Data-Oriented Interface Responsibility Patterns: Types of Information Holder Resources

Olaf Zimmermann
University of Applied Sciences of Eastern Switzerland, Rapperswil, Switzerland

Cesare Pautasso
Software Institute, Faculty of Informatics, USI Lugano, Switzerland

Daniel Lübke
iQuest GmbH, Hanover, Germany

Uwe Zdun
University of Vienna, Faculty of Computer Science, Software Architecture Research Group, Vienna, Austria

Mirko Stocker
University of Applied Sciences of Eastern Switzerland, Rapperswil, Switzerland

ABSTRACT

Remote Application Programming Interfaces (APIs) are used in almost any distributed system today, for instance in microservices-based systems, and are thus enablers for many digitalization efforts. API design not only impacts whether software provided as a service is easy and efficient to develop applications with, but also affects the long term evolution of the software system. In general, APIs are responsible for providing remote and controlled access to the functionality provided as services; however, APIs often are also used to expose and share information. We focus on such data-related aspects of microservice APIs in this paper. Depending on the life cycle of the information published through the API, its mutability and the endpoint role, data-oriented APIs can be designed following patterns such as *Operational Data Holder*, *Master Data Holder*, *Reference Data Holder*, *Data Transfer Holder*, and *Link Lookup Resource*. Known uses and examples of the patterns are drawn from public Web APIs as well as application development and integration projects we have been involved in.

CCS CONCEPTS

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

ACM Reference Format:
Olaf Zimmermann, Cesare Pautasso, Daniel Lübke, Uwe Zdun, and Mirko Stocker. 2020. Data-Oriented Interface Responsibility Patterns: Types of Information Holder Resources. In *European Conference on Pattern Languages of Programs 2020 (EuroPLoP '20), July 1–4, 2020, Virtual Event, Germany.* ACM, New York, NY, USA, 25 pages. https://doi.org/10.1145/3424771.3424821

1 INTRODUCTION

Microservices architectures have evolved from previous incarnations of Service-Oriented Architectures (SOAs) [17]. They consist of independently deployable, scalable and changeable services, each having a single responsibility. These responsibilities model business capabilities. Microservices often are deployed in lightweight virtualization containers, encapsulate their own state, and communicate via message-based remote APIs in a loosely coupled fashion. Microservices solutions leverage polyglot programming, polyglot persistence, as well as DevOps practices including decentralized continuous delivery and end-to-end monitoring [24], [27], [45].

Microservice APIs designers must address concerns such as [50]:

- How many services should be exposed?
- Which service cuts let services and their clients deliver user value jointly, but couple them loosely?
- How often do services and their clients interact to exchange data? How much and which data should be exchanged?

The Microservice API Patterns (MAP) website[1] covers and organizes this design space providing guidance distilled from the experience of API design experts. This paper deals with a specific issue that is always encountered when designing API endpoints:

Which architectural roles do API endpoints play?

Service identification activities might lead to a list of *candidate API endpoints* to satisfy such diverse goals (for instance, resources in RESTful HTTP APIs). At the beginning of a project or product development, these interfaces are yet unspecified (or only partially specified). Service designers have to address semantic concerns and find an appropriate business granularity for services. Simplistic statements such as "Service-Oriented Architecture (SOA) services are coarse-grained by definition, while microservices are fine-grained; you cannot have both in one system" or "always prefer fine-grained over coarse-grained services" are insufficient as project requirements and stakeholder concerns differ [31]. Context always matters [35]; cohesion and coupling criteria come in many forms. As a result, the non-functional requirements for service design often are conflicting [48].

In response to such challenges, our responsibility patterns cover two distinct main architectural roles for API endpoints: *Processing Resources* are resources whose primary function is to handle incoming action requests or commands, whereas *Information Holder Resources* are resources whose primary function is to expose storage and management of data or meta-data (including its creation and

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
EuroPLoP '20, July 1–4, 2020, Virtual Event, Germany
© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-7769-0/20/07…$15.00
https://doi.org/10.1145/3424771.3424821

[1] https://microservice-api-patterns.org/

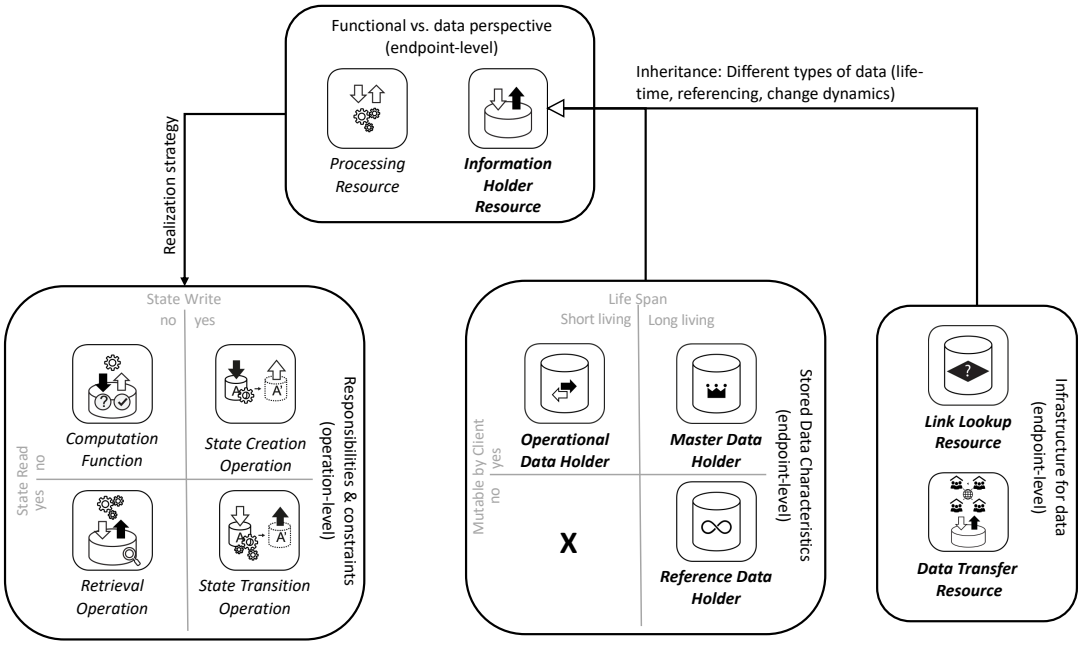


Figure 1: Type of Information Holders and their relations to other role and responsibility patterns (bold pattern names: scope of this paper).

retrieval). The following patterns represent different types of data holders, refining the general *Information Holder Resource* pattern:

- *Operational Data Holder*: A resource that stores short-living, operational (a.k.a. transactional) data.
- *Master Data Holder*: A resource that stores long-living and frequently referenced, but still mutable data.
- *Reference Data Holder*: A resource that stores long-living (often simple) data that cannot be altered by clients.
- *Data Transfer Resource*: A resource whose primary function is to offer a shared data exchange between other resources.
- *Link Lookup Resource*: A resource whose primary function is supporting clients that follow or dereference links to other resources.

Each resource (be it a *Processing Resource* or an *Information Holder Resource*) is offered via an *endpoint* which in turn offers different *operations*. These operations may have different *responsibilities*:

- *State Creation Operation*: A write-only operation that creates state in the endpoint.
- *Retrieval Operation*: An read-only operation that finds and delivers data, but does not change server-side data.
- *State Transition Operation*: An operation that performs one or more activities causing a server-side state change.
- *Computation Function*: An operation that computes a result solely from its input and does not read or write server-side state.

To design these responsibilities properly, understanding the architectural role of the endpoint is essential. In this paper, we will not cover *Processing Resource* and the four operation responsibilities any further[2], but rather provide patterns that help to better understand and design the architectural endpoint role from a data-centric point of view. In particular, we will first introduce a generic *Information Holder Resource* pattern and then cover five specializations of it.

Figure 1 shows these *Information Holder Resource* patterns and their relations among each other, as well as to the other endpoint role and operation responsibility patterns. The patterns covered in this paper are highlighted with a bold pattern name. Other patterns are described in a companion paper ZLZPS:2020:MAP-A.

The remainder of this paper is structured as this. Section 2 presents related work; Section 3 provides an overview of our pattern language, its categories and patterns published so far. It also introduces the API design vocabulary used in the pattern texts as well as our pattern template. Section 4 presents the six patterns. Section 5 summarizes the paper and provides an outlook.

2 RELATED WORK

2.1 Data on the outside vs. data on the inside

Structuring data exchanges without breaking information hiding is a hard problem for which no single solution exists. According to Helland, "data on the outside" differs from "data on the inside" significantly [15]. Data access and usage profiles drive many data modeling decisions, both for data on the inside and for data on the outside. However, inside and outside data have diverging mutability, lifetime, accuracy, consistency and protection needs.

[2]These patterns are covered in detail in our paper "Interface Responsibility Patterns: Processing Resources and Operation Responsibilities" [49].

2.2 Responsibility-Driven Design (RDD)

The patterns in this paper have vastly different invocation, processing, and state management characteristics. To order and structure the design space, we adopt terminology and *role stereotypes* from *Responsibility-Driven Design (RDD)*[3]. In RDD, a stereotype is "a conventional, formulaic, and oversimplified conception, opinion, or image". An application is "a set of interacting objects", an object ia "an implementation of one or more roles" (here: microservice). A role is "a set of related responsibilities", a responsibility is "an obligation to perform a task or know information". A collaboration is "an interaction of objects or roles (or both)", a contract is "an agreement outlining the terms of a collaboration" [41].

Our microservice API design terms relate to the more general RDD concepts in the following way: API operations take over a responsibility, and API and their endpoints assemble these responsibilities into roles. The collaborations then arise from calls to API operations (a.k.a. service invocations). The *API Description*, presented in a previous paper [25], specifies the contract.

RDD defines these role stereotypes:

- An interfacer "transforms information and requests between distinct parts of a system".
- A service provider "performs work on demand".
- A controller "makes decisions and closely directs others' actions" and a coordinator "mechanically reacts to events".
- An information holder "knows and provides information"
- A structurer "maintains relationships between objects and information about those relationships" [41].

All API endpoints can be seen as (remote) *interfacers* that provide and protect access to *service providers*, *controllers/coordinators*, and *information holders/structurers*. Specifically, we (re-)use the following role stereotype in this paper: *information holder* (under this very name).

3 RECAP: THE MAP LANGUAGE 2016-2019

The patterns introduced in this paper are an extension of our prior works: In particular, we first introduced "Interface Representation Patterns" to structure messages in remote APIs, including *Atomic Parameter*, *Parameter Tree*, and *Pagination* [51]. Next, we presented "Interface Quality Patterns" that deal with runtime qualities and the communication of API qualities (between provider and client), including *Service Level Agreement* and *Wish List* [34]. The third slice, presented last year, focused on the versioning and evolution of *API Descriptions* and their implementations ("Interface Evolution Patterns"); patterns included *Version Identifier* and *Two in Production* [25]. The companion paper "Interface Responsibility Patterns: Processing Resources and Operation Responsibilities" [49] gives a more elaborate introduction to and overview of the pattern categories our language is organized into.

3.1 Domain model

We have generalized the concepts and terminology that we found in remote API platforms and integration technologies such as HTTP, gRPC, WSDL/SOAP (to name just a few) into a platform-independent domain model. We described this domain model in a previous EuroPLoP paper [25]; its vocabulary is used throughout our pattern language and also in the following pattern texts.

An *API endpoint* is a provider-side end of a communication channel and a specification of where the *API* resources are located so that *APIs* can be accessed by *API clients*. Each *API endpoint* belongs to an *API*; one *API* can have different endpoints. The *API* exposes *operations*.

3.2 Pattern template

We use the following template for our patterns: The *context* establishes preconditions for pattern applicability. The *problem* specifies a design issue to be resolved. The *forces* explain why the problem is hard to solve – architectural design issues and conflicting quality attributes are often referenced here; a non-solution may be pointed out as well. The *solution* answers the design question from the problem statement, describes how the solution works and which variants (if any) exist. It also gives an example and shares implementation hints. The *consequences* section discusses to which extent the solution resolves the pattern forces; it may also include additional pros and cons and identify alternative solutions. *Known uses* report real-world pattern applications. Finally, relations to other patterns are explained and additional pointers and references given under *more information*.

4 TYPES OF INFORMATION HOLDERS

We introduce five specializations of the general *Information Holder Resource* in this section. Three of these differ by life span and mutability of the data (i.e., operational data, master data, reference data); the other two have a special purpose (loosely coupled information exchange, address management). Table 1 gives an overview.

4.1 Pattern: *Information Holder Resource*

a.k.a. Generic Information Service, Data Entity Resource, Siloed/Isolated Data Holder

Context. A domain model[4], a conceptual entity-relationship diagram[5] or another form of glossary of key application concepts and their interconnections have been specified. The model contains entities that have an identity and a life cycle as well as attributes; entities cross-reference each other.

From this analysis and design work, it has become apparent that structured data will have to be used in multiple places in the distributed system being designed; hence, these shared data structures have to be made accessible from multiple remote clients.

It is not possible or not easy to hide the shared data structured behind domain logic (i.e., processing-oriented actions such as business activities and commands); the application under construction does not have a workflow or other processing nature.

[3] http://www.wirfs-brock.com/PDFs/A_Brief-Tour-of-RDD.pdf

[4] http://www.scaledagileframework.com/domain-modeling/

[5] https://www.visual-paradigm.com/guide/data-modeling/what-is-entity-relationship-diagram/

Table 1: Problem-solution pairs of the six patterns presented in this paper.

Pattern Name	Problem	Solution
Information Holder Resource	How can domain data be exposed in an API, but its implementation still be hidden? More specifically, 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? How to segregate operations into separate CRUD-like APIs by data lifetime, link structure, and mutability?	Add an *Information Holder Resource* endpoint to the API, representing a data-oriented entity. Expose Create, Read, Update, Delete (CRUD) as well as search operations in this endpoint to access and manipulate this entity.
Operational Data Holder	How can an API support clients that want to create, read, update, and/or delete instances of domain entities that are rather short-lived, change often during daily business operations and have many outgoing relations?	Tag an *Information Holder Resource* as *Operational Data Holder* endpoint and add API operations to it that allow API client to Create, read, update, and delete its data often and fast.
Master Data Holder	How can I create, read, update, and (possibly) delete data that lives long, does not change frequently, and is referenced often by other data directly or indirectly?	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.
Reference Data Holder	How should data that is referenced in many places, lives long, and is immutable for clients be treated in API contracts? How can such reference data be used in requests to and responses from arbitrary endpoints (*Processing Resources* [49] or *Information Holder Resources*)?	Provide a special type of *Information Holder Resource* endpoint, a *Reference Data Holder* as a single point of reference for the static, immutable data. Provide read operations, but no create, update, or delete operations. Update the reference data elsewhere (backend, separate management API).
Data Transfer Resource	How can two or more communication participants exchange data without knowing each other, without being available at the same time, and without having to wait until the data has been completely transferred?	Introduce a special type of *Information Holder Resource*, a *Data Transfer Resource* endpoint with a globally unique, network-accessible address that two or more clients can use as a shared data exchange blackboard. Add at least one *State Creation Operation* and one *Retrieval Operation* to it.
Link Lookup Resource	How can message representations refer to other, possibly many and frequently changing, API endpoints and operations without binding the message recipient to the actual addresses of these endpoints? How can clients deal with broken links?	Introduce a special type of *Information Holder Resource*, a dedicated *Link Lookup Resource* endpoint that exposes special *Retrieval Operation* operations that return single instances or collections of *Link Elements* that represent the current addresses of the referenced API endpoints.

