Introduction to Microservice API Patterns (MAP)

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

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

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

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

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

Abstract

The Microservice API Patterns (MAP) language and supporting website premiered under this name at Microservices 2019. MAP distills proven, platform- and technology-independent solutions to recurring (micro-)service design and interface specification problems such as finding well-fitting service granularities, rightsizing message representations, and managing the evolution of APIs and their implementations. In this paper, we motivate the need for such a pattern language, outline the language organization and present two exemplary patterns describing alternative options for representing nested data. We also identify future research and development directions.

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

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

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

1 Motivation

It is hard to escape the term *microservices* these days. Much has been said about this rather advanced approach to system decomposition since its inception a few years ago [10]. The basic elements of a microservice-based message exchange are introduced in Figure 1.

Early adopters' experiences suggest that service design requires particular attention if microservices are supposed to deliver on their promises [16]:

- How many (micro-)service operations should be exposed in Application Programming Interfaces (APIs)?
- 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?
- What are suitable message representation structures and nesting levels, and how do these change throughout service life cycles?

Joint Post-proceedings of the First and Second International Conference on Microservices (Microservices 2017/2019).
Editors: Luís Cruz-Filipe, Saverio Giallorenzo, Fabrizio Montesi, Marco Peressotti, Florian Rademacher, and Sabine Sachweh; Article No. 4; pp. 4:1–4:17

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

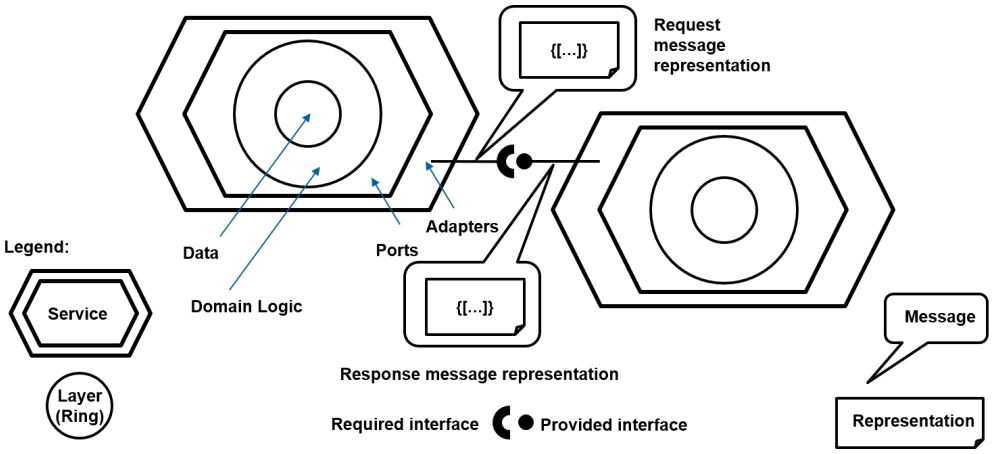

Figure 1 Microservices, represented as hexagons, exchange request and response message representations via platform-independent *ports* and technology-specific *adapters*. The inner structure of the services is sketched in onion form: each ring represents a local logical layer (e.g., *logic*, *data*).

- How can the meaning of message representations be agreed upon – and how to stick to these contracts in the long run?

To address these and related design issues and choose working combinations out of the many possible design options, application context and requirements must be analyzed. Our Microservice API Patterns (MAP) cover and organize this design space. Before we describe MAP and present two example patterns in the following sections, let us first recapitulate what microservices actually are (and where they came from).

1.1 A Consolidated Definition of Microservices

Microservices architectures have evolved from previous incarnations of Service-Oriented Architectures (SOAs) [5]. 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 [22], [13], [10].

When it comes to protocol selection, message-based APIs such as RESTful HTTP or queue-based event sourcing and streaming have come to dominate over remote procedure calls, including their object-oriented variants [15]. JSON is a particularly popular data serialization and message exchange format in many developer communities today.

1.2 Service Design Challenges

Microservices architectures include many remote APIs. The data representations exposed by these APIs must not only meet the information and processing needs of clients and other services, but also be designed and documented in an understandable and maintainable way.

