OST
Ostschweizer Fachhochschule

APIs as Service Activators:

Tackling the Hard Parts of Integration Design

ZEUS2023

ABOUT CALL FOR PAPERS PROGRAM VENUE ORGANIZATION

15th Central European Workshop on Services and their Composition

Olaf Zimmermann

February 16, 2023

olaf.zimmermann@ost.ch

IFS INSTITUTE FOR SOFTWARE

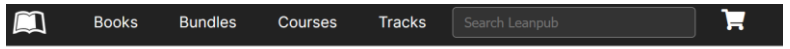
Abstract

API stands for application programming interface, but might as well mean *access* to *services* via protocol for *integration*. Message-based APIs and the services they expose must be carefully designed to achieve qualities such as composability, efficiency, and evolvability; project context and application domain challenges drive the architectural decisions required regarding communication, coordination and consistency.

This talk introduces a stepwise, incremental and iterative design practice that leverages proven principles and patterns to jumpstart greenfield API design and service engineering. For brownfield scenarios, it proposes an interface refactoring catalog to resolve design smells frequently occurring in practice. Common design tradeoffs in these scenarios are discussed in the form of reusable Architectural Decision Records (ADRs).

Who is ZIO?

- Software Architect
- Lecturer, Author, Blogger
- Open Source Contributor
  - Context Mapper
    (DSL for Domain-Driven Design)
  - API patterns, message/interface description MDSL
  - Interface Refatoring Catalog
  - Markdown ADRs, Y-Statements as a compact form of ADRs
  - Lakeside Mutual microservices (sample application)

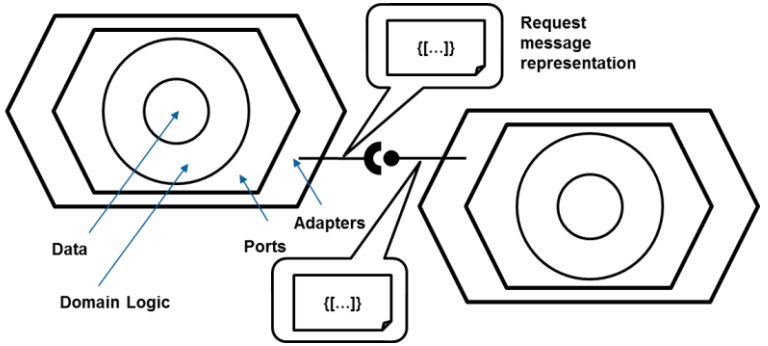


Design Practice Reference
Guides and Templates to Craft Quality Software in Style

https://leanpub.com/dpr

MDSL

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



www.api-patterns.org

https://medium.com/olzzio

Agenda

- Context and motivation
  - Stepwise API and service design (by example)
  - Pattern languages as knowledge brokers
- A pattern language for API design
  - Overview, domain model
  - Selected patterns
- Refactoring to patterns
- Pattern selection decisions and other hard design concerns (tradeoffs)
- Concluding thoughts
  - Pattern adoption
  - Open research questions

Agenda

- **Context and motivation**
  - **Stepwise API and service design (by example)**
  - **Pattern languages as knowledge brokers**
- A pattern language for API design
  - Overview, domain model
  - Selected patterns
- Refactoring to patterns
- Pattern selection decisions and other hard design concerns (tradeoffs)
- Concluding thoughts
  - Pattern adoption
  - Open research questions

Sample Scenario: Conference Management

API user story 1: As a mobile app for a conference attendees,
I want to download a session overview from the system backend and, on demand, detailed information about individual sessions (speaker, title, abstract, time, location, etc.)
so that they enjoy a positive learning experience and find their way through the conference.

API user story 2: As the review management application at the conference organizer,
I want to move submissions (talk proposals) in the conference management system through a "receive – review - (accept | reject) – inform – schedule" process
so that a high-quality conference program is assured.

How much time and effort do you need to turn an API user story into a working Web API prototype (RESTful HTTP)?

API and Service Design Step by Step (Simplified!)

User Story

- Agile practice
- Use cases also possible

Demo instructions:
https://ozimmer.ch/practices/2020/06/10/ICWEKeynoteAndDemo.html

generate

Analysis Model

- Domain Driven Design (DDD): Subdomains, Entities

generate initially

Design Model

- DDD: Bounded Contexts with Aggregates
- Bounded Context Types: Feature, System
- JHipster JDL

generate

API Contract

- MDSL (API Contract DSL)
- Open API Specification (OAS), gRPC Protocol Buffers, GraphQL Schema, …

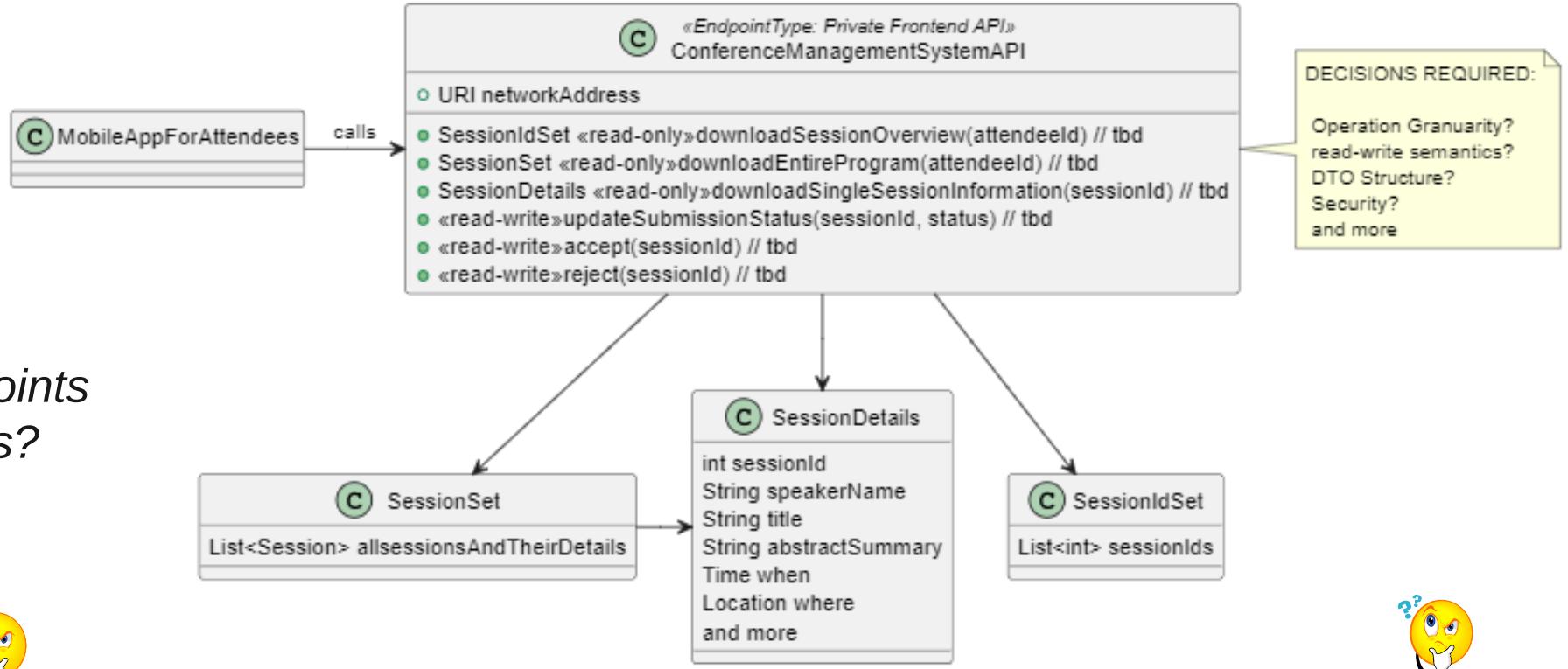
generate stubs

Clients, Providers

- JHipster
- Spring Boot
- JUnit

Elaborate version described as SSD activity in DPR:
https://socadk.github.io/design-practice-repository/activities/SDPR-StepwiseServiceDesign.html

Initial Design Model (aka Candidate Endpoint List)



How many endpoints and operations?

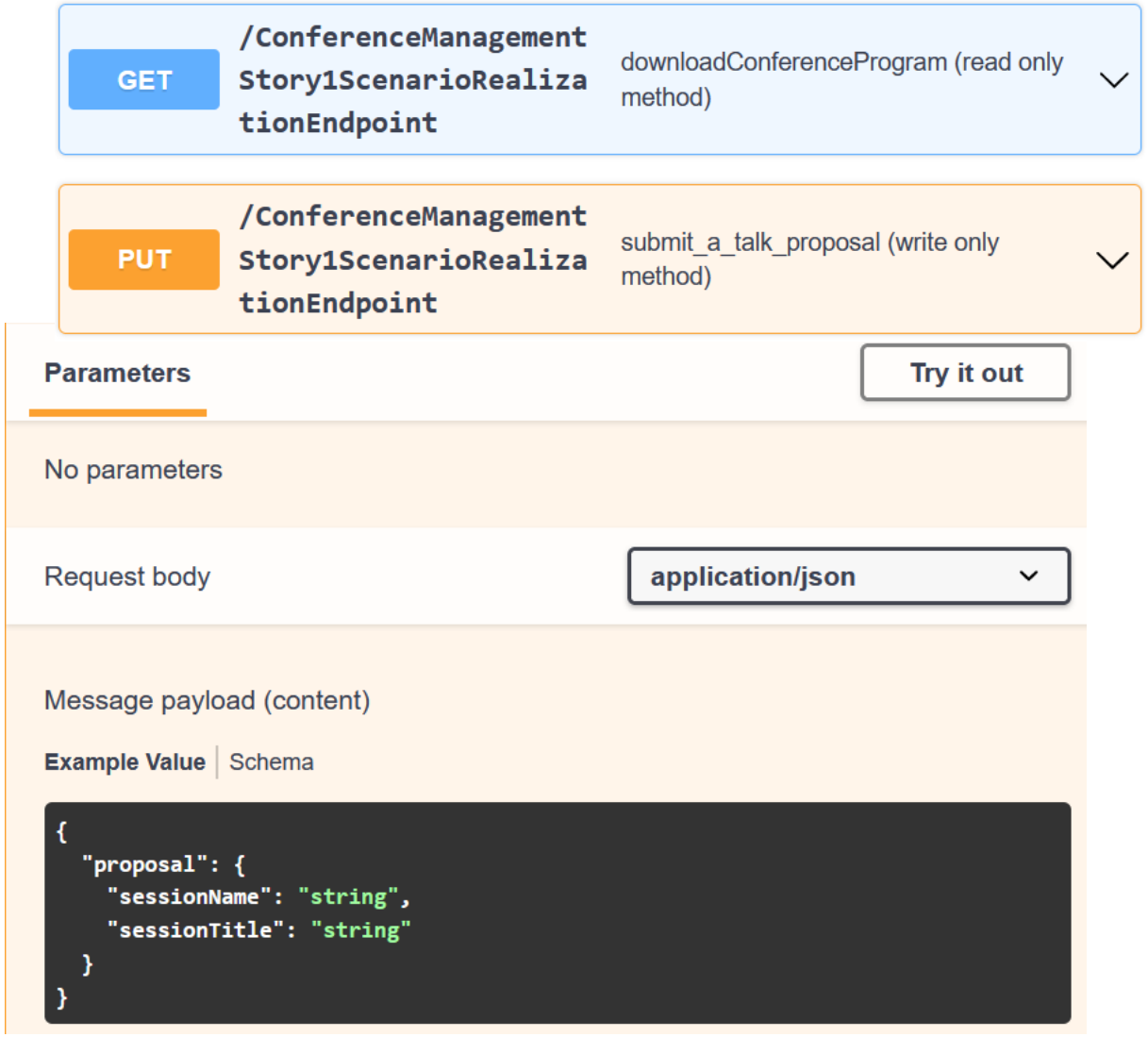
Many small or few large messages?