Problem. How can domain data be exposed in an API, but its implementation still be hidden?

More specifically, 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? For instance, how to deal with race conditions?

How to segregate operations into separate CRUD-like APIs by data lifetime, link structure, and mutability?

Forces. Dealing with structured, possibly replicated data is one of the most challenging design issues in distributed systems; microservice APIs are no exception. Generally speaking, key factors that influence this general design issue are:

- *Modeling approach*, for instance endpoint identification method, and its impact on coupling
- *Quality attribute conflicts and tradeoffs* such as concurrency, consistency; data quality and integrity; recoverability and availability; mutability and immutability

- *Compliance with architectural design principles*, e.g., when making architectural decisions[6] about logical layers and physical tiers

The detailed forces that arise from these general concerns, as well as their relations, are discussed in the five concrete, specific types of information holders featured as separate patterns (Sections 4.2 to 4.6 of this paper): *Operational Data Holder*, *Master Data Holder*, *Reference Data Holder*, *Data Transfer Resource*, and *Link Lookup Resource*.

A key decision is whether the endpoint should have activity (processing) semantics or data-oriented (entity state) semantics. This pattern explains how to emphasize data; its *Processing Resource* [49] sibling focusses on action/activity orientation.

Details. *Modeling approach and and its impact on coupling.* Some software engineering and Object-Oriented Analysis and Design (OOAD)[7] methods balance processing and structural aspects in their steps, artifacts, and techniques; some put a strong emphasis on either computing or data. Domain-Driven Design (DDD)[8], for instance, is an example of a balanced approach. Entity-Relationship Diagrams focus on data structure and relationships rather than behavior. If a data-centric modeling and API endpoint identification approach is chosen, there is a risk that many CRUD (Create, Read, Update, Delete) APIs operating on data are exposed, which can have a negative impact on dependency management (information hiding principle violated) and data quality (because arbitrary data manipulations can be performed from any authorized client). CRUD-oriented data abstractions in interfaces introduce operational and semantic coupling.

Quality attribute conflicts and tradeoffs. One example of such tradeoff is the desire for data currentness/freshness vs. the effort required to keep it consistent and accurate (P. Helland: "Data on the Outside vs. Data on the Inside" [15]). Design time qualities such as simplicity and clarity, runtime qualities such as performance, availability and scalability and evolution time qualities such as maintainability and flexibility often conflict with each other. Cross-cutting concerns such as application security also make it difficult to deal with data in APIs. A decision to exposing or publish internal data through an API cannot be made without considering which consumer should have the correct read/write access rights and what are the consequences for consumers if such data may become temporarily or permanently unavailable in the future.

Compliance with architectural design principles. The API under construction might be part of a project that has already established a logical and a physical software architecture; it should also play nice w.r.t. organization-wide architectural decisions [42], for instance those establishing architectural principles such as loose coupling, logical and physical data independence or microservices tenets such as independent deployability[9]. Such principles might include suggestive or normative guidance if and how data can be exposed in APIs; a number of pattern selection decisions are required, with those principles serving as decision drivers [43], [18]. The information holder patterns in this paper provide the concrete alternatives and criteria for making such architectural decisions[10].

Non-solution. One could think of hiding all data structures behind processing-oriented API operations and Data Transfer Objects (DTOs) analogous to object-oriented programming (i.e., local object-oriented APIs expose access methods and facades while keeping all individual data members private). Such approach is feasible and promotes information hiding; however, it may limit the opportunities to deploy, scale, and replace remote components (services) independently of each other (because either many fine-grained, chatty API calls are required or data has to be stored redundantly). It also introduces an undesired extra level of indirection sometimes (for instance, when building data-intensive applications and integration solutions).

Another possibility would be to give direct access to the database so that consumers can see for themselves what data is available and directly read and even write it if allowed. The API in this case becomes a tunnel to the database, where consumers can send arbitrary queries and transactions through it; databases such as CouchDB provide such API out-of-the-box. This solution completely removes the need to design an API, since the internal representation of the data is directly exposed to consumers. However, by breaking basic information hiding principles, it also results in a tightly coupled architecture where it will be impossible to ever touch the database schema without affecting every API consumer. This solution also introduces security threats.

Solution. Add an *Information Holder Resource* endpoint to the API, representing a data-oriented entity. Expose Create, Read, Update, Delete (CRUD), and search operations in this endpoint to access and manipulate this entity. Define and manage reference links to other endpoints.

Make the endpoint remotely accessible for one or more API clients by providing a unique logical address. In the API implementation, coordinate calls to these operations to protect the data entity.

How it works. Let each operation of the *Information Holder Resource* have one and only one of the following *operation responsibilities*:

- *State Creation Operations* [49] create the entity that is represented by the *Information Holder Resource*.
- *Retrieval Operations* [49] access and read an entity, but do not update it. They may search for and return collections of such entities.
- *State Transition Operations* [49] access existing entities and modify/update them fully or partially.

For each operation, design the request and, if needed, response message structures. For instance, represent entity relationships as *Link Elements* (another pattern in our language). If basic reference data such as country codes or currency codes are looked up, the response message typically is an *Atomic Parameter* [51]; if a rich, structured domain model entity is looked up, the response is more likely to contain a *Parameter Tree* [51] that represents the

[6] https://ozimmer.ch/practices/2020/04/27/ArchitectureDecisionMaking.html
[7] https://en.wikipedia.org/wiki/Object-oriented_analysis_and_design
[8] https://en.wikipedia.org/wiki/Domain-driven_design
[9] https://www.ifs.hsr.ch/uploads/tx_icscrm/1_msa-pospaperzio4summersoc2016v15nc.pdf
[10] https://ozimmer.ch/practices/2020/04/27/ArchitectureDecisionMaking.html

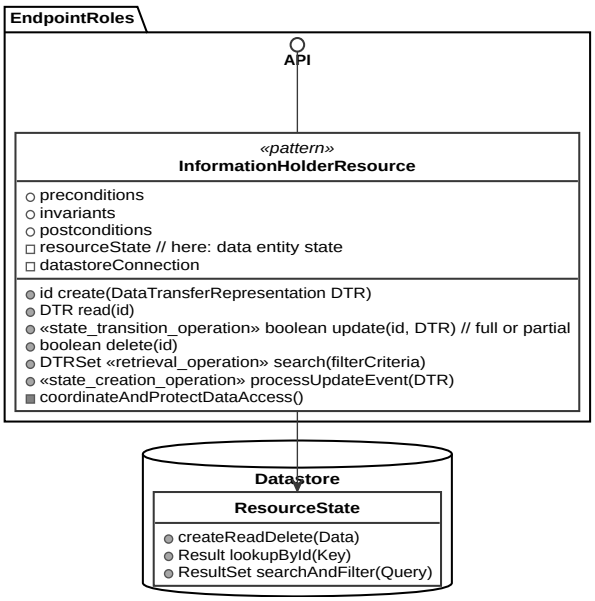


Figure 2: Information Holder Resources model and expose general data-oriented API designs. This endpoint role groups information-oriented responsibilities. Its operations create, read, update, or delete the data held. Searching for data sets is also supported.

data transfer representation of the looked up information. Define operation-level preconditions and postconditions as well as invariants to protect the resource state. Figure 2 sketches this solution.

Decide whether the *Information Holder Resource* should be a Stateful Component[11] or a Stateless Component[12]. In the latter case, there still is state but the entire resource state management is outsourced to a backend system.

Define the quality characteristics of the new endpoint and its operation as well (e.g., transactionality, idempotence, access control, accountability, and consistency):

- Introduce access/modification control and coordination policies (e.g., *API Keys* [34], conversation patterns[13]).
- Protect the concurrent data access by applying an optimistic or a pessimistic locking strategy from the database and concurrent programming literature. For instance, consider the patterns in [29] and [47].
- Implement consistency preserving checks (which may support strict[14] or eventual[15] consistency).

Five patterns in our language refine this general solution to data-oriented API endpoint modeling: *Operational Data Holder*, *Master Data Holder*, *Reference Data Holder*, *Data Transfer Resource*, and *Link Lookup Resource*. See Sections 4.2 to 4.6 of this paper.

[11]http://www.cloudcomputingpatterns.org/stateful_component/
[12]http://www.cloudcomputingpatterns.org/stateless_component/
[13]https://www.enterpriseintegrationpatterns.com/patterns/conversation/
[14]http://www.cloudcomputingpatterns.org/strict_consistency/
[15]http://www.cloudcomputingpatterns.org/eventual_consistency/

Example. The `Customer Core` microservice in the Lakeside Mutual sample[16] exposes master data. Its semantics and its operations (e.g., `changeAddress(...)`) are data- rather than action-oriented (the service is consumed by other microservices that are *Processing Resources* [49]):

```
@RestController
@RequestMapping("/customers")
public class CustomerInformationHolder {
    @ApiOperation(
        value = "Change a customer's address.")
    @PutMapping(
        value = "/{customerId}/address")
    public ResponseEntity<AddressDto> changeAddress(
        @ApiParam(
            value = "the customer's unique id",
            required = true)
        @PathVariable CustomerId customerId,
        @ApiParam(
            value = "the customer's new address",
            required = true)
        @Valid @RequestBody AddressDto requestDto) {
            [...]
    }

    @ApiOperation(
        value = "Get a specific set of customers.")
    @GetMapping(
        value = "/{ids}")
    public ResponseEntity<CustomersResponseDto>
      getCustomer(
        @ApiParam(
        value = "a comma-separated list of customer ids",
            required = true)
        @PathVariable String ids,
        @ApiParam(
          value = "a comma-separated list of the fields
            that should be included in the response",
            required = false)
        @RequestParam(
            value = "fields", required = false,
            defaultValue = "")
        String fields) {
            [...]
      )
    }
```

Implementation hints. Architects and developers who decide to apply and realize *Information Holder Resources* should take the following advice into consideration:

- Model the life cycle of the entities owned and controlled by instances of this pattern to identify their behavioral characteristics (e.g., master data vs. transactional data), e.g. starting from their appearance in use cases or user stories. Do not

[16]https://github.com/Microservice-API-Patterns/LakesideMutual/tree/master/customer-core

let the *Information Holder Resource* and its implementation turn into an anemic domain model[17].
- Do not unveil implementation details such as indexing and encodings in the technical part of the *API Description* [25] (rationale: adhere data independence principles from database design and information management best practices).
- Make any transferred data immutable once it leaves the API (only allow changing it through the API).
- Provide metadata as required to promote syntactic and semantic interoperability (for instance, data provenance information, timestamps, and data protection means).
- Do not directly access *Information Holder Resources* from Web clients unless the usage scenario is truly data-oriented (for instance as in storage sharing services such as Dropbox and ownCloud, see Known Uses of this pattern); rather call *Information Holder Resources* from *Processing Resources* to decouple the Web client from the storage.
- Do not let implementation details such as Object/Relational (O/R) mapper configurations or SQL snippets slip into the API. Transferring data sent to the API directly into the SQL used to query the underlying data store can expose the microservice to SQL injection attacks [12].
- Version the endpoint adequately, for instance with the help of *Semantic Versioning* and *Version Identifiers* [25]. Also define an evolution roadmap for the data definitions and version the data exchanged through the API (i.e., the request and response message structured exposed by the *Information Holder Resource*).

Consequences.

Resolution of forces.

+ *Modeling approach and API endpoint/service identification method.* Introducing *Information Holder Resources* often is the consequence of a data-centric approach to API modeling. It depends on the scenario at hand and the project goals/product vision whether such approach is adequate; while activity- or process-orientation is often preferred, is simply is not natural in a number of scenarios.
+ *Compliance with architectural design principles.* Processing will typically shift to the consumer of the *Information Holder Resource*. The *Information Holder Resource* then is solely responsible for acting as a reliable source of linked data (*Master Data Holder Reference Data Holder*, *Link Lookup Resource*), relationship sink (*Operational Data Holder*) or both (*Data Transfer Resource*).
– *Quality attribute conflicts and tradeoffs.* Using the API as an *Information Holder Resource* requires to carefully consider security, data protection, consistency, availability and coupling implications. Not all consumers may be authorized to access each *Information Holder Resource*. If they do, data consistency has to be preserved for concurrent access of multiple consumers. Likewise, consumers must deal with the consequences of temporary outages, e.g., by introducing an appropriate caching and offline data replication and synchronization strategy. Any change to the *Information Holder Resource* content, meta-data and representation formats needs to be controlled to avoid breaking consumers.
– *Compliance with architectural design principles. Information Holder Resources* have the reputation to increase coupling and violate the information hiding principle (see below).

The detailed qualities are determined on the operation level (e.g., *State Creation Operations* and *Retrieval Operations*).

Further discussion. *Modeling approach and API endpoint/service identification method.* Data-oriented methods are well suited to identify *Information Holder* endpoints, but sometimes go too far.[18] The counter position is taken by a post in M. Nygard's blog[19] for a responsibility-based strategy for avoiding pure *Information Holder Resources*, which he refers to as "entity service anti-pattern": He recommends to always evolve away from this pattern[20] (because it creates high semantic and operational coupling) and rather "focus on behavior instead of data" (which we describe as *Processing Resource* and "divide services by life cycle in a business process" (which we see as one of several service identification strategies). In our opinion, this advice goes too far as well: *Information Holder Resources* do have their place, but any usage should be a conscious decision motivated and justified by the business and integration scenario at hand — because of the impact on coupling that Nygard describes. For some data, it might be better indeed not to expose it at the API level but hide it behind *Processing Resources*.

Compliance with architectural design principles. The introduction of *Information Holder Resource* endpoints may break higher-order principles such as strict logical layering that forbids direct access to data entities in the presentation layer. It might be required to refactor the architecture [44] (or grant an explicit exception to the rule).

NFR tradeoffs. Quality attribute trees can steer the selection process [18]. In practice, the decision between availability and consistency is not as binary and strict as the CAP theorem suggests, which is discussed by its original authors in a 12-year retrospective and outlook[21] [4]. The Backup, Availability, Consistency (BAC) theorem adds an additional quality concern that causes conflicts [29]. The theoretical limitations or their practical implications cannot be argued or designed away; patterns such as the ones in this language can help to identify and manage them in clearly visible and well defined places in the architecture [32].

If several fine-grained *Information Holders* appear in an API, many calls might be required to get a user story realized, and data quality is hard to ensure (because it becomes a shared, distributed responsibility). Consider hiding several of them behind any type of *Processing Resource* or introduce composite *Information Holder Resources*.

