Pattern-oriented API Refactoring: Addressing Design Smells and Stakeholder Concerns

Mirko Stocker
Eastern Switzerland University of Applied Sciences (OST)
Switzerland
mirko.stocker@ost.ch

Olaf Zimmermann
Eastern Switzerland University of Applied Sciences (OST)
Switzerland
olaf.zimmermann@ost.ch

Stefan Kapferer
Eastern Switzerland University of Applied Sciences (OST)
Switzerland
stefan.kapferer@ost.ch

Abstract

In distributed systems, remote Application Programming Interfaces (APIs) let architectural components such as microservices communicate with each other; interoperability and satisfactory developer experience are key stakeholder concerns. In response to changing requirements and insights from development and operations, API endpoints and the request and response messages of the exposed operations are actively designed and then modified during the entire life cycle of the system. Refactoring is a crucial practice in agile software development, widely adopted in practice at the code level. Architectural refactoring has been researched but has not been adopted nearly as widely as code-level refactoring. This paper continues our work on refactoring remote APIs, which we introduced at EuroPLoP 2023. We present a second slice of seven API refactorings pulled from our online Interface Refactoring Catalog, many of which target API design patterns: *Extract Information Holder*, *Inline Information Holder*, *Extract Operation*, *Rename Operation*, *Make Request Conditional*, *Encapsulate Context Representation*, and *Introduce Version Identifier*. Besides context, problem, and step-by-step solution, we also motivate the refactorings by stakeholder concerns and identify the design smells that refactoring can address. All refactorings are illustrated with implementation code snippets, excerpts from API specification, and/or examples of messages exchanged at runtime. The paper concludes with an outlook to future work.

CCS Concepts

• **Software and its engineering** → **Patterns**; *Designing software.*

Keywords

application programming interface, cloud computing, design patterns, enterprise application integration, interface definition languages, refactoring, software

ACM Reference Format:
Mirko Stocker, Olaf Zimmermann, and Stefan Kapferer. 2024. Pattern-oriented API Refactoring: Addressing Design Smells and Stakeholder Concerns. In *29th European Conference on Pattern Languages of Programs, People, and Practices (EuroPLoP 2024), July 03–07, 2024, Irsee, Germany.* ACM, New York, NY, USA, 24 pages. https://doi.org/10.1145/3698322.3698334

EuroPLoP 2024, July 03–07, 2024, Irsee, Germany
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1683-6/24/07
https://doi.org/10.1145/3698322.3698334

1 Introduction

Creating robust, scalable, and maintainable software systems is a constant challenge. Developers have to adapt and enhance their software to meet changing requirements, accommodate new features, and address different quality aspects that meet the goals and expectations of stakeholders. Refactoring and design patterns play a key role in shaping the quality and longevity of software solutions.

Refactoring is the disciplined activity of restructuring code without altering its external behavior. It is an essential agile practice that allows developers to eliminate technical debt, enhance code readability, and improve the overall maintainability of software systems [9]. Refactoring serves as a cornerstone for maintaining a codebase, eliminating design and architectural smells that can accumulate over time. Design *smells* signal potential issues in the code's structure and design. By recognizing and addressing these design smells through thoughtful refactoring, developers can ensure the long-term technical sustainability of their software solutions. Refactoring is typically aimed at code-level elements such as classes, methods, and variables. However, the same principles can be applied to APIs and their architectural elements.

Design patterns complement the principles of refactoring. Patterns encapsulate best practices and "communicate wisdom and insight in computer/software systems design" [23]. They provide a common language for developers to communicate and share knowledge. Through refactoring, software can also be aligned with a design pattern to improve its understandability [18]. Many of the patterns that our refactorings use come from a pattern language for microservice and remote API design, first published in EuroPLoP proceedings 2017 to 2020 [21, 32, 35, 36, 40] and published in book-length in "Patterns for API Design: Simplifying Integration with Loosely Coupled Message Exchanges" [41]. Appendix A provides an overview of the API design patterns from this book that are referenced in this paper.

This paper presents a catalog of refactorings that target APIs and their architectural elements. We call these refactorings *interface refactorings* to distinguish them from code-level refactorings. In our API domain model in "Patterns for API Design" [41], we describe an API as "a collection of endpoints" that offer "operations" to "communication participants" (also called "API clients" and "API providers" depending on their role in the communication). Clients of the API exchange structured request and response messages with the API provider. Figure 1.1 shows the targets of our refactorings in terms of this domain model.

The Interface Refactoring Catalog (IRC) currently features 24 refactorings. Many of these refactorings use API design patterns as their targets. A first slice of eight IRC entries was published

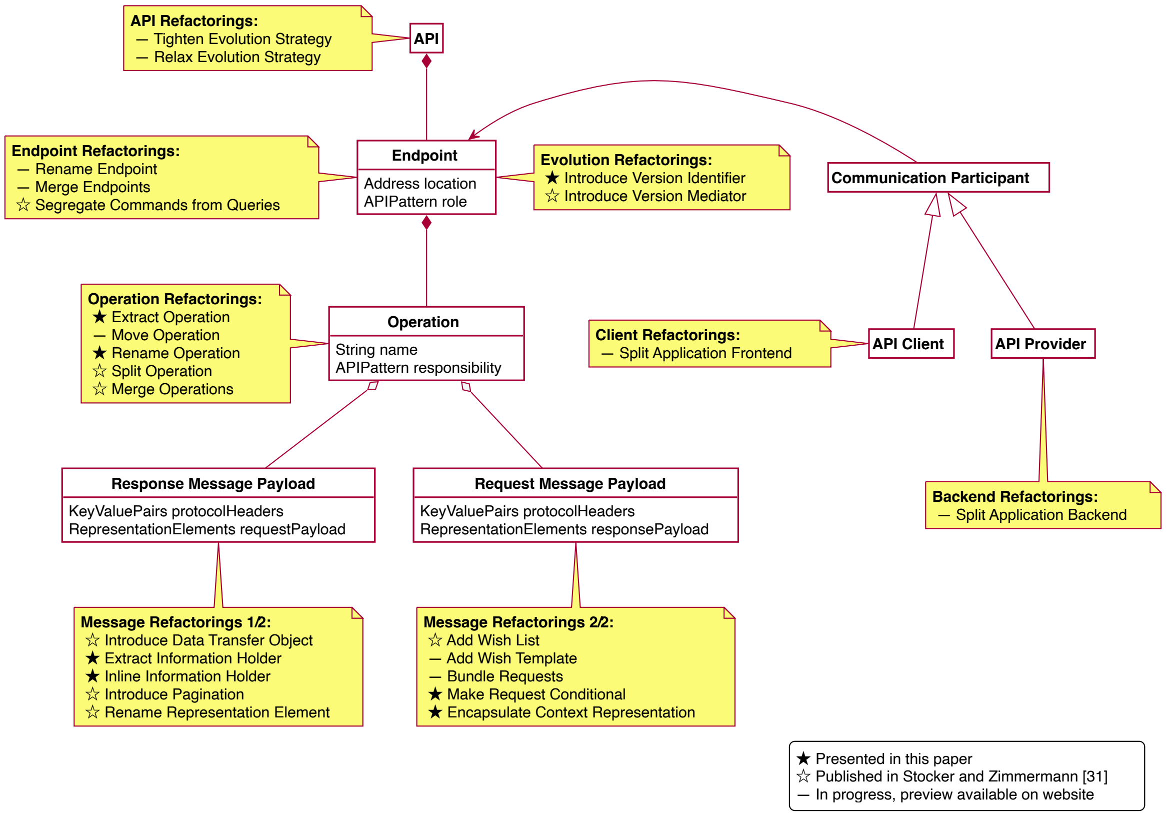
API Refactorings:
— Tighten Evolution Strategy
— Relax Evolution Strategy
API
Endpoint Refactorings:
— Rename Endpoint
— Merge Endpoints
☆ Segregate Commands from Queries
Endpoint
Address location
APIPattern role
Evolution Refactorings:
★ Introduce Version Identifier
☆ Introduce Version Mediator
Communication Participant
Operation Refactorings:
★ Extract Operation
— Move Operation
★ Rename Operation
☆ Split Operation
☆ Merge Operations
Operation
String name
APIPattern responsibility
Client Refactorings:
— Split Application Frontend
API Client
API Provider
Response Message Payload
KeyValuePairs protocolHeaders
RepresentationElements requestPayload
Request Message Payload
KeyValuePairs protocolHeaders
RepresentationElements responsePayload
Backend Refactorings:
— Split Application Backend
Message Refactorings 1/2:
☆ Introduce Data Transfer Object
★ Extract Information Holder
★ Inline Information Holder
☆ Introduce Pagination
☆ Rename Representation Element
Message Refactorings 2/2:
☆ Add Wish List
— Add Wish Template
— Bundle Requests
★ Make Request Conditional
★ Encapsulate Context Representation
★ Presented in this paper
☆ Published in Stocker and Zimmermann [31]
— In progress, preview available on website

Figure 1.1: Refactorings by targeted API element (as defined in API domain model from Zimmermann et al. [41])

in "API Refactorings to Patterns: Catalog, Template and Tools for Remote Interface Evolution" [31] (Note that we use the terms API refactoring and interface refactoring interchangeably). This paper presents the next slice; the entire catalog is available at interface-refactoring.github.io. Five of the refactorings in this paper target API design patterns (*Extract* and *Inline Information Holder*, *Make Request Conditional*, *Encapsulate Context Representation*, *Introduce Version Identifier*); the remaining two ones change API structure and names (*Extract Operation*, *Rename Operation*).

While the refactorings captured in this template are not patterns in the classical sense [11], they share many properties with software design patterns: refactorings are also applied in a specific context to solve a particular problem. Different forces apply, requiring trade-offs and decisions. While patterns are mined from known uses, our refactorings are derived from our own professional experience. Their presentation is inspired by literature such as the "Refactoring" book [9], with selected code refactorings from the book being transcribed and transferred to the domain of APIs.

Usage examples of the refactorings are shown in the context of Lakeside Mutual, a fictitious insurance company that serves as an example scenario to demonstrate microservices [39] and domain-driven design [3]. The example also demonstrates many of the API design patterns from Zimmermann et al. [41]; its sample applications consist of several Spring Boot microservices that provide APIs to frontends to create, read, update, and delete insurance policies as well as product and customer master data.

The refactorings are introduced on the API contract level a) using OpenAPI Specification (OAS), a notation to describe HTTP APIs, b) Context Mapper Language (CML) [16], a Domain-specific Language (DSL) for Domain-Driven Design (DDD), and c) Microservice Domain-Specific Language (MDSL), a DSL that allows describing APIs in a technology-agnostic way. To show the changes before and after the refactorings from a client's perspective, we also use the curl command line tool that shows HTTP request and response messages as well as git diff output. Examples at the code level examples are given as well.

The remainder of this paper is structured in the following way. Section 2 discusses related patterns and pattern languages. Section 3 gives an overview of our interface refactoring catalog and presents the refactorings mentioned above. Section 4 summarizes and concludes the paper.

Our layout conventions are as follows: Refactorings are set in *Italics*; those not presented in this paper either link to the IRC website or to our previously published paper (Stocker and Zimmermann [31]). Pattern names appear in SMALL CAPS. We use #hash-tags for quality attributes, e.g., #performance, to discern them from smell names such as ***God endpoint***. When elements in the code examples are referenced in the text, their names are set in `Courier`.

2 Related Work

We already covered related work on the subject rather extensively in our EuroPloP 2023 paper [31]. Therefore, this section concentrates on related patterns.

Most of the interface refactorings presented in this paper use API design patterns from the "Patterns for API Design" [41] book as their targets. Other pattern languages and individual patterns cover API design aspects as well; some of our API refactorings target those patterns. We summarize particularly important ones in the following.

The Data Transfer Object (DTO) pattern of the "Patterns of Enterprise Application Architecture (P of EAA)" [8] book by Martin Fowler is used in many API designs. A DTO is "an object that carries data between processes in order to reduce the number of method calls. [...] The fields in a Data Transfer Object are fairly simple, being primitives, simple classes like strings and dates, or other Data Transfer Objects." Strangler Fig Application by Martin Fowler, or the Backends For Frontends pattern by Sam Newman, cover relevant aspects and are used in our refactorings. Similarly, some patterns of the Domain-Driven Design (DDD) approach by Evans [3] such as Published Language, Aggregate or Entity touch on aspects of remote API design; a practitioner view on the interrelation of remote APIs and DDD is presented in [29]. Likewise, the Cloud Computing Patterns by Fehling et al. [4] cover aspects such as workload types and application tiers that are not only relevant in cloud computing scenarios but for software architecture in general and for API design as well.

Lilienthal and Schwentner [20] present Domain-Driven Refactorings to transform software systems to improve the maintainability of legacy systems. Hohpe and Pillai introduce Refactoring to Serverless to "improve the design of your serverless application by replacing application code with automation code, while using the same programming language."

In the following Section 3 we discuss related work for each refactoring in the seven subsections called "Related Content".

3 The Interface Refactoring Catalog (Second Slice)

In this section, we present seven refactorings. Five of them target the introduction of API design patterns: *Extract Information Holder*, *Inline Information Holder*, *Make Request Conditional*, *Encapsulate Context Representation*, *Introduce Version Identifier*. We also introduce two refactorings changing API structure and names, *Extract Operation* and *Rename Operation*.

Table 1 lists the refactorings along with the design smells and the stakeholder concerns that they address.

All refactorings presented in this section start from a *context and motivation* and introduce several *stakeholder concerns*, including quality attributes and design forces. An *initial position sketch* shows which API parts or architectural elements are targets of the refactoring. In addition to the stakeholder concerns, each refactoring also lists *design smells* that indicate problems with an existing solution. Smells are "structures in the design that indicate violations of fundamental design principles and negatively impact design quality" [33]. See our website for navigating the refactorings by stakeholder concerns and by smells. Starting from the initial position, a series of *instructions* transforms the design into a *target solution sketch* that details how the refactoring can be applied and validated. Each refactoring comes with a concrete *example* that shows the refactoring in action. A discussion of *hints and pitfalls to avoid* follows. The coverage of each refactoring closes with a subsection about *related content*.