While microservice API design and implementation might seem to be simple and straightforward from the distance, a closer look unveils that a lot of interesting problems are awaiting API teams:

- *Requirements diversity*: The wants and needs of API clients differ from one another, and keep on changing. API providers have to decide whether they want to offer good-enough compromises in a single unified API or try to satisfy all client requirements individually.
- *Design mismatches*: What backend systems can do (in terms of functional scope and quality), and how they are structured (in terms of endpoint and data definitions), might be different from what clients expect. These differences have to be overcome.
- *Open vs. closed systems*: API clients and providers often have conflicting goals. For instance, the desire to innovate and market dynamics such as competing API providers trying to catch up on each other may cause more change and possibly incompatible evolution strategies than clients are able or willing to accept. Publishing an API means opening up a system and giving up some control, thus limiting the freedom to change it. Clients might use data that is exposed by an API in unexpected ways.
- *Stability vs. flexibility*: Microservices help to enable frequent releases, e.g., in the context of DevOps practices such as continuous delivery. Changes are released at an ever increasing pace. In contrast, APIs should stay as stable as possible to avoid breaking client code. This constant conflict needs to be resolved by microservice API designers.

These conflicting requirements and stakeholder concerns must be balanced; many design trade-offs can be observed:

- *Few operations that carry lots of data back and forth vs. many chatty, fine-grained interactions.* Which choice is better in terms of performance, scalability, bandwidth consumption and evolvability?
- *Stable, standardized, elaborate interfaces vs. fast changing, specialized, focused ones.* How to find a balance between breadth and depth? How to keep the interfaces compatible without sacrificing their extensibility?
- *Data consistency vs. reliability and fast response times.* Should state changes be reported via coordinated API calls or via reactive event sourcing and streaming? Should commands and queries be separated architecturally? To which extent can and should consistency, availability, and recoverability (backup) requirements be satisfied? [14]

1.3 Existing Design Heuristics

One can find many excellent books providing deep advice about using RESTful HTTP, e.g., which HTTP verb or method to pick to implement a particular operation, or how to apply asynchronous messaging including routing, transformation, and guaranteed delivery [1], [7]. Strategic Domain-Driven Design [3], [19] can assist with service identification. SOA, cloud and microservice infrastructure patterns have already been proposed, and structuring data storages also is understood well. Our previous publications [17] and [23] cover such related works; the MAP website also gives reading recommendations[1].

Structuring data exchanges without breaking information hiding remains hard; no single solution exists. According to Helland [4], "data on the outside" differs from "data on the inside" significantly. Data access/usage profiles drive many data modeling decisions, both for data on the inside and for data in the outside. However, inside and outside data have diverging mutability, lifetime, accuracy, consistency and protection needs.

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

2 Microservice API Patterns (MAP) Scope and Organization

Microservice API Patterns (MAP) takes a broad view on microservice API design and evolution, from the perspective of data on the outside, i.e., the message representations and payloads exchanged when APIs are called (as shown in Figure 1). These messages are structured as *representation elements* which differ in their meaning and structure as API endpoints and their operations have different architectural roles and responsibilities. Critical design choices about the message structure and semantics strongly influence the design time and runtime qualities of an API and its underlying microservices implementations. Many options exist, with very different characteristics. The API designs evolve over time.

2.1 Patterns as Knowledge Sharing Vehicles

Software patterns are proven knowledge sharing vehicles with a 25-year track record [6]. We decided for the pattern format to share API design advice because:

- Pattern names aim at forming a domain vocabulary, a *ubiquitous language* [3]. For instance, Hohpe's and Woolf's Enterprise Integration Patterns [7] have become the lingua franca of queue-based messaging; they are implemented in a number of frameworks and tools. Such ubiquitous language for API design is missing to date.
- The forces and consequences sections of patterns support informed decision making, for instance about desired and achievable quality characteristics (but also downsides of certain designs). The design challenges and trade-offs identified in Section 1 frame and support such design discussions.
- Patterns are soft around their edges: they only sketch solutions and do not provide blueprints to be followed blindly.
- Patterns are not invented, but mined from practical experience and then curated and hardened via peer feedback.

2.2 Knowledge Categories

MAP addresses the following questions, which also define pattern categories:

- The *structure* of messages and the message elements that play critical roles in the design of APIs. What is an adequate number of representation elements for request and response messages? How are these elements structured? How can they be grouped and annotated with supplemental usage information (metadata)? [23]
- The impact of message content on the *quality* of the API. How can an API provider achieve a certain level of quality of the offered API, while at the same time using its available resources in a cost-effective way? How can the quality trade-offs be communicated and accounted for? [17]
- The roles and *responsibilities* [20] of API operations. Which is the architectural role played by each API endpoint and its operations? How do these roles and their responsibilities impact microservice size and granularity?
- API descriptions as a means for API governance and *evolution* over time. How to deal with life cycle management concerns such as support periods and versioning? How to promote backward compatibility and extensibility? How to communicate breaking changes? [11]

Two more categories complete the language scope, *foundation* and *identification* (not covered here due to space constraints). See Figure 2 for an overview.