Note that our usage and interpretation of the message exchange pattern *Request-Reply* [19] does not make any assumptions about

[17] https://www.martinfowler.com/bliki/AnemicDomainModel.html

[18] One of the classic cognitive biases is that every construction problem looks likes a nail if you know how to use a hammer and have one at hand.
[19] http://www.michaelnygard.com/blog/2018/01/services-by-lifecycle/
[20] http://www.michaelnygard.com/blog/2018/04/evolving-away-from-entities/
[21] https://www.infoq.com/articles/cap-twelve-years-later-how-the-rules-have-changed

the number of connections on the application- and transport protocol level. For instance, request-response uses a single HTTP connection according to the default SOAP/HTTP binding[22]. However, two interactions could also be used, one for supplying the request/input message and one for the response/output message; this has been specified by the ancient WS-Addressing standard[23], which also defines how the callback address is transmitted.

You might be wondering whether a new sort of data (e.g., analytical data, monitoring data) would have to be reflected by a new pattern in our language (by whatever nomenclature). This can be thought of, but is not required; arguably, the three dimensions and characteristics that we chose to carve out in the patterns (i.e., lifetime, mutability, and link reference structure) are particularly important when designing APIs and then implementing and deploying them (for instance, to cloud offerings). Analytical data can be seen as a special type of *Reference Data* (as it will not change); monitoring data has operational character. As a fallback, API designers can simply talk about the more general *Information Holder Resource* pattern if they struggle with our distinction by life time, mutability, and reference management.

Known Uses. *Information Holders* can be found in many public Web APIs and in middleware; they are seen less but do exist in business information systems:

- The Star Wars API[24] positions itself as data-oriented; it provides six *Information Holder Resources*: films, people, planets, species, starships, and vehicles. Its operations include *Retrieval Operations* (HTTP GETs), supporting searches by name and, in some cases, other attributes such as starship model.
- Document-oriented databases such as CouchDB and MongoDB provide native and direct access to the stored documents via HTTP interfaces; hence, these documents qualify as instances of the pattern. See for instance `GET /{db}/_all_docs` in the CouchDB API[25].
- Information management products deal with and expose *Information Holder Resources* by definition. Examples of such products include Master Data Management (MDM)[26] and Product Information Management (PIM)[27] systems; yet other systems deal with customer relationship data.
- Account information, billing statements, currency codes exposed in the APIs of cloud providers also qualify as known uses.
- Storage offerings such Dropbox, ownCloud and Amazon S3 provide abstractions such as file system space and key-value buckets; their APIs therefore also implement the *Information Holder Resource* pattern.
- A large data analytics solution currently being developed by a Swiss telecommunication service provider also uses this pattern to configure Hadoop jobs and supporting file systems.
- A Spring sample[28] uses this pattern. It is criticized for its impact on coupling by M. Nygard in this blog post[29].

Pattern uses in enterprise and government SOAs include:

- The Dynamic Interface described in an OOPSLA 2004 experience report[30] features a service that allows API clients to request an overview of selected bank customers and their financial transactions.
- Terravis [2] offers a `GetParcelIndex` operation, which can be called with different search parameters. The operation returns a list of `EGRIDs` (i.e., electronic parcel identifiers), which uniquely identify parcels Swiss-wide and can be used to retrieve detailed parcel information from the federated land registry systems.
- Usage scenarios such as open government data[31], partner information inventory, e-government data[32], and "show-only" data, e.g. mapping of partner id to user view in the real-estate process hub Terravis [26].

Related Patterns. This general *Information Holder Resource* pattern has several refinements that differ w.r.t. mutability, relationships, and instance lifetimes: *Operational Data Holder*, *Master Data Holder*, and *Reference Data Holder*. See Sections 4.2 to 4.4 of this paper.

The *Lookup Resource* pattern is another specialization; the lookup results may be *Information Holder Resources*. Finally, *Data Transfer Resource* holds temporary data owned by the clients. See Sections 4.5 and 4.6 of this paper.

The *Processing Resource* [49] pattern represents complementary semantics and is an alternative to this pattern.

State Creation Operations and *Retrieval Operations* can typically be found in *Information Holder Resources*, modeling create, read, update, and delete semantics. Stateless *Computation Functions* and read-write *State Transition Operations* are less common, but also permitted.

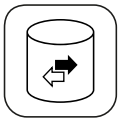
Information Holder is a role stereotype in Responsibility-Driven Design (RDD) [41]. Implementations of this pattern often can be seen as an API pendant to the *Repository* pattern in Domain-Driven Design (DDD) [8], [36]. *Information Holder Resources* are often implemented with one or more *Entities* from DDD, possibly grouped into an Aggregate. Note that no one-to-one correspondence between *Information Holder Resource* and *Entities* should be assumed because the primary job of the tactic DDD patterns is to organize the business logic layer of a system, not a (remote) *Service Layer* [11].

Other Sources. Chapter 8 in "Process-Driven SOA" is devoted to business object integration and dealing with data [16]. "Data on the Outside versus Data on the Inside"[33] by P. Helland explains the differences between data management on API and API implementation level [15]; the article is commented in this blog post[34].

[22] https://www.w3.org/2000/xp/Group/1/10/11/2001-10-11_Framework_HTTP_Binding
[23] https://www.w3.org/2002/ws/addr/
[24] https://swapi.co/
[25] http://docs.couchdb.org/en/2.1.1/api/
[26] https://en.wikipedia.org/wiki/Master_data_management
[27] https://en.wikipedia.org/wiki/Product_information_management
[28] https://spring.io/blog/2015/07/14/microservices-with-spring
[29] http://www.michaelnygard.com/blog/2017/12/the-entity-service-antipattern/
[30] http://soadecisions.org/soad.htm#oopsla04
[31] https://de.wikipedia.org/wiki/Open_Data
[32] https://opendata.swiss/en/
[33] http://cidrdb.org/cidr2005/papers/P12.pdf
[34] https://blog.acolyer.org/2016/09/13/data-on-the-outside-versus-data-on-the-inside/

The online article "Understanding RPC Vs REST For HTTP APIs"[35] talks about RPC vs. REST, but taking a closer look it actually (also) is about deciding between *Information Holder Resources* and *Processing Resources*.

Various consistency management patterns exist. We refer the reader to [9] which features patterns such as *Strong Consistency* and *Eventual Consistency*. A blog post from the Amazon Web Services CTO also covers this topic in depth [38].



4.2 Pattern: *Operational Data Holder*

a.k.a. Transaction(al) Data Holder, Secondary Data Access and Modification

Context. A domain model, an entity-relationship diagram or a glossary of key business concepts and their interconnections have been specified; it has been decided to expose some of these data entities in an API by way of *Information Holder Resource* instances.

The data specification unveils that the entity lifetimes/update cycles differ significantly (for instance, from seconds, minutes and hours to months, years and decades) and that the frequently changing entities participate in relationships with slower-changing ones. For instance, fast-changing data may mostly act as link sources while slow-changing data mostly appear as link targets.[36]

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?

Forces. Particularly relevant design time and runtime qualities when dealing with frequently changing data that is related to other data entities (including slower moving ones) include:

- *Processing speed* for daily content update operations
- Business *agility* and update flexibility, schema update *flexibility* and speed
- *Conceptual integrity and consistency*, e.g., of outgoing relationships

Details. *Processing speed (for daily content update operations).* Depending on the business context, services dealing with operational data must be extremely fast, with low response time both for reading and updating.

Business agility and update flexibility, schema update flexibility and speed. Depending on the business context, services dealing with operational data must also be easy to change (e.g, when performing A/B testing with parts of the live users) on the data definition (schema) level.

Conceptual integrity and consistency (e.g., of outgoing relationships). The created and modified operational data must meet the high accuracy and quality standards if the system is subject to audits, for instance in system and process assurance audits of financially relevant business objects in enterprise applications a.k.a. business information systems (called "Fin-BOs" in [20]). Operational data might be owned, controlled and managed by a (vertical or horizontal) integration partner, and might have many outgoing links (relations) to similar data and longer lived, less frequently changing data (i.e., master data; see sibling pattern *Master Data Holder*). Consumers expect that the referred entities will be correctly accessible, after the interaction with an operational data resource has successfully completed.

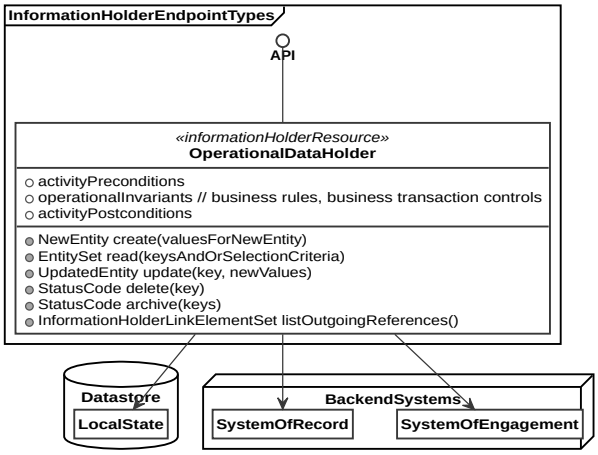


Figure 3: Operational Data Holder (Sketch). Operational data has a short to medium lifetime and may change a lot. It may reference master data and other operational data.

Non-solution. One could think of treating all data equally to promote solution simplicity, irrespective of its lifetime and relationship characteristics. However, such unified approach might only yield a mediocre compromise that meets all of the above needs somehow, but does not excel with regard to any of them. If, for instance, operational data is treated as master data, one might end up with an over-engineered API w.r.t consistency and reference management that also leaves room for improvement w.r.t. processing speed and change management.

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 (CRUD) its data often and fast.

Optionally, expose additional operations to give the *Operational Data Holder* domain-specific responsibilities. For instance, a shopping basket might offer fee and tax computations, product price update notifications, discounting, and other state-transitioning operations.

How it works. The request and response messages of such *Operational Data Holders* often take the form of *Parameter Trees* [51] or *Annotated Parameter Collections* (if accompanied by metadata); however, the other types of request and response message structure can also be found in practice. Figure 3 sketches the solution.

[35] https://www.smashingmagazine.com/2016/09/understanding-rest-and-rpc-for-http-apis/

[36] The context of this pattern is the similar to that of its sibling pattern *Master Data Holder*, but acknowledges and points out that the lifetimes and relationship structure of these two types of data differs (in German: "Stammdaten" vs. "Bewegungsdaten", see [10], [40]).

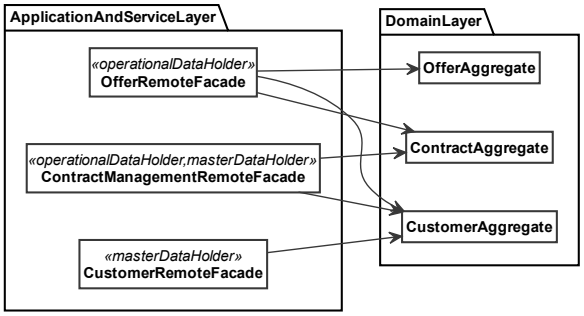


Figure 4: Examples of an Operational Data Holder (and Master Data Holders): offers reference contracts and customers, contracts reference customers. In this example, the remote facades access multiple aggregates isolated from each other.

Sometimes even operational data is kept for a long time: in a world of big data analytics and business intelligence insights, operational data is often archived for analytical processing, e.g., in data marts and data warehouses or semantic data lakes[37].

One must be aware of relationships with master data and consider adding them into messages via *Embedded Entity* or *Linked Information Holder* instances, two of our patterns not contained in this paper but covered in [50].

Example. Lakeside Mutual[38], our sample application from the insurance domain, manages operational data such as claims and risk assessments that are exposed as Web services and REST resources (Figure 4).

All basic and advanced structural patterns can be applied when designing the request and response messages of the operations of the *Operational Data Holder*, for instance *Pagination* [51]. Their applicability heavily depends on the actual data semantics. For instance, entering items into a shopping basket might expect a *Parameter Tree* [51] and return a simple success flag as an *Atomic Parameter* [51]. The checkout activity then might require multiple complex parameters (*Parameter Forest* [51]) and return the order number and the expected delivery date in an *Atomic Parameter List* [51]. The deletion of operational data can be triggered by sending a single *Id Element* and might return a simple success flag and/or *Error Report* representation.

Protect instances of this pattern with microservices infrastructure patterns such as *Circuit Breaker*[39] that shuts down an outgoing communication channel in case of connectivity problems (to avoid an increase of the stress level of the overall system and its components) and *Bulk Head*[40] that shuts down parts of a system (or services landscape) temporarily to protect other parts [28]. In the context of *Information Holder Resources*, outbound calls from *Operational Data Holders* to *Master Data Holders* can for instance be protected by a Circuit Breaker; closely related *Operational Data Holders*, for instance those forming/exposing an Aggregate in domain-driven designs [36], can be guarded by a joint *Bulk Head*.

Implementation hints. When realizing *Operational Data Holders*, one should consider to:

- Model the entities so that they actually serve an information need that can be traced back to a user story or use case.
- Test as realistically as possible, for instance with all data combinations and variations of cardinalities (e.g., account id vs. customer id in a core banking system: 1:1 or n:m?).
- Include the financially relevant *Operational Data Holders* in the *system and process assurance audits* [20].
- If justified by project-level quality requirements, optimize data access for mixed read/write operations; for instance, create read-only replicas and/or apply the *Command Query Responsibility Segregation (CQRS)*[41] pattern by introducing a command endpoint (that offers *State Creation Operations* and *State Transfer Operations* [49]) and a query endpoint (that offers *Retrieval Operations* [49]).
- Decide for an evolution strategy [25]: an *Operational Data Holder* may be subject to *Aggressive Deprecation* or only offer a *Limited Lifetime Guarantee*; mission-critical data might be released with a *Two in Production* policy.

Consequences.

Resolution of forces. The pattern primarily serves as a "marker pattern" in API documentation, helping with making technical interfaces "business-aligned", which is one of the SOA principles and microservices tenets identified in [45]:

+ The less inbound dependencies an *Operational Data Holder* has, the *easier to update* it is. Its limited life time helps to support backward compatibility when evolving the API.
+ There are many tactics[42] and patterns whose goal is to design *Operational Data Holders* that *perform and scale well*. For instance, relaxing its consistency properties can improve its availability. Turning it into a *Stateless Component*[43] promotes horizontal scalability.
– The *consistency and availability management* of *Operational Data Holders* may prioritize the conflicting requirements differently than *Master Data Holders* (depending on the domain and scenario); eventual consistency might be preferred over strict consistency at times.

Even if this pattern and suited related ones are chosen and applied well, the API implementations can still harm extensibility, performance, consistency, and availability.[44]

Further discussion. The distinction between master data and operational data is somewhat subjective and depending on application context; data that is needed only temporarily in one application might be a core asset in another one. For instance, think about purchases in an online shop: while the shopper only cares about the order until it it delivered and paid for (unless there is a warranty

[37] https://en.wikipedia.org/wiki/Data_lake
[38] https://github.com/Microservice-API-Patterns/LakesideMutual
[39] https://microservices.io/patterns/reliability/circuit-breaker.html
[40] https://docs.microsoft.com/en-us/azure/architecture/patterns/bulkhead
[41] https://martinfowler.com/bliki/CQRS.html
[42] https://www.viewpoints-and-perspectives.info/home/perspectives/performance-and-scalability/
[43] https://www.cloudcomputingpatterns.org/stateless_component/
[44] Such quality-related forces and patterns addressing them have many, many-to-many relationships.

case or (s)he wants to return the good or repeat the same order in the future), the shop provider will probably keep all details forever to be able to analyze buying behavior over time (customer profiling, product recommendations, and targeted advertisement).