3.1 Refactoring: Extract Information Holder

3.1.1 Context and Motivation. An API operation returns multiple related, possibly deeply nested data structures to provide clients with a rich dataset in a single response. We call such data elements Embedded Entities [39]. For example, in an e-commerce application, the request for the profile of a customer might also return their complete purchasing history. This API is very convenient for clients requiring all the information simultaneously. However, it might not be appropriate for all use cases; some API clients might want to retrieve selected purchasing data through subsequent individual requests when they need it.

As an API client, I prefer to retrieve related data elements step-by-step over having to process large structured data sets appearing in a single response message so that I can process individual responses and the data in them quickly and on demand.

3.1.2 Stakeholder Concerns.

- **#performance, #green-software** Assembling, transferring, and processing a response utilizes resources both on the provider and client side. These resources should not be wasted but handled with care and respect for the environment and the energy consumed. Bandwidth and computing power are examples of valuable and costly resources.
- **#evolvability, #coupling** Systems and components evolve at different speeds. Hence, they should not depend on each other unless this is justified in the business requirements. Data dependencies often introduce unwanted coupling that is difficult to detect and resolve.
- **#data-currentness** Data returned by an API might age at different rates. In the e-commerce shop scenario, for instance, the master data of customers (e.g., names, shipping addresses) will change less frequently than transactional data (such as orders). API clients might want to cache some of the data retrieved, which is harder if faster-changing data is embedded in slower-changing data.
- **#security** Not all API clients have the same access privileges. More fine-grained data Retrieval Operations make it easier to enforce related controls and rules, avoiding the risk that restricted data "slips through" accidentally. To revisit the e-commerce scenario, what if the shop software also includes public ratings of products that show the name and picture of the rating customer? Here, only limited and carefully selected information about the customer should be returned.

3.1.3 Initial Position Sketch. The API implementation shown in Figure 3.1 (a) returns Data Elements that contain further nested data.

The refactoring targets response messages in API operations that return rich data representation elements.

Refactoring	Design Smells Addressed	Stakeholder Concerns
Extract Information Holder (Section 3.1) As an API client, I prefer to retrieve related data elements step-by-step over having to process large structured data sets appearing in a single response message so that I can process individual responses and the data in them quickly and on demand.	God endpoint, data lifetime mismatches, overfetching, sell what is on the truck	#performance, #green-software, #evolvability, #coupling, #data-currentness, #security
Inline Information Holder (Section 3.2) As the API provider, I want to reduce indirection by embedding referenced information holder resources so that clients have to issue fewer requests when working with linked data.	Underfetching, leaky encapsulation	#performance, #green-software, #usability, #developer-experience, #offline-support
Extract Operation (Section 3.3) As the API provider, I want to focus the responsibilities of an endpoint on a single role so that a) API clients serving a particular stakeholder group understand the API design intuitively and b) the release roadmap and scaling of the endpoint can be optimized for each group of stakeholders and clients.	Role and/or responsibility diffusion, low cohesion, REST principle(s) constraints, god endpoint, wrong cuts	#reliability, #stability, #single-responsibility-principle, #independent-deployability, #scalability, #security
Rename Operation (Section 3.4) As an API provider, I want to express the responsibilities of an operation in its name so that client developers, API developers, and non-technical stakeholders (end users, product managers) understand the API — and each other in conversations about the API.	Curse of knowledge, role and/or responsibility diffusion, ill-motivated naming conventions, sloppy naming, cryptic or misleading name	#maintainability, #understandability (including #explainability and #learnability)
Make Request Conditional (Section 3.5) As an API provider, I want to be able to tell clients that they already have a recent version of some data so that I do not have to send this data again.	High latency/poor response time, spike load, polling proliferation	#performance, #green-software, #data-access-characteristics, #developer-experience, #simplicity
Encapsulate Context Representation (Section 3.6) As a conversation participant, I want to consolidate all technical metadata in a single place and keep it close to the domain data so that clients and providers can prepare and process it jointly and so that protocols can be switched if that is required to satisfy requirements and constraints that change over time.	Tight coupling to a communication protocol, quality-of-service (QoS) fragmentation and dispersion	#developer-experience, #learnability, #interoperability, #modifiability, #security, #auditability
Introduce Version Identifier (Section 3.7) As an API provider, I want to communicate versions and their compatibility properties explicitly so that clients can react accordingly on changes that affect them during API evolution.	Tacit semantic changes up to incompatibilities creep in, resistance to change caused by uncertainty	#maintainability, #compatibility, #developer-experience

Table 1: Refactorings in the order in which they are presented in this paper along with a goal statement expressed in the form of a user story, the addressed design smells, and the affected stakeholder concerns in the form of quality attributes.

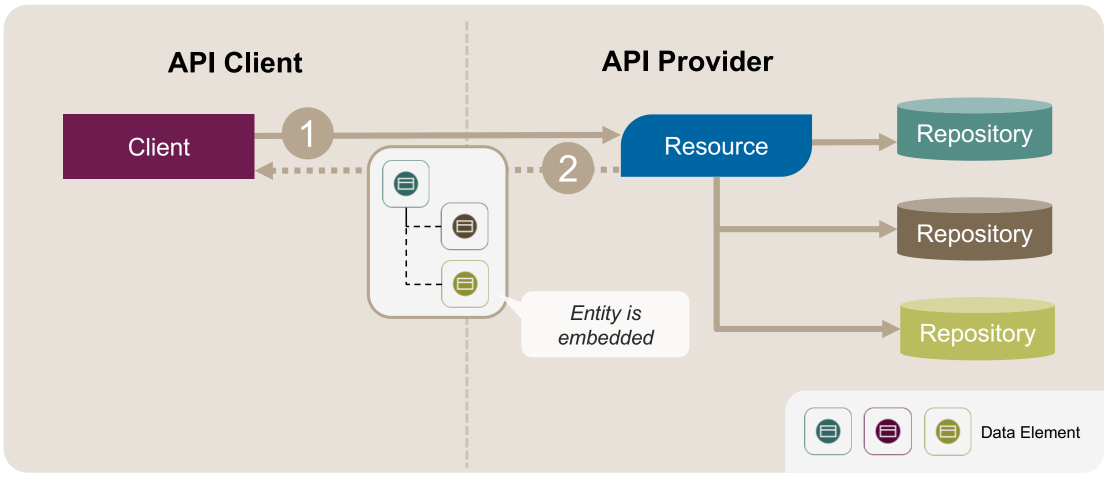

Figure 3.1 (a): Extract Information Holder: Initial Position Sketch: An API provider responds to a request from a client (1) with a message (2) that contains several, possibly nested, Data Elements. The client does not require all the received data.

3.1.4 Design Smells.

God endpoint The endpoint offering this operation might have to access many data sources or backend systems to assemble the response. Derived from the "God Class" smell in object-oriented design, the term describes a class or an object that controls numerous other system parts [27]. Many such dependencies on external systems and data make the API implementation harder to operate and evolve.

Data lifetime mismatches Conflating Data Elements with different lifetimes makes caching, especially cache invalidation, harder. This may happen when slow-changing master data contains fast-changing transactional data (for example, in an Operational Data Holder), but also if transactional data that is often refreshed by clients contains embedded master data that infrequently changes.

Overfetching Clients throw away parts of the received data because the API design follows a one-size-fits-all approach, and the provider includes all data in responses that any present or future client might be interested in. For example, in an e-commerce API, product procurement information might only interest a few clients, while most want to learn about current prices and items in stock.

Sell what is on the truck Implementation data is exposed just because it is there, without any client-side use case.

3.1.5 Instructions. As a preparation for the refactoring, make sure that the following preconditions are met:

1. Decide on which parts of the message to extract. See the Embedded Entity and Linked Information Holder patterns for advice [41].
2. Ensure the API offers a dedicated Retrieval Operation for the data that is currently embedded and will be extracted. If this is not already the case, apply the *Split Operation* [31, pages 12-15] refactoring first. An *Extract Operation* or *Segregate Commands from Queries* [31, pages 20-22] refactoring might also be appropriate to avoid the god endpoint smell.
3. (Optional) If the API operation does not already use a dedicated Data Transfer Object (DTO), apply the *Introduce Data Transfer Object* [31, pages 4-7] refactoring to decouple the API response message from the internal data model. The presence of a DTO allows changing the response message structure without affecting the internal data model. Depending on how deep the Embedded Entity is nested in the response data structure, the *Introduce Data Transfer Object* [31, pages 4-7] refactoring may have to be applied several times. You might be using a programming language or framework where this step is not required. In that case, you can just skip it as long as you have a means to modify the response message structure.

Replace an Embedded Entity with a Linked Information Holder in the following steps:

1. Add a Link Element to the response message that points clients to a Retrieval Operation in an Information Holder Resource. This link realizes/applies the Linked Information Holder pattern; when a DTO is present, it is placed in it.
2. Adjust the tests to the new response structure and run them to observe the changed responses.
3. (Optional) Deprecate or remove the Embedded Entity in the original response message.
4. Clean up the implementation code. For example, services, utilities, or repositories previously used to retrieve the embedded data might not be required anymore here; hence, they should either be moved or removed.
5. Check security policies to ensure that clients can access the linked data.
6. Adjust API clients under your control to issue additional API calls to retrieve the data available at the endpoint referenced in the new Link Element as needed.
7. Update API Description [22], version number, sample code, tutorials, etc., as required. API directories and gateways might have to be updated as well.

3.1.6 Target Solution Sketch (Evolution Outline). The client can use the Link Element returned in response to the initial request to retrieve the related data in a follow-up call, as shown in Step 3 in Figure 3.1 (b).

To reap the full benefits of this refactoring, backward compatibility has to be given up. In the first step, the Embedded Entity could be marked as deprecated to give the clients time to adjust. At a time defined and announced when applying the refactoring, the Embedded Entity is removed from the message payload. The Limited Lifetime Guarantee pattern in Lübke et al. [21] describes this lifecycle management strategy in detail.

3.1.7 Example(s). The following API Description shows an endpoint to retrieve a CustomerProfileDTO, which includes the Embedded Entity PurchaseOrderDTOs.

```
API description ECommerceAPI

data type CustomerProfileId {"id": ID<string>}

data type CustomerProfileDTO {
  "id": CustomerProfileId,
  "name": Data<string>,
  <<Embedded_Entity>> "purchaseHistory": PurchaseOrderDTO*
}

data type PurchaseOrderDTO "DTODesignToBeContinued"

endpoint type CustomerProfileEndpoint
serves as INFORMATION_HOLDER_RESOURCE
exposes
  operation getCustomerProfile
    with responsibility RETRIEVAL_OPERATION
    expecting payload CustomerProfileId
    delivering payload CustomerProfileDTO

API provider ECommerceAPIProvider
  offers CustomerProfileEndpoint

API client ECommerceClient
  consumes CustomerProfileEndpoint
```

This example uses the MDSL notation introduced in Zimmermann et al. [41].

Having applied the refactoring, the client will now receive a link (notice the purchaseHistory link in CustomerProfileDTO):

```
data type CustomerProfileDTO {
  "id": CustomerProfileId,
  "name": Data<string>,
---  <<Embedded_Entity>> "purchaseHistory": PurchaseOrderDTO*
+++  <<Linked_Information_Holder>>
+++    "purchaseHistory": Link<string>
}

data type PurchaseOrderDTO "DTODesignToBeContinued"

+++ endpoint type PurchaseHistoryEndpoint
+++ serves as INFORMATION_HOLDER_RESOURCE
+++ exposes
+++   operation getPurchaseHistory
+++     with responsibility RETRIEVAL_OPERATION
+++     expecting payload CustomerProfileId
+++     delivering payload PurchaseOrderDTO*

API provider ECommerceAPIProvider
  offers CustomerProfileEndpoint
+++   offers PurchaseHistoryEndpoint
```

3.1.8 Hints and Pitfalls to Avoid. Comparing the Target Solution Sketch from Figure 3.1 (b) with the Initial Position Sketch shown in Figure 3.1 (a) shows that the first resource now accesses fewer repositories to assemble the response message. This enables further architectural refactorings such as *Split Application Backend*.

Monitor the API to maintain and challenge the rationale for pattern usage. If most or all client calls follow the given Linked Information Holder, consider embedding the target element in the original representation again using the *Inline Information Holder* refactoring. A deeper discussion of the benefits and liabilities of the two patterns involved in this refactoring and its inverse, Embedded Entity and Linked Information Holder, can be found in the pattern texts in Zimmermann et al. [41].

For the specific question of whether it is preferable to exchange several small messages or a few larger ones, please refer to our article What is the Right Service Granularity in APIs?

3.1.9 Related Content. The inverse API refactoring is *Inline Information Holder*.

If there is no operation to retrieve the linked data yet, the *Split Operation* [31, pages 12-15] refactoring can be used to create one.

After a *Split Operation* refactoring, *Extract Information Holder* can be used to further "split" the response messages of the operations.

The Wish List and Wish Template patterns (and related *Add Wish List* [31, pages 7-10] and *Add Wish Template* refactorings) offer alternative solutions to the problem of how an API client can inform the API provider at runtime about the data it is interested in.

Context Mapper [15], a modeling framework and Domain-specific Language (DSL) for Domain-Driven Design (DDD), implements a refactoring called Split Aggregate by Entity. A DDD Aggregate [3] establishes a transactional boundary around a group of Entities that are persisted together; a data-centric Aggregate could be exposed via an Information Holder Resources on the API level. Splitting such an Aggregate therefore can be seen to correspond to splitting or extracting parts from an API-level Information Holder Resource.

