

- **Context Mapper architecture**
  - Modelled with Context Mapper DSL
  - UML generated

The reverse engineering and discovery component can generate CML Context Maps from existing source code. This allows to reverse engineer the architecture model in projects with existing monoliths or microservices.

Provides the Context Mapper DSL (CML) modeling language to express architectures on the basis of Strategic Domain-driven Design (DDD) patterns.

The Service Cutter integration into Context Mapper allows to analyze the Context Map with respect to coupling criteria and supports to suggest improved Context Maps. The Service Cutter library exposes an API (Open Host Service and Published Language) used by Context Mapper to generate the new decompositions.

The generators allow to generate other representations of the architecture derived by a given CML Context Map.
Microservice API Patterns (MAP) Categories

- **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?

- **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 trade-offs be communicated and accounted for?

- **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?

- **Evolution Patterns:**
  - Recently workshopped (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
Mini-Exercise: Can MAP serve as a map/guide to API design?

- Let’s have a look at the language organization and selected patterns…
  - [http://microservice-api-patterns.org](http://microservice-api-patterns.org)
    - Website public since 2/2019; experimental preview site available to beta testers
  - Sample patterns (suggestions):
    - Request Bundle, Embedded Entity, Wish List, API Key, Two in Production

Microservice API Patterns

Microservice API Patterns (MAP) take a broad view on API design and evolution, primarily focusing on message representations – the payloads exchanged when APIs are called. These payloads have structure. The representation elements in the payloads differ in their meanings as API endpoints and their operations have different architectural responsibilities. Furthermore, the chosen representation structures strongly influence the design time and runtime qualities of an API.

Our Microservice API Patterns capture proven solutions to design problems commonly encountered when specifying and implementing message-based APIs in terms of their structure, responsibilities, and quality.