The *Operational Data Holder* pattern can help to satisfy regulatory requirements expressed as compliance controls, for instance "all purchase orders reference a customer that actually exists in a system of record and in the real world" (to avoid fraud); see this example[45] of insufficient data management and controls [20].

Known Uses. Most, if not all, business applications deal with multiple instances of this pattern; many of these are exposed in solution-internal APIs or community APIs as well as public ones (which leads to "Software-as-a-Service* offerings). A few known uses are:

- The Cargo Tracking system that serves as Domain-Driven Design Sample Application[46] keeps operational data such as `Cargo`, `RouteSpecification` and `Itinerary` progress (leg/hop arrival is expressed as handling events) and exposes them to the application frontend via its (rather thin) application and presentation layer. See the root entity of the Cargo Aggregate[47] and its contained entities.
- Several of the "method families" in the Slack Web API[48] are data-oriented; the "conversations"[49] family, for instance, deals with operational data.
- Tweets and posts in social networks such as Twitter and Facebook also qualify as known uses if/as exposed in *Public APIs*.

As a process integration platform, Terravis [2] deals with many different process types and thus many transaction-specific data holders. For instance, when a loan is transferred from one bank to another, the securities also have to be transferred – in exchange for payments. Within this process, the two involved banks must negotiate the terms by offering (and accepting) a payment promise and a creditor release. These two entities depend on each other, and are only relevant until the transaction has been completed. Thus, they represent operational data (rather than master data).

Account endpoints in core banking APIs such as [3] also qualify as pattern instances; actually, the accounts can also be seen to be long-lived master data, while account transactions are truly operational.

Related Patterns. The alternative to this pattern are *Master Data Holder* and *Reference Data Holder* (instances of which live longer and have more incoming references); a less data- and more action-oriented alternative is *Processing Resource*. All operation responsibilities patterns including *State Creation Operation* and *State Transfer Operation* can be used in *Operational Data Holder* endpoints.

The *Data Type Channel* pattern in [19] describes how to organize a messaging system by message semantics and syntax (e.g., Query, Price Quote and Purchase Order). These channel types can be organized according to the terminology introduced by our pattern category *Responsibility Patterns*.

[45] https://en.wikipedia.org/wiki/Enron_scandal
[46] http://dddsample.sourceforge.net/characterization.html
[47] https://github.com/citerus/dddsample-core/tree/master/src/main/java/se/citerus/dddsample/domain/model/cargo
[48] https://api.slack.com/web
[49] https://api.slack.com/methods#conversations

Operational Data Holders referencing other *Operational Data Holders* may chose to include this data in the form of an *Embedded Entity*. On the contrary, references to *Master Data Holders* often are not included/embedded but externalized via *Linked Information Holder* [50] references.

Other Sources. See *Master Data Holder* for a brief discussion of the connection of our patterns to Domain-Driven-Design [8].

"Software Systems Architecture: Working With Stakeholders Using Viewpoints and Perspectives"[50] [33] has an information viewpoint.

"Data on the Outside versus Data on the Inside"[51] by P. Helland explains design forces and constraints for data exposed in APIs and application-internal data [15].

4.3 Pattern: *Master Data Holder*

a.k.a. Master Data Resource, Primary Data Access and Modification

Context. A domain model, an entity-relationship diagram, a glossary, or a similar dictionary of key concepts and their interconnections have been specified; it has been decided to expose some of these data entities in an API by way of *Information Holder Resources*.

The data specification unveils that the lifetimes and update cycles of these *Information Holder Resource* endpoints differ significantly (for instance, from seconds, minutes and hours to months, years and decades). Long-living data typically has many incoming relationships, whereas shorter-living data often references long-living data (outgoing relationships). In many application scenarios, data that is referenced in multiple places and lives long has high data quality and data protection needs. The data access profiles of these two types of data differ substantially.[52]

Problem. How can I create, read, update, and (possibly) delete data that lives long, does not change frequently, and is referenced often by other data directly or indirectly?[53]

Forces. The top-level forces that have to be resolved when dealing with any *Information Holder Resource* are discussed in the *Information Holder Resource* pattern. Additional concerns specific to master data are:

- Master data *quality*
- Master data *protection*
- *Data under external control*, for instance master data management systems

Details. *Master data quality.* Master data should have high quality as it is used (in)directly and/or implicitly in many places, from daily business to strategic decision making. If it is not stored and managed in a single place, uncoordinated updates, software bugs and other unforeseen circumstances may lead to inconsistencies

[50] https://www.viewpoints-and-perspectives.info/home/viewpoints/information/
[51] http://cidrdb.org/cidr2005/papers/P12.pdf
[52] The context of this pattern is the similar to that of its alternative pattern *Operational Data Holder*, but emphasizes that the lifetimes and relationship structure of these two types of data differs.
[53] Such data is often called *master data* and is contrasted to *operational data* a.k.a. transaction data (in German: "Stammdaten" vs. "Bewegungsdaten", see [10], [40]).

that are hard to detect. If it is stored centrally, access to it might be slow due to access contention, communication and coordination overhead.

Master data protection. Irrespective of its storage and management policy, master data must be well protected with suitable access control policies, as it make an attractive target for attacks, and the consequences of data breaches can be severe.

Data under external control. Master data may be owned and managed by dedicated systems, often purchased by (or developed in) a separate organizational unit (for instance, master data management systems specializing on product or customer data). Data ownership and audit procedures differ, and the data has a monetary value appearing in balance sheets of enterprises. Therefore its definitions and interfaces are hard to influence and change; due to the external influence on its lifecycle, master data may evolve at a different speed than operational data that references it. An external hosting (strategic outsourcing) of the specialized master data management systems further complicates the integration scenario.

Non-solution. One could think of treating all entities/resources equally to promote solution simplicity, irrespective of their lifetime and relationship patterns. However, such an approach runs the risk of not addressing the concerns of stakeholders such as security auditors, data owners and stewards, and hosting providers (and, last but not least, the real-world correspondents of the data, for instance customers and other system users) satisfyingly.

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 (that must meet compliance requirements).

Optionally, offer other life cycle events or state transitions in this *Master Data Holder* endpoint. Also optionally, expose additional operations to give the *Master Data Holder* domain-specific responsibilities. For instance, an archive might offer time-oriented retrieval, bulk creations and purge operations.

How it works. A *Master Data Holder* is a special type of *Information Holder Resource*. Figure 5 shows its specific design elements.

This type of information holder offers certain operations to follow references; unlike the *Reference Data Holder*, it offers operations to manipulate the data via the API.

The request and response messages of *Master Data Holders* often take the form of *Parameter Trees* [51]; however, more atomic types of request and response message structure can also be found in practice: master data *creation* operations typically receive a simple to medium complex *Parameter Tree* because master data might be complex but is often created in one go, e.g., if entered completely by a user in a form (such as an account creation form). They usually return an *Atomic Parameter* [51] or an *Atomic Parameter List* [51] to report the *Id Element* or *Link Element* that identifies the master data entity uniquely/globally and reports whether the creation request was successful or not (for instance, using the *Error Report* pattern). Reasons for failure can be duplicate keys, violations of business rules and other invariants, or internal server-side processing errors (for instance, temporary unavailability of backend systems).

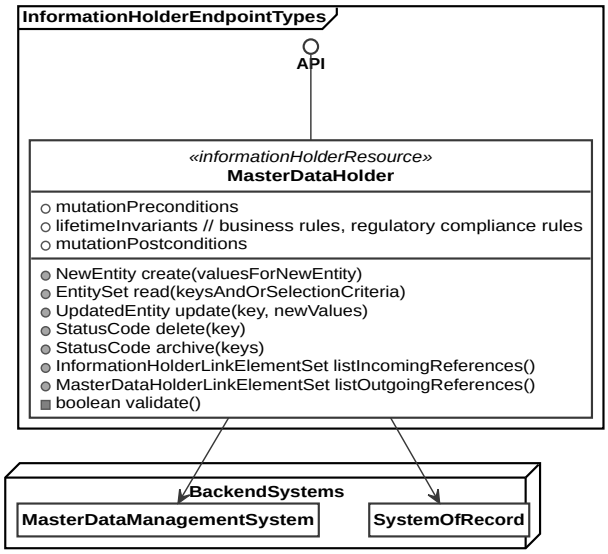


Figure 5: Master Data Holder (Sketch). Master data lives long and is frequently referenced by other master data and by operational data. It therefore faces specific quality and consistency requirements.

A master data *update* may come in two forms:

- as *coarse-grained* operation that replaces most or all attributes in a master data entity such as customer or product.
- as *fine-grained* operation that updates only one or a few of the attributes in a master data entity, for instance the address of a customer (but not its name) or the price of a product (but not its supplier and taxation rules).

Read access to master data is often performed via *Retrieval Operations* that offer parameterized search-and-filter query capabilities (possibly expressed declaratively).

Deletion might not be desired or hard to implement. It might also be mandatory to offer due to legislation: *Delete* operations on master data are complicated and risky due to the large amount of incoming references; sometimes, it might not be permitted to delete it for legal reasons. Hence, master data often is not deleted at all, but set into a "immutable/archived" state in which updates are no longer possible. This also allows keeping audits trails and historic data manipulation/access journals because master data changes are often critical and thus must be non-repudiable. If deletion is really necessary (and this can be a regulatory requirement as well), the data may be actually be hidden from (some) consumers but still preserved in a hidden or invisible state (unless legislation forbids this).

In an HTTP resource API, the address (URI) of a master data resource can be widely shared among clients referencing it, which can access it via HTTP GET (a read-only method that supports caching). The creation and update calls make use of POST, PUT, and PATCH methods, respectively [1].

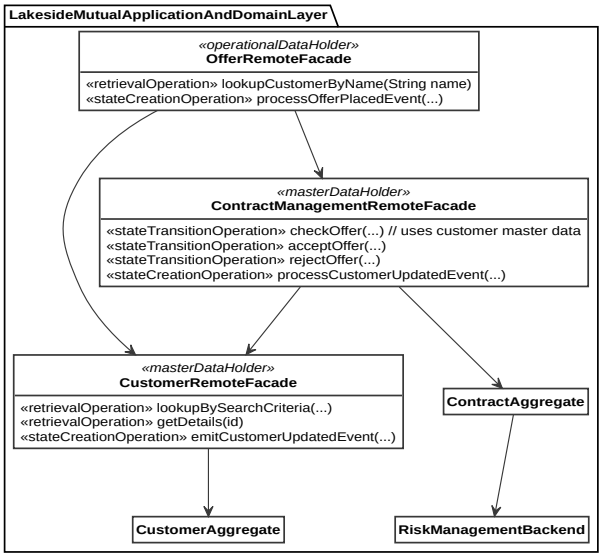


Figure 6: Example of Operational Data Holder and Master Data Holder interplay. In this example, the remote facades access each other and domain-layer aggregates.

Example. Lakeside Mutual[54], our sample application from the insurance domain, features master data such as customers and contracts that are exposed as Web services and REST resources (Figure 6), thus applying the *Master Data Holder* pattern.

Implementation hints. When introducing *Master Data Holders* into API architectures, one should consider these hints:

- Be aware of relationships with other master data and carefully consider whether to reference them or to include them in messages. Due to the amount of incoming references, *Master Data Holders* often are not included in responses from *Operational Data Holders* via *Embedded Entities* [50], but rather externalized via *Linked Information Holders* [50]. Analyze collection of API use cases across clients to decide between these two patterns. Avoid circular dependencies, as they make understanding the data and changing it much harder.
- If the (many) clients of *Master Data Holders* have different information needs, patterns such as *Wish List* and *Wish Template* can be used to reduce unnecessary data transfer. Another alternative is known as the Backends For Frontends[55] pattern.
- Consider to separate read access to master data from write access, e.g., applying Command-Query Responsibility Segregation (CQRS)[56].
- Use the conceptual separation of *Master Data Holders* and *Operational Data Holders* to drive service cuts[57]. Do not update master data as a hidden side effect of updates to *Operational Data Holders*, but make this dependency and impact on application state explicit in the API
- By definition, master data is "shareable data" because it has incoming references. Therefore, a crisp exact definition and strong syntactical validation is required in many scenarios (e.g., of phone numbers and addresses across countries). You can consider to assigned this responsibility to an explicit *Validation Service* [49] operation.
- Write and keep conversion utilities; do not rely on publicly available ones that might disappear or commercial products that might be discontinued over long time periods.
- Invest in data quality initiatives, incentives, and metrics for all business-critical master data, particularly if this data included in data analytics efforts (algorithms, reports) driving strategic business decisions. If insufficient emphasis is put on data quality and overcoming heterogeneity, a "data lake" as defined in the semantic big data management community easily morphs into a data "swamp" (or data "bog").
- Define Service Level Objectives (SLOs) for all operations and group them into a *Service Level Agreement* [34] for the *Master Data Holder*. Include data freshness information in these SLOs/SLAs.
- Decide for an evolution strategy [25]: a *Master Data Holder* may offer a rather long but still *Limited Lifetime Guarantee*. A *Two in Production* policy is often chosen so that the many clients can evolve more independently.
- Backup data in different formats to make it future-proof (and test the restore functions regularly, e.g., after software updates).
- Consider a *Master Data Management (MDM)*[58] solution or product as service realization (but acknowledge that this is a nontrivial make-or-buy-decision with long term consequences).

Consequences.

Resolution of forces.

+ Tagging an API endpoint as a *Master Data Holder* can help to achieve the required focus on *data quality* and *data protection*.
– Master data by definition has many inbound dependencies and might also have outbound ones. Additional patterns are required to ensure the consistency, freshness, etc. of these link relationships.

Tagging an API endpoint as a *Master Data Holder* alone does not resolve any of the forces; the implementation hints should be taken into account when doing so during endpoint realization. Additional patterns from other languages are eligible to resolve the individual forces [14], [13], [6].

Further discussion. Master data often is a valuable company asset that appears in the balance sheet (or turns a company into an acquisition target). Hence, it is particularly important to plan its future evolution in a roadmap that respects backward compatibility, considers digital preservation, and protects the data from theft and tampering.