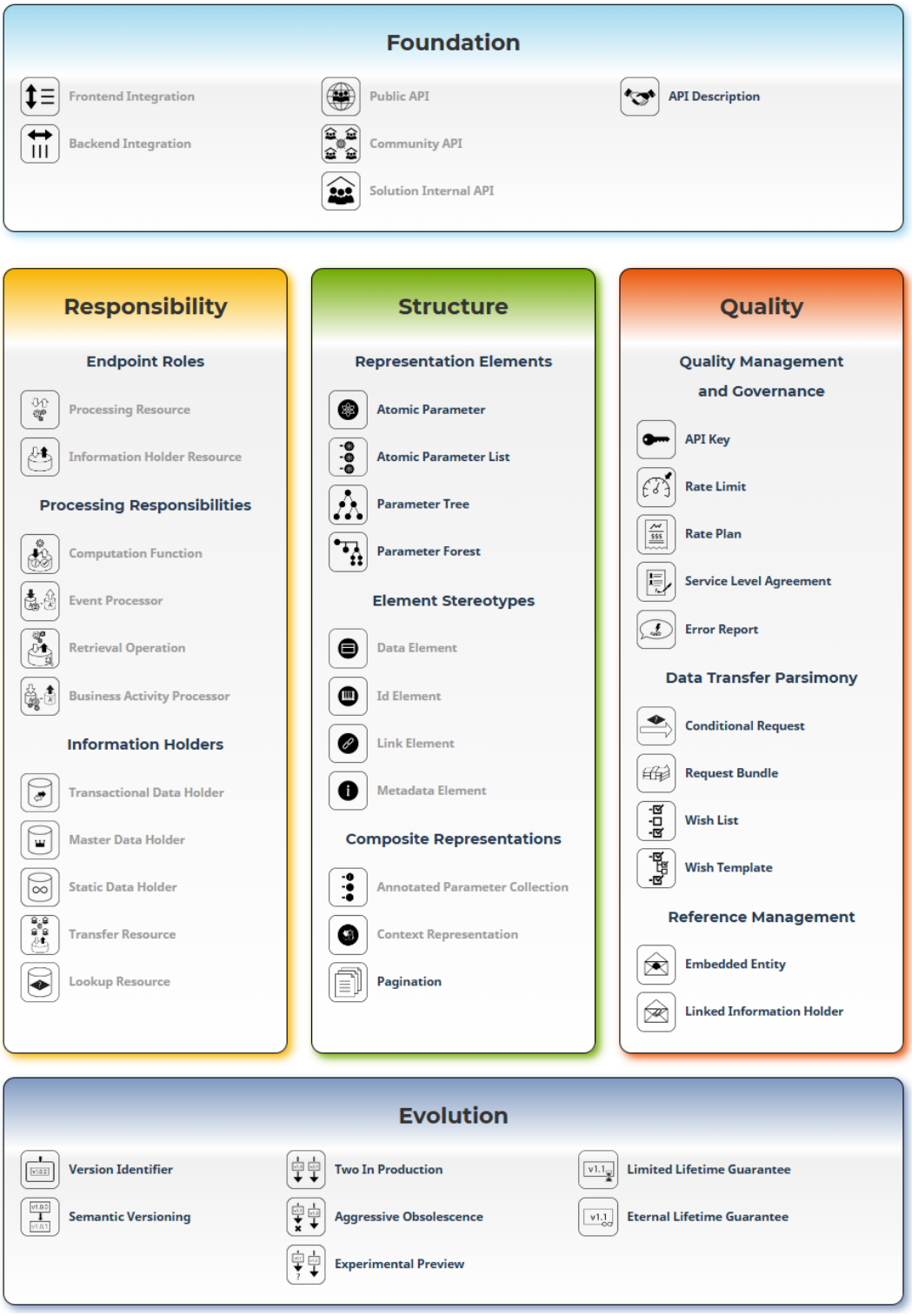

Figure 2 The MAP language is organized into categories, three of which have subcategories. Patterns set in bold/black are already available online at the time of writing; the grayed out ones are currently being mined. The identification category is not available yet. Visit www.microservice-api-patterns.org for an interactive, up-to-date version of this pattern index.

3 Pattern Examples: In-/Excluding Nested Data Representations

In this section, we introduce two patterns from the quality category not featured in peer-reviewed publications yet, *Embedded Entity* and *Linked Information Holder*. They provide two alternatives for representing related data elements: inclusion (nesting) and linkage (referencing).

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

References to other patterns are formatted like this in this paper: *Pattern Name*.