As another example not related to APIs but Web application frontend design, consider the difference between single and multi-page websites. All information is available on a single page regardless of whether it is relevant to each reader. In a multi-page design, the home page gives an overview, and additional information is provided via hyperlinks that can be followed on demand.

3.2 Refactoring: Inline Information Holder

3.2.1 Context and Motivation. An API provides several endpoints that give clients access to data-centric Information Holder Resources. The resources are related and refer to each other, for instance, via hyperlinks. For example, operational data such as an order in an e-commerce shop may reference a Master Data Holder describing the products.

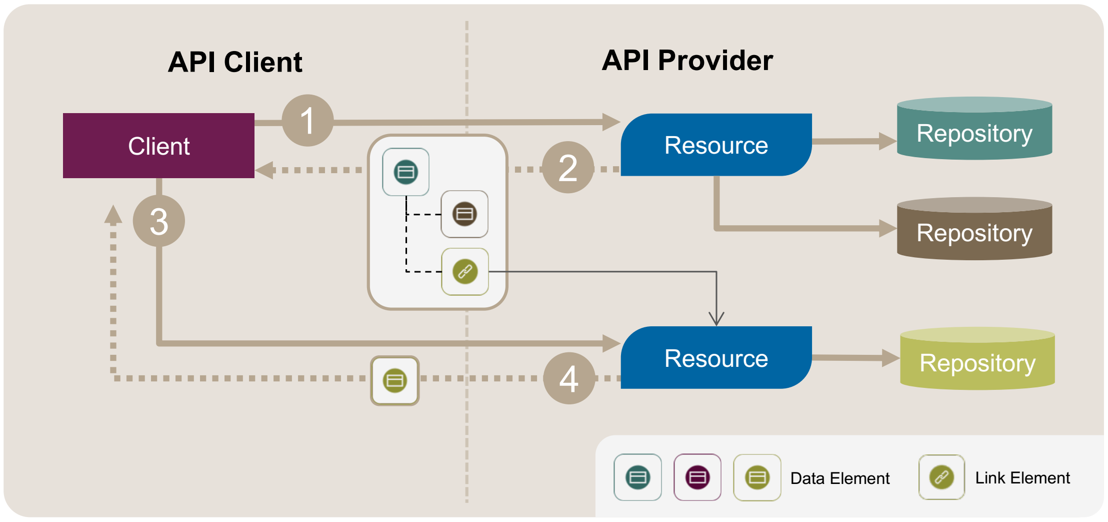

Figure 3.1 (b): Extract Information Holder: Target Solution Sketch: An API client requests (1) a resource from a provider, which responds with a message (2) containing a Linked Information Holder. The client can then request (3) this data when it needs this data. The provider responds (4) with a Data Element that was embedded in the response in the Initial Position Sketch.

Clients of the API are interested in the data of several of these linked Information Holder Resources. To access this distributed data, the clients have to send multiple requests.

As the API provider, I want to reduce indirection by embedding data that is available from one or more referenced Information Holder Resources so that my clients have to issue fewer requests when working with linked data.

3.2.2 *Stakeholder Concerns.*

#performance, #green-software Both API clients as well as providers are interested in keeping the latency and bandwidth usage low and using as few resources as possible.

#usability, #developer-experience An API that provides clients all the required data with as few requests as possible may be easier to use than an API where the client has to issue many requests and requires complex state management on the client side to keep track of multiple API calls. See the blog post API Design Review Checklist: Questions Concerning the Developer Experience (DX) for hints on improving the developer experience.

#offline-support When the connection is unstable or if one wants to build offline functionality into an app, one large request is often better than many small ones.

3.2.3 *Initial Position Sketch.*

The initial response shown in Figure 3.2 (a) contains a Link Element that refers to another information holder of the API. The client has to follow the link to retrieve the data of the referenced resource.

In terms of the API specification, the response Data Transfer Object (DTO) `PolicyDto` contains the `customerId` of the referenced customer and a link to it (notation: OpenAPI Specification, simplified for brevity):

```
paths:
  '/policies/{policyId}':
    get:
      summary: Get a single policy.
      parameters:
        - name: policyId
          in: path
          description: the policy's unique id
          required: true
          type: string
      responses:
        '200':
          description: A single policy.
          schema:
            $ref: '#/definitions/PolicyDto'
definitions:
  PolicyDto:
    type: object
    properties:
      ...
      customerId:
        type: string
      link:
        type: string
```

The response message uses a DTO to transfer the data.

This refactoring targets an operation in an endpoint and its response message.

3.2.4 *Design Smells.*

Underfetching Clients have to issue many requests to get the required data, harming performance.

Leaky encapsulation The implementation data model is leaking through the API. For example, a relational database has been exposed via an API with an endpoint for each table in the database, and now clients must resolve the foreign key relationships between tables.

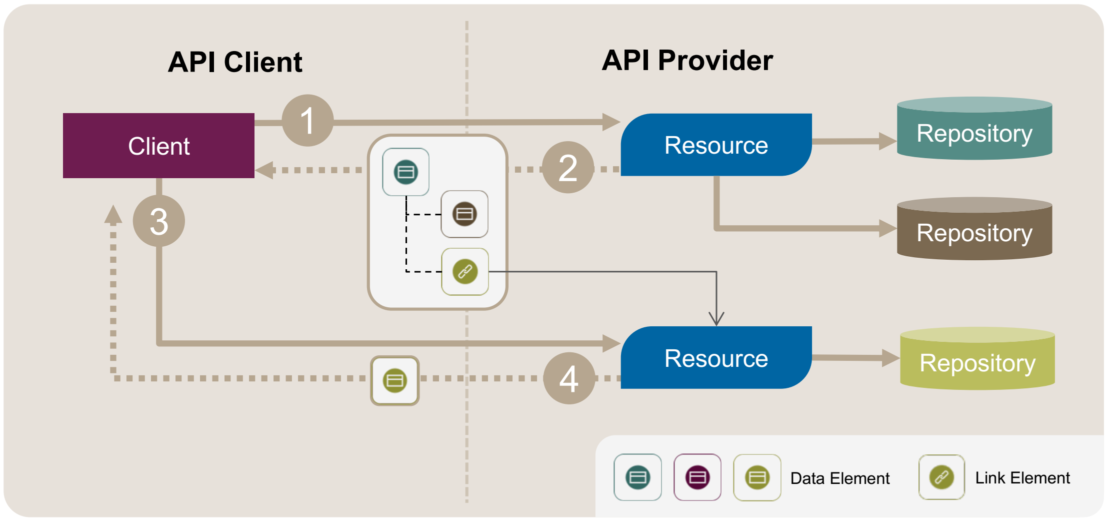

Figure 3.2 (a): Inline Information Holder: Initial Position Sketch: A response message (2) of an API operation that is requested (1) contains a link to a secondary resource that the client has to retrieve using a follow-up request (3, 4). Note that this figure is identical to Figure 3.1 (b), which presents the Target Solution Sketch of the inverse refactoring Extract Information Holder.

3.2.5 Instructions. Instead of providing clients with a hyperlink to fetch the related data, the response message of the API operation includes the referenced data:

1. Decide which linked data to inline/embed into the message. For a discussion of the tradeoffs involved, see the Embedded Entity and Linked Information Holder patterns [41].
2. To transfer the data, insert a new attribute to the DTO.
3. Retrieve the additional entity or value from the repository and add it to the DTO instance.
4. If present, e.g., when using Hypertext as the Engine of Application State (HATEOAS), remove the superfluous link to the resource whose data is now inlined. Only perform this removal if backward compatibility is not needed.
5. Adjust the tests to the new response structure.
6. Clean up the implementation code if necessary (observing the "Rule of Three" of refactoring[1]), for example, by moving duplicated code to a common location.
7. Adjust API clients under your control to remove obsolete API calls, but find and use the inlined data instead.
8. Adjust API Description, version number, sample code, tutorials, etc., as needed.

The link to the referenced resource can be kept in the response message to maintain backward compatibility. In this case, old clients can still follow the link, and updated clients can use the inlined data directly.

3.2.6 Target Solution Sketch (Evolution Outline). After the refactoring, the linked information is included in the initial response, saving the client an additional request. This solution is sketched in Figure 3.2 (b).

[1]The Rule of Three states that when you copy and paste code for the third time, you should extract it into a method [9]. Not to be confused with the Rule of Three of the Patterns community: call it a pattern if there are at least three known uses (https://wiki.c2.com/?RuleOfThree).

The implementation effort on the client also decreases: state management is less complex when fewer requests are needed to fetch required data. These benefits are countered by increased message size, leading to longer transfer times and higher processing and database retrieval effort for the endpoint, which might not be needed by clients after all.

Regarding the API specification, the response DTO now contains the additional data (see the lines at the bottom marked with +++). The link to the referenced resource can be removed if backward compatibility is not needed (see the lines marked with ---).

```
paths:
  '/policies/{policyId}':
    get:
      summary: Get a single policy.
      parameters:
        - name: policyId
          in: path
          description: the policy's unique id
          required: true
          type: string
      responses:
        '200':
          description: A single policy.
          schema:
            $ref: '#/definitions/PolicyDto'
definitions:
  PolicyDto:
    type: object
    properties:
      ...
      customerId:
        type: string
 +++  customer:
 +++    type: object
 +++      properties:
 +++        customerId:
```

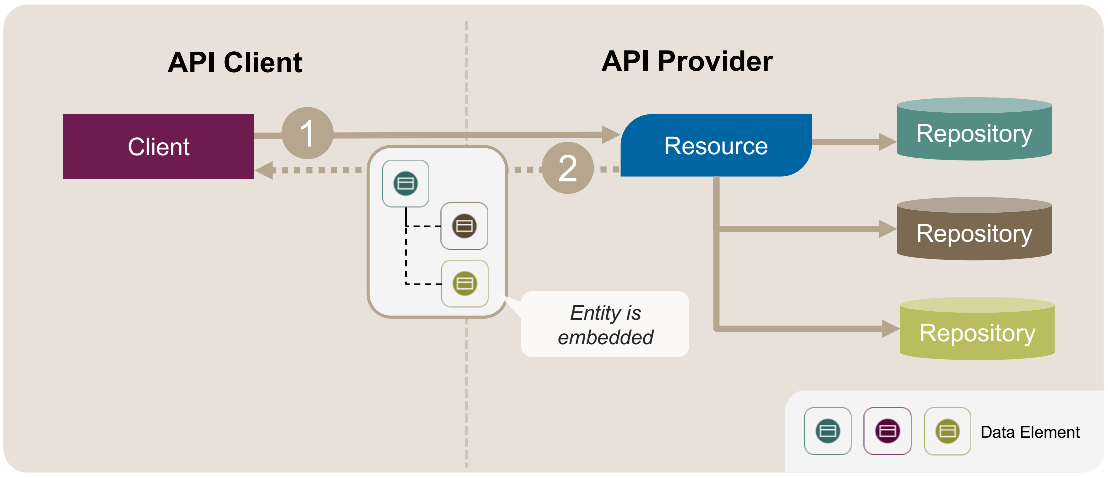

Figure 3.2 (b): Inline Information Holder: Target Solution Sketch: The client requests (1) a resource through the API. The API implementation responds with a rich response message (2) that contains all the data. Note that this figure is identical to Figure 3.1 (a), which presents the Initial Solution Sketch of the inverse refactoring Extract Information Holder.

```
+++          type: string
+++        firstname:
+++          type: string
+++        lastname:
+++          type: string
---   link:
---     type: string
```

3.2.7 *Example(s).* The Policy Management backend microservice of Lakeside Mutual, a fictitious insurance company, contains an endpoint to retrieve the details of a specific policy, along with a reference to the customer through their `customerId`. The following listing shows two requests made using the curl command line tool. It sends an HTTP GET request to the specified URL. The response to this request is a JSON object.

```
curl http://localhost/policies/fvo5pkqerr

{
  "policyId" : "fvo5pkqerr",
  "customerId" : "rgpp0wkpec",
  "creationDate" : "2021-07-07T13:40:52.201+00:00",
  "policyPeriod" : {
    "startDate" : "2018-02-04T23:00:00.000+00:00",
    "endDate" : "2018-02-09T23:00:00.000+00:00"
  },
  ...
}

curl http://localhost/customers/rgpp0wkpec

{
  "customerId" : "rgpp0wkpec",
  "firstname" : "Max",
  "lastname" : "Mustermann",
  ...
}
```

We start the refactoring by adding a new attribute to the DTO:

```
public class PolicyDto extends RepresentationModel {
    private String policyId;
--- private String customerId;
+++ private CustomerDto customer;
    private Date creationDate;
...
```

Depending on the backward compatibility requirements, the `customerId` can be kept in the DTO. Otherwise, it can be removed, as shown above. To fetch the data for the `customer`, the endpoint implementation uses the `customerService`, a Java class residing in the business logic layer of the sample application, to look up the customer and add it to the response DTO:

```
@ApiOperation(value = "Get a single policy.")
@GetMapping(value = "/{policyId}")
public ResponseEntity<PolicyDto> getPolicy(
    @ApiParam(value = "the policy's unique id")
    @PathVariable PolicyId policyId) {
    logger.debug("Fetching policy with id '{}'",
      policyId.getId());
    Optional<PolicyAggregateRoot> optPolicy =
      policyRepository.findById(policyId);
    PolicyAggregateRoot policy = optPolicy.get();
    PolicyDto response = PolicyDto.fromDomainObject(policy);
+++ CustomerDto customer = customerService.
+++   getCustomer(policy.getCustomerId());
+++ response.setCustomer(customer);
    return ResponseEntity.ok(response);
}
```

Note that we use the Java Web framework Spring Boot in this example. The annotations `@GetMapping` and `@PathVariable` are used to instruct Spring that this is an HTTP endpoint that expects a `PolicyId` in its path. The annotations prefixed with `@Api` are used to generate an OpenAPI Specification file from the source code and serve as further documentation.