[54] https://github.com/Microservice-API-Patterns/LakesideMutual
[55] https://samnewman.io/patterns/architectural/bff/
[56] https://www.infoq.com/articles/microservices-aggregates-events-cqrs-part-2-richardson/
[57] https://github.com/ServiceCutter/ServiceCutter/wiki/CC-8-Content-Volatility
[58] https://en.wikipedia.org/wiki/Master_data_management

Known Uses. Known uses of *Master Data Holders* are many, appearing in sample applications, public Web APIs, and in real-world enterprise and government information systems:

- The Cargo Tracking system that serves as Domain-Driven Design Sample Application[59] holds master data such as ports and possible routes and exposes this data to the application frontend via its (rather thin) application and presentation layer. See the root entity of the Cargo Aggregate[60] and its contained entities.
- In the e-commerce application that serves as example for Chris Richardson's microservices patterns[61] the Inventory Service and the Account Service use this pattern.
- Several of the "method families" in the Slack Web API[62] are data-oriented; the "users"[63] family, for instance, deals with master data.
- In the order management SOA described in [46], customers with their billing plans and the managed telephony network qualify as master data. Specific business services are dedicated to updating it.
- The UID service offered by the Swiss government[64] exposes company information that qualifies as *master data*. The public part of the API supports company searches and tax number validation.
- A major German car manufacturer offers a REST level 2 user profile management service for all its clients; this service features known uses of several responsibility patterns. For instance, it validates data strictly, converts addresses and phone numbers to country-specific standards, and offers create, read, update (but no search) operations in its Swagger/Open API contracts exposed in an *API Description* [25] website. One objective of this elaborate design is to comply with the EU General Data Protection Regulation (GDPR)[65].
- The Terravis system [2] offers a Web service to query federated master data concerning parcels, rights, and persons in Swiss land registers. Terravis also offers a service which can be used to query rich master data including name, address and contact information for all banks, notaries, and land registries participating in the platform.

Related Patterns. The *Master Data Holder* pattern has two alternative patterns *Reference Data Holder* (immutable) and *Operational Data Holder* (shorter lived, less incoming references). Each operation exposed by an endpoint with *Master Data Holder* semantics requires a request and response message structure that can be expressed with patterns such as *Atomic Parameter List* [51].

Domain-Driven Design (DDD) does not distinguish between master data and operational data in its tactic patterns [8]; both operational data and master data may be part of the *Published Language* and appear in dedicated *Bounded Contexts* and *Aggregates* as *Entities* (see [36]).

Other Sources. The notion of master data vs. operational/transaction(al) data comes from literature in the database community (more specifically, information integration) and in business informatics ("Wirtschaftsinformatik" in German, [10]). It also plays an important role in Online Analytical Processing (OLAP), Data Warehouses, and Business Intelligence (BI) efforts [22]. Such efforts are predecessors of the current big data analytics trend.

"Data on the Outside versus Data on the Inside"[66] [15] does not distinguish between master data and operational data, but still is a good read when it comes to designing *Mater Data Holders*.

4.4 Pattern: *Reference Data Holder*

a.k.a. Immutable Endpoint/Immutable Data Holder, Static Data Resource, Reference Data Lookup Table

Context. A requirements specification unveils that some data is referenced in most if not all system parts, but changes only very rarely (if ever); these changes are of administrative nature and not caused by API clients operating during everyday business. Such data is called *reference data*.[67] It comes in many forms: units of measurement, zip codes, country codes, currency codes, geo locations, etc.[69]

The data transfer representations in the request and response messages of API operations may either contain – or point at – reference data to satisfy the information needs of a message receiver.

Problem. How should data that is referenced in many places, lives long, and is immutable for clients be treated in API contracts? How can such reference data be used in requests to and responses from API endpoints such as *Processing Resources* [49] or *Information Holder Resources*?

Forces. The following specific forces have to be resolved when dealing with static, immutable data:

- *Performance vs. consistency trade-off for read access*
- *Do not repeat yourself (DRY)*

Details. *Performance vs. consistency trade-off for read access.* Since static reference data rarely changes, it may pay off to introduce a cache to reduce round-trip access response time and reduce server traffic if it is referenced and read a lot. Such replication tactics have to be designed carefully so that they function as desired and do not make the end-to-end system overly complex and hard to maintain. For instance, caches should not grow too big and replication has to be able to tolerate network partitions (outages); if the static data does change (on schema or on content level), updates have to be applied consistently. Examples: new zip

[59] http://dddsample.sourceforge.net/characterization.html
[60] https://github.com/citerus/dddsample-core/tree/master/src/main/java/se/citerus/dddsample/domain/model/cargo
[61] http://microservices.io/patterns/microservices.html
[62] https://api.slack.com/web
[63] https://api.slack.com/methods#users
[64] https://www.isb.admin.ch/isb/de/home/e-services-bund/services/uid-webservice.html
[65] https://www.eugdpr.org/
[66] http://cidrdb.org/cidr2005/papers/P12.pdf
[67] See "Data on the Outside vs. Data on the Inside"[68] by P. Helland, for an introduction to reference data (in the broad sense of the word).
[69] See https://en.wikipedia.org/wiki/Reference_data for links to inventories/directories of reference data.

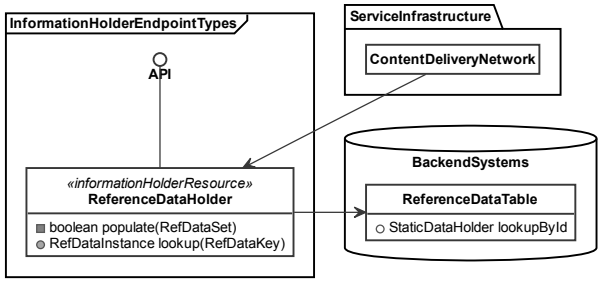


Figure 7: Reference Data Holder (Sketch). Static reference data lives long but never changes. It is referenced often and in many places.

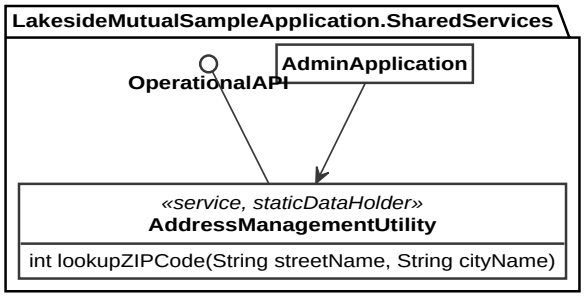


Figure 8: Reference Data Holder (Example): ZIP code lookup

codes in a country, transition from local currencies to Euro (EUR) in Europe.

Do not repeat yourself (DRY). Since reference data rarely changes (if ever), there is a temptation to simply hard code it within the API consumers, or if using a cache, retrieve it once and then store a local copy forever. Such designs work well in the short run and might not cause any problems – until the data and its definitions are to change.[70] Since the DRY (do not repeat yourself) principle is violated, the change will impact every client, and if clients are out of reach, it may no longer or not even be possible to update them.

Non-solution. One could treat static and immutable reference data just like dynamic data that is both read and written. This works fine in many scenarios, but misses opportunities to optimize the read access, for instance, via data replication in Content delivery Networks (CDNs) and might lead to unnecessary duplication of storing and computing efforts.

Solution. Provide a special type of *Information Holder Resource* endpoint, a *Reference Data Holder*, as a single point of reference for the static, immutable data.[71] Provide read operations, but no create, update, or delete operations in this endpoint. Update the reference data elsewhere if needed (backend, separate management API).

The *Reference Data Holder* may allow clients to retrieve the entire reference data set so that they can copy it locally (e.g., for accessing it multiple times), partially filter its content before doing so (e.g., to implement some auto-completion feature in a form), or to lookup individual entries of the reference data (e.g., for validation purposes).

How it works. The request and response messages of *Reference Data Holders* often take the form of *Atomic Parameters* [51] or *Atomic Parameter Lists* [51], for instance when the reference data is unstructured and merely enumerates certain flat values; however, the other types of request and response message structure can also be found in practice. Figure 7 sketches the solution.

The currency list can be copy-pasted all over the place (as it never changes) or it can be retrieved and cached from the *Reference Data Holder* API as described here. Such API can provide a complete enumeration of the list (to initialize and refresh the cache) or feature the ability to project/select the content (for instance, a list of European currency names), or allow clients to check if some value is present in the list for client-side validation ("does this currency exist"?).

The operations of *Reference Data Holders* offer direct access to a reference data table. Such `lookupById` can map a short identifier (e.g., a provider-internal surrogate key) to a more expressive, human-readable identifier and/or entire data set.

The pattern does not prescribe any type of implementation; for instance, a relational database might come across as an over-engineered solution when managing a list of currencies; a file-based key-value store or Indexed Sequential Access Method (ISAM)[72] files might be sufficient. Key-value stores such as a Redis or document-oriented NoSQL database may also be considered.

Example. There is one main usage scenarios for this pattern, simple value data lookup (e.g., for country codes, currency codes). Figure 8 shows an instance of the pattern that allows API clients to lookup zip codes.

Implementation hints. Architects and developers that decide to feature explicit *Reference Data Holders* in their API architectures should consider following tips:

- Do not reinvent the wheel: to enhance the reusability of the endpoint, look for standard(ized) formats and content, for instance when dealing with lists of country codes. Both de facto and de jure standards may be considered, both public and organization-wide ones.
- Document data provenance information such as de-jure or de-facto standards and their versions that define the data and its structure as *Metadata Elements*.
- When outsourcing commodity reference data management, note that his may introduce an external dependency for information that may be of critical importance for normal operation (what if the list of countries disappears or, even worse, gets polluted with incorrect data?).
- Define a management interface in addition to the operational interface, for instance to allow for batch updates and to obtain API usage statistics.

[70] For instance, it was sufficient to use two digits for calendar years until 20 years ago.
[71] To use terms from data warehousing and master data management: "single version of the truth" or "golden copy".
[72] https://en.wikipedia.org/wiki/ISAM

- Maintain a regression test suite to be prepared for the rare but not impossible event that the reference data and its structure do change.
- Decide for an evolution strategy: a *Reference Data Holder* is one of the few situations in which an *Eternal Lifetime Guarantee* [25] may make sense economically.
- *Reference Data Holder* lookups should be included in IT audits and systems and system process assurance audits to make sure that proper compliance controls are in place [20]. These controls protect the reference data from being tampered with as well as other security threats that may harm the accurate functioning of the data lookup and, in turn, all business services depending on them.

Consequences.

Resolution of forces.

+ Explicit, separate *Reference Data Holders* avoid unnecessary repetition (so DRY force resolved). The purpose of the *Reference Data Holder* is to give a central point of reference for helping disseminate the data while keeping control over it.
+ Read performance can be optimized; immutable data can be replicated rather easily (no risk of inconsistencies).

– Explicit, separate *Reference Data Holders* have to be developed, documented, managed, and maintained. This causes effort.

Further discussion. Performance optimization for read access. The pattern hides the actual data behind the API and therefore allows the API provider to introduce proxies, caches and read-only replicas behind the scenes. The only effect that is visible to the API clients is an improvement (if done right) in terms of quality properties such as response times and availability, possibly expressed in the *Service Level Agreement* [34] that accompanies the functional API contract.

Do not repeat yourself (DRY). Client no longer have to implement reference management on their own, at the expense of introducing a dependency on a remote API call. This positive effect can be viewed as a form of data normalization[73] as known from database design and information management.

If a standalone *Reference Data Holder* turns out to cause more work and complexity than it adds value (in terms of data normalization and performance improvements), one can consider to merge the simple static/immutable reference data with an already existing, more complex and somewhat more dynamic *Master Data Holder* endpoint in the API by way of an interface refactoring [21].

Known Uses. Publicly visible known uses include:

- The country abstractions/endpoints in the public, donation-funded API RESTCountries[74] are instances of this pattern. Similar services exist for currency codes.
- Taxonomies such as the topic keywords used in digital libraries such as the ACM Digital Library Computing Classification System[75] and IEEE Xplore qualify when exposed in APIs such as the IEEE Xplore API Portal[76].
- Geographic data such as maps, coordinates and zip codes also change rarely (although they still do change over long periods of time), and many APIs for them can be seen as *Reference Data Holders*.
  – An example is the Overpass API for Open Street Map[77].
  – The location database behind the Open Weather Map API[78] uses its own `city IDs` (for more than 200k cites) and also works with country codes, zip codes, and geo coordinates `long(itude)` and `lat(itudes)`.
- The unique company identifiers[79] ("UIDs") assigned by the Swiss government and available via a Web service qualify as *reference data*.

The core banking integration SOA described in [3] works with region codes and product/market categories.

Many digital archives must guarantee that stored documents are never changed and will therefore return them as static data. For instance, Terravis [2] uses such a digital archive and offers only read operations (i.e., *Retrieval Operations*) on existing documents. Moreover, Terravis offers a process meta-data search feature/service that returns information about process instances that will never change once a process instance has been completed, but might be amended before because additional information about a process instance is collected (e.g., process outcome.) For example, a notary might invite another bank to participate in a land register transaction. This newly added bank will be added to the reference data of the process instance.

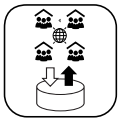
Examples of somewhat unexpected changes to static/immutable reference data include the introduction of the Euro currency in many European countries and new zip codes in Germany in the 1990s; APIs that work with such abstractions must make the data catalog version and data types (for instance, units of measures for distances and weights) explicit.

Related Patterns. The *Master Data Holder* pattern is an alternative to *Reference Data Holder*. It also represents long-living data, which still is mutable. *Operational Data Holders* represent more ephemeral data.

The *Reference Management* subcategory of our *Quality Patterns* features two related patterns, *Embedded Entity* and *Linked Information Holder* [50]. Simple static data is often embedded (which eliminates the need for a dedicated *Reference Data Holder*), but can also be linked (with the link pointing at a *Reference Data Holder*).

Other Sources. A definition of the term *reference data* can be found at TechTarget[80]: "consisting of sets of values, statuses or classification schema".

[73] http://searchsqlserver.techtarget.com/definition/normalization
[74] https://restcountries.eu/
[75] https://dl.acm.org/ccs
[76] https://developer.ieee.org/
[77] https://wiki.openstreetmap.org/wiki/Overpass_API
[78] http://openweathermap.org/api
[79] https://www.isb.admin.ch/isb/de/home/e-services-bund/services/uid-webservice.html
[80] http://whatis.techtarget.com/definition/reference-data



4.5 Pattern: *Data Transfer Resource*

a.k.a. Connector Resource, Integration Resource, Share, Temporary Data Store, Transient Information Holder

Context. Two or more communication participants want to exchange data. The number of exchange participants may change over time, and their existence only partially be known to each other. They might not always be active at the same time. For instance, additional participants may want to access the same data after it has already been shared by its source.

Participants may also only be interested in accessing the latest version of the shared information and do not need to observe every change applied to it. Communication participants may not be able to install any messaging middleware beyond a basic HTTP client library locally.

Problem. How can two or more communication participants exchange data without knowing each other, without being available at the same time, and even if the data has already been sent before its recipients became known?

Forces. The following forces have to be resolved in this setting:

- *Coupling (dimensions: time and location)*
- *Communication constraints*
- *Reliability*
- *Scalability*
- *Storage space efficiency*
- *Latency*
- *Ownership management*

Details. *Coupling (time).* Communication participants may not be able to communicate synchronously (at the same time) as their availability and connectivity state may change over time. The more communication participants want to exchange data, the more unlikely it is that all will be ready to send and receive messages at the same time.