Include nested data or link to it?

Sample API: Rapid Prototyping (DEMO)

HTTP command and JSON created with:

- MDSL Web, MDSL Tools
  - https://mdsl-web.up.railway.app/
- Open API Specification (OAS) tools
  - https://editor.swagger.io/
- Next up would be:
  - JUnit tests
  - API implementation, e.g. Spring Boot beans @Controller, @Component, @Entity



API Design is Simple?!

Complexity of systems

Project pressure, legacy constraints

Quality goals, security needs

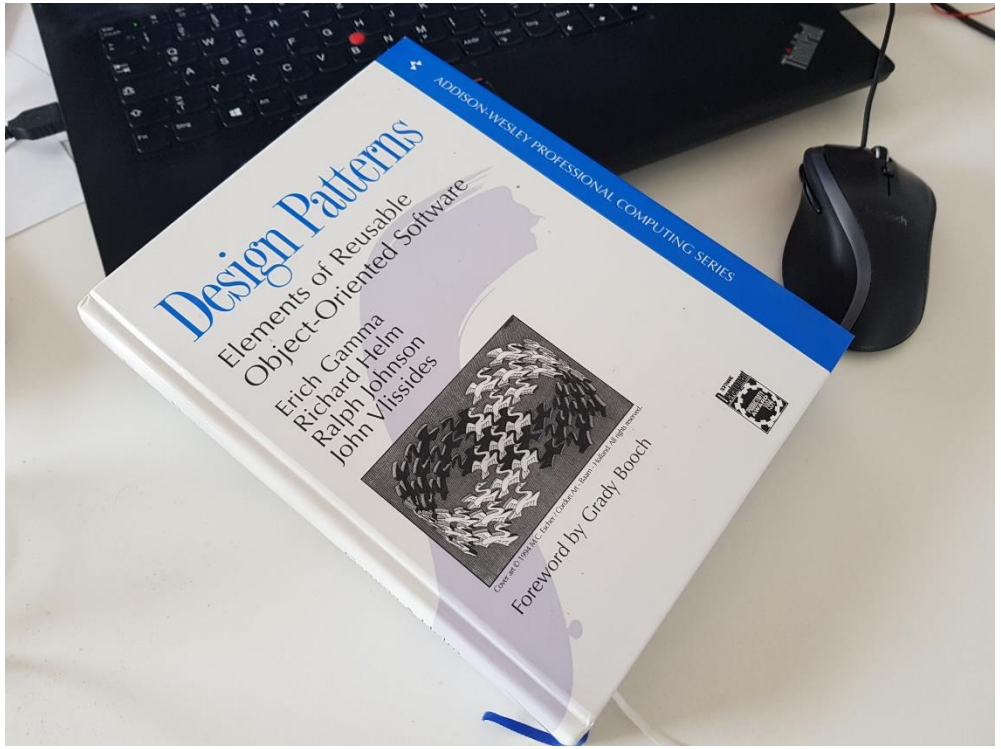
What are the urgent and important questions?

Architectural knowledge helps, even the experienced!
Patterns structure problem and solution domain
and can offer context-specific design guidance.

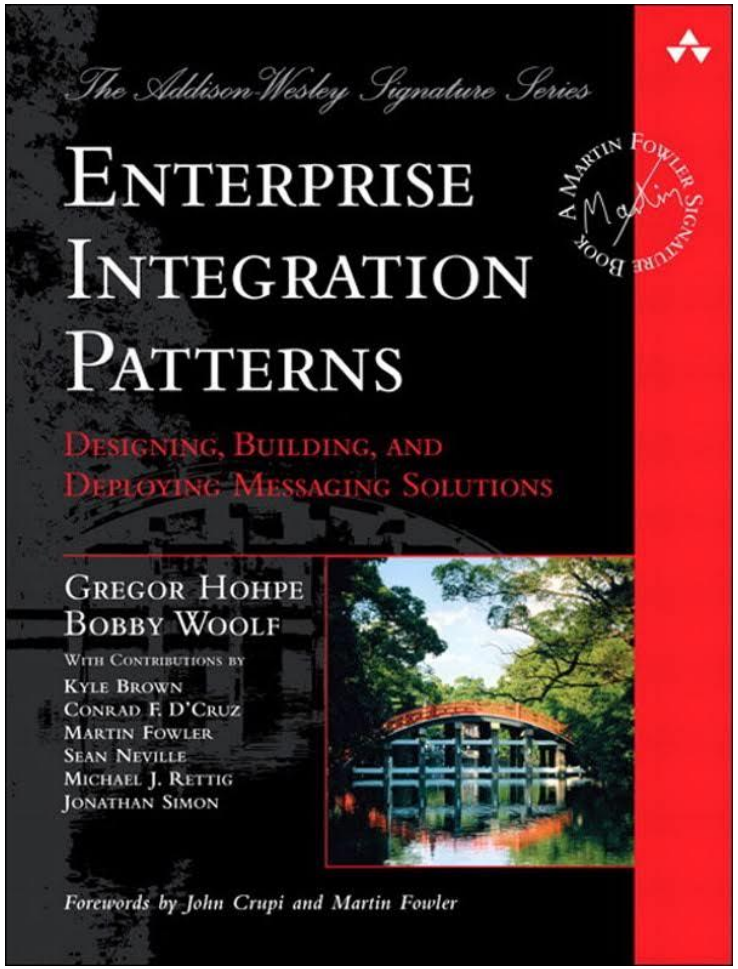
Why Patterns?

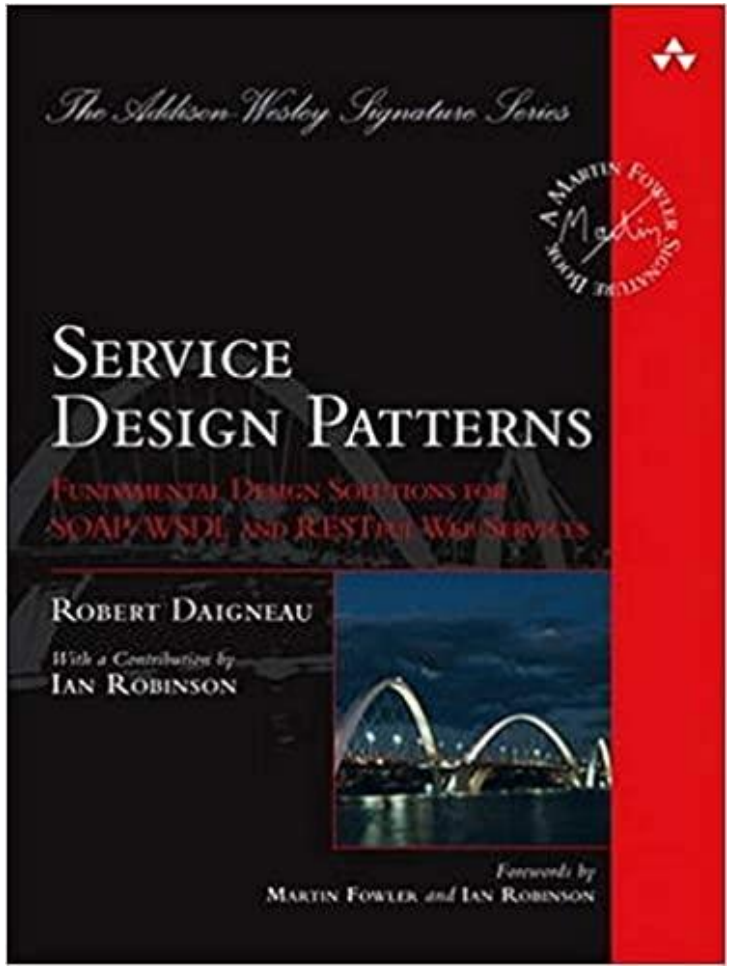
Patterns collect and document experience with proven solutions to common problems