The customer data is now part of the response message. The client can access the data without issuing a second request:

```
curl http://localhost/policies/fvo5pkqerr

{
  "policyId" : "fvo5pkqerr",
  "customer" : {
    "customerId" : "rgpp0wkpec",
    "firstname" : "Max",
    "lastname" : "Mustermann",
    ...
  },
  "creationDate" : "2021-07-07T13:40:52.201+00:00",
  "policyPeriod" : {
    "startDate" : "2018-02-04T23:00:00.000+00:00",
    "endDate" : "2018-02-09T23:00:00.000+00:00"
  },
  ...
}
```

3.2.8 Hints and Pitfalls to Avoid. The referenced information holder should be part of the same API endpoint. Otherwise, performing the refactoring might introduce undesired dependencies between backend services.

An API endpoint may now interact with more backends or databases than before. This additional dependency might not be desired from a separation of concerns standpoint, for instance, when considering role- or attribute-based authorization [13].

See the Embedded Entity and Linked Information Holder patterns for a deeper discussion of the benefits and liabilities of each pattern.

3.2.9 Related Content. Extract Information Holder inverses this refactoring.

The Wish List and Wish Template patterns both offer alternative solutions to the problem of how an API client can inform the API provider at runtime about the data it is interested in, known as response shaping.

The Backends For Frontends pattern by Sam Newman is another approach to tailoring a backend to the specific needs of a client.

As also mentioned in the inverse refactoring *Extract Information Holder*, Context Mapper [15] implements these refactorings on domain-level (DDD). While *Extract Information Holder* corresponds to Split Aggregate by Entity, *Inline Information Holder* would be established with Merge Aggregates in Context Mapper and DDD.

3.3 Refactoring: Extract Operation

3.3.1 Context and Motivation. One or more API endpoints, for instance, HTTP resources, have been developed, tested, and deployed. One of these endpoints offers multiple operations for clients to call. These operations work with multiple domain concepts. Their functional and technical responsibilities differ regarding stakeholder groups (users, developers, etc.) and their addressed quality concerns. Some operations are process- or activity-oriented, while others offer data storage. At least one operation differs concerning read and/or write access characteristics, access control policies, and data protection requirements. As a consequence, the endpoint serves multiple roles in the API architecture. The operations differ regarding their evolution (e.g., the frequency of changes requiring/leading to new releases).

As the API provider, I want to focus the responsibilities of an endpoint on a single role so that API clients serving a particular stakeholder group understand the API design intuitively, and the release roadmap and scaling of the endpoint can be optimized for each group of stakeholders and clients.

3.3.2 Stakeholder Concerns.

#reliability and #stability Independent endpoints that do not share the same execution context and resources can be deployed independently; operations co-located in a single endpoint, however, share their deployment characteristics. For instance, if a long-running operation causes an API provider-internal error, its sibling operations might suffer from quality-of-service degradations as well. Nygard [25] uses the term stability: "A robust system keeps processing transactions, even when transient impulses, persistent stresses, or component failures disrupt normal processing."

#single-responsibility-principle Architectural principles are affected positively or negatively when APIs are refactored. Here, the Purposeful, style-Oriented, Isolated, channel-Neutral, and T-shaped (POINT) principles for API design apply; extracting an endpoint can improve **P**, **O**, and **I** (but might harm **T** when looking at a single endpoint and not an entire API).

#independent-deployability, #scalability Endpoints can be deployed and then scaled separately, which is one of the defining tenets of microservices-based systems [34]. The fewer operations an endpoint exposes, the easier it is to optimize the scaling for those operations.

#security With multiple operations co-located within a single endpoint, it can be challenging to enforce fine-grained access control policies. Refactoring this endpoint into multiple specialized ones allows for more granular control over access permissions and authorization rules. The security requirements for the data an API endpoint exposes may also differ; hence, separating operations can make it easier to apply data protection measures that ensure confidentiality.

3.3.3 Initial Position Sketch. The design for this interface refactoring looks as follows (notation: MDSL):

```
endpoint type SomeEndpoint
exposes
  operation operation1
    expecting payload "RequestMessage1"
    delivering payload "ResponseMessage1"
   operation operation2
    expecting payload "RequestMessage2"
    delivering payload "ResponseMessage2"
```

See Figure 3.3 (a) for a graphical representation of this Initial Position Sketch.

The refactoring targets are an API endpoint (for instance, an HTTP resource identified with a URI) and one of its operations (for instance, an HTTP verb/method supported by the resource).

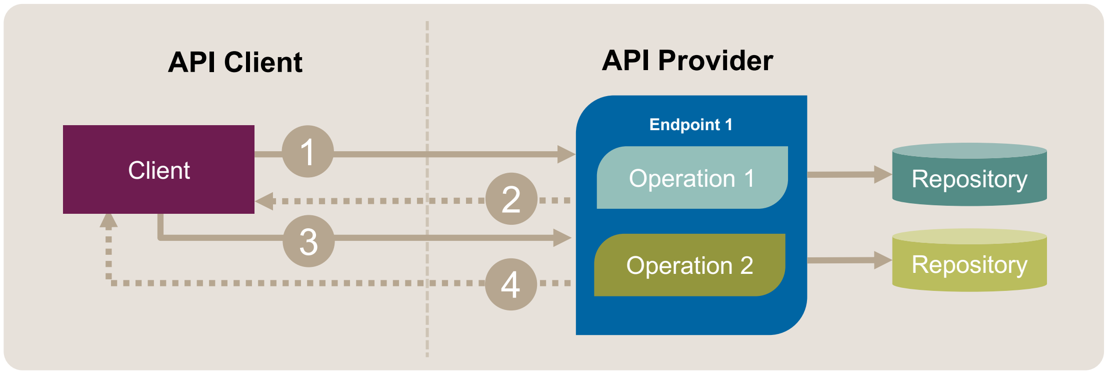

Figure 3.3 (a): Extract Operation: Initial Position Sketch. A client uses two different operations (message exchanges 1-2 and 3-4) in an API endpoint for its communication with the API.

3.3.4 *Design Smells.*

Role and/or responsibility diffusion The endpoint is both an Information Holder Resource and a Processing Resource, or an Information Holder Resource exposes different types of data in different operations (for instance, both master data and operational data). The endpoint operations may have rather diverse functional and technical responsibilities (read vs. write, for instance). As a consequence of one or more of these smells, it is hard to explain the endpoint purpose.

Low cohesion The operations in the endpoint deal with multiple, not necessarily related domain concepts. Consequently, the endpoint has more than one reason to change during its evolution. It serves multiple stakeholder groups and/or its implementation is developed and maintained by multiple teams.

REST principle(s) constraints A key design constraint imposed by the REST style used by many HTTP APIs is the "unified interface," which mandates the use of standard HTTP verbs (POST, GET, PUT, PATCH, DELETE, etc.). These verbs come with certain restrictions; for instance, GET and PUT operations should be idempotent. Sometimes, REST constraints limit extensibility when a resource identified by a single URI runs out of verbs for its operations [28].

God endpoint The endpoint and its operations implementations might have to access many data sources or backend systems to assemble responses to requests. Many such dependencies on external systems and data make the API implementation more complicated to operate and evolve. In object-oriented design, a class or object that controls many other system parts is called a "God Class" [27].

Wrong cuts The endpoint might have been designed to serve multiple purposes, and the operations might have been chosen to be co-located in the same endpoint. This design decision might have been made based on the wrong assumptions or requirements, leading to a design that is hard to maintain and evolve.

3.3.5 *Instructions.*
Follow these steps to extract an endpoint:

1. Remove the operation from the API Description of the source endpoint.
2. Check the general security policies and the client rights management. For example, authorization rules that use endpoint existence and names to determine whether a client application and end-user are permitted to perform an operation might have to be adjusted.
3. Refactor at the code level. For instance, create an additional REST controller class when working with Java and HTTP in Spring and move the implementation of the chosen operation.
4. Create an API Description for the new endpoint that only exposes the extracted operation.
5. Adjust the existing integration tests or add additional ones to verify that the original and new endpoints meet their API Descriptions (both in terms of functional and non-functional characteristics).
6. Evaluate whether the roles and responsibilities of the two endpoints are well-separated and that the refactoring resulted in endpoints with higher cohesion.
7. Inform all API clients about the change and the version that will introduce it. Provide migration information (or support the transition on a technical level, for instance, with an HTTP redirect [7]).

If necessary, repeat these steps with the remaining operations until the roles and responsibilities of the endpoint have been clarified and the smells resolved.

3.3.6 *Target Solution Sketch (Evolution Outline).*
The following simple and abstract MDSL sketch specifies the result of the refactoring at an abstract level (see Figure 3.3 (b) for a graphical representation):

```
endpoint type SomeEndpoint
exposes
  operation operation1
    expecting payload "RequestMessage1"
    delivering payload "ResponseMessage1"

endpoint type ExtractedNewEndpoint
exposes
```

```
operation operation2
    expecting payload "RequestMessage2"
    delivering payload "ResponseMessage2"
```

Note that this sketch does not show signs of bad smells in terms of semantics or qualities; the following example does.

3.3.7 Example. Sometimes, it makes sense to separate commands from queries (see *Segregate Commands from Queries* [31, pages 20-22]). This refactoring is a particular case of endpoint extraction. Hence, the following example can be seen as an example of both *Segregate Commands from Queries* and *Extract Operation.* It starts from a Domain-Driven Design (DDD) featuring a single AGGREGATE [3].

```
Aggregate PublicationEndpoint {
  Service PublicationManagementFacade {
    // a state creation/state transition operation:
    @PaperId add(@PublicationEntryDTO newEntry);

    // retrieval operations:
    @PublicationArchive dumpPublicationArchive();
    Set<@PublicationEntryDTO>
      lookupPublicationsFromAuthor(String author);
    String exportAsBibtex(@PaperId paperId);

    // computation operations (stateless):
    String convertToBibtex(@PublicationEntryDTO entry);
  }
}
```

The notation in the above snippet is Context Mapper Domain-Specific Language (CML) [15]. Context Mapper is a modeling framework for DDD that provides a domain-specific language. DDD can be seen as a form of pattern-oriented, object-oriented analysis and design; "Design Practice Reference" contains introductions to tactic and strategic DDD[37].

This single publication management AGGREGATE (and API endpoint) can be split into two, leading to this design:

```
Aggregate PublicationCommandsEndpoint {
  Service PublicationManagementCommandFacade {
    // a state creation/state transition operation:
    @PaperId add(@PublicationEntryDTO newEntry);

    // computation operations (stateless):
    String convertToBibtex(@PublicationEntryDTO entry);
  }
}

Aggregate PublicationQueriesEndpoint {
  Service PublicationManagementQueryFacade {
    // retrieval operations:
    @PublicationArchive dumpPublicationArchive();
    Set<@PublicationEntryDTO>
      lookupPublicationsFromAuthor(String author);
    String exportAsBibtex(@PaperId paperId);
  }
}
```

Note that this design violates principles such as single responsibility, high cohesion, and low coupling because Bibtex-related operations appear in both endpoints. In response, the *Move Operation* refactoring can be applied on `convertToBibtex`. A third endpoint that exposes the two BibTeX-related operations can also be introduced.

3.3.8 Hints and Pitfalls to Avoid. When applying this refactoring, API designers have to make sure that:

- Concurrent access to business logic and database from two presentation layers, a.k.a. API endpoints, does not cause issues such as lost updates, phantom reads, deadlocks, and so on [8].
- Performance and independent deployability improve as desired (loose coupling of the original and new endpoint). Extracting an endpoint to focus on a single role redistributes the existing responsibilities and logic across multiple endpoints. This redistribution could affect the performance of the API, especially if there are increased interdependencies or additional network calls are introduced. Proper load testing and performance analysis should be conducted to ensure that the refactored API can handle the expected workload and achieve satisfactory response times.
- Maintainability does not suffer because of design erosion, duplication of PUBLISHED LANGUAGE [3], and so on. The refactored endpoints may depend on other services or resources within the system. It is essential to carefully manage and coordinate these dependencies to ensure the refactored endpoints can operate independently and reliably.

3.3.9 Related Content. The *Extract Information Holder* refactoring can be applied in preparation for this refactoring.

When following the BACKENDS FOR FRONTENDS pattern, it might be helpful to extract an endpoint to serve a particular frontend.

This refactoring is reverted by *Merge Endpoints*. *Segregate Commands from Queries* [31, pages 20-22] describes endpoint extraction for a particular reason. *Move Operation* has a similar purpose and nature but does not create a new endpoint.

The STRANGLER FIG APPLICATION pattern describes an approach to migrating a legacy system incrementally by replacing specific functionality with new applications and services instead of replacing it *immediately*. The *Extract Operation* refactoring applied to the strangled legacy system can support such an approach. A backend system exposing multiple service endpoints is generally easier to update incrementally (and replace eventually) than a more monolithic one. The blog post "Refactoring Legacy Code with the Strangler Fig Pattern" provides detailed step-by-step explanations.

3.4 Refactoring: Rename Operation

3.4.1 Context and Motivation. An API endpoint, for instance, an HTTP resource, has been developed, tested, and deployed. The name of one of the operations of the endpoint does not represent its semantics well; there is a mismatch between the operation name and the performed operation. It is not easy to comprehend.

As an API provider, I want to express the responsibilities of an operation in its name so that client developers, API developers, operators, and non-technical stakeholders such as end users and product managers understand the API — and each other in conversations about the API.