Coupling (location). The location of communication participants may be unknown to the other participants. It may not be possible to directly address all participants due to asymmetric network connectivity, making it difficult, for example, for senders to know how to reach the recipients of the data exchange that are hidden behind a Network Address Translation (NAT)[81] table or a firewall.

Communication constraints. Some communication participants may be unable to directly talk to each other. For instance, clients in the client/server architectural style by definition are not capable of accepting incoming connections. Also, some communication participants may not be allowed to install software required for communication locally (e.g., messaging middleware). In such cases, indirect communication is the only possibility.

Reliability. Networks cannot be assumed to be reliable, and clients are not always active at the same time. Hence, any distributed data exchanges must be designed to deal with temporary network partitions and system outages.

Scalability. The number of recipients may not be known at the time the data is sent. This number could also become very large and increase access requests in unexpected ways. This, in turn, may harm throughput and response times. Also, scaling up the amount of data can be an issue: the amount of data to be exchanged may grow unboundedly and beyond the capacity limits of individual messages (as defined by the communication and integration protocols used).

Storage space efficiency. The data to be exchanged has to be stored somewhere, and sufficient storage space must be available. The amount of data to be shared must be known as there may be limits on how much data can be transferred (bandwidth) or stored.

Latency. Direct communication tends to be faster than indirect communication via relays or intermediaries.

Ownership management. Ownership of the shared information has to be established to achieve explicit control over its availability lifecycle. The initial owner is the participant sharing the data; however there may be different parties responsible for cleanup: the original sender (interested in maximizing the reach of the shared data), the intended recipient (who may or not want to read it multiple times), or the host of the transfer resource (who must keep storage costs in check).

Non-solution. One could think of using enterprise integration patterns such as *Publish-Subscribe Channel* [19] offered by Message-Oriented Middleware (MOM) such as ActiveMQ, Apache Kafka or Rabbit MQ, but then the clients would have to run their own local messaging system endpoint to receive and process incoming messages. MOM needs to be installed and operated, which adds to the overall systems management effort [19].

Solution. Introduce a special type of *Information Holder Resource*, a *Data Transfer Resource* endpoint with a globally unique, network-accessible address that two or more clients can use as a shared data exchange blackboard. Add at least one *State Creation Operation* and one *Retrieval Operation* to it.

Decide on data ownership and its transfer; prefer client ownership over provider ownership (in this case).

Figure 9 sketches the solution.

How it works. Multiple applications (in API client roles) can use the shared *Data Transfer Resource* as a medium to exchange information which is originally created by one of them and then transferred to the shared resource. Once the information has been published in the shared resource, any additional client that knows the URI of the shared resource and is authorized to do so may retrieve it, update it, add to it, and delete it (when the data is no longer useful for any client application).

The shared *Data Transfer Resource*[82] establishes a shared, asynchronous data flow channel between its clients to mediate all interactions among them. As a result, clients can exchange data without having to directly connect to each other, or – perhaps even more importantly – without having to address each other directly and without being up and running at the same time. Hence, it decouples them in time (no need to be available at the same time) and makes their location irrelevant — as long as they all can reach the shared *Data Transfer Resource*.

[81] https://en.wikipedia.org/wiki/Network_address_translation

[82] a.k.a. blackboard or, for younger readers, whiteboard.

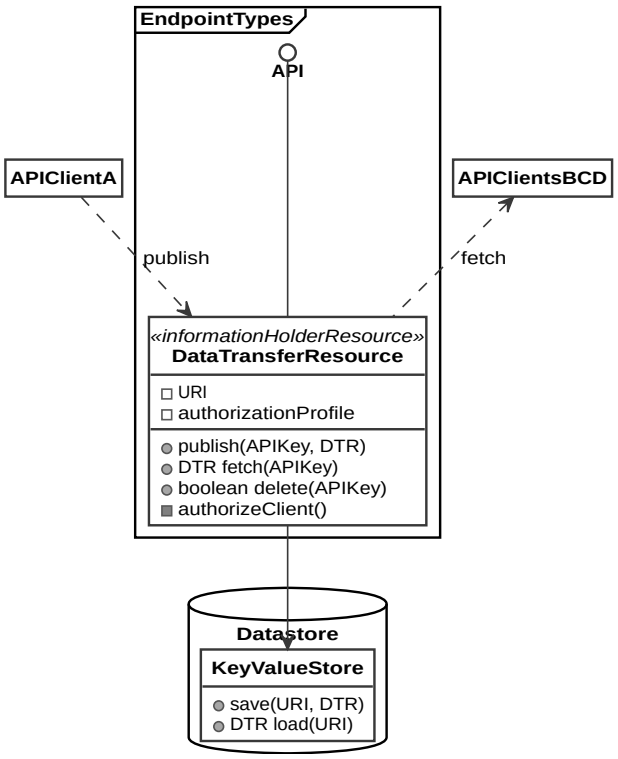


Figure 9: Data Transfer Resource (Sketch). A Transfer Resource endpoint holds temporary data and decouples two or more API clients. The pattern instance acts as a software connector (or data exchange blackboard) between these clients. Data ownership remains with the loosely coupled application clients.

How do client negotiate the URI for the shared resource? Clients may need to agree in advance about the shared resource address or they may dynamically discover it using a dedicated *Link Lookup Resource*. Also, it is possible that the first client sets the URI while publishing the original content and informs the others about it via some other communication channel, or by registering the address with a *Link Lookup Resource*, whose identity has been, again, agreed upon in advance by all clients.

HTTP support for the pattern: From an implementation perspective, this solution is directly supported in HTTP, whereby client A first performs a PUT request to publish the information on the shared resource, uniquely identified by a URI, and then client B performs a GET request to fetch it from the shared resource. Note that the information published on the shared resource does not disappear as long as no clients perform an explicit DELETE request. Client A publishing the information to the shared resource can do so reliably, as the HTTP PUT request is idempotent. Likewise, if the subsequent GET request fails, Client B may simply retry it to be able to eventually read the shared information. Figure 10 illustrates the HTTP realization of the pattern.

Clients cannot know whether other clients have retrieved the information from the shared resource. To address this limitation,

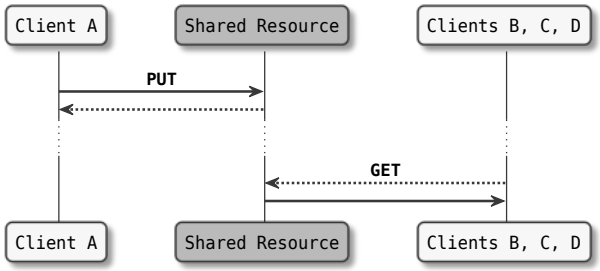


Figure 10: Data Transfer Resource (HTTP Realization)

Figure 11: Relay Resource (Sketch)

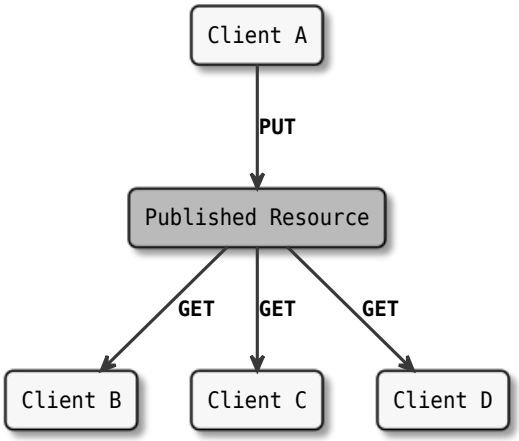


Figure 12: Published Resource (Sketch)

the shared resource can track access traffic and offer additional metadata about the delivery status so that it is possible to inquire whether and how many times the information has been fetched after it has been published. Such *Metadata Elements* may also help with the garbage collection of shared resources that are no longer in use.

Variants. Access patterns and resource lifetimes may differ, which suggests the following variants of this pattern:

1. *Relay Resource*: There are two clients only, one that writes and one that reads (Figure 11). Ownership is shifted from the writer to the reader.
2. *Published Resource*: One client writes as before, but then a very large, unpredictable number of clients read it at different times (maybe years later), as shown in Figure 12. Routing patterns can be supported this way, e.g. *Recipient List* [19]. The original writer determines for how long the shared resource remains publicly available to the multiple readers.

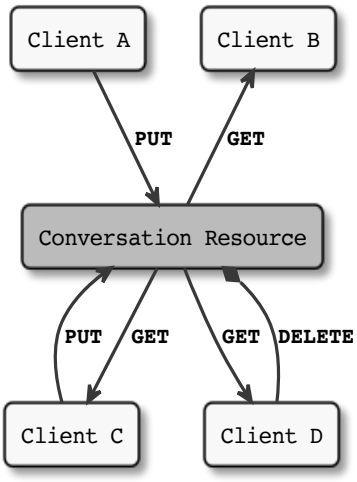


Figure 13: Conversation Resource (Sketch)

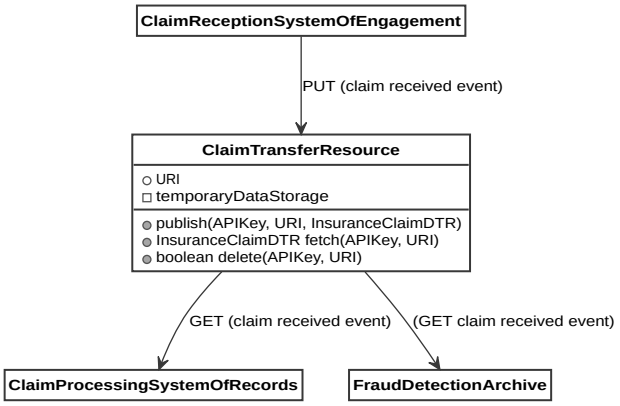


Figure 14: Claims management data flow as an example of a Data Transfer Resource; access is controlled with an API Key.

3. *Conversation Resource*: Many clients read and write and eventually delete the shared resource (Figure 13). Any participant owns (i.e., can both update or delete) the transfer resource.

Example. The example in Figure 14 instantiates the pattern for an integration interface in a fictitious insurance scenario. The `ClaimReceptionSystemOfEngagement` is the data source, and a `ClaimTransferResource` decouples the two data sinks (`ClaimProcessingSystemOfRecords`, `FraudDetectionArchive`) from it.

Implementation hints. Architects and developers that decide to introduce *Data Transfer Resources* into an integration architecture or microservices-based system should consider the following advice:

- Include the *Data Transfer Resource* endpoint into the systems management procedures and other operations concepts. For instance, purge the resource in regular intervals or alert an administrator if it is untouched for a certain amount of time or grows beyond a predefined quota. *Dead Letter Queue* and *Message Expiration* are related "Enterprise Integration Patterns" [19].
- Stick to other design and implementation advice for HTTP resource APIs, for instance regarding URI design, Web linking, and content negotiation. The "RESTful Web Services Cookbook" presents related recipes [1].
- You also may want to choose a REST maturity level[83] consciously and provide rationale for the decision; for *Data Transfer Resources*, level 2 (proper use of URIs and verbs) may not be sufficient as the URI of the shared resource has to be dealt with explicitly (i.e., included in response message bodies, which corresponds to maturity level 3).

Consequences.

Resolution of forces.

+ Coupling (time and location): asynchronous and indirect communication are supported.
+ Communication constraints: clients which cannot directly connect use the transfer resource as a shared blackboard.
+ Reliability by idempotent transfers.
+ Scalability both of amount of exchanged data and number of clients reading or writing it.
+ It depends on the chosen variant how flexible the ownership is/should and can be.
– Client polling: clients are unable to receive notifications from the transfer resource.
– Storage space efficiency: the *Data Transfer Resource* provider has to allocate sufficient space.
– Latency: indirect communication requires two hops between participants, which however do not have to be available at the same time. In this pattern, the ability to transfer data across large periods of time and multiple participants takes priority over the performance of the individual transfer.

Further discussion. The pattern combines the benefits of messaging and shared data repositories; flexibility of data flow and asynchrony [30]. Let us go through the forces and pattern properties one by one (in the context of HTTP and Web APIs).

Client constraint. Clients sometimes cannot directly talk to each other because:

1. They are clients (so not supposed to receive any incoming request).
2. They are running behind a firewall/Network Address Translator (NAT) which only allows outgoing connections.
3. They are running inside a Web browser which only allows sending HTTP requests to and receiving responses from a Web server.
4. They are not running at the same time.

If direct connectivity is impossible, then an indirect route may still work. The shared *Data Transfer Resource* provides such intermediary element and can serve as a shared information blackboard, which is reachable from both clients and remains available even when some of the clients temporarily disappear.

[83] https://martinfowler.com/articles/richardsonMaturityModel.html

Reliability/asynchrony and systems management. When using the messaging style, the connection from the client to the middleware can be a local one (the messaging system broker process then takes care of the remote messaging, guaranteeing message delivery). Such "programming without a call stack" is conceptually harder and more error prone than blocking remote procedure invocations, but also more powerful when done properly [19]. When applying the *Data Transfer Resource* pattern, the client-to-resource connection always is a remote one. Moreover, HTTP cannot guarantee message delivery. However, the idempotency of the PUT and GET methods in HTTP can mitigate the problem because the sending clients can retry calls to the *Data Transfer Resource* until the upload or download succeeds. When using such idempotent HTTP methods to access the shared resource, the middleware or the receiver do not have to detect and remove duplicate messages.

Scalability. The amount of data that can be stored on a Web resource is bound by the capacity of the data storage/file system that underlies the Web server. The amount of data that can be transferred to and from the Web resource within one standard HTTP request/response is virtually unlimited according to the protocol and therefore limited only by the underlying middleware implementations and client/server hardware capacity.

Data ownership. Depending on the pattern variant, data ownership – the right but also the obligation to ensure the validity of the shared resource content and to clean it up eventually – can stay with the source, be shared among all parties aware of its URI, or, be transferred to the *Data Transfer Resource*. The latter option is adequate if the source originally publishing the data is not expected to be present until all recipients have had a chance to read it.

Once a *Data Transfer Resource* has been introduced into an integration architecture, additional design issues arise:

- *Access control:* Depending on the type of information being exchanged, clients reading from the resource trust that the resource was initialized by the right sources. So in some scenarios, only authorized clients may be allowed to read from or write to the shared resource.
- *Lack of coordination:* Clients may read from and write to the shared resource at any time, even multiple times. There is little coordination between writers and readers beyond being able to detect empty (or non-initialized) resources.
- *Optimistic locking:* Multiple clients writing at the same time may run into conflicts, which should be reported as an error.
- *Polling:* Some clients cannot receive notifications when the shared resource state is changed and must resort to polling to be able to fetch the most recent version.
- *Garbage collection:* The *Data Transfer Resource* cannot know whether any client that has completed reading will be the last one; hence, there is a risk of leaking data unless it is explicitly removed. Housekeeping is required: purging *Transfer Resource* which have outlived their usefulness avoids waste of storage resources.

Known Uses. DropBox and ownCloud created a business model for usage of this pattern (Software-as-a-Service, SaaS) and provide related integration APIs (see here[84] and here[85]). The Doodle scheduling service is not a API, but a Web application that uses this pattern as well.