3.1 Pattern: *Embedded Entity*

a.k.a. Inlined Entity Data; Embedded Document (Nesting)

3.1.1 Context

The information required by a communication participant contains structured data. This data includes multiple elements that relate to each other in certain ways. For instance, a master data element such as a customer profile may *contain* other elements providing contact information including addresses and phone numbers, or a periodic business report may *aggregate* source information such as monthly sales figures summarizing individual business transactions. API clients work with several of the related information elements when processing response messages or producing request messages.[2]

3.1.2 Problem

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

3.1.3 Forces

When deciding for or against this pattern, you have to consider its impact on:

- Performance and scalability
- Flexibility and modifiability
- Data quality
- Data freshness and consistency

[2] Note that this is (almost) the same context as in the sibling pattern *Linked Information Holder*.

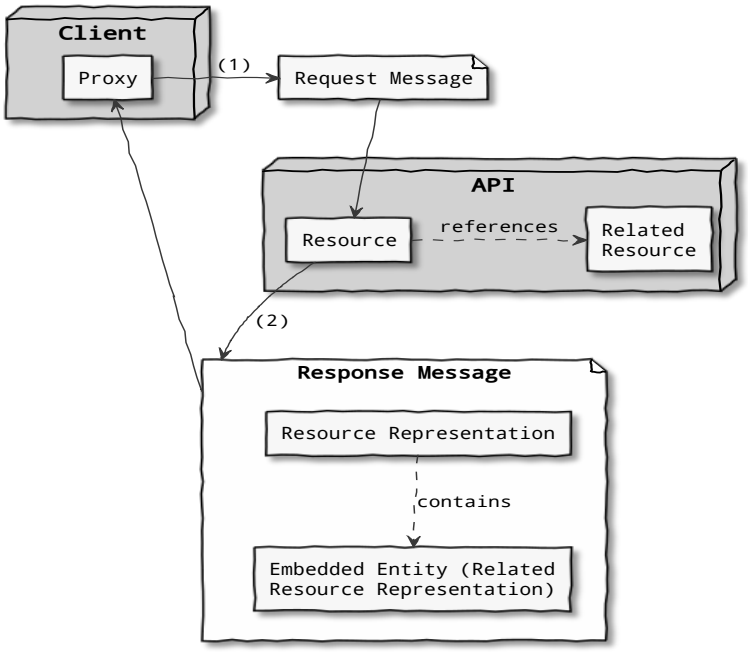

Figure 3 Sketch of Embedded Entity pattern (entities are represented as HTTP resources).

Traversing all relationships between information elements to include all possibly interesting data may require complex message representations and lead to large message sizes. It is unlikely and/or difficult to ensure that all recipients will require the same message content.

3.1.3.1 Non-solution

One could simply define one API endpoint per information element. This endpoint is accessed whenever API clients process data from that information element, e.g., when it is referenced from another one. But if API clients use such data in many situations, this solution leads to many subsequent requests to follow the references. This could possibly make it necessary to coordinate request execution and introduce conversation state, which harms scalability and availability; distributed data also is more difficult to keep consistent than local data.

3.1.4 Solution

For any relationship that the client has to follow, embed an *Entity Element*[3] in the message that contains the data of the target entity (instead of linking to the target entity). For instance, if a purchase order has a relation to product master data, let the purchase order message hold a copy of all relevant information stored in the product master data. Figure 3 shows a solution sketch of *Embedded Entity*.

[3] All patterns that are already published, but not contained in this paper can be found online: `https://microservice-api-patterns.org/`.

3.1.4.1 How it works

Define a *Parameter Tree*[4] or an *Atomic Parameter List* that includes an *Entity Element* for the referenced relationship. Provide an additional *Metadata Element* to denote the relationship type if needed.

Analyze outgoing relationships in the *Entity Element* and consider embedding them in the message as well, but only if this additional data is also used by the API client in enough cases. Repeat this analysis up to reaching the "transitive closure" where all reachable entities have either been included or excluded.

Review each source-target relationship carefully: is the target entity really needed on the API client side in enough cases? A "yes" answer warrants transmitting relationship information as *Embedded Entities*; otherwise transmitting references to *Linked Information Holders* might be sufficient.

Document the existence and the meaning of the embedded entity relationships in the *API Description*.