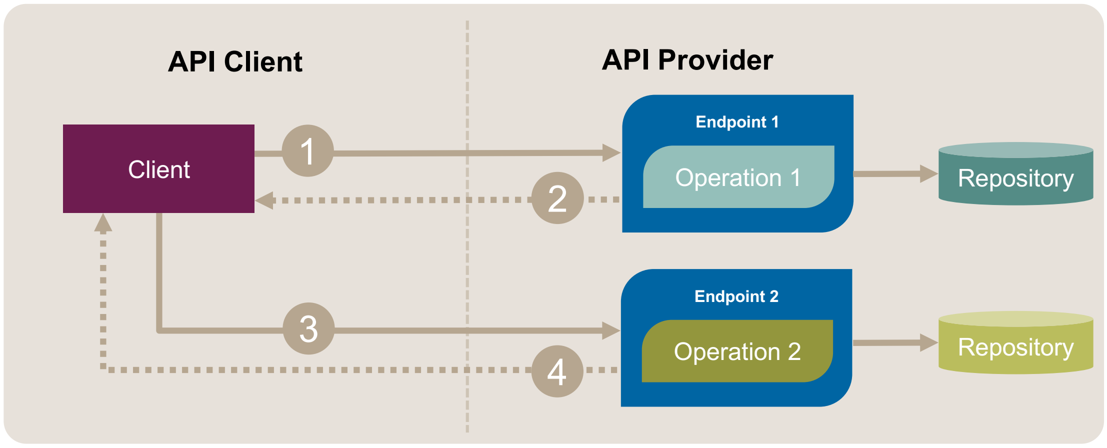

Figure 3.3 (b): Extract Operation: Target Solution Sketch. The two conversations (message exchanges 1-2 and 3-4) with the API now go to operations residing in two different API endpoints.

3.4.2 *Stakeholder Concerns.*

#maintainability APIs should be changed as much as required and as little as possible, and ripple effects be avoided; source code and documentation on client and provider side using a particular name have to be updated if this name changes. Expressive operation names help with orientation during API evolution. Debugging and trouble shooting is also easier if logs and error reports contain meaninfgul names.

#understandability (incl. #explainability, #learnability) All stakeholders involved in development and operations should be able to grasp what an API is supposed to do (and actually does) with ease; it should be straightforward to teach API usage. On the contrary, educated guesses and implicit assumptions are likely to cause misunderstandings that lead to technical risk and defects later on.

3.4.3 *Initial Position Sketch.*

This refactoring deals with a single operation.

This is a rather trivial initial design to be improved with this refactoring, specified in the Microservice Domain-Specific Language (MDSL) notation:

```
endpoint type GenericEndpointOfUnknownRole
exposes
  operation hardToGraspName
    expecting payload "SomeRequestMessage"
```

It is unlikely that an identifier such as `hardToGraspName` is part of the vocabulary of any application domain or genre that the API deals with (such as finance, e-commerce/retail, or distributed control system in a factory).

3.4.4 *Design Smells.*

Curse of knowledge The operation name is easy to comprehend – but only for the developers of the API implementation on the provider side. On the contrary, client developers miss required context information.[2]

Role and/or responsibility diffusion The operation is doing something, but the effects of the operation execution are not clear. For instance, it is not specified whether it reads and/or writes provider-side data and application state. The domain model abstractions/concepts that it works with remain fuzzy. Precision might be harmed and ambiguities introduced.

Ill-motivated naming conventions Knowing what no one else knows could be seen as a pragmatic approach to job security; obscuring operation names might be part of such a strategy. However, the attitude driving such naming decisions can be considered unprofessional or unethical [38]; API design and documentation should be seen as a service provided to the client community (and other stakeholders).

Sloppy naming Another example of good intentions gone wrong is trying to be funny when naming program(ming) artifacts; endpoint and operation names are not the most suited places for humor or irony because they distract from the facts.

Cryptic or misleading name The name of the operation is not only difficult to understand but also misleading. It might suggest that the operation does something that it does not, which may have been caused by a change in the implementation of the operation without updating the name.

3.4.5 *Instructions.*

1. Rename the operation in the abstract API contract and any technical API Description (for instance, its OpenAPI description).
2. Refactor on the code level; for instance, apply "Rename Element" or "Rename Method" as offered by many Java IDEs. Optionally, implement a new stub that merely redirects clients to the new endpoint operation; in HTTP, this can be achieved with URL redirection and status code `301` [7].
3. Adjust the test cases and run them (to keep the builds "green").

[2]The term "curse of knowledge" originates from technical writing. See, for instance, hint 5 in the blog post "Technical Writing Tips and Tricks" https://ozimmer.ch/authoring/2020/04/24/TechWritingAdvice.html and a video lecture by Steven Pinker referenced in that post.

4. Update all supporting documentation such as API reference and guides, examples, and tutorials. Check and update security rules if necessary.
5. Inform all known API clients about the change, ideally with detailed instructions how to migrate. Code snippets that can be copy-pasted easily will be appreciated by the client maintainers.

3.4.6 *Target Solution Sketch (Evolution Outline).*

The following MDSL sketch outlines how to improve the naming on an abstract, conceptual level:

```
endpoint type DomainNounAndRDDRoleStereotype
exposes
  operation verbFromDomainLanguage
    expecting payload "DomainLevelTransferObject1"
    delivering payload "DomainLevelTransferObject2"
```

Note that two similar refactorings were applied as well, *Rename Endpoint* and *Rename Representation Element* [31, pages 18-20].

3.4.7 *Example.*

In a publication management system, the remote service layer of a Web application might expose a Community API for Backend Integration, two foundational API patterns. The service might look as follows:

```
Service JabrefAPI {
    @PaperId add(@PublicationEntryDTO newEntry);
```

The notation in this example is CML, the tactic domain-driven design language supported by Context Mapper. Renaming the rather generic names yields this API design:

```
Service PublicationManagementFacade {
  // a state creation/state transition operation
  // (DTO = Data Transfer Object)
  @PaperId addPublicationToArchive(
    @PublicationEntryDTO publicationInformation);
```

Endpoint name, operation name, and parameter name are now free of technical jargon (except for the pattern names `Facade` and `DTO`, with the acronym being explained in the comment).

3.4.8 *Hints and Pitfalls to Avoid.*

Consider consulting the following artifacts (before and after the refactoring):

- Naming conventions on organization, unit, or project level. If you cannot locate such conventions, use this opportunity to establish them. Rename consistently and document the rationale for your naming decisions. For instance, such conventions might state that operation names start with verbs and followed by a noun from the domain vocabulary (see example above).
- Glossaries and the Ubiquitous Language established by a domain model [37]. Note that some community members advise that `BankAccountAggregate` is a bad choice of name while others recommend this domain-pattern pairing convention.
- Coding guidelines, both general and language-specific.

Avoid special characters such as underscore _ in operation names because middleware and tools might not handle them correctly. The same holds for natural language-specific characters such as German “Umlaute”.[3]

[3]As a test, do ä, ö, ü render properly when you read this?

Be careful with metaphors when naming things. Some might not be understood by parts of the target audience, others might cause unwanted reactions. Baseball fans, for example, know what a “curveball” and a “pitcher” are, but this sport is not as global as others. Apply the ones that you do choose consistently and do not mix them wildly.[4] Ask yourself: Would the current name also work in another domain? In how many projects/APIs can this name be found?

Note that this refactoring is not straightforward to apply in HTTP resource APIs due to the REST principle “uniform interface”: One cannot rename the predefined HTTP PUT verb to something that has domain semantics. This constraint represents a deliberate design decision and is inherent to/in the REST style. It is one reason the Web works: it is not necessary to recompile the Web browser when there is a breaking change in the HTML layout of a website. URIs, however, can be changed; hence, the *Rename Endpoint* refactoring often is eligible.

3.4.9 *Related Content.*

This refactoring reverts itself. In code refactoring, there is *Rename Method* [9].

Move Operation is another operation-level refactoring.

The hints in “The Art of Readable Code” [2] also apply to API naming. Many programming language communities also have naming guidelines, such as the C++ Core Guidelines feature naming suggestions. A CppCon 2019 talk by Kate Gregory titled “Naming is Hard: Let’s Do Better” has good advice that applies when choosing API element names.

3.5 Refactoring: Make Request Conditional

3.5.1 *Context and Motivation.*

An API endpoint provides data that changes rarely, and thus, some clients request and receive the same data frequently. Preparing and retransmitting data already available to the clients is unnecessary and wasteful.

As an API provider, I want to be able to tell clients that they already have the most recent version of certain data so that I do not have to send this data again.

3.5.2 *Stakeholder Concerns.*

#performance, #green-software Response, throughput, and processing times concern API clients and providers. Unused data that is prepared, transported, and processed wastes resources, which should be avoided.

#data-access-characteristics API clients might use caching and do not want to retrieve data they already have.

#developer-experience, #simplicity Knowing when and how long to cache which data might be challenging for API clients and providers. Permanent or temporary storage is required. These valid concerns have to be balanced with the desire for performance.

3.5.3 *Initial Position Sketch.*

Figure 3.5 (a) shows the initial position sketch for this refactoring. The client requests some data from the API. Later, the client wants to ensure that the data is still up to date and sends a second request for the same data.

[4]What happens if an elephant enters a room as a Visitor or crosses a Bridge? Should the Flyweight pattern be applied then? Or does it make sense to build a Factory in this case? What will Observers think about this Strategy? [10]

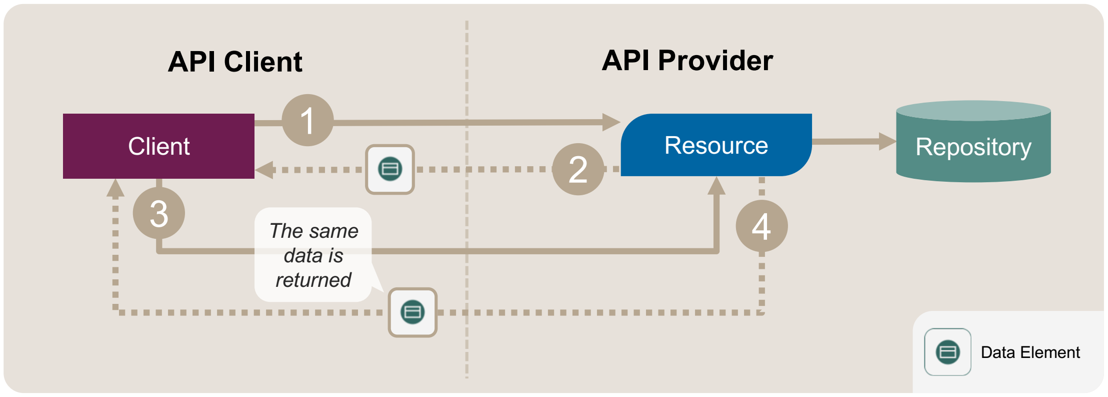

Figure 3.5 (a): Make Request Conditional: Initial Solution Sketch: In the first message exchange (1–2), the endpoint returns one or more Data Elements. Later on (3), the client requests the data from the endpoint again. Because nothing has changed, the provider returns the same data (4) as in the previous response.

This refactoring targets a single API operation and its request and response messages.

3.5.4 *Design Smells.*

High latency/poor response time Load on the API provider is unnecessarily high because the same data is processed and transferred many times over.

Spike load Regular requests for large amounts of data can cause Periodic Workload or Unpredictable Workload [4] for CPU and memory, for instance, when a relatively large JSON object representing the requested data has to be constructed (on the provider side) and read (on the client side).

Polling proliferation Clients that participate in long-running conversations and API call orchestrations ping the server for the current status of processing (“are you done?”). They do so more often than the provider-side state advances.

3.5.5 *Instructions.*

Instead of transmitting the same data repeatedly, the request can be conditional. Condition information is exchanged as metadata to allow the communication participants to determine whether the client already has the latest data version.

1. Decide for one of the two variants of the Conditional Request pattern: data can a) be *timestamped* or b) responses be *fingerprinted* (by calculating a hash code of the response body) [41].
2. Adjust the API specification and implementation to include a conditional Metadata Element in both request and response messages. The request metadata should be optional so that it can be omitted in initial requests; optionality also brings backward compatibility. For the response message, check if the transport protocol provides a special status for this case and consider using it (such as HTTP status code `304 Not Modified`).
3. In the API implementation, evaluate the condition – for example, by comparing the previously mentioned timestamps or fingerprints/hashes – and respond with an appropriate message.
4. Create additional unit or integration tests for the API implementation that validate combinations of metadata presence or absence (with changed and unchanged data).
5. If several operations in the API use Conditional Requests, investigate whether your framework offers a way to implement this functionality in a generic way.
6. Adjust the API client implementations that you oversee (for instance, API usage examples) to utilize the new feature: send conditions and keep previously received data. Adjust the API tests as well.
7. Document the changes, for example, in a changelog data release notes, and release a new API version.

This refactoring can be applied incrementally, for instance, to a single operation or a group of operations. Backwards compatibility is preserved by making the condition metadata optional in the request.

3.5.6 *Target Solution Sketch (Evolution Outline).*

Comparing the solution in Figure 3.5 (b) to the initial position sketch, we see that follow-up requests return a special response message indicating that the data has not changed. The client can then continue to use the data it has already received.

3.5.7 *Example(s).*

The Customer Core microservice of the Lakeside Mutual sample application implements conditional requests in its `WebConfiguration` class. Classes annotated with `@Configuration` can be used to customize the configuration of the Spring MVC framework. The fingerprint-based variant of Conditional Request is applied in its request and response messages:

```
@Configuration
public class WebConfiguration implements WebMvcConfigurer {
    ...

    /**
     * This is a filter that generates an ETag value based
     * on the content of the response. This ETag is compared
     * to the If-None-Match header of the request. If these
     * headers are equal, the response content is not sent,
     * but rather a 304 "Not Modified" status instead.
```

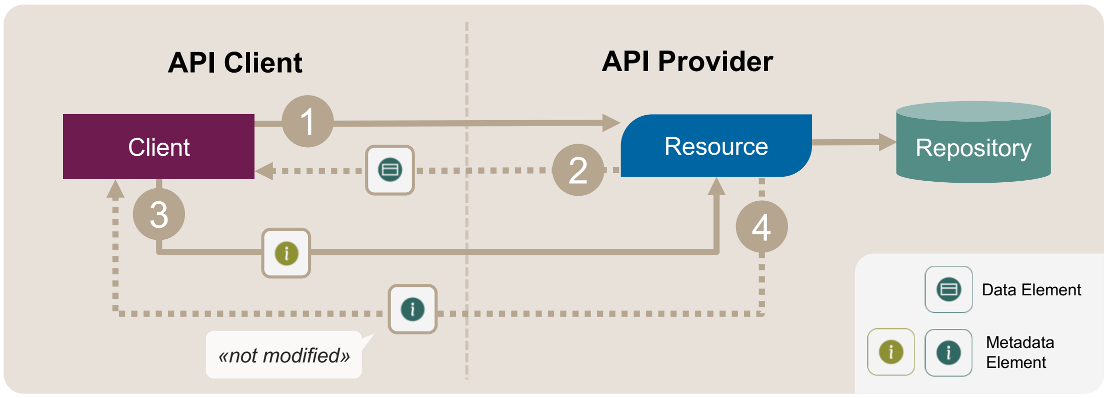

Figure 3.5 (b): Make Request Conditional: Target Solution Sketch: The first exchange (1–2) is the same as in the initial position. In the second request though, the client includes the condition metadata (3) in its request, which in turn allows the provider to respond with a special "not modified" message (4) if the data has not changed.

```
     *
     * By marking the method as @Bean, Spring can call this
     * method and inject the dependency into other components,
     * following the inversion of control principle.
     * */
    @Bean
    public Filter shallowETagHeaderFilter() {
        return new ShallowEtagHeaderFilter();
    }
}
```

The `ShallowEtagHeaderFilter` class is already included in the Spring Framework. Because it is implemented as a filter applied to all requests and responses, the implementation of the individual operations does not have to be adjusted. A consequence of this implementation, and the reason why it is called "shallow" ETag, is that responses are still assembled, hashed and replaced with a `304 Not Modified` response if the hash matches the ETag header.

Alternatively, a Version Identifier could be introduced in the (meta)data to avoid having to retrieve and hash the entire data. This is also supported by Spring Data REST for classes that have an `@Version` property:

```
@Entity
public class CustomerAggregateRoot implements RootEntity {

    @Version
    Long version;

    @EmbeddedId
    private CustomerId id;

    ...
}
```

3.5.8 *Hints and Pitfalls to Avoid.* Before and when making requests conditional, ask yourself:

- How does the additional overhead to calculate the hashes, or the extra storage used by timestamps and versioning numbers compare to the expected savings?
- Does the condition cover all the data returned in the response? For example, when the data contains nested structures, a change in a contained element must be detected. Otherwise, clients might work with stale data.
- How does a Conditional Request count towards a Rate Limit [41]?

Be careful when combining Conditional Requests with a Wish List or Wish Template. The data might not have changed, but the client could request different parts of it. In this case, the cached data is unlikely to be sufficient.

Do not mindlessly start caching all API responses on the client side. Cache design is hard to get right. For instance, knowing when to invalidate cache entries is not trivial [17].

The Conditional Request pattern and this refactoring assume that the server is responsible for evaluating the condition. However, it may make sense for the client to evaluate the condition in order to avoid sending a request to the server. For example, a client could consult the HTTP `Expires` header to decide whether the data retrieved from the server is still current [5]. This doesn't guarantee that the client has the latest data, but depending on the use case, that may not be a problem.

3.5.9 *Related Content.* The online presentation of the Conditional Request pattern coverage presents an example leveraging the Spring framework.

An operation that returns nested data holders that change more or less often than the containing data can prevent this refactoring from being applied. In that case, applying the *Extract Information Holder* refactoring first to separate the nested data holders from the containing data can help. Chapter 7 of Zimmermann et al. [41] provides a comprehensive introduction to API quality.

Our catalog includes an *Introduce Version Identifier* refactoring that focuses on versioning endpoints, not Data Elements.

Conditional requests in Hypertext Transfer Protocol (HTTP/1.1) are defined by RFC 7232 [6].

3.6 Refactoring: Encapsulate Context Representation

3.6.1 Context and Motivation. An API endpoint and its operations have been defined. API client and provider must exchange context information about their interaction, such as the client's location, Quality-of-Service (QoS) control data, or data used to authenticate, authorize, and bill clients. API client-provider interactions might be part of conversations within and across API endpoints, possibly involving external systems as well.

As a conversation participant, I want to consolidate all context information in a single place and keep it close to the domain data so that clients and providers can prepare and process it along with that data. This also allows switching protocols if that is required to satisfy requirements and constraints that change over time.

3.6.2 Stakeholder Concerns.

#developer-experience and #learnability Accessing protocol headers is different from accessing message payload; different local APIs and/or platform-specific libraries have to be used. Consolidating information in the payload reduces the learning and implementation effort.

#interoperability and #modifiability Less protocol-specific functionality means fewer changes are required when one protocol is replaced by another.

#security and #auditability In multiprotocol scenarios, end-to-end security guarantees can only be given and enforced when the information in protocol headers is aggregated and correlated somehow. System and process assurance auditors appreciate if all relevant compliance information can be found in a single place (that is adequately protected) [12].

3.6.3 Initial Position Sketch. Technical metadata such as API Keys, session IDs, or other QoS properties (for example, correlation identifiers, priority levels, time-to-live information, transactional policies, bandwidth requirements, packet-loss tolerance, or latency constraints) travel exclusively in the form of protocol headers. The initial position sketch in Figure 3.6 (a) shows a response message with a payload and several protocol headers.

The refactoring targets request and/or response messages of one or more operations. Operations that form a conversation may or may not appear in the same API endpoint.

3.6.4 Design Smells.

Tight coupling to a communication protocol Most of the network and communication protocols define their own header formats and fields; HTTP is an example. Some of these protocols support custom headers, and others do not. Using protocol-specific headers locks the communication participant in; this can be positive or negative, depending on context and requirements.

Quality-of-Service (QoS) fragmentation and dispersion Several protocols might be used in conversations, such as HTTP, gRPC, and asynchronous messaging (AMQP). API clients and providers must go to multiple places to gather or produce all required context information, which can be error-prone, time-consuming, and cause technical debt.

3.6.5 Instructions.

1. Design a data structure, the Context Representation, to represent the context information.
2. Add this data structure to the request and/or response operations of the targeted operations, add DTOs where necessary.
3. Implement a client-provider message exchange that produces and consumes instances of the new data structure.
4. Update the API Description with information about the syntax and semantics of the new message payload part. Provide data usage examples in the documentation, including valid and invalid values (or value ranges).
5. Inform clients about their new options and/or liabilities to work with the new Context Representation.

Note that it might be required to keep the context information in its current place, e.g., in protocol headers, for backward compatibility reasons. In this case, the refactoring allows clients to choose between the old and new ways of providing context information.

3.6.6 Target Solution Sketch (Evolution Outline). Applying these steps leads to the solution sketched in Figure 3.6 (b).

3.6.7 Example(s). The following MDSL code snippet shows an API endpoint with an operation that expects context information in the headers:

```
data type KeyValuePair {
  "name": ID<string>,
  "property": D<string>
}+

endpoint type SampleService
  exposes
    operation sampleOperationInitial
      expecting
        headers {
          "apiKey":ID<int>,
          "sessionId":D<int>?,
          "otherQosProperties":
            KeyValuePair*
        }
        payload
          "regularRequestPayload":D<string>
      delivering
        payload "someUnspecifiedResponseData"
```

After the context information has been encapsulated, the `apiKey`, `sessionId`, and `otherQosProperties` from the header have moved. They now appear in a Data Transfer Object called `RequestContext` that is part of the request payload:

```
data type RequestContext {
  "apiKey":ID<int>,
  "sessionId":D<int>?,
  "otherQosProperties": KeyValuePair*
}

data type KeyValuePair {
  "name": ID<string>,
  "property": D<string> }+

endpoint type SampleService
  exposes
    operation sampleOperationTarget
```

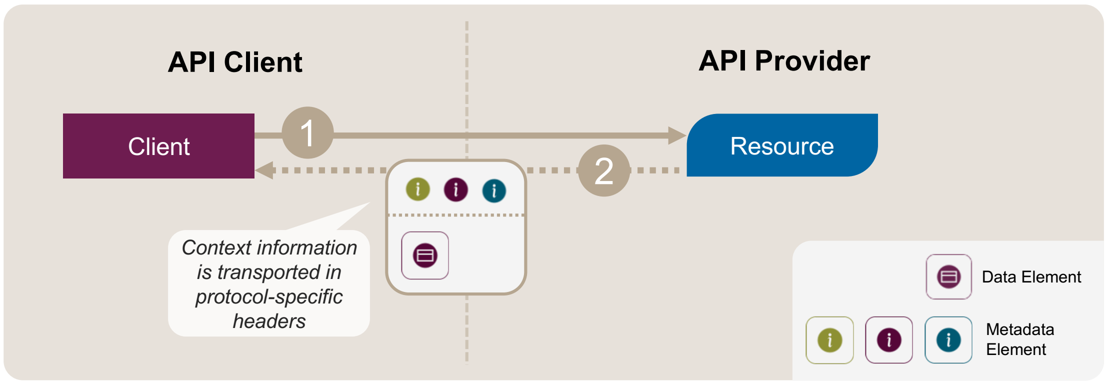

Figure 3.6 (a): Encapsulate Context Representation: Initial Position Sketch. API client and provider exchange a message that contains context information as METADATA ELEMENTS in the protocol header.

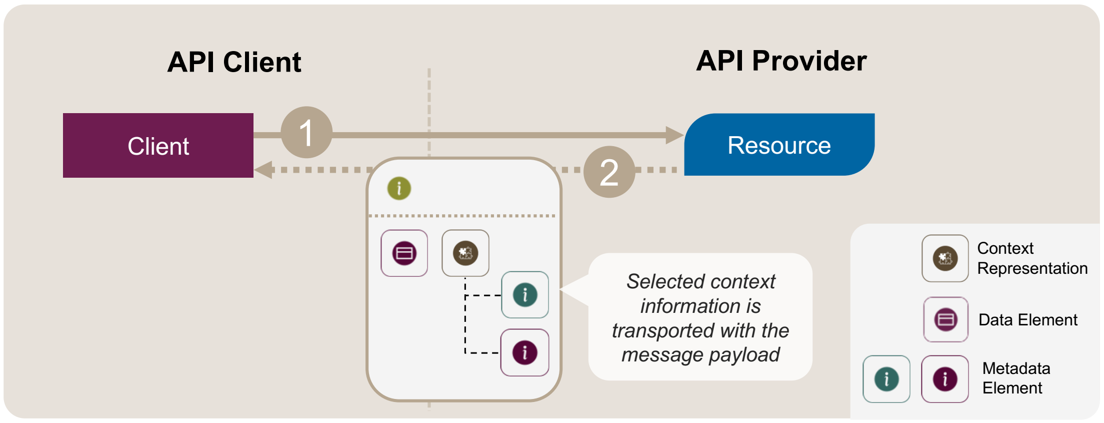

Figure 3.6 (b): Encapsulate Context Representation: Target Solution Sketch. In addition to protocol-specific metadata transported in protocol headers, application-level METADATA ELEMENTS are bundled and included in the payload of the response message.

```
expecting
  payload {
   <<Context_Representation>>
     "requestContext": RequestContext,
   <<Data_Element>>
     "regularRequestPayload":D<string>
  }
delivering
  payload "someUnspecifiedResponseData"
```

3.6.8 Hints and Pitfalls to Avoid. Before and when encapsulating context information, make sure to:

- Decide whether the context has a local or global scope with respect to operation invocations in one or more API endpoints. The pattern variants discussed in Zimmermann et al. [41] provide detailed information about this decision.
- Decide whether request or response messages should contain a message payload-level CONTEXT REPRESENTATION (or both). Contextualizing requests is more common; for instance, think about client location, API user data, WISH LIST items, as well as credentials used to authenticate, authorize, and bill clients. Response contexts can also be observed in practice.
- Strive for a reusable data structure design across operations (and endpoints, if possible). Prefer de-jure or de-facto industry standards over own creations to define the inner structure of the QoS information in the CONTEXT REPRESENTATION if possible. For example, RFC 7807 [24] defines a standard way to carry problem details in HTTP response messages.

It might be required but not possible to encrypt the data in protocol headers; this would be a reason why this refactoring is eligible. Suppose the payload is encrypted but contains context information used for message routing (for instance, in an API GATEWAY [26]). In that case, the refactoring might cause undesired decrypt/encrypt steps in the intermediary.

3.6.9 Related Content. Steps 1 and 2 of this refactoring can be seen as an instance of *Introduce Data Transfer Object* [31, pages 4-7].

Undoing the content encapsulation is possible, but our Interface Refactoring Catalog does not contain an explicit inverse refactoring at present.

3.7 Refactoring: Introduce Version Identifier

3.7.1 Context and Motivation. An API has been deployed to a production environment and is used by clients. The provider is evolving the API with new or improved functionality. Existing clients might have to be adjusted when a new version is released.

As an API provider, I want to indicate versions and their compatibility properties explicitly at design time and runtime so that clients can react accordingly to changes that affect them during API evolution.

3.7.2 Stakeholder Concerns.

#maintainability There are many reasons to change an API (besides quality refactorings [30]). It should be changed as much as required and as little as possible, and ripple effects be avoided. One of the first steps in related maintenance tasks is determining which system parts should be changed. The impact of the major and minor changes on other parts should be kept at a minimum, but it is worth communicating when such changes occur.