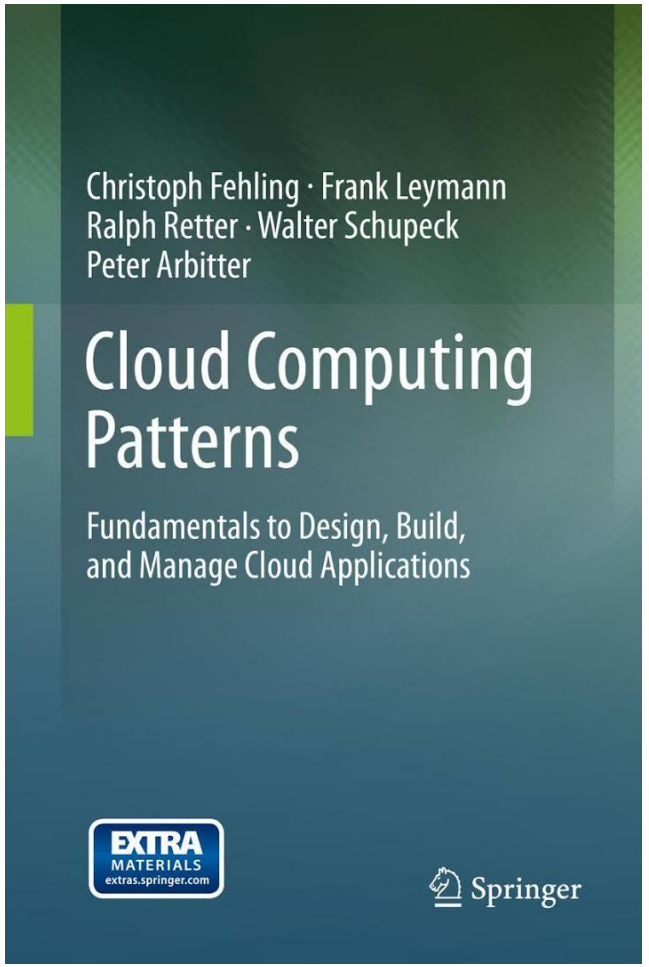
- Many templates:
  - *Name*, Icon
  - *Context*: Intent, Motivation and Applicability
  - *Solution* Structure and its *Forces*
  - *Consequences*: Benefits and Liabilities
  - *Examples* and Implementation Hints
  - Pointers to *Related Patterns*
  - *Known Uses*
- Community processes/practices:
  - Shepherding (coaching), writers workshops



(Selected) Existing Patterns Relevant for API Design



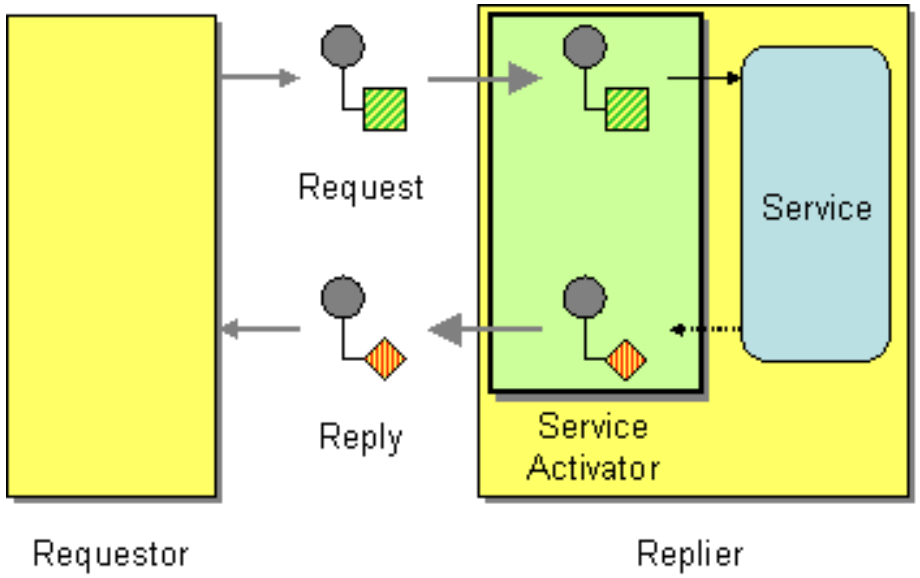




Enterprise Integration Pattern (2003): Service Activator

- https://www.enterpriseintegrationpatterns.com/patterns/messaging/MessagingAdapter.html

How can an application design a service to be invoked both via various messaging technologies and via non-messaging techniques?

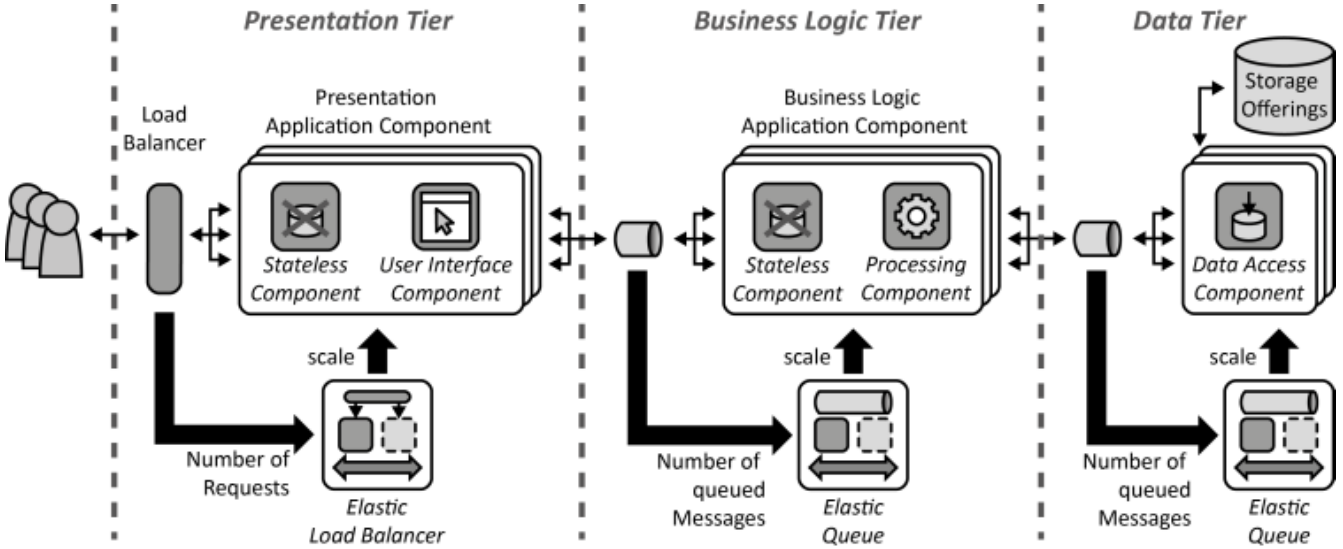


Design a *Service Activator* that connects the messages on the channel to the service being accessed.

Cloud Computing Pattern (2013): Three-Tiered Cloud Application

- https://www.cloudcomputingpatterns.org/three_tier_cloud_application/

How can presentation logic, business logic, and data handling be decomposed into separate tiers that are scaled independently?



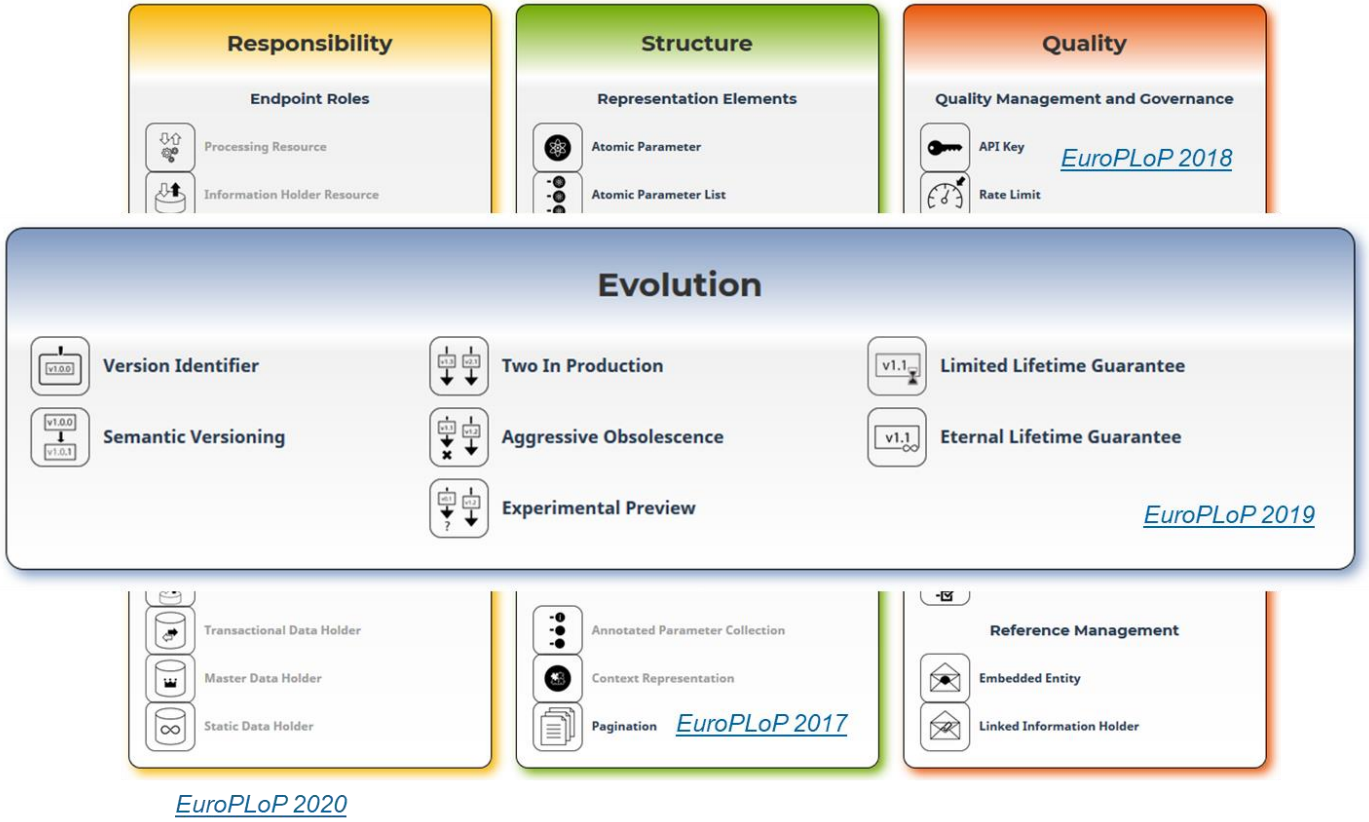
The application is decomposed into three [backend] tiers, where each tier is elastically scaled independently.