3.1.4.2 Example

Lakeside Mutual[5], a microservices sample application, contains a service called `Customer Core` that aggregates several information items (here: entities and value objects from Domain-Driven Design) in its operation signatures. An API client can access this data via an HTTP resource API. This API contains several instances of the pattern. Applying the *Embedded Entity* pattern, a response message might look as follows:

```
GET http://localhost:8080/customers/a51a-433f-979b-24e8f0
```

```
{
  "customer": {
    "id": "a51a-433f-979b-24e8f0"
  },
  "customerProfile": {
    "firstname": "Robbie",
    "lastname": "Davenhall",
    "birthday": "1961-08-11T23:00:00.000+0000",
    "currentAddress": {
      "streetAddress": "1 Dunning Trail",
      "postalCode": "9511",
      "city": "Banga"
    },
    "email": "rdavenhall0@example.com",
    "phoneNumber": "491 103 8336",
    "moveHistory": [{
      "streetAddress": "15 Briar Crest Center",
      "postalCode": "",
      "city": "Aeteke"
    }]
  },
```

[4] See `https://microservice-api-patterns.org/`.
[5] `https://github.com/Microservice-API-Patterns/LakesideMutual`

```
  "customerInteractionLog": {
    "contactHistory": [],
    "classification": "??"
  }
}
```

The referenced information items are all fully contained in the response message (e.g., `customerProfile`, `customerInteractionLog`); no URIs (links) to other resources appear. Note that `customerProfile` actually embeds nested data (`currentAddress`, `moveHistory`), while the `customerInteractionLog` is empty in this exemplary data set.

3.1.4.3 Implementation hints

When embedding entity relationships in message representations, keep in mind to:

- Document data characteristics such as owner, provenance, lifetime, and last update in the *API Description*; consider to introduce corresponding *Metadata Elements* if the data is used by a sufficient amount of clients requiring additional explanations.
- Distinguish transactional data from master data and other reference data when embedding it (to account for their different life cycles, evolution roadmaps and validity timeframes).
- Secure the message so that the content part with the highest protection need is covered adequately; this might (or might not) be the *Embedded Entity*. If the security requirements of link source and target differ substantially, consider switching to the sibling pattern *Linked Information Holder*.
- Be careful with consumer-side caching and replicating parts or all of the embedded data as this may introduce consistency, concurrency, and/or data ownership issues, especially when mixing transactional data with master data in one message.
- Test the use of *Embedded Entities* with all valid and invalid cardinalities. More specifically, empty, one, few, or many referenced data items should appear in different test cases.
- Monitor message sizes at runtime to prepare for interface refactoring such as switching to *Linked Information Holder* (see discussion below) or introducing *Pagination*.
- Define compatibility rules and service evolution policies [11] when introducing *Embedded Entites*. The more related entities a message includes and the more complex its payload is, the more likely it is to change as a whole and in parts. As a client, behave as a *Tolerant Reader* [2]: Clients should not assume that all related entities will always be included and might have to be ready to follow a link in case the information is not embedded.

3.1.5 Consequences

The pattern meets the "all in one" requirement articulated by the problem statement, but this may lead to large messages that are expensive to transfer. If some clients do not have to receive all the data, then parts of the payload could have been omitted.

3.1.5.1 Resolution of forces

+ An *Embedded Entity* reduces the number of calls required: If the required information is included, the client does not have to create a request to obtain it.
+ Embedding entities can lead to a reduction in the number of endpoints, because no dedicated endpoint to retrieve some information is required.

– Embedding entities leads to larger response messages which take longer to transfer.

– It can be difficult to anticipate what information different clients require to perform their tasks. As a result, there is a tendency to include more data than needed by (most) clients in an *Embedded Entity*, which leads to yet larger message sizes. Such design can be found in many *Public APIs* serving large and possibly unknown clients.
– Large messages that contain unused data consume more bandwidth than necessary. However, if most or all of the data is actually used, sending many small messages might actually require more bandwidth than sending one large message (e.g., for header and metadata sent with the smaller messages multiple times).
– If the embedded entities change with different speed (e.g., a fast-changing transactional entity refers to immutable master data), retransmitting all entities causes unnecessary overhead as messages with partially changed content cannot be cached. Consider switching to a *Linked Information Holder* (and maybe additionally apply the *Conditional Request* pattern for the linked entity).
– Once included and exposed in an *API Description*, it is hard to remove an *Embedded Entity* in a backward-compatible manner (as clients may have begun to rely on it).

3.1.5.2 Alternatives

If reducing message size is your main design goal, you can also define a *Wish List* or, even more expressive, a *Wish Template* to minimize the data to be transferred by letting consumers dynamically describe which subset of the data they need.

API Gateways[6] can also help when dealing with different information needs. They can either provide two alternative APIs that use the same backend interface, and/or collect and aggregate information from different endpoints and operations (which makes them stateful).

3.1.6 Known Uses

Many public APIs with complex response messages use the *Embedded Entity* pattern:

- When retrieving an issue with the GitHub v3 API[7], the response also contains the full information about the milestone the issue is assigned to.
- A tweet in the Twitter REST API[8] contains the entire user information, including for example the number of followers the user has.
- Many operations in the Microsoft Graph API apply this pattern. For instance, the user resource representations[9] contain structured attributes that represent (sub-)entities (but also link to other resources via *Linked Information Holders*). For instance, the response body of `List events` contains an array of `attendees` that are identified by their email addresses, but also have a `type` and a `status`.