An article by B. Rücker introduces event command transformation[86], mentions asynchronous Web-based integration, and provides working examples.

A major German car manufacturer offers a REST level 2 user profile management service for all its clients; this service features a different, queue-based type of *Data Transfer Resource* to decouple clients from each other and from servers. In the spirit of the Reactive Manifesto[87], it uses asynchronous message passing (here: Amazon SQS). The queue is populated in the implementation of one API operation. Another API operation offers clients an opportunity to poll (look for) new appearances in the queue via a Web resource; if there are none, the call does not block but returns a "come back later" message after 30 seconds. Clients can select what they are interested in; the server also filters by authentication roles (which can be seen as an implementation of a *Retrieval Operation* [49]). Consumption policies and message structure vary.

Related Patterns. The pattern differs from other types of *Information Holder Resources* with respect to data access. The *Data Transfer Resource* exclusively owns and controls its own data store; the only way to access its content is via the published API of the *Data Transfer Resource*. Instances of other *Information Holder Resource* types may work with data that is accessed and possibly even owned by other parties (e.g. backend systems and their non-API clients). Likewise the *Data Transfer Resource* acts both as a data source and data sink, unlike other *Information Holders* which may either store operational data, master data, or reference data. A *Link Lookup Resource* only exposes metadata; a *Data Transfer Resource* can hold any data.

65 patterns for asynchronous messaging are described in [19]. A *Data Transfer Resource* can be seen as a Web-based realization of a *Message Channel*, supporting message routing and transformation, as well as several message consumption options (e.g., *Competing Consumers* and *Idempotent Receiver*). Queue-based messaging and Web-based software connectors (as described by this *Data Transfer Resource* pattern) can be seen as two different but related integration styles; these styles are compared in [30].

Blackboard pattern is a POSA 1 pattern [5], intended to eligible in a different context, but similar in its solution sketch. The Remoting Patterns book [37] describes the remoting style *shared repository*; our *Data Transfer Resource* can be seen as the API for a Web-flavored shared repository.

Other Sources. *Interfacer* is a role stereotype in Responsibility-Driven Design (RDD) that describes a related but more generic programming-level concept [41].

[84] https://www.dropbox.com/developers/documentation/http/overview
[85] https://doc.owncloud.com/server/developer_manual/core/apis/ocs-capabilities.html
[86] https://www.infoq.com/articles/microservice-event-choreographies#
[87] https://www.reactivemanifesto.org/

4.6 Pattern: *Link Lookup Resource*

a.k.a. Address Data Holder, API Directory, Endpoint Repository, Inventory/Discovery Resource, Service Registry

Context. The message representations in request and response messages of an API operation must satisfy the information needs of the message receivers entirely. To do so, these messages may contain references to other API endpoints. Sometimes, it is not desirable to expose such endpoint references to all clients directly because such direct exposure adds coupling (thus harming location and reference autonomy[88]).

Two reasons to avoid an address coupling between communication participants are:

- As an API provider, I want to be able to change the destinations of links freely when evolving API while workload grows and requirements change.
- As an API client, I do not want to have to change code and configuration (e.g., application startup procedures) when the naming and structuring conventions for links change on the provider side.

Problem. How can message representations refer to other, possibly many and frequently changing, API endpoints and operations without binding the message recipient to the actual addresses of these endpoints?

Forces. When structuring an API that deals with linked resources and data (in the broadest sense of these words), conflicting concerns exist:

- *Cohesion* within one endpoint and *coupling* between endpoints
- *Dynamic endpoint references*: flexible runtime changeability of endpoint references
- *Number of endpoints and API complexity*
- *Centralization vs. de-centralization*
- *Message sizes, number of calls, resource use*
- *Dealing with broken links*

Details. *Cohesion and coupling. Information Holder Resource* endpoints typically have rich interfaces, exposing multiple operations to create, read, update, delete the held data. If comprehensive lookup capabilities are also added to the endpoint contract, all features that deal with one particular data element are in one place (which is good), but this place becomes difficult to document and learn (and also to maintain and test); it can be seen to already violate the Single Responsibility Principle[89] (i.e., do one thing and do it right, and have only one reason to change).

Dynamic endpoint references. Solutions for binding references to endpoints at design or deployment time, including hard-coded references in the clients as well as more sophisticated binding schemes, are often not flexible enough to deal with situations in which dynamic changes to endpoint references at runtime are required. Endpoints that are taken offline temporarily for maintenance or load balancing with a dynamic number of receiving endpoints a typical examples. Another usage scenario are intermediaries and redirecting helpers that help overcome formatting differences after new API versions have been introduced.

Number of endpoints and API complexity. The coupling problem could be avoided by having a specific endpoint only for getting the address of another endpoint (e.g., an *Information Holder Resource*). But this would in the extreme case that all endpoints require such functionality, double the number of endpoints which would harden API maintenance and increase complexity of the API.

Centralization vs. de-centralization. Hard-coding references and having one specific endpoint per *Information Holder Resource*, discussed before, are highly de-centralized solutions; other designs could centralize the binding of references instead. However, any centralized solution will receive more traffic than partially autonomous, distributed ones.

Message sizes, number of calls, resource use. An alternative solution to consider any form of references used in clients, is to avoid them following the *Embedded Entity* pattern [50]. However, this increases message sizes. This has to be contrasted to any solutions for managing references to endpoints in clients, which generally requires more calls (but sometimes less). All these considerations influence the resource use in terms of server processing resources and network bandwidth.

Dealing with broken links. Consumers following references will assume these references lead to the correct API endpoint. If such references no longer work because the API endpoint address has changed, consumer may either fail as they are no longer able to connect to the API or access out-of-date information from a previous API version.

Non-solution. A simple approach could be to add lookup operations (a.k.a. special types of *Retrieval Operations* that return *Link Elements*) to already existing endpoints (such as *Information Holder Resources* and/or *Processing Resources*). This solution is workable but compromises cohesion within the endpoints and couples endpoints in the reference and location autonomy dimensions.

Solution. Introduce a special type of *Information Holder Resource*, a dedicated *Link Lookup Resource* endpoint that exposes special *Retrieval Operation* operations that return single instances or collections of *Link Elements* that represent the current addresses of the referenced API endpoints.

How it works. These *Link Elements* may point both at data-oriented *Information Holder Resources* endpoints as well as action-oriented *Processing Resources* [49].

The most basic *Link Lookup Resource* uses a single *Atomic Parameter* for the request message to identify the lookup target by its primary key, e.g., a plain/flat, but globally unique string identifier.[90] On the next level of client convenience, an *Atomic Parameter List* can be used if multiple lookup options and query parameters exist (this way, the lookup mode can/has to be specified by the client). The *Link Lookup Resource* returns global, network-accessible references to the held information (each taking the form of a *Link Element*, possibly amended with *->Metadata ELements that disclose the link type).

[88] https://www.cloudcomputingpatterns.org/loose_coupling/
[89] https://en.wikipedia.org/wiki/Single-responsibility_principle
[90] Such unique identifiers are also used to create *API Keys*.

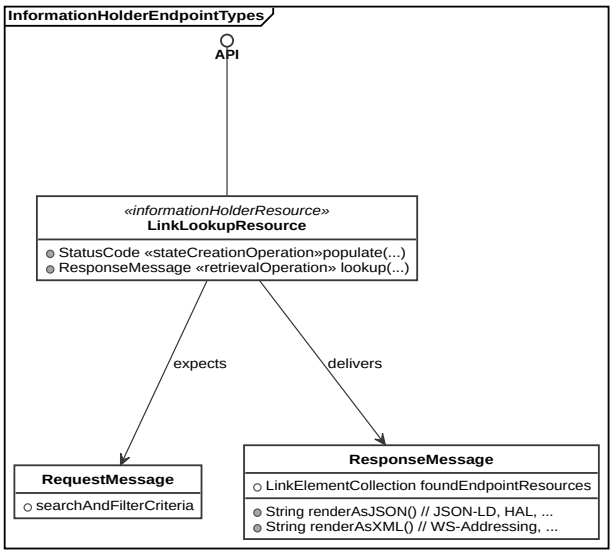


Figure 15: Link Lookup Resource (Sketch). A Lookup Resource is an API endpoint that merely holds information about other ones.

If the network addresses of one or more *Information Holder Resources* (or any of its refinements dealing with operational data, master data, reference data or serving as temporary *Data Transfer Resource*) are returned, the client can access these resources subsequently to obtain attributes, relationship information, and so on. Figure 15 sketches this solution.

Variants. When the *Link Elements* point at *Processing Resources* [49] rather than *Information Holder Resources*, a variant of this pattern is constituted:

Hypertext as the Engine of Application State (HATEOAS). HATEOAS is one of the defining characteristics of truly RESTful Web APIs according to the definitions of the REST style [7], [39].

The addresses of a few root endpoints (a.k.a. home resources) are published (i.e., communicated to prospective API clients); the addresses of related services can then be found in each response. The clients parse the responses to discover the URIs of subsequent *Processing Resource*. If a *Processing Resource* is referenced this way, the control flow and application state management become dynamic and highly decentralized; the operation-level pattern *State Transition Operation* [49] covers this REST principle in detail.

Example. In the Lakeside Mutual sample application, we can define two operations to find *Information Holder Resources* that represent customers (notation: Microservice Domain-Specific Language (MDSL)[91], a new style- and technology-independent service contract modeling language [21]):

```
API description MAPLinkLookupResource

data type URI P // protocol, domain, path, parameters

endpoint type LinkLookupResourceInterface // sketch
exposes
 operation lookupInformationHolderByLogicalName
   expecting payload
     <<Identifier_Element>> "name": ID
   delivering payload
     <<Link_Element>> "endpointAddress": URI

 operation lookupInformationHolderByCriteria
   expecting payload {
     "filter": P
   }
   delivering payload {
     <<Link_Element>> "uri": URI* // 0..m cardinality
   }

API provider CustomerLookupResource
  offers LinkLookupResourceInterface
```

If multiple results of the same type are returned, the *Link Lookup Resource* turns into a *Collection Resource* [1].

Implementation hints. Architects and developers that decide to use dedicated *Link Lookup Resources* should take the following advice into consideration:

- Do not create "über-directories" but design in a user requirement-driven way, for instance ask and answer "how do you find this information?" questions.[92]
- Apply hyperlink formats that are supported by tools such as HAL or JSON-LD; add link type information.
- *Conditional Requests* may help to meet advanced performance requirements. Caching is another option, but known to be generally difficult[93] to design and implement right.
- Ensure that write access is granted only to authorized parties, as consumers implicitly trust the lookup results as they follow them.
- To promote microservices principles such as agility and autonomy, make sure that all endpoints have a single reason to change [31]. This is particularly important for intermediaries such as *Link Lookup Resources* because by definition each of them add at least two dependencies to the overall services and API landscape.
- Define appropriate life cycle models and versioning strategies for the directory entries (i.e., the actual resources looked up) to avoid cluttering. Sunset entries when you can; apply evolution patterns [25] such as *Aggressive Deprecation* and *Limited Lifetime Guarantee* to govern their life cycle.

Consequences.

Resolution of forces.

+ The pattern decouples clients and providers in terms of location autonomy.

[91] https://microservice-api-patterns.github.io/MDSL-Specification/

[92] Universal Description Discovery and Integration (UDDI)], https://en.wikipedia.org/wiki/Web_Services_Discovery#Universal_Description_Discovery_and_Integration, one of the core XML Web services technologies from the early 2000s [52], had a reputation of not having followed this advice sufficiently in its API design and data model.

[93] https://martinfowler.com/bliki/TwoHardThings.html

+ The pattern promotes high cohesion within and low coupling within one endpoint as the lookup responsibility is separated from the actual processing and information retrieval.
– The pattern causes extra calls and increases the number of endpoints.
– The pattern causes operational costs; the lookup resource must be kept current.

Further discussion. Number of endpoints vs. cohesion. Usage of the pattern improves cohesion within endpoints at the expense of adding additional, specialized ones.

Performance. The pattern usage has a negative impact on the number of calls clients are required to send (liability) unless caching is introduced to mitigate this effect and lookup calls are only performed after detecting broken links. The pattern can only improve performance if the overhead for looking up the *Information Holder Resource* (or other provider-internal data storage) over an API operation boundary (so making two calls) does not exceed the savings achieved by leaner message payloads (of each operation).

Amount of endpoints. If the combination of a *Linked Information Holder* [50] with a *Lookup Resource* turns out to add more overhead than performance and flexibility gains, the *Link Lookup Resource* can be replaced with a direct link; if the direct linking still leads too overly chatty message exchanges (conversations) between API clients and API providers, the referenced data could be flattened as an instance of *Embedded Entity* [50].

Hypermedia usage is one of the defining constraints of the REST style and required to implement HATEOAS[94]. One has to decide whether the hypermedia should refer to the resources responsible for server-side processing (of any endpoint type) directly or whether a level of indirection should be introduced to further decouple clients and endpoints (this pattern).

Centralization vs. de-centralization. A maintenance budget for a shared APIs is often hard to get approved and renewed as it competes with requests for feature and project funding. While this is a general observation, it is particularly challenging to deal with when relatively simple features with a context-crossing nature have to be developed; hence, lookups to centralized *Link Lookup Resources* that cross domain boundaries should be handled with care.

Dealing with broken links. The added indirection can help to change the system runtime environment. For example, when directly using URIs, system might be harder to change because server names change etc. The REST principle of HATEOAS solves this problem for the actual names; only hardcoded client-side links are problematic. Microservices middleware such as API gateways[95] can be used as well; however, such usage adds complexity to the overall architecture as well as additional runtime dependencies.

Infrastructure-level service discovery can be used alternatively. For instance patterns such as Service Registry[96], Client-Side Discovery[97], and Self Registration[98] have been captured.

Known Uses. Instances of *Link Lookup Resource* can be found in public Web APIs and sample applications:

- The `Cargo Repository` in the Cargo Aggregate[99] of the Domain-Driven Design Sample Application[100] implements two basic find operations.
- Slack, which discloses an elaborate OpenAPI contract for its endpoints and operations[101], has the notion of object types[102] which can be seen as pattern instances (if exposed as API endpoints).

Terravis [2] internally offers and utilizes a lookup service that returns concrete API endpoints for given banks, notaries, and land registries. All these parties are referred to by a so-called Business Partner Identifier (BPID), which can be sent to the lookup service which then returns all known API endpoints offered by the identified partner.

Terravis also requires all parties to offer a service endpoint that returns a list of all supported APIs with their respective versions as a *Version Identifier* [25], as well as the actual endpoint. Terravis will query this service daily and cache the offered API versions and endpoints for use throughout the day.

Web Service Inspection Language (WSIL)[103] support in SOAP-based Web services endpoints and Universal Description, Discovery, and Integration (UDDI) APIs can be seen as obsolete implementations and known uses of *Link Lookup Resource* concepts.

Related Patterns. Instances of this pattern can return links to any of the endpoint types/roles, often to *Information Holder Resources*.