Agenda

- Context and motivation
  - Stepwise API and service design (by example)
  - Pattern languages as knowledge brokers
- **A pattern language for API design (and message design)**
  - **Overview, domain model**
  - **Selected patterns**
- Refactoring to patterns
- Pattern selection decisions and other hard design concerns (tradeoffs)
- Concluding thoughts
  - Pattern adoption
  - Open research questions

2016-2022: Microservice API Patterns (MAP)

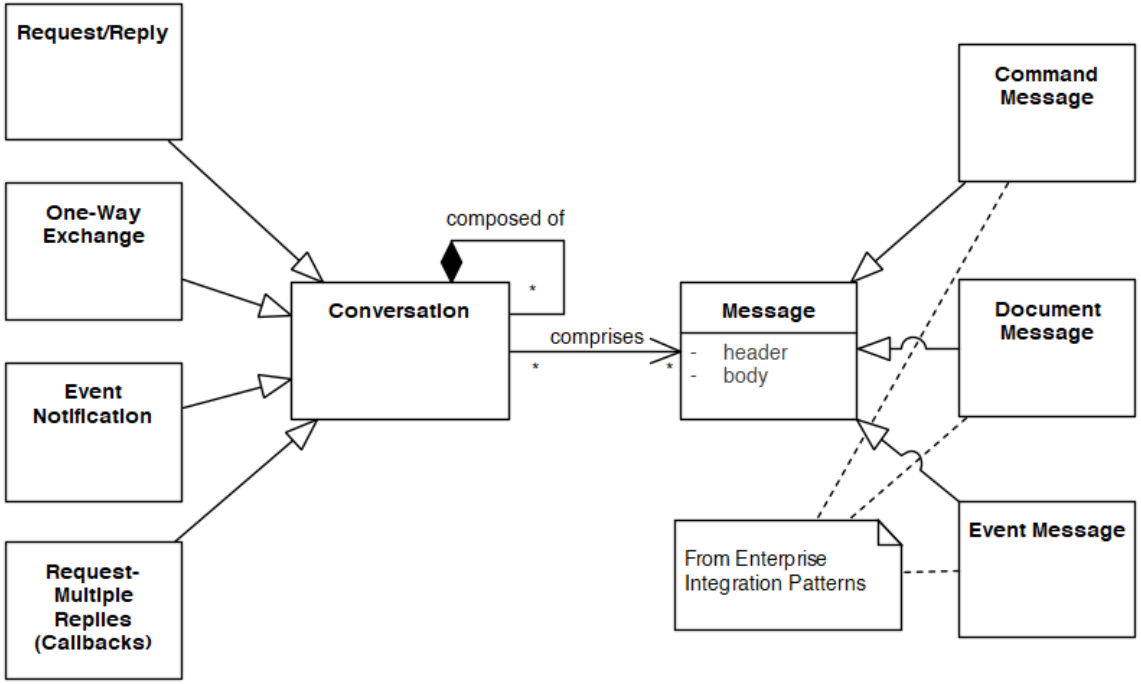
- Distilled solution patterns from projects
  - Not invented, but curated (five authors)
- 44 patterns in five categories
  - Focus: architecture (endpoint roles), data (message representations), versioning (evolution strategies)
- API-related concepts, independent of (but suited for) specific technologies:
  - RESTful HTTP, SOAP
  - GraphQL, gRPC, CORBA
  - Messaging systems, event streaming

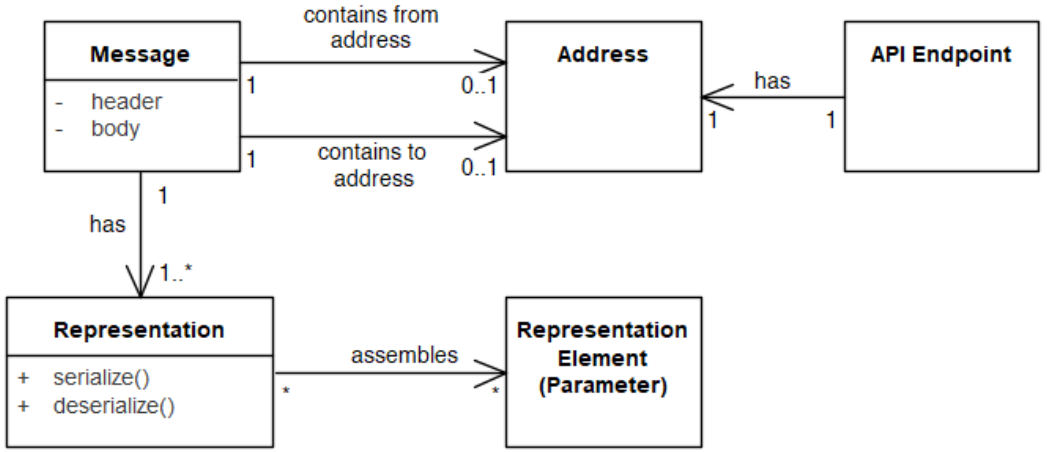


www.api-patterns.org

A Domain Model for APIs

- *From Chapter 1 of "Patterns for API Design"*
  - Available as sample content at Amazon.com
  - Serves as vocabulary in pattern texts

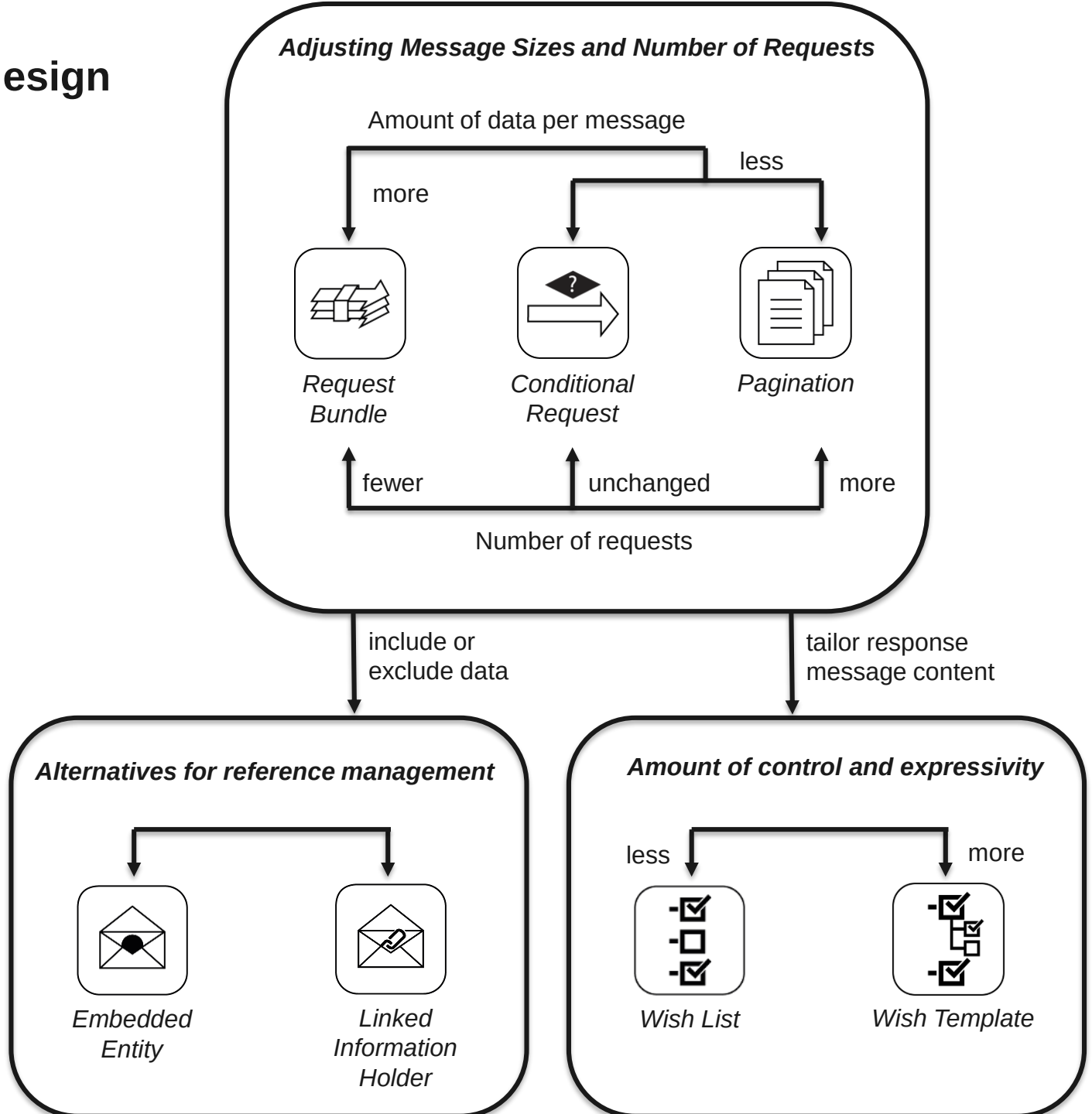




- Abstracts from remoting technologies such as REST, gRPC, GraphQL, WSDL/SOAP, …
  - Example API Endpoint corresponds to one or more HTTP resources (identified by URIs)
  - Endpoints expose operations (not shown here)
  - Validity demonstrated in MDSL tools that map and bind API descriptions structured according to this domain model to these technologies (and more)

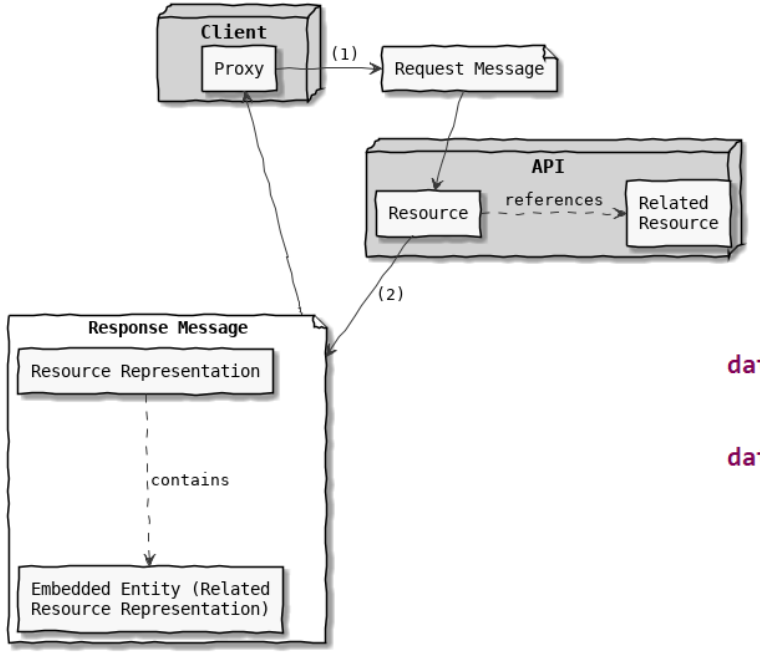
Quality Category

How much data do API client and API provider exchange? And how often?



online version of pattern

Sample Pattern: Embedded Entity



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

```
data type PaperCollectionDTO { "paperCollectionId":D<long>,
   <<Embedded_Entity>>
   "referenceEntryList":PaperItemDTO*}
data type PaperItemDTO
   { "referenceEntryId":D<int>,
   "authors":D<string>*,
   "title":D<string>,
   "venue":D<string>,
   "publisher":D<string>,
   "doi":D<string>}
```

Forces: Performance, scalability; flexibility and modifiability; data quality, freshness, consistency.

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).

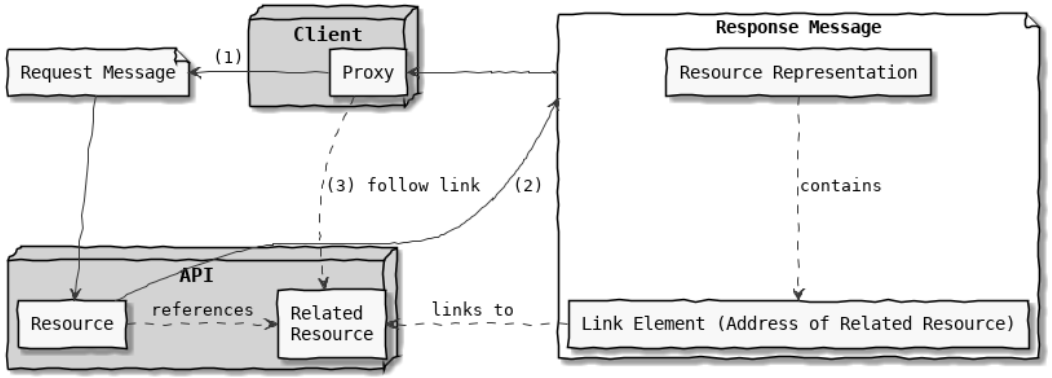
Sample Pattern: Linked Information Holder

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.

```
data type PaperDirectory { "paperCollectionId":D<long>,
   <<Linked_Information_Holder>>
   "referenceEntryPointer":PaperItemInformationHolderLink*}
data type PaperItemInformationHolderLink {
   <<Link_Element>> "uri":L<string>
}
```

online version of pattern



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.

Sample Pattern: Pagination (1/2)

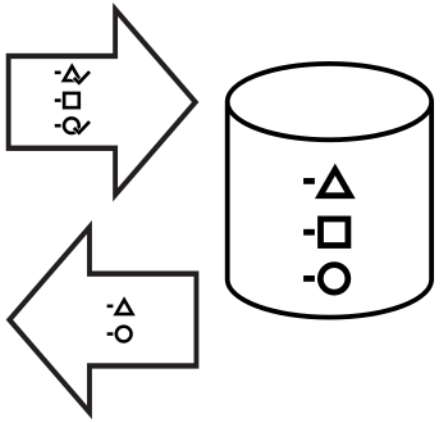
- **Context:**
  - An API endpoint and its calls have been identified and specified.
- **Problem:**
  - *How can an API provider optimize a response to an API client that should deliver large amounts of data with the same structure?*
- **Forces:**
  - Data set size and data access profile (user needs), 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

Sample Pattern: Pagination (2/2)

- **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

Sample Pattern: Wish List (1/2)

- **Problem:**
  - *How can an API client inform the API provider at runtime about the data it is interested in?*
- **Forces (selection):**
  - Client diversity, message size vs. number of messages
  - Endpoint complexity
- **Solution:**
  - *As an API client, provide a Wish List in the request that enumerates all desired data elements of the requested resource.*



https://api-patterns.org/patterns/quality/dataTransferParsimony/WishList

Sample Pattern: Wish List (2/2)

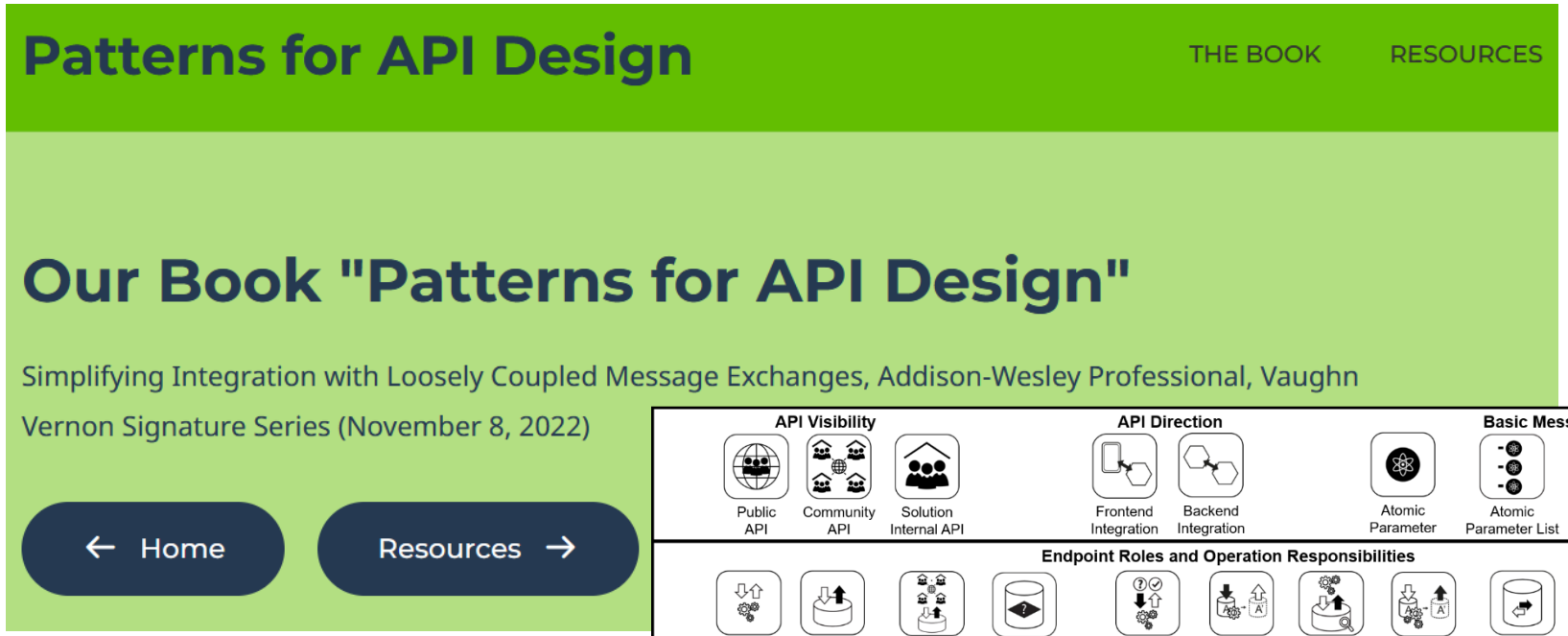
- **Known uses:**
  - Found in many Web and product APIs, e.g. Atlassian Jira
- **Variations:**
  - Expansion, wild cards (*), query expression (GraphQL!)
- **Alternative:**
  - Wish Template (structured, mock object rather than flat name list)

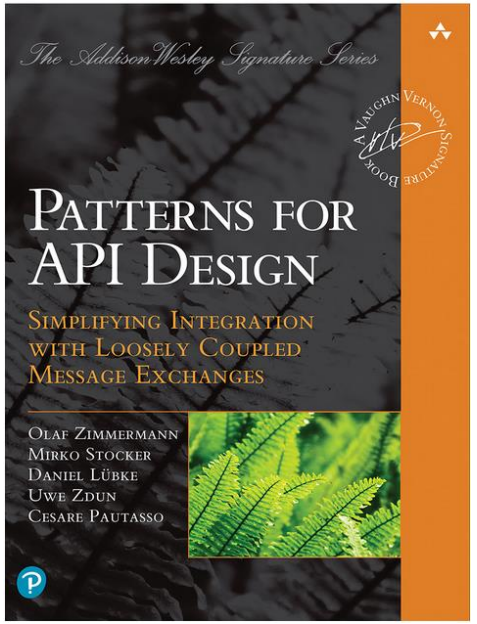
- **Example:**

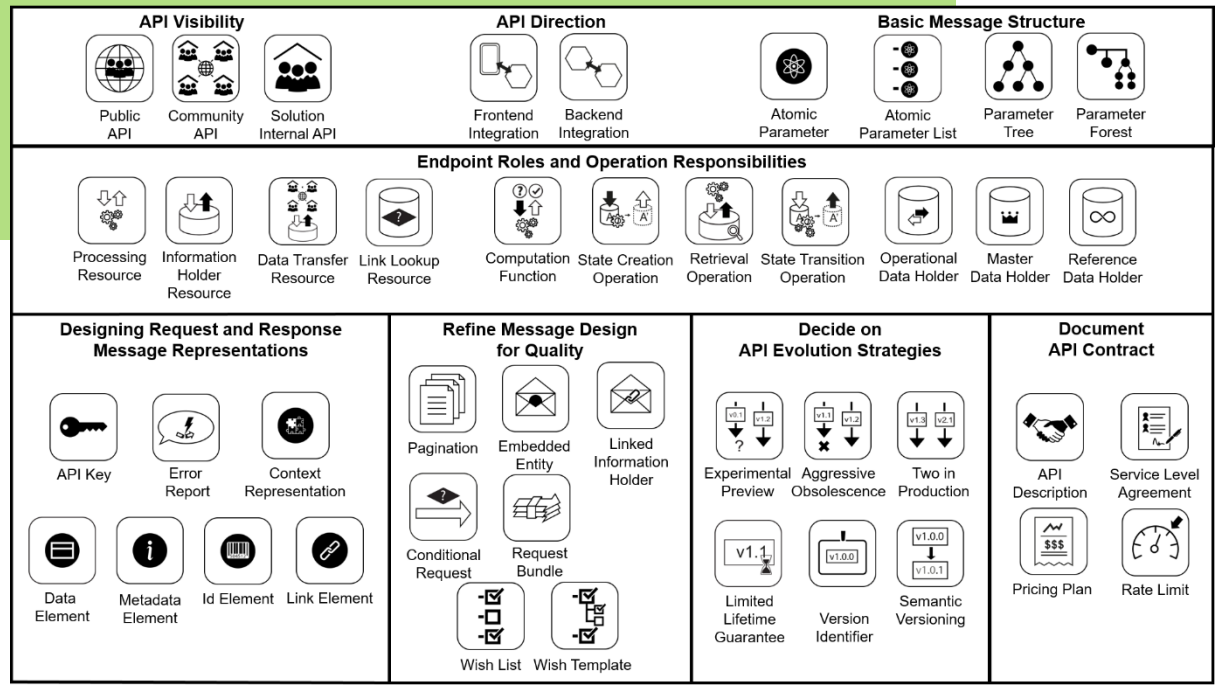
```
curl -X GET
http://localhost:8080/customers/gktlipwhjr?fields=
customerId,birthday,postalCode

{
  "customerId": "gktlipwhjr",
  "birthday": "1989-12-31T23:00:00.000+0000",
  "postalCode": "8640"
}
```

Website and Book: api-patterns.org







Agenda

- Context and motivation
  - Stepwise API and service design (by example)
  - Pattern languages as knowledge brokers
- A pattern language for API design (and message design)
  - Overview, domain model
  - Selected patterns
- **Refactoring to patterns**
- Pattern selection decisions and other hard design concerns (tradeoffs)
- Concluding thoughts
  - Pattern adoption
  - Open research questions

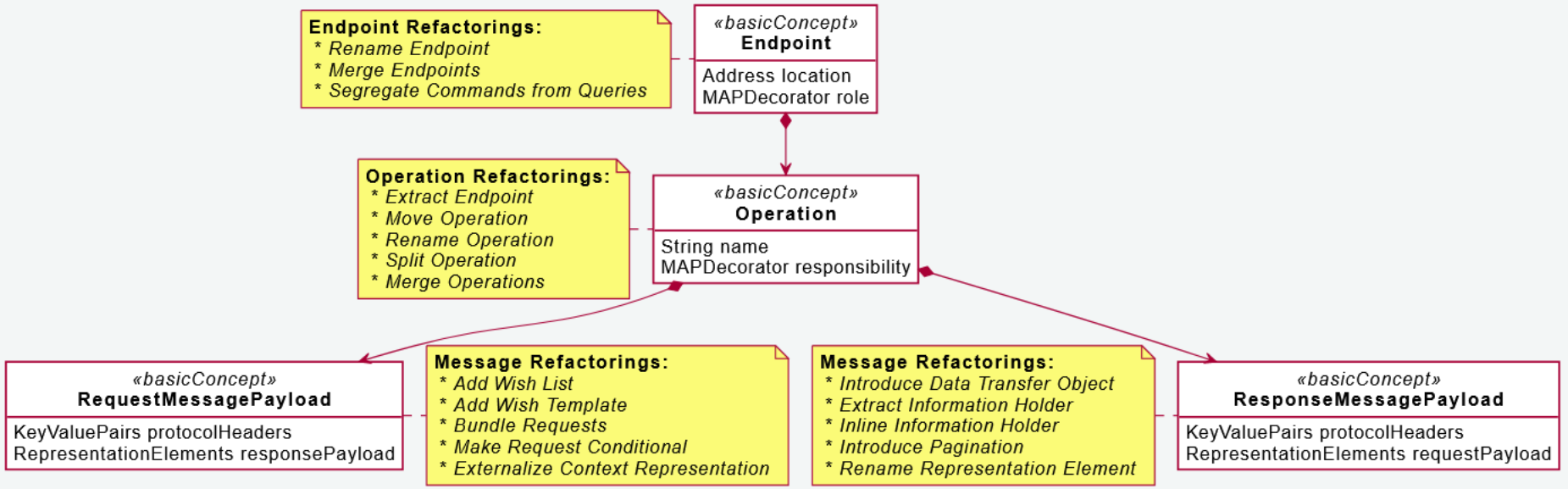
https://interface-refactoring.github.io/ (joint work with Mirko Stocker)

Interface Refactoring Catalog (Emerging)

News (08/2022): The patterns featured in this catalog form the core of a new book in Vaughn Vernon's Signature Series at Pearson, scheduled for publication in December 2022. Read more about these exciting news on the API design patterns website.

Published Refactorings

An overview of the refactorings, set in *italics*, is (links available on the Refactorings by Target page):



https://interface-refactoring.github.io/ (joint work with Mirko Stocker)

From API Design Smells to Refactorings: Smell Browser

API does not get to the POINT	2	Atomicity and consistency management issues	1	Change log jitter or commit chaos	1
Client community smaller than expected	1	Cloud-native traits violated	1	Combinatorial explosion of input options	1
Cryptic or misleading name	2	Curse of knowledge	2	Data lifetime mismatches	2
Endpoint implementation spaghetti	1	Evolution strategy does not match client expectations	1	Extreme decomposition	1
Feature/release inertia a.k.a. stale roadmap	2	God endpoint	2	High coupling	1
High latency/poor response time	4	Lack of trust and confidence	1	Large and/or partially unknown user base	1
Leaky encapsulation	3	Low cohesion	1	Overfetching	3
Polling proliferation	1	Quality-of-Service (QoS) fragmentation and scattering	1	REST principle(s) violated	2
Resistance to change caused by uncertainty	3	Role and/or responsibility diffusion	4	Same backend system and/or domain data processed by multiple endpoints	1
Security by obscurity	1	Single responsibility spread	2	Sloppy or ill-motivated naming conventions	1
Structured artifact serialized and therefore strangled	1	Tacit semantic changes up to incompatibilities creep in	1	Tight coupling of data contract	1
Tight coupling to a communication protocol	1	Too coarse-grained security or data privacy	2	Underfetching	3

MDSL Web: Refactorings as Model Transformations

Target Endpoint:

ConferenceMan

Target Operation:

downloadSessio

```
API description APISupportingConferenceManagementStory1
data type SessionInformation {"sessionId":D, "title":D, "abstract":D, "speaker":D, "time":D , "location":D}

endpoint type ConferenceManagementStory1ScenarioRealizationEndpoint supports scenario ConferenceManagementStory1Scenario
exposes
operation downloadSessionInformation with responsibility RETRIEVAL_OPERATION
  expecting payload  "attendeeId": ID<int>
  delivering payload SessionInformation*

scenario ConferenceManagementStory1Scenario
  story ConferenceManagementStory1
    a "ConferenceManagementApp" wants to "downloadSessionInformation" with "sessionId" and "title" and "abstract" and "speaker"
```

Refactoring/Transformation (Reference: Interface Refactoring Catalog, MDSL Transformations):

Select the transformation or refactoring to perform:

Refactor/Transform | Move to IDL Generation and Download

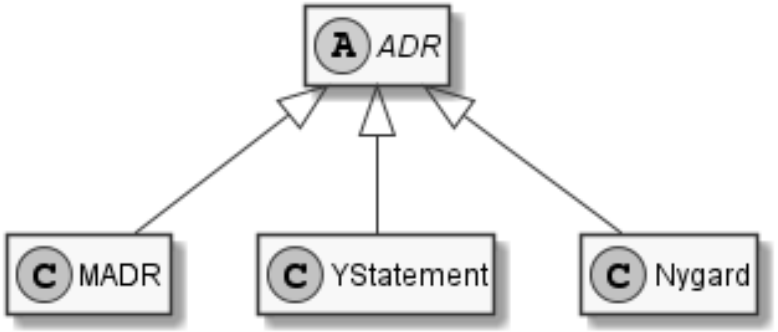
Both forward engineering and refactoring supported by MDSL transformations