Plenty of APIs offered by custom enterprise information systems and master data management products also realize the pattern.

---

[6] `https://microservices.io/patterns/apigateway.html`
[7] `https://developer.github.com/v3/issues/#get-a-single-issue`
[8] `https://developer.twitter.com/en/docs/tweets/post-and-engage/api-reference/get-statuses-show-id`
[9] `https://developer.microsoft.com/en-us/graph/docs/api-reference/v1.0/resources/user`

3.1.7 More Information

3.1.7.1 Related Patterns

Linked Information Holder describes a complementary solution for the reference management problem and can also be seen as an alternative (as explained above).

Wish List or *Wish Template* can help to fine-tune the content in an *Embedded Entity*, as explained above.

3.1.7.2 Other Sources

See Section 7.5 in [18] for additional advice and examples ("Embedded Document (Nesting)").

3.2 Pattern: *Linked Information Holder*

a.k.a. Linked Entity, Data Reference; Compound Document (Sideloading)

3.2.1 Context

An API exposes structured data to meet the information needs of the communication participants. This data contains elements that relate to each other (e.g., a master information element may *contain* other elements providing detailed information or a performance report for a period of time may *aggregate* raw data such as individual measurements). API clients want to work with several of the related information elements when processing response messages or producing request messages.[10]

3.2.2 Problem

When exposing structured, possibly deeply nested information elements in an API, how can you avoid sending large messages containing lots of data that is not always useful for the message receiver in its entirety?

3.2.3 Forces

A general rule of thumb in distributed systems is that request and response messages should not be too large so that the network and the endpoint processing resources are not over-utilized. That said, message recipients possibly would like to follow many or all of the relationships to access information elements related to the elements requested. If the related elements are not included, information about their location and their content is required, as well as access information. This information set has to be designed, implemented and evolved; the resulting dependency has to be managed.

The sibling pattern *Embedded Entity* list additional forces that apply to both patterns.

[10] This context is (almost) the same as that of the sibling pattern *Embedded Entity*.

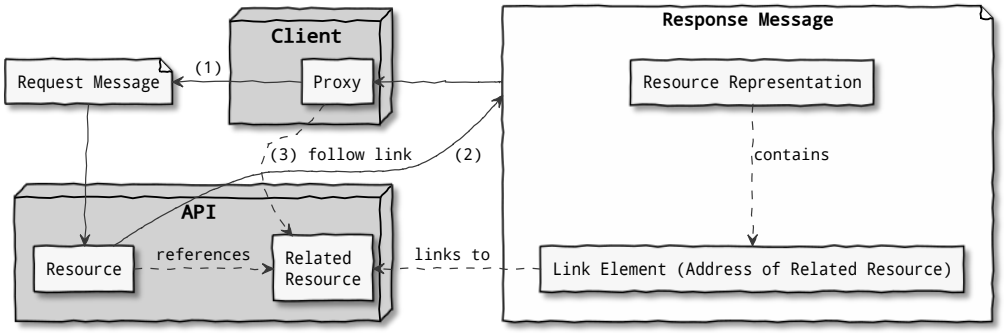

Figure 4 Sketch of Linked Information Holder pattern.

3.2.3.1 Non-solution

One option is to always (transitively) include all the related information elements of each transmitted element in request and response messages throughout the API (as described in the *Embedded Entity* pattern). However, this approach can harm performance of individual calls and lead to large, wasteful messages containing data not required by some clients.

3.2.4 Solution

Add a *Link Element*[11] to the message that references an API endpoint. Introduce an API endpoint that represents the linked entity, for instance, an *Information Holder Resource* for the referenced information element. This element might be an entity from the domain model[12] that is exposed by the API. Figure 4 outlines this solution.

3.2.4.1 How it works

Include the location information (i.e., host and port), expressed in the logical naming and/or addressing scheme of the API, when referencing the endpoint via *Link Elements* in request and response messages. This typically requires a *Parameter Tree* to be used in the representation structure; in simple cases, an *Atomic Parameter List* might suffice.

Identify the *Link Element* with a name. If more information about the relation should be sent to clients, annotate this *Link Element* with details about the corresponding relationship, for instance, a *Metadata Element* specifying the type and characteristics of the relationship. Clients and providers must agree on the semantics (i.e., meaning) of the link relationships, and be aware of coupling and side effects introduced.

Document the existence and the meaning of the *Linked Information Holder* in the *API Description*. Specify the cardinalities on both ends of the relation. One-to-many relationships can be modeled as collections, for instance by transmitting multiple elements as *Atomic Parameter Lists*. Many-to-many relationships can be modeled as two such one-to-many relationships, one linking the source entities to the targets, and one linking the target entities to the sources. Such design may require the introduction of an additional API endpoint representing the relation rather than its source and target.