The pattern can be combined with *Retrieval Operations* (on operation level). For instance, a *Retrieval Operation* instances may return *Id Elements* pointing at *Information Holder Resources* indirectly (that in turn return the data); the *Link Lookup Resource* turns the *Id Element* into a *Link Elements*.

The *Collection Resource* recipe 2.3 in the *RESTful Web Services Cookbook* [1] can be seen as a RESTful HTTP pendant of this pattern, adding add and remove support; Chapter 14 then discusses discovery.

This pattern is an API-specific version/refinement of the more general *Lookup* pattern described in [23] and [37]. At a more abstract level, the pattern also is a specialization of the *Repository* pattern described in [8].

Other Sources. SOA books cover related concepts such as service repositories and registries. In Responsibility-Driven Design (RDD) terms, a *Link Lookup Resource* acts as a *structurer* [41].

5 CONCLUSIONS AND OUTLOOK

The knowledge captured in this paper (and its companion paper [49] that focuses on action-oriented endpoints and operation responsibilities) already has been used as guidance for making architectural decisions in industry projects. Our patterns are applicable not only to microservice APIs, but also to any remote API leveraging plain

[94] https://en.wikipedia.org/wiki/HATEOAS
[95] https://microservices.io/patterns/apigateway.html
[96] https://microservices.io/patterns/service-registry.html
[97] https://microservices.io/patterns/client-side-discovery.html
[98] https://microservices.io/patterns/self-registration.html
[99] https://github.com/citerus/dddsample-core/tree/master/src/main/java/se/citerus/dddsample/domain/model/cargo
[100] http://dddsample.sourceforge.net/characterization.html
[101] https://api.slack.com/specs/openapi/v2/slack_web.json
[102] https://api.slack.com/types
[103] http://www.onjava.com/pub/a/onjava/2002/10/16/wsil.html

document messages rather than stateful protocols or remote objects, synchronous ones using direct HTTP exchanges as well as asynchronous ones based on message queues. A reflection of the evolution of our pattern language since 2017 can be found online: "MAP Retrospective and Outlook"[104].

Selected patterns are implemented in the *Lakeside Mutual*[105] scenario and sample application. Lakeside Mutual is a fictitious insurance company that implemented its core business capabilities for customer, contract, and risk management as a set of microservices with corresponding application frontends. Furthermore, the emerging *Microservice Domain Specific Language (MDSL)*[106] features all patterns introduced in this paper as endpoint decorators, and our *Software/Service/API Design Practice Repository (DPR)* features the responsibility patterns in Step 5 of its stepwise service design method[107].

For the future, we consider to extend our pattern collection with further patterns that belong to other categories: for instance, additional structural representation patterns are currently being mined, captured, and validated. We also consider to cover implementation aspects of *Information Holders* and its refining patterns presented in this paper: concurrent access and modifications require some kind of business- or system-level transaction management, which can be implemented in several ways, including ACID-style consistency control as well as more relaxed BASE[108] approaches, including sagas[109] and other forms of business-level compensation.

ACKNOWLEDGMENTS

We want to thank our shepherd Stefan Sobernig, EuroPLoP 2020 writers' workshop participants, students and members of our professional networks who helped to investigate public Web APIs, donated candidate patterns and known uses, and reviewed drafts of pattern candidates and language structure. The work of Olaf Zimmermann and Mirko Stocker on MDSL and DPR is supported by the Hasler Foundation. The work of Cesare Pautasso and Uwe Zdun was supported by the API-ACE project, funded by SNF project 184692 and FWF (Austrian Science Fund) project I 4268.

REFERENCES

[1] Subbu Allamaraju. 2010. *RESTful Web Services Cookbook.* O'Reilly.
[2] Walter Berli, Daniel Lübke, and Werner Möckli. 2014. Terravis – Large Scale Business Process Integration between Public and Private Partners. In *Lecture Notes in Informatics (LNI), Proceedings INFORMATIK 2014*, Erhard Plödereder, Lars Grunske, Eric Schneider, and Dominik Ull (Eds.), Vol. P-232. Gesellschaft für Informatik e.V., Gesellschaft für Informatik e.V., 1075–1090.
[3] Michael Brandner, Michael Craes, Frank Oellermann, and Olaf Zimmermann. 2004. Web services-oriented architecture in production in the finance industry. *Informatik-Spektrum* 27, 2 (2004), 136–145. https://doi.org/10.1007/s00287-004-0380-2
[4] E. Brewer. 2012. CAP twelve years later: How the "rules" have changed. *Computer* 45, 2 (Feb 2012), 23–29.
[5] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. 1996. *Pattern-Oriented Software Architecture - Volume 1: A System of Patterns.* Wiley.
[6] Robert Daigneau. 2011. *Service Design Patterns: Fundamental Design Solutions for SOAP/WSDL and RESTful Web Services.* Addison-Wesley Professional. http://www.servicedesignpatterns.com/
[7] Thomas Erl, Benjamin Carlyle, Cesare Pautasso, and Raj Balasubramanian. 2013. *SOA with REST - Principles, Patterns and Constraints for Building Enterprise Solutions with REST.* Prentice Hall. I–XXXII, 1–577 pages.
[8] Eric Evans. 2003. *Domain-Driven Design: Tacking Complexity In the Heart of Software.* Addison-Wesley.
[9] Christoph Fehling, Frank Leymann, Ralph Retter, Walter Schupeck, and Peter Arbitter. 2014. *Cloud Computing Patterns: Fundamentals to Design, Build, and Manage Cloud Applications.* Springer.
[10] Otto K. Ferstl and Elmar J. Sinz. 2006. *Grundlagen der Wirtschaftsinformatik.* Oldenbourg.
[11] Martin Fowler. 2002. *Patterns of Enterprise Application Architecture.* Addison-Wesley.
[12] William G Halfond, Jeremy Viegas, and Alessandro Orso. 2006. A classification of SQL-injection attacks and countermeasures. In *Proceedings of the IEEE International Symposium on Secure Software Engineering*, Vol. 1. IEEE, 13–15.
[13] Robert Hanmer. 2007. *Patterns for Fault Tolerant Software.* Wiley.
[14] D.C. Hay. 1996. *Data Model Patterns: Conventions of Thought.* Dorset House. https://books.google.ch/books?id=a7VQAAAAYAAJ
[15] Pat Helland. 2005. Data on the Outside Versus Data on the Inside. In *CIDR 2005, Second Biennial Conf. on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2005, Online Proceedings.* 144–153. http://cidrdb.org/cidr2005/papers/P12.pdf
[16] Carsten Hentrich and Uwe Zdun. 2011. *Process-Driven SOA: Patterns for Aligning Business and IT.* Auerbach Publications.
[17] Gregor Hohpe. 2007. SOA Patterns: New Insights or Recycled Knowledge? Online article. https://www.enterpriseintegrationpatterns.com/docs/HohpeSOAPatterns.pdf
[18] Gregor Hohpe, Ipek Ozkaya, Uwe Zdun, and Olaf Zimmermann. 2016. The Software Architect's Role in the Digital Age. *IEEE Softw.* 33, 6 (2016), 30–39. https://doi.org/10.1109/MS.2016.137
[19] Gregor Hohpe and Bobby Woolf. 2003. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions.* Addison-Wesley.
[20] Klaus Julisch, Christophe Suter, Thomas Woitalla, and Olaf Zimmermann. 2011. Compliance by design–Bridging the chasm between auditors and IT architects. *Computers & Security* 30, 6 (2011), 410–426.
[21] Stefan Kapferer and Olaf Zimmermann. 2020. Domain-driven Service Design - Context Modeling, Model Refactoring and Contract Generation. In *Proc. of the 14th Advanced Summer School on Service-Oriented Computing (SummerSOC'20) (to appear).* Springer CCIS.
[22] Ralph Kimball and Margy Ross. 2002. *The Data Warehouse Toolkit: The Complete Guide to Dimensional Modeling* (2nd ed.). John Wiley.
[23] Michael Kircher and Prashant Jain. 2004. *Pattern-Oriented Software Architecture, Volume 3: Patterns for Resource Management.* Wiley.
[24] James Lewis and Martin Fowler. 2014. Microservices: a definition of this new architectural term. https://martinfowler.com/articles/microservices.html. https://martinfowler.com/articles/microservices.html
[25] Daniel Lübke, Olaf Zimmermann, Mirko Stocker, Cesare Pautasso, and Uwe Zdun. 2019. Interface Evolution Patterns - Balancing Compatibility and Extensibility across Service Life Cycles. In *Proc. of the 24th European Conference on Pattern Languages of Programs (EuroPLoP '19).*
[26] Daniel Lübke and Tammo van Lessen. 2016. Modeling Test Cases in BPMN for Behavior-Driven Development. *IEEE Software* 33, 5 (Sept.-Oct. 2016), 15–21.
[27] Sam Newman. 2015. *Building Microservices: Designing Fine-Grained Systems.* O'Reilly.
[28] Michael Nygard. 2018. *Release It! Design and Deploy Production-Ready Software* (2nd ed.). Pragmatic Bookshelf.
[29] Guy Pardon, Cesare Pautasso, and Olaf Zimmermann. 2018. Consistent Disaster Recovery for Microservices: the BAC Theorem. *IEEE Cloud Computing* 5, 1 (12 2018), 49–59. https://doi.org/10.1109/MCC.2018.011791714
[30] Cesare Pautasso and Olaf Zimmermann. 2018. The Web as a Software Connector: Integration Resting on Linked Resources. *IEEE Software* 35 (January/February 2018), 93–98. https://doi.org/10.1109/MS.2017.4541049
[31] Cesare Pautasso, Olaf Zimmermann, Mike Amundsen, James Lewis, and Nicolai M. Josuttis. 2017. Microservices in Practice, Part 1: Reality Check and Service Design. *IEEE Software* 34, 1 (2017), 91–98. https://doi.org/10.1109/MS.2017.24
[32] Cesare Pautasso, Olaf Zimmermann, Mike Amundsen, James Lewis, and Nicolai M. Josuttis. 2017. Microservices in Practice, Part 2: Service Integration and Sustainability. *IEEE Software* 34, 2 (2017), 97–104. https://doi.org/10.1109/MS.2017.56
[33] Nick Rozanski and Eóin Woods. 2005. *Software Systems Architecture: Working With Stakeholders Using Viewpoints and Perspectives.* Addison-Wesley Professional.
[34] Mirko Stocker, Olaf Zimmermann, Daniel Lübke, Uwe Zdun, and Cesare Pautasso. 2018. Interface Quality Patterns - Communicating and Improving the Quality of Microservices APIs. In *Proc. of the 23nd European Conference on Pattern Languages of Programs (EuroPLoP '18).*

[104] https://ozimmer.ch/patterns/2020/04/29/MAPRetrospective.html
[105] https://github.com/Microservice-API-Patterns/LakesideMutual
[106] https://microservice-api-patterns.github.io/MDSL-Specification/
[107] https://github.com/socadk/design-practice-repository/blob/master/activities/SDPR-StepwiseServiceDesign.md
[108] https://queue.acm.org/detail.cfm?id=1394128
[109] https://microservices.io/patterns/data/saga.html

[35] Francisco Torres. 2015. Context is King: What's Your Software's Operating Range? *IEEE Software* 32, 5 (2015), 9–12. https://doi.org/10.1109/MS.2015.121

[36] Vaughn Vernon. 2013. *Implementing Domain-Driven Design.* Addison-Wesley Professional.

[37] Markus Voelter, Michael Kircher, and Uwe Zdun. 2004. *Remoting Patterns - Foundations of Enterprise, Internet, and Realtime Distributed Object Middleware.* J. Wiley & Sons, Hoboken, NJ, USA.

[38] Werner Vogels. 2009. Eventually Consistent. *Commun. ACM* 52, 1 (January 2009), 40–44. https://doi.org/10.1145/1435417.1435432

[39] Jim Webber, Savas Parastatidis, and Ian Robinson. 2010. *REST in Practice: Hypermedia and Systems Architecture* (1st ed.). O'Reilly Media, Inc.

[40] Andrew White, David Newman, Debra Logan, and John Radcliffe. 2006. Mastering master data management. G00136958.

[41] Rebecca Wirfs-Brock and Alan McKean. 2002. *Object Design: Roles, Responsibilities, and Collaborations.* Pearson Education.

[42] Uwe Zdun, Mirko Stocker, Olaf Zimmermann, Cesare Pautasso, and Daniel Lübke. 2018. Guiding Architectural Decision Making on Quality Aspects in Microservice APIs. In *16th International Conference on Service-Oriented Computing ICSOC 2018.* 78–89. http://eprints.cs.univie.ac.at/5956/

[43] Olaf Zimmermann. 2009. *An architectural decision modeling framework for service-oriented architecture design.* Ph.D. Dissertation. University of Stuttgart, Germany. http://elib.uni-stuttgart.de/opus/volltexte/2010/5228/

[44] Olaf Zimmermann. 2015. Architectural Refactoring: A Task-Centric View on Software Evolution. *IEEE Software* 32, 2 (Mar.-Apr. 2015), 26–29. https://doi.org/10.1109/MS.2015.37

[45] Olaf Zimmermann. 2017. Microservices Tenets. *Comput. Sci.* 32, 3-4 (July 2017), 301–310. https://doi.org/10.1007/s00450-016-0337-0

[46] Olaf Zimmermann, Vadim Doubrovski, Jonas Grundler, and Kerard Hogg. 2005. Service-oriented architecture and business process choreography in an order management scenario: rationale, concepts, lessons learned. (2005), 301–312.

[47] Olaf Zimmermann, Jonas Grundler, Stefan Tai, and Frank Leymann. 2007. Architectural Decisions and Patterns for Transactional Workflows in SOA. In *Service-Oriented Computing - ICSOC 2007, Fifth International Conference, Vienna, Austria, September 17-20, 2007, Proceedings (Lecture Notes in Computer Science)*, Bernd J. Krämer, Kwei-Jay Lin, and Priya Narasimhan (Eds.), Vol. 4749. Springer, 81–93. https://doi.org/10.1007/978-3-540-74974-5_7

[48] Olaf Zimmermann, Pal Krogdahl, and Clive Gee. 2004. Elements of service-oriented analysis and design.

[49] 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 (EuroPLoP '20).*

[50] 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))*, Luís Cruz-Filipe, Saverio Giallorenzo, Fabrizio Montesi, Marco Peressotti, Florian Rademacher, and Sabine Sachweh (Eds.), Vol. 78. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 4:1–4:17. https://doi.org/10.4230/OASIcs.Microservices.2017-2019.4

[51] Olaf Zimmermann, Mirko Stocker, Daniel Lübke, and Uwe Zdun. 2017. Interface Representation Patterns: Crafting and Consuming Message-Based Remote APIs. In *Proc. of the 22nd European Conference on Pattern Languages of Programs (EuroPLoP '17).* ACM, Article 27, 36 pages. https://doi.org/10.1145/3147704.3147734

[52] Olaf Zimmermann, Mark Tomlinson, and Stefan Peuser. 2003. *Perspectives on Web Services: Applying SOAP, WSDL and UDDI to Real-World Projects.* Springer Science & Business Media.