#compatibility Explicit versions, possibly introducing breaking changes, might appear costly and anti-agile or not RESTful at first glance. However, fixing bugs caused by not knowing about versioning mismatches is often expensive.

#developer-experience Since Version Identifiers can be placed in protocol headers (in most protocols) or in the message payload, additional learning and decision making is required. Accessing protocol headers differs from accessing payload in terms of code to be written, tool and library support, and portability.

3.7.3 Initial Position Sketch. The entire API or individual parts, such as endpoints, operations, or message parts can be versioned. Here, we primarily target endpoint versioning.

All clients invoke operations exposed by a single, unversioned endpoint. Figure 3.7 (a) shows this rather simple Initial Position Sketch.

3.7.4 Design Smells.

Tacit semantic changes up to incompatibilities creep in While the technical API contract remains unchanged, the meaning of the received or returned data might change over time. Such semantic mismatches between older and newer versions should be documented in the API Description and examples and then caught during testing, ideally in an automated fashion. Implicit versioning an applying the Tolerant Reader pattern [1] might hide such changes and their impact for quite some time.

Resistance to change caused by uncertainty API providers may hesitate to implement necessary changes due to a lack of clarity in their strategy for evolving the API. Clients might be reluctant to upgrade to new versions because they are unable to assess the imposed changes on their side.

See "Interface Evolution Patterns — Balancing Compatibility and Extensibility across Service Life Cycles" [21] for other smells related to API versioning and countermeasures.

3.7.5 Instructions. The introduction and continued use of explicit an Version Identifier has to be planned, executed, and sustained:

1. Decide on scope and naming conventions for the Version Identifier, for instance, following the Semantic Versioning specification when assigning and communicating version numbers (currently standing at Version 2.0.0).[5]
2. Define an evolution roadmap, selecting one or more lifecycle management strategies to define the lifetime of the version; see the related refactorings *Tighten Evolution Strategy* and *Relax Evolution Strategy*.
3. Decide where to place the Version Identifier for each API element to be versioned. For instance, possible locations are endpoint address, message payload, and protocol header.
4. Put the version-enhanced endpoint addresses in the API implementation code or update the message construction code, depending on where the Version Identifier has been added.
5. Update the API documentation with the new Version Identifier(s) and meta-information about the meaning of this version information (for instance, consequences of certain version numbers regarding compatibility).

The pattern description of Version Identifier provides more information about versioning scopes, identifier placement (location), and compatibility considerations [41].

3.7.6 Target Solution Sketch (Evolution Outline). Once a Version Identifier has been introduced, clients can choose which version of an endpoint they want to work with (assuming that multiple versions are supported, as described in the Two in Production pattern). Figure 3.7 (b) illustrates this new, more flexible setup.

3.7.7 Example(s). Depending on the chosen location of the Version Identifier, clients enact their usage decision in the message payload or a header (see section "Instructions" above). In HTTP, it may be part of the endpoint address (relative URI path):

```
GET /customers/1234
Accept: text/json+customer; version=1.0
...
```

or

```
GET /v2/customers/1234
...
```

The API Stylebook compiled by "API Handyman" Arnaud Lauret points at many additional examples in its design topic "Updates and Versioning".

3.7.8 Hints and Pitfalls to Avoid. Before and when applying this refactoring, make sure to:

- Involve API clients in the decisions about and planning of API evolution (assuming that they are known and willing to participate).
- Stay backward-compatible whenever possible, but do not hesitate to upgrade the major version when necessary.
- Provide migration aids such as change logs, code snippets, and mappings of identifiers and parameters from old to new versions.

[5] Note that the Semantic Versioning specification has a Version Identifier and applies Semantic Versioning itself.

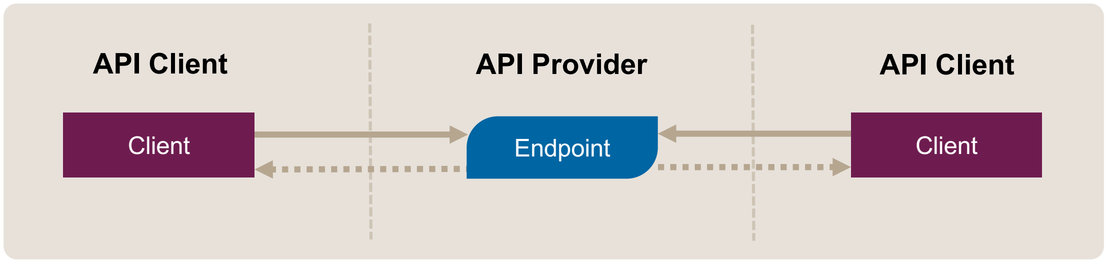

Figure 3.7 (a): Introduce Version Identifier: Initial Position Sketch. An API provider has deployed an API with a single endpoint that clients use.

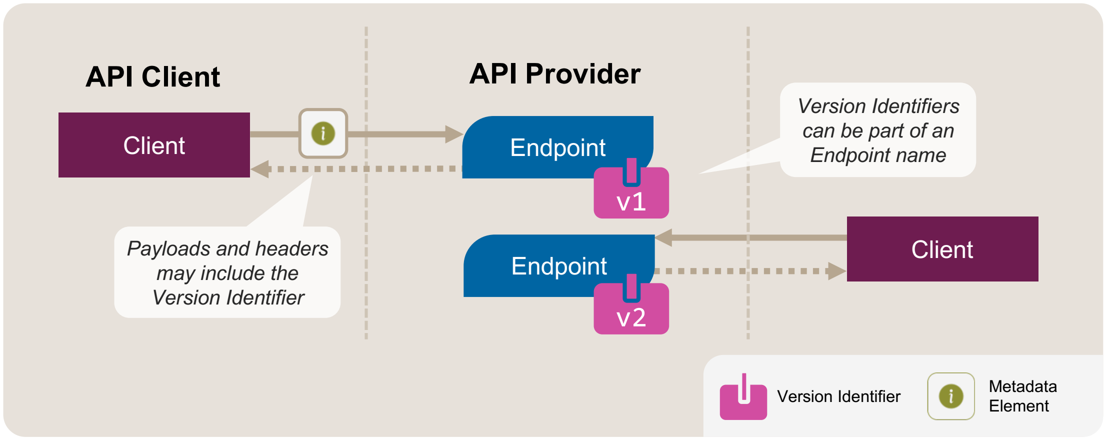
API Client
API Provider
Client
Endpoint
v1
Version Identifiers can be part of an Endpoint name
Endpoint
v2
Client
Payloads and headers may include the Version Identifier
Version Identifier
Metadata Element

Figure 3.7 (b): Introduce Version Identifier: Target Solution Sketch. The API provider has introduced a VERSION IDENTIFIER to the API, allowing clients to choose which version of an endpoint they want to work with.

- Be aware of design challenges caused by automatic routing. For example, VERSION IDENTIFIERS in encrypted message payloads might not be visible to intermediaries and, therefore, can not be used for routing purposes.

"Interface Evolution Patterns — Balancing Compatibility and Extensibility across Service Life Cycles" provides further hints. For example, "stick to a standardized and consistent versioning strategy, e.g., decide which objects to version consistently (operations, data types, etc.) or which versioning schema to use (e.g., SEMANTIC VERSIONING)" [21].

3.7.9 Related Content. Other refactorings dealing with API evolution are *Introduce Version Mediator* [31, pages 22-26], *Tighten Evolution Strategy*, and *Relax Evolution Strategy*.

The evolution patterns in Zimmermann et al. [41] cover versioning. For instance, there is a pattern called SEMANTIC VERSIONING. The concept of SERVICE LEVEL AGREEMENT (SLA) is captured in pattern form as well; an SLA pertains to a single or multiple versions and should specify and reference those versions explicitly.

Arnaud Lauret's book on Web API design covers the topic [19], and the blog post "5 Ways to Version APIs" also discusses options with pros and cons.

James Higginbotham provides advice regarding "When and How Do You Version Your API?.

4 Summary

In this paper, we introduced seven interface refactorings from our Interface Refactoring Catalog: *Extract Information Holder*, *Inline Information Holder*, *Extract Operation*, *Rename Operation*, *Make Request Conditional*, *Encapsulate Context Representation*, and *Introduce Version Identifier*. These seven refactorings comprise the second slice of our Interface Refactoring Catalog (IRC), which we first presented at EuroPLoP 2023 [31].

Future work may concern collecting further API refactorings, connecting our work with other refactoring initiatives (e.g., domain model refactoring), and resuming tool research and development. We are also interested in validating our refactorings with real-world APIs and in exploring the relationship between interface refactorings and interface definition languages, such as Smithy, TypeSpec, and OpenAPI.

API testing and monitoring are potential areas of future research as well. We want to investigate how refactoring can improve the testability, observability, and maintainability of APIs. In addition, we are interested in green software and how refactoring can improve the environmental sustainability of APIs. For example, by transferring less data (*Make Request Conditional*, *Extract Information Holder*), we can reduce the amount of data processed and transferred, which can lead to energy savings and reduce the environmental footprint of software systems [14].

Acknowledgments

We would like to thank the contributors to the interface refactoring catalog. A big thank you goes to our EuroPLoP 2024 shepherd Thomas Mejstrik for his comprehensive and insightful feedback, which will make our work more accessible to a wider audience. We also want to thank the participants of the EuroPLoP 2024 Writers' Workshop B Andreas Fiesser, Diogo Maia, Eduardo Guerra, João Francisco Daniel, Marc Schmidt, Paulo Gabriel Gadelha Queiroz, Pierre Schnizer, and Waheedullah Sulaiman Khail for their valuable feedback. A small project grant from the Hasler Foundation partially supported the research presented in this paper.