[11] See `https://microservice-api-patterns.org/`.
[12] `https://en.wikipedia.org/wiki/Domain_model`

3.2.4.2 Example

A sample application for `Customer Management` could work with a `Customer Core` service API that aggregates several information elements from the domain model of the application, in the form of entities and value objects from Domain-Driven Design (DDD). Its API client could access this data through a `Customer Information Holder`, implemented as a REST controller in Spring Boot.

If the `Customer Information Holder` implements the *Linked Information Holder* pattern for the `customerProfile`, a response message looks as this:

```
GET http://localhost:8080/customers/a51a-433f-979b-24e8f0

{
  "customer": {
    "id": "a51a-433f-979b-24e8f0"
  },
  "links": [{
    "rel": "customerProfile",
    "href": "http://localhost:8080/customers/a51a-433f-979b-24e8f0/profile"
  }, {
    "rel": "moveHistory",
    "href": "http://localhost:8080/customers/a51a-433f-979b-24e8f0/moveHistory"
  }],
  "email": "rdavenhall0@example.com",
  "phoneNumber": "491 103 8336",
  "customerInteractionLog": {
    "contactHistory": [],
    "classification": "??"
  }
}
```

The `customerProfile` can then be retrieved by a subsequent GET request to the provided URI link. The `moveHistory` has been factored out as well, so the pattern is applied twice in this example.

3.2.4.3 Implementation hints

When adding links to message representations to turn relationship targets into API endpoints, it is good practice to:

- Keep the naming scheme and structure of the *Link Elements* consistent and be reluctant to change it. For instance, keep the URI naming scheme consistent in HTTP resource APIs.
- Define *compliance controls* if the link relations are subject to system and process assurance audits as discussed in [8].[13]
- Run regression tests on the source of a link when the link destination changes its interface or implementation.

[13] An example of such a control is a pre- and postcondition check at the API endpoint boundary that enforces the correct cardinality of a link from a purchase order to customer (e.g., there has to be one and only one customer per order). Such design-by-contract approaches can be implemented as a foreign key relationship if both order and customer are stored in the same relational database. In a distributed services architecture, both sides of the relationship can modify the data independently of each other, which might introduce inconsistencies.

- Monitor performance continuously to preserve and challenge the rationale for pattern usage. If most or all client calls follow the given link, consider embedding the target element in the original representation to reduce traffic (see *Embedded Entity* pattern).
- Adhere to REST constraints and related recommended practices when using RESTful HTTP (see [1]): Linked reference data is a cornerstone of the Hypertext as the Engine of Application State (HATEOAS) tenet that is required to reach REST maturity level 3[14].

3.2.5 Consequences

This pattern is often applied when referencing rich information holders serving multiple usage scenarios: not all message recipients require the full set of referenced data, for instance, when *Master Data Holders* such as customer profiles or a product records are referenced. Following the link, message recipients can obtain the required subsets on demand.

When introducing link elements into message representations, an implicit promise is made to the recipient that these links can be followed successfully; the provider might not be willing to keep such promise infinitely. Even if a long lifetime of the linked endpoint is guaranteed, links still may break; for instance, when the data organization or location changes. Clients should expect this and be able to follow redirects or referrals to the updated links. To minimize breaking links on the provider side, the provider should invest in maintaining link consistency. For instance, a *Lookup Resource* can be used to solve this problem.

3.2.5.1 Resolution of forces

+ Linking instead of embedding entities results in smaller messages and uses less resources in the communications infrastructure when exchanging individual messages. This needs to be contrasted to the possible higher resource use due to transfer of multiple messages as links get followed.
+ If some linked data changes often, only that data needs to be requested again.

– Additional requests are required to dereference the linked information.
– Linking instead of embedding entities might result in the use of more resources in the communications infrastructure as multiple messages are required to follow the links.
– Additional *Information Holder Resource* endpoints have to be provided for the linked entities, causing development and operations effort and cost.

3.2.5.2 Alternatives

The patterns in the *Quality Patterns* category that help reduce the amount of data exchanged can be used alternatively. For instance, *Conditional Request*, *Wish List*, and *Wish Template* are eligible; the structure pattern *Pagination* is an option too.

3.2.6 Known Uses

Many public Web APIs apply this pattern, for instance the JIRA Cloud REST API[15] when reporting the links between issues in the Get issue link[16] call.