https://mdsl-web.up.railway.app

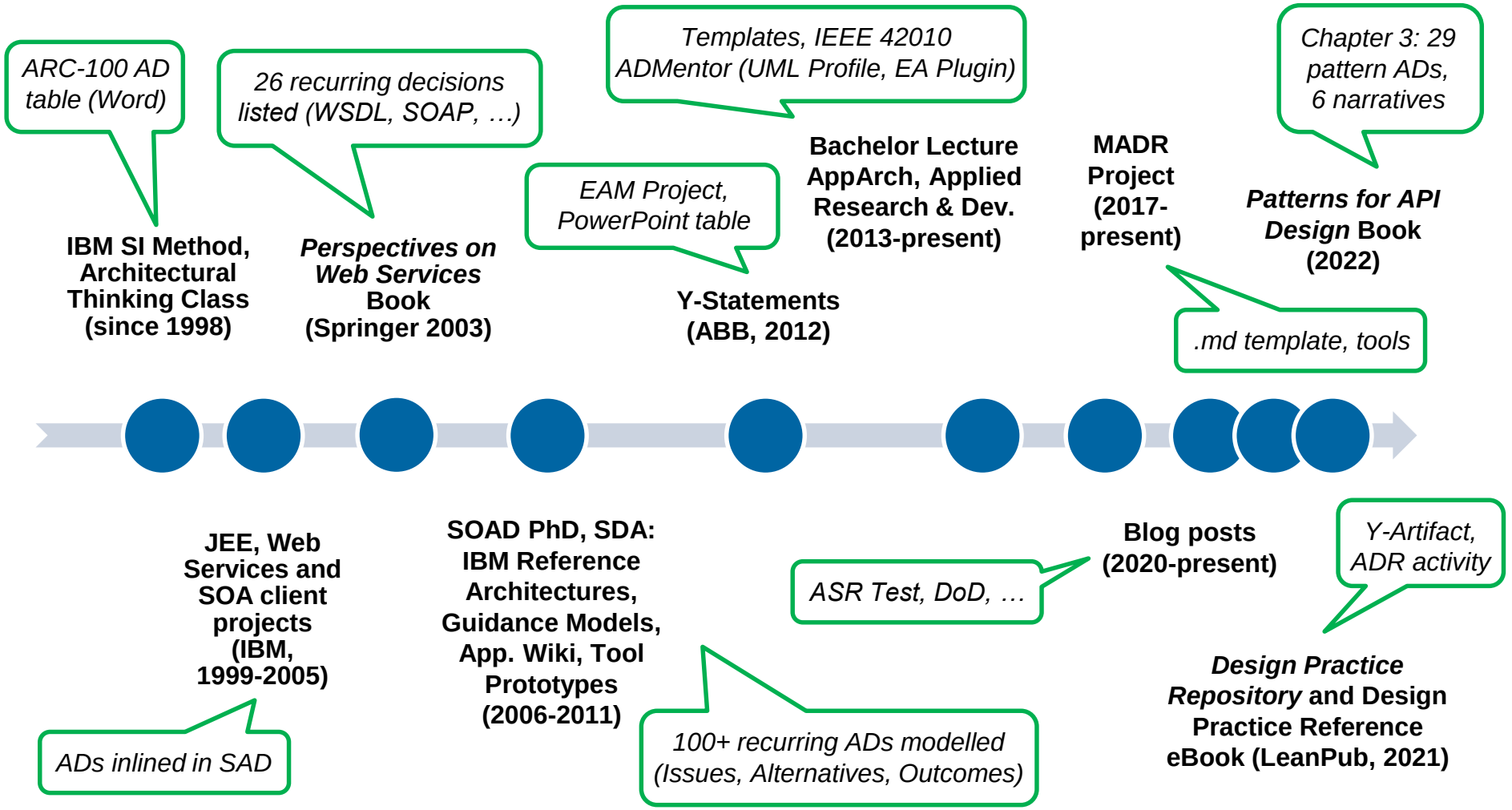
Agenda

- Context and motivation
  - Stepwise API and service design (by example)
  - Pattern languages as knowledge brokers
- A pattern language for API design (and message design)
  - Overview, domain model
  - Selected patterns
- Refactoring to patterns
- **Pattern selection decisions and other hard design concerns (tradeoffs)**
- Concluding thoughts
  - Pattern adoption
  - Open research questions

https://adr.github.io/



My Architectural Decision (AD) Journey 1998 – 2022

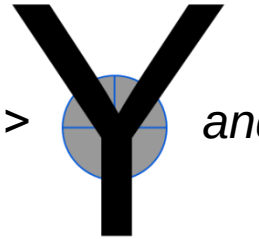


Y-Template (ABB Software Dev. Improvement Initiative)

Reference: ABB, SATURN 2012

In the context of <use case uc and/or component co>,

… facing <non-functional concern c>,

… we decided for <option o1>  *and neglected <options o2 to on>,*

… to achieve <quality q>,

… accepting downside <consequence c>.

(somewhat) adopted by the community, examples:
https://cards42.org/#adr and https://herbertograca.com/2019/08/12/documenting-software-architecture/

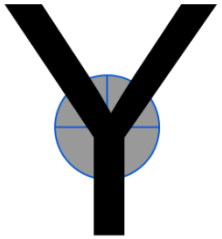
Exercise: API Design with Patterns (Sample System)

- Let's capture a pattern selection AD (in the Conference Management scenario)
  - See Chapter 3 of our book for examples ☺

In the context of API user story 1 (session information download),

facing client diversity, (sometimes) slow networks and sub-second response time requirements,

we decided to expose two Retrieval Operations in the API endpoint, one embedding detailed information and one linking to it

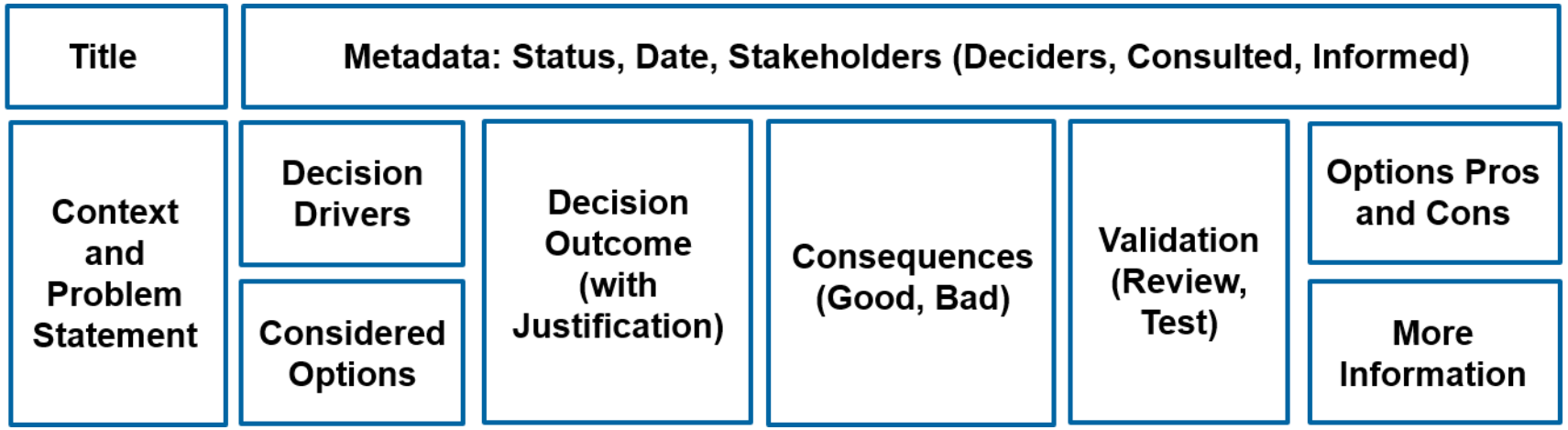


and neglected usage of other quality patterns (such as Wish List and Pagination)

to achieve the required performance and a splendid developer experience,

accepting that implantation and test effort almost doubles.

Markdown Architectural/Any Decision Records (MADR)



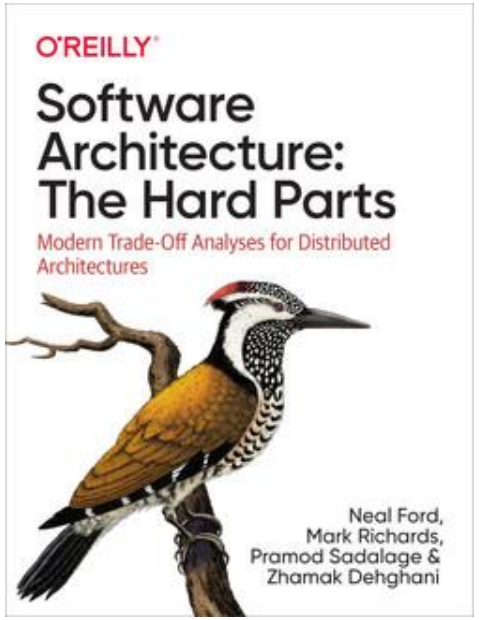
Where is the Markdown?
[In the GitHub repo](https://madr.hithub.io)

"The Markdown ADR (MADR) Template Explained and Distilled" on Medium has details

… and there is a ZEUS 2018 paper about template, concepts, tools (Oliver Kopp et al)

API/Service Design: The Hard Parts

Video **by Neil Ford**, TL;DR: "tradeoffs, always"



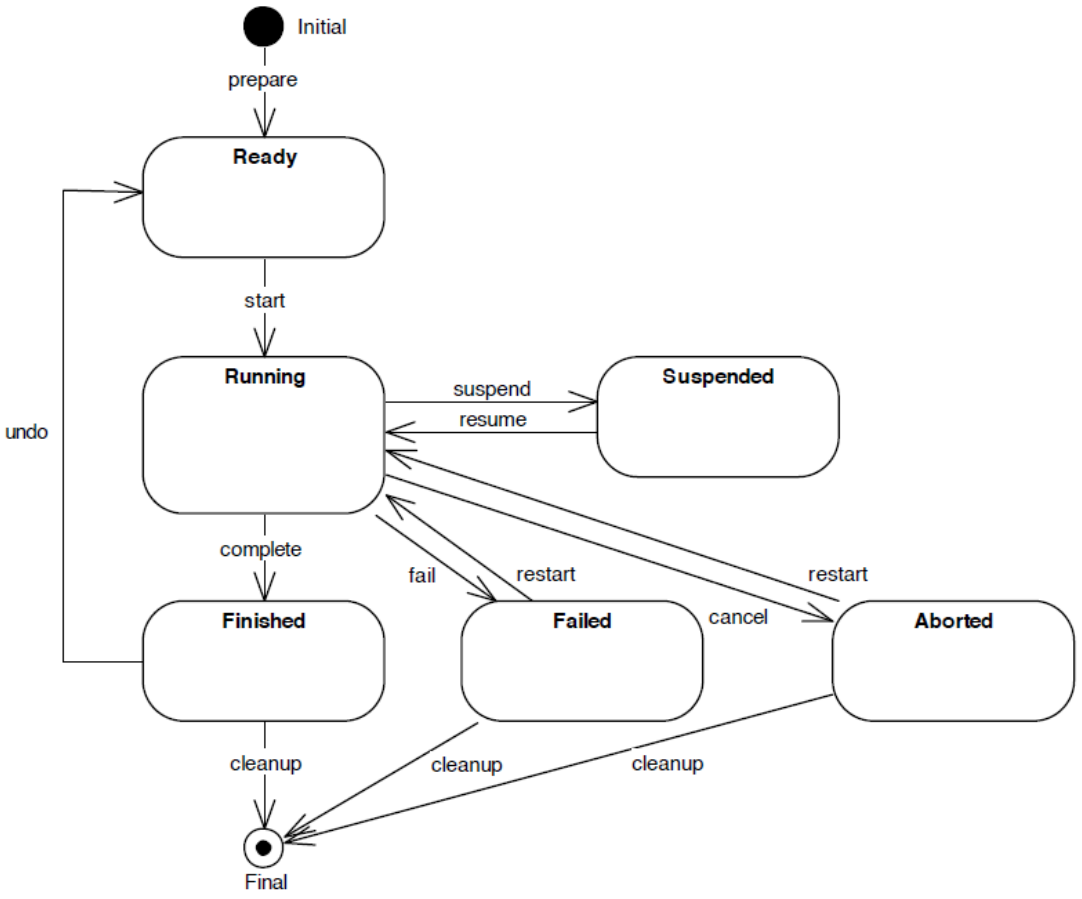
- **Communication**
  - Synchronous vs. asynchronous
- **Coordination**
  - Central(ized) vs. decentral(ized)
- **Consistency**
  - Strict vs. eventual

What are the pros and cons of these options?

What is the relationship between these recurring ADs and API design?

Frontend BPM vs. BPM-as-a-Service

- Business Activity Processor variant of State Transition Operation pattern
  - Book, website archive, EuroPLoP 2020 paper
- Each API operation realizes one state transition, whose mileage may wary:
  - Entire business process
  - Single activity
- General state machine to be adapted according to domain requirements
  - E.g. DDD, Context Mapper DSL and tools
- Technology mapping:
  - HTTP POST, PUT, PATCH
  - Mutations in GraphQL



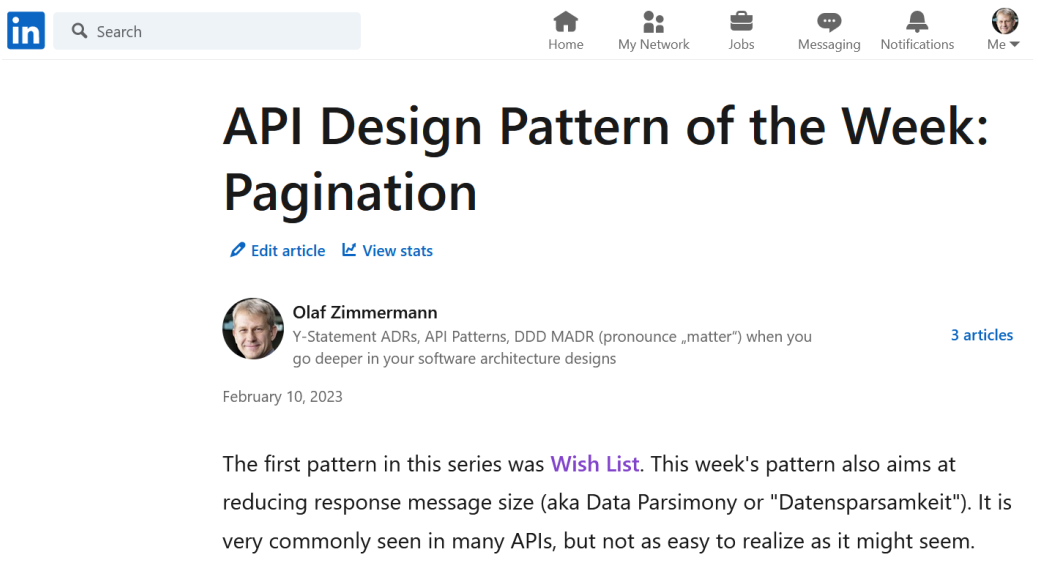
Agenda

- Context and motivation
  - Stepwise API and service design (by example)
  - Pattern languages as knowledge brokers
- A pattern language for API design
  - Overview, domain model
  - Selected patterns
- Refactoring to patterns
- Pattern selection decisions and other hard design concerns (tradeoffs)
- **Concluding thoughts**
  - **Pattern adoption**
  - **Open research questions**

Discussion

- Do the patterns names work for you?
- Do the presented solutions make sense?
- Which patterns might be missing?

- Pros and cons of the options
  - for API design/architecture decisions
- Sample scenarios



In collaboration with Microservices Community

Ongoing Research, Future Research Topics

- API refactoring, architectural refactoring knowledge and tools
  - First slice of catalog submitted to EuroPLoP 2023
- Language, framework and tool support for the patterns
  - MDSL, Jolie (a microserivce programming language)
- Methods and tools for reengineering "monolith to microservices"
  - Bring back the notion of guidance models (ADs first)? See Chapter 3 of our book.
- Events and APIs (aka asynchronous integration)
- AI/ML for API design and documentation?
- Ethical (Empathic?) Software Engineering

More Information: api-patterns.org and blogs

- https://ozimmer.ch/blog/ und (shortened subset) https://docsoc.medium.com/

Introduction to Microservice API Patterns (MAP)

Olaf Zimmermann
University of Applied Sciences of Eastern Switzerland, Rapperswil, Switzerland
ozimmerm@hsr.ch

Mirko Stocker
University of Applied Sciences of Eastern Switzerland, Rapperswil, Switzerland
mirko.stocker@hsr.ch

Daniel Lübke
iQuest GmbH, Hanover, Germany
ich@daniel-luebke.de

Cesare Pautasso
Software Institute, Faculty of Informatics, USI Lugano, Switzerland
c.pautasso@ieee.org

Uwe Zdun
University of Vienna, Faculty of Computer Science, Software Architecture Resear
Vienna, Austria
uwe.zdun@univie.ac.at

Abstract

The Microservice API Patterns (MAP) language and supporting website premierec
at Microservices 2019. MAP distills proven, platform- and technology-indepen
recurring (micro-)service design and interface specification problems such as fi
service granularities, rightsizing message representations, and managing the evolu
their implementations. In this paper, we motivate the need for such a pattern
the language organization and present two exemplary patterns describing altern
representing nested data. We also identify future research and development direc

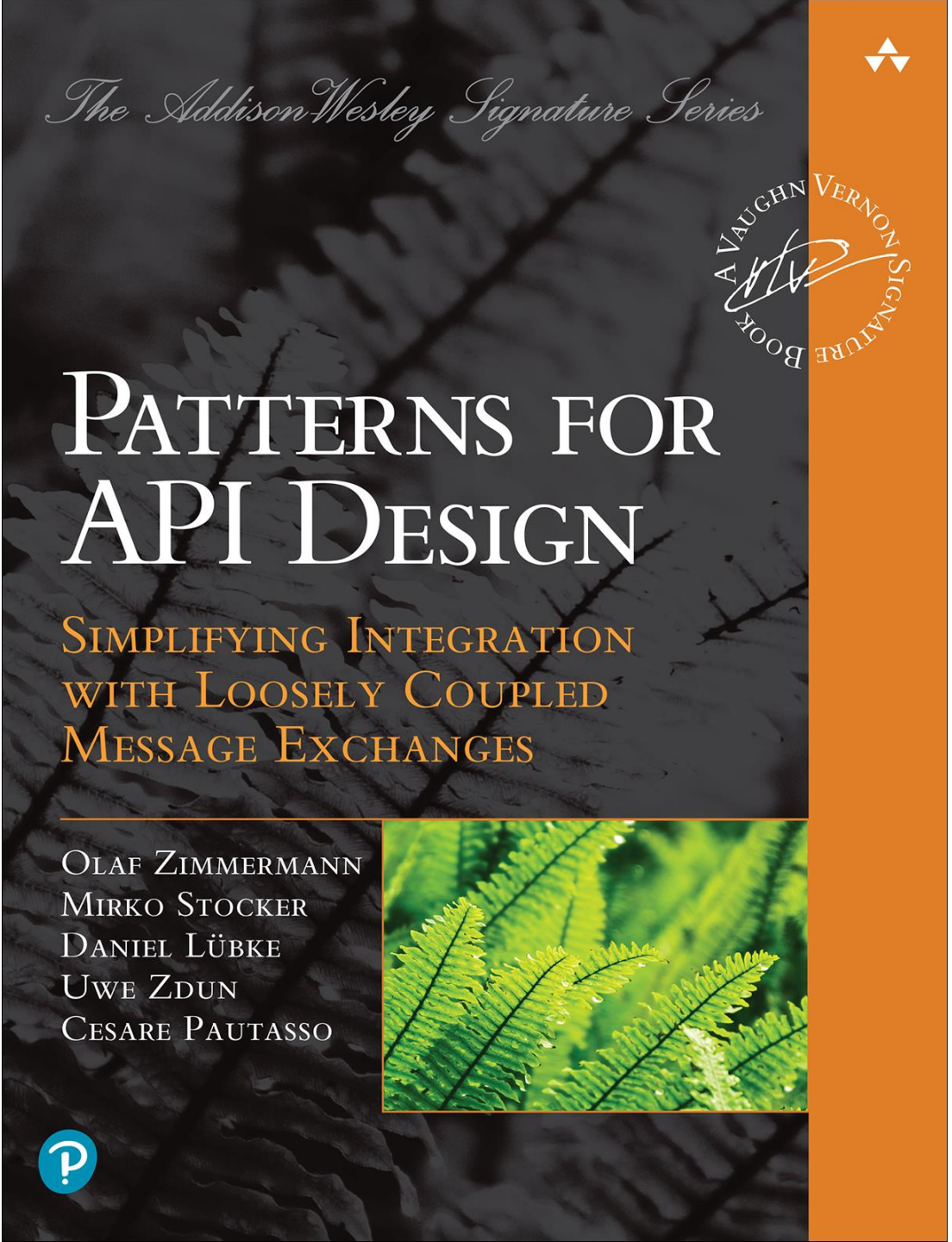
2012 ACM Subject Classification Software and its engineering → Patterns; Softw
eering → Designing software

Keywords and phrases application programming interfaces, distributed systems, enterprise applica-
tion integration, service-oriented computing, software architecture

Digital Object Identifier 10.4230/OASIcs.Microservices.2017-2019.4

https://microservice-api-patterns.org/publications

https://www.youtube.com/watch?v=cNp7ys0g0Bs



Order & Save 35%* on eBook at informit.com/api-patterns

- Use code **API-PATTERNS** during checkout
- Offer only good at informit.com
- eBook – DRM-Free PDF & EPUB

Print books available**
Please check your local or online store where you purchase technical related books.

*Discount code API-PATTERNS is only good at informit.com and cannot be used on the already discounted book + eBook bundle or combined with any other offer. Discount offer is subject to change.

**If your order print books from InformIT, your order is subject to import duties and taxes, which are levied once the package reaches the destination country.