References

[1] Robert Daigneau. 2011. *Service Design Patterns: Fundamental Design Solutions for SOAP/WSDL and RESTful Web Services.* Addison-Wesley.
[2] Trevor Foucher Dustin Boswell. 2011. *The Art of Readable Code.* O'Reilly Media, Inc.
[3] Eric Evans. 2003. *Domain-Driven Design: Tacking Complexity In the Heart of Software.* Addison-Wesley.
[4] Christoph Fehling, Frank Leymann, Ralph Retter, Walter Schupeck, and Peter Arbitter. 2014. *Cloud Computing Patterns: Fundamentals to Design, Build, and Manage Cloud Applications.* Springer. https://doi.org/10.1007/978-3-7091-1568-8
[5] Roy T. Fielding, Mark Nottingham, and Julian Reschke. 2022. HTTP Caching. RFC 9111. https://doi.org/10.17487/RFC9111
[6] Roy T. Fielding and Julian Reschke. 2014. Hypertext Transfer Protocol (HTTP/1.1): Conditional Requests. RFC 7232. https://doi.org/10.17487/RFC7232
[7] Roy T. Fielding and Julian Reschke. 2014. Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content. RFC 7231. https://doi.org/10.17487/RFC7231
[8] Martin Fowler. 2002. *Patterns of Enterprise Application Architecture.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.
[9] Martin Fowler. 2018. *Refactoring* (2 ed.). Addison-Wesley, Boston, MA.
[10] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-oriented Software.* Addison-Wesley.
[11] Neil B. Harrison. 2003. Advanced Pattern Writing Patterns for Experienced Pattern Authors. In *Proc. Eighth European Conference on Pattern Languages of Programs (EuroPLoP).* 1–20.
[12] Klaus Julisch, Christophe Suter, Thomas Woitalla, and Olaf Zimmermann. 2011. Compliance by design - Bridging the chasm between auditors and IT architects. *Computers and Security* 30 (09 2011), 410–426. https://doi.org/10.1016/j.cose.2011.03.005
[13] Stefan Kapferer and Samuel Jost. 2017. *Attributbasierte Autorisierung in einer Branchenlösung für das Versicherungswesen - Analyse, Konzept und prototypische Umsetzung.* Bachelor Thesis. University of Applied Sciences of Eastern Switzerland (HSR FHO), https://eprints.ost.ch/id/eprint/602/.
[14] Stefan Kapferer, Mirko Stocker, and Olaf Zimmermann. 2024. Towards responsible software engineering: combining value-based processes, agile practices, and green metering. In *IEEE International Symposium on Technology and Society, ISTAS 2024, Puebla, Mexico, September 18-20, 2024.* IEEE. accepted for publication.
[15] Stefan Kapferer and Olaf Zimmermann. 2020. Domain-Driven Service Design. In *Service-Oriented Computing*, Schahram Dustdar (Ed.). Springer International Publishing, 189–208. https://doi.org/10.1007/978-3-030-64846-6_11
[16] Stefan Kapferer and Olaf Zimmermann. 2021. Domain-Driven Architecture Modeling and Rapid Prototyping with Context Mapper. In *Model-Driven Engineering and Software Development*, Slimane Hammoudi, Luís Ferreira Pires, and Bran Selić (Eds.). Springer International Publishing, Cham, 250–272.
[17] Phil Karlton. 2009. *Two hard things.* https://martinfowler.com/bliki/TwoHardThings.html
[18] Joshua Kerievsky. 2004. *Refactoring to Patterns.* Pearson Higher Education.
[19] Arnaud Lauret. 2019. *The Design of Web APIs.* Manning.
[20] Carola Lilienthal and Henning Schwentner. 2025. *Domain-Driven Transformation: Modularize and Modernize Legacy Software.* Addison-Wesley Professional.
[21] Daniel Lübke, Olaf Zimmermann, Cesare Pautasso, Uwe Zdun, and Mirko Stocker. 2019. Interface Evolution Patterns: Balancing Compatibility and Extensibility across Service Life Cycles. In *Proceedings of the 24th European Conference on Pattern Languages of Programs* (Irsee, Germany) *(EuroPLop '19).* Association for Computing Machinery, New York, NY, USA, Article 15, 24 pages. https://doi.org/10.1145/3361149.3361164
[22] Daniel Lübke, Olaf Zimmermann, Cesare Pautasso, Uwe Zdun, and Mirko Stocker. 2019. Interface Evolution Patterns: Balancing Compatibility and Extensibility across Service Life Cycles. In *Proceedings of the 24th European Conference on Pattern Languages of Programs* (Irsee, Germany) *(EuroPLop '19).* Association for Computing Machinery, New York, NY, USA, Article 15, 24 pages. https://doi.org/10.1145/3361149.3361164
[23] Gerard Meszaros and Jim Doble. 1997. A Pattern Language for Pattern Writing. *Pattern Languages of Program Design* 3 (1997), 529–574.
[24] Mark Nottingham and Erik Wilde. 2016. Problem Details for HTTP APIs. RFC 7807. https://doi.org/10.17487/RFC7807
[25] Michael T. Nygard. 2018. *Release It!* (2 ed.). Pragmatic Bookshelf, Raleigh, NC. https://learning.oreilly.com/library/view/release-it-2nd/9781680504552/
[26] Chris Richardson. 2018. *Microservices Patterns.* Manning.
[27] Arthur J. Riel. 1996. *Object-Oriented Design Heuristics.* Addison-Wesley, Reading, MA.
[28] Souhaila Serbout, Cesare Pautasso, Uwe Zdun, and Olaf Zimmermann. 2022. From OpenAPI Fragments to API Pattern Primitives and Design Smells. In *26th European Conference on Pattern Languages of Programs* (Graz, Austria) *(EuroPLoP'21).* Association for Computing Machinery, New York, NY, USA, Article 21, 35 pages. https://doi.org/10.1145/3489449.3489998
[29] Apitchaka Singjai, Uwe Zdun, and Olaf Zimmermann. 2021. Practitioner Views on the Interrelation of Microservice APIs and Domain-Driven Design: A Grey Literature Study Based on Grounded Theory. In *18th IEEE International Conference On Software Architecture (ICSA 2021).* https://doi.org/10.5281/zenodo.4493865
[30] Mirko Stocker and Olaf Zimmermann. 2021. From Code Refactoring to API Refactoring: Agile Service Design and Evolution. In *Service-Oriented Computing*, Johanna Barzen (Ed.). Springer International Publishing, Cham, 174–193. https://doi.org/10.1007/978-3-030-87568-8_11
[31] Mirko Stocker and Olaf Zimmermann. 2023. API Refactorings to Patterns: Catalog, Template and Tools for Remote Interface Evolution. In *Proceedings of the 28th European Conference on Pattern Languages of Programs* (Irsee, Germany) *(EuroPLop '23).* Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3628034.3628073
[32] Mirko Stocker, Olaf Zimmermann, Daniel Lübke, Uwe Zdun, and Cesare Pautasso. 2018. Interface Quality Patterns – Communicating and Improving the Quality of Microservices APIs. In *23rd European Conference on Pattern Languages of Programs 2018.* https://doi.org/10.1145/3282308.3282319
[33] Girish Suryanarayana, Ganesh Samarthyam, and Tushar Sharma. 2014. *Refactoring for Software Design Smells: Managing Technical Debt* (1st ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.
[34] Olaf Zimmermann. 2017. Microservices tenets. *Comput. Sci. Res. Dev.* 32, 3-4 (2017), 301–310. https://doi.org/10.1007/S00450-016-0337-0
[35] Olaf Zimmermann, Daniel Lübke, Uwe Zdun, Cesare Pautasso, and Mirko Stocker. 2020. Interface Responsibility Patterns: Processing Resources and Operation Responsibilities. In *Proc. of the European Conference on Pattern Languages of Programs* (Online) *(EuroPLoP '20).*
[36] Olaf Zimmermann, Cesare Pautasso, Daniel Lübke, Uwe Zdun, and Mirko Stocker. 2020. Data-Oriented Interface Responsibility Patterns: Types of Information Holder Resources. In *Proc. of the European Conference on Pattern Languages of Programs* (Online) *(EuroPLoP '20).*
[37] Olaf Zimmermann and Mirko Stocker. 2021. *Design Practice Reference - Guides and Templates to Craft Quality Software in Style.* LeanPub. https://leanpub.com/dpr
[38] Olaf Zimmermann, Mirko Stocker, and Stefan Kapferer. 2024. Bringing ethical values into agile software engineering. In *Smart Ethics in the Digital World: Proceedings of the ETHICOMP 2024. 21th International Conference on the Ethical and Social Impacts of ICT.* Universidad de La Rioja, 90–93.
[39] Olaf Zimmermann, Mirko Stocker, Daniel Lübke, Cesare Pautasso, and Uwe Zdun. 2020. Introduction to Microservice API Patterns (MAP). In *Joint Post-proceedings of the First and Second International Conference on Microservices (Microservices 2017/2019) (OpenAccess Series in Informatics (OASIcs), Vol. 78)*, Luís Cruz-Filipe, Saverio Giallorenzo, Fabrizio Montesi, Marco Peressotti, Florian Rademacher, and Sabine Sachweh (Eds.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 4:1–4:17. https://doi.org/10.4230/OASIcs.Microservices.2017-2019.4
[40] Olaf Zimmermann, Mirko Stocker, Daniel Lübke, and Uwe Zdun. 2017. Interface Representation Patterns - Crafting and Consuming Message-Based Remote APIs. In *22nd European Conference on Pattern Languages of Programs (EuroPLoP 2017).* 1–36. https://doi.org/10.1145/3147704.3147734
[41] Olaf Zimmermann, Mirko Stocker, Daniel Lübke, Uwe Zdun, and Cesare Pautasso. 2022. *Patterns for API Design: Simplifying Integration with Loosely Coupled Message Exchanges.* Addison-Wesley Professional.

A Summary of Patterns for API Design

In the refactorings presented in this paper, we refer to 24 patterns from Zimmermann et al. [41]. The pattern name, icon, problem and solution summary statements are listed in Table 2 to provide a quick reference. For more details, we refer the reader to the API Patterns website and book.

Pattern Name	**Pattern Summary (Problem and Solution)**
API Description	*Problem:* Which knowledge should be shared between an API provider and its clients? How should this knowledge be documented? *Solution:* Create an API Description that defines request and response message structures, error reporting, and other relevant parts of the technical knowledge to be shared between provider and client. In addition to static and structural information, also cover dynamic or behavioral aspects, including invocation sequences, pre- and postconditions, and invariants. Complement the syntactical interface description with quality management policies as well as semantic specifications and organizational information.
API Key	*Problem:* How can an API provider identify and authenticate clients and their requests? *Solution:* As an API provider, assign each client a unique token — the API Key — that the client can present to the API endpoint for identification purposes.
Backend Integration	*Problem:* How can distributed applications and their parts, which have been built independently and are deployed separately, exchange data and trigger mutual activity while preserving system-internal conceptual integrity without introducing undesired coupling? *Solution:* Integrate the backend of a distributed application with one or more other backends (of the same or other distributed applications) by exposing its services via a message-based remote Backend Integration API.
Community API	*Problem:* How can the visibility of and the access to an API be restricted to a closed user group that does not work for a single organizational unit but for multiple legal entities (such as companies, nonprofit/nongovernment organizations, and governments)? *Solution:* Deploy the API and its implementation resources securely in an access-restricted location so that only the desired user group has access to it — for instance, in an extranet. Share the API Description only with the restricted target audience.
Conditional Request	*Problem:* How can unnecessary server-side processing and bandwidth usage be avoided when frequently invoking API operations that return rarely changing data? *Solution:* Make requests conditional by adding Metadata Elements to their message representations (or protocol headers) and processing these requests only if the condition specified by the metadata is met.
Context Representation	*Problem:* How can API consumers and providers exchange context information without relying on any particular remoting protocols? How can identity information and quality properties in a request be made visible to related subsequent ones in conversations? *Solution:* Combine and group all Metadata Elements that carry the desired information into a custom representation element in request and/or response messages. Do not transport this single Context Representation in protocol headers, but place it in the message payload. Separate global from local context in a conversation by structuring the Context Representation accordingly. Position and mark the consolidated Context Representation element so that it is easy to find and distinguish from other Data Elements.
Data Element	*Problem:* How can domain/application-level information be exchanged between API clients and API providers without exposing provider-internal data definitions in the API? How can API client and API provider be decoupled from a data management point of view? *Solution:* Define a dedicated vocabulary of Data Elements for request and response messages that wraps and/or maps the relevant parts of the data in the business logic of an API implementation.
Embedded Entity	*Problem:* How can one avoid sending multiple messages when their receivers require insights about multiple related information elements? *Solution:* For any data relationship that the client wants to follow, embed a Data Element in the request or response message that contains the data of the target end of the relationship. Place this Embedded Entity inside the representation of the source of the relationship.

Table 2: Patterns from Zimmermann et al. [41] mentioned in this paper, with their problem and solution summaries.

Pattern Name	**Pattern Summary (Problem and Solution)**
Information Holder Resource	*Problem:* How can domain data be exposed in an API, but its implementation still be hidden? How can an API expose data entities so that API clients can access and/or modify these entities concurrently without compromising data integrity and quality? *Solution:* Add an Information Holder Resource endpoint to the API, representing a data-oriented entity. Expose create, read, update, delete, and search operations in this endpoint to access and manipulate this entity. In the API implementation, coordinate calls to these operations to protect the data entity.
Limited Lifetime Guarantee	*Problem:* How can a provider let clients know for how long they can rely on the published version of an API? *Solution:* As an API provider, guarantee to not break the published API for a fixed timeframe. Label each API version with an expiration date.
Link Element	*Problem:* How can API endpoints and operations be referenced in request and response message payloads so that they can be called remotely? *Solution:* Include a special type of Id Element, a Link Element, to request or response messages. Let these Link Elements act as human- and machine-readable, network-accessible pointers to other endpoints and operations. Optionally, let additional Metadata Elements annotate and explain the nature of the relationship.
Linked Information Holder	*Problem:* How can messages be kept small even when an API deals with multiple information elements that reference each other? *Solution:* Add a Link Element to messages that pertain to multiple related information elements. Let this Link Element reference another API endpoint that represents the linked element.
Master Data Holder	*Problem:* How can I design an API that provides access to master data that lives for a long time, does not change frequently, and will be referenced from many clients? *Solution:* Mark an Information Holder Resource to be a dedicated Master Data Holder endpoint that bundles master data access and manipulation operations in such a way that the data consistency is preserved and references are managed adequately. Treat delete operations as special forms of updates.
Metadata Element	*Problem:* How can messages be enriched with additional information so that receivers can interpret the message content correctly, without having to hardcode assumptions about the data semantics? *Solution:* Introduce one or more Metadata Elements to explain and enhance the other representation elements that appear in request and response messages. Populate the values of the Metadata Elements thoroughly and consistently; process them as to steer interoperable, efficient message consumption and processing.
Operational Data Holder	*Problem:* How can an API support clients that want to create, read, update, and/or delete instances of domain entities that represent operational data: data that is rather short-lived, changes often during daily business operations, and has many outgoing relations? *Solution:* Tag an Information Holder Resource as Operational Data Holder and add API operations to it that allow API clients to create, read, update, and delete its data often and fast.
Processing Resource	*Problem:* How can an API provider allow its clients to trigger an action in it? *Solution:* Add a Processing Resource endpoint to the API exposing operations that bundle and wrap application-level activities or commands.
Rate Limit	*Problem:* How can the API provider prevent API clients from excessive API usage? *Solution:* Introduce and enforce a Rate Limit to safeguard against API clients that overuse the API.

Pattern Name	**Pattern Summary (Problem and Solution)**
Retrieval Operation	*Problem:* How can information available from a remote party (the API provider, that is) be retrieved to satisfy an information need of an end user or to allow further client-side processing? *Solution:* Add a read-only operation `ro: (in,S) -> out` to an API endpoint, which often is an Information Holder Resource, to request a result report that contains a machine-readable representation of the requested information. Add search, filter, and formatting capabilities to the operation signature.
Semantic Versioning	*Problem:* How can stakeholders compare API versions to detect immediately whether they are compatible? *Solution:* Introduce a hierarchical three-number versioning scheme `x.y.z`, which allows API providers to denote different levels of changes in a compound identifier. The three numbers are usually called major, minor, and patch versions.
Service Level Agreement	*Problem:* How can an API client learn about the specific quality-of-service characteristics of an API and its endpoint operations? How can these characteristics, and the consequences of not meeting them, be defined and communicated in a measurable way? *Solution:* As an API product owner, establish a structured, quality-oriented Service Level Agreement that defines testable service-level objectives.
Two in Production	*Problem:* How can a provider gradually update an API without breaking existing clients but also without having to maintain a large number of API versions in production? *Solution:* Deploy and support two versions of an API endpoint and its operations that provide variations of the same functionality but do not have to be compatible with each other. Update and decommission the versions in a rolling, overlapping fashion.
Version Identifier	*Problem:* How can an API provider indicate its current capabilities as well as the existence of possibly incompatible changes in order to prevent malfunctioning of API clients due to undiscovered interpretation errors? *Solution:* Introduce an explicit version indicator. Include this Version Identifier in the API Description and in the exchanged messages. To do the latter, add a Metadata Element to the endpoint address, the protocol header, or the message payload.
Wish List	*Problem:* How can an API client inform the API provider at runtime about the data it is interested in? *Solution:* As an API client, provide a Wish List in the request that enumerates all desired data elements of the requested resource. As an API provider, deliver only those data elements in the response message that are enumerated in the Wish List ("response shaping").
Wish Template	*Problem:* How can an API client inform the API provider about nested data that it is interested in? How can such preferences be expressed flexibly and dynamically? *Solution:* Add one or more additional parameters to the request message that mirror the hierarchical structure of the parameters in the corresponding response message. Make these parameters optional or use Boolean as their types so that their values indicate whether or not a parameter should be included.