[14] `https://martinfowler.com/articles/richardsonMaturityModel.html`
[15] `https://docs.atlassian.com/jira/REST/cloud/`
[16] `https://developer.atlassian.com/cloud/jira/platform/rest/#api-api-2-issueLink-linkId-get`

Certain calls in the Microsoft Graph API also apply this pattern: for instance, user resource representations[17] contain scalar and complex attributes as "Properties", but also link to other resources such as Calendar (under "Relationships").

RESTful HTTP APIs on maturity level 3 apply this pattern if the links representing application state transfer deal with both master data and transactional data resources and their representations. An example is Spring Restbucks[18].

3.2.7 More Information

3.2.7.1 Related Patterns

The sibling pattern *Embedded Entity* provides an alternative to *Linked Information Holder*, transmitting related data rather than referencing it.

Linked Information Holders typically reference *Information Holder Resources*. The referenced *Information Holder Resources* can be combined with *Lookup Resource* to cope with potentially broken links, as outlined previously.

Linked Service[19] is a similar pattern in [2], but less focused on data. "Web Service Patterns" [12] has a *Partial DTO Population* pattern which solves a similar problem.

3.2.7.2 Other Sources

See Section 7.4 in [18] for additional advice and examples, to be found under "Compound Document (Sideloading)".

The Backup, Availability, Consistency (BAC) theorem investigates data management issues further [14].

4 Conclusion

This paper introduced Microservice API Patterns (MAP), a volunteer project compiling a pattern language for the design and evolution of Microservice APIs. The language is organized into six pattern categories at present; the MAP website[20] provides additional navigation aids such as a cheat sheet and pattern filtering by scope, phase, role, and quality attributes. The patterns, their known uses and the examples have been mined from public Web APIs as well as application development and software integration projects the authors and their industry partners have been involved in [21]. We previously published 18 patterns at pattern conferences; this paper introduced two more.

In our future work, we plan to fill gaps throughout our six pattern categories. The next patterns will describe additional structural representations as well as the architectural roles and responsibilities of endpoints and operations within an API. Patterns capturing API endpoint and service identification strategies and tactics as well as corresponding artifacts yet have to be mined: Context Maps, Bounded Contexts and Aggregates from Domain-Driven Design (DDD) [3] seem to be particularly promising starting points for microservice API design, and tools for strategic DDD are beginning to emerge [9]. We have also begun to work on a technology-independent service contract language that incorporates our patterns as first-class language elements, as well as tools to create API specifications from DDD context maps, existing code, and as other specification languages such as Open API Specification. Other future tools may search for pattern instances and provide metrics.

[17] https://developer.microsoft.com/en-us/graph/docs/api-reference/v1.0/resources/user
[18] https://github.com/odrotbohm/spring-restbucks
[19] http://www.servicedesignpatterns.com/ClientServiceInteractions/LinkedService
[20] https://microservice-api-patterns.org/

References

1 Subbu Allamaraju. *RESTful Web Services Cookbook*. O'Reilly, 2010.

2 Robert Daigneau. *Service Design Patterns: Fundamental Design Solutions for SOAP/WSDL and RESTful Web Services*. Addison-Wesley Professional, 2011. URL: `http://www.servicedesignpatterns.com/`.

3 Eric Evans. *Domain-Driven Design: Tacking Complexity In the Heart of Software*. Addison-Wesley, 2003.

4 Pat Helland. 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*, pages 144–153, 2005. URL: `http://cidrdb.org/cidr2005/papers/P12.pdf`.

5 Gregor Hohpe. SOA patterns: New insights or recycled knowledge? `https://www.enterpriseintegrationpatterns.com/docs/HohpeSOAPatterns.pdf`, 2007. URL: `https://www.enterpriseintegrationpatterns.com/docs/HohpeSOAPatterns.pdf`.

6 Gregor Hohpe, Rebecca Wirfs-Brock, Joseph W. Yoder, and Olaf Zimmermann. Twenty Years of Patterns' Impact. *IEEE Software*, 30(6):88, 2013. `doi:10.1109/MS.2013.135`.

7 Gregor Hohpe and Bobby Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley, 2003.

8 Klaus Julisch, Christophe Suter, Thomas Woitalla, and Olaf Zimmermann. Compliance by design–Bridging the chasm between auditors and IT architects. *Computers & Security*, 30(6):410–426, 2011.

9 Stefan Kapferer. A Domain-specific Language for Service Decomposition. Term project, University of Applied Sciences of Eastern Switzerland (HSR FHO), 2018. URL: `https://eprints.hsr.ch/722`.

10 James Lewis and Martin Fowler. Microservices: a definition of this new architectural term. `https://martinfowler.com/articles/microservices.html/`, 2014. URL: `https://martinfowler.com/articles/microservices.html/`.

11 Daniel Lübke, Olaf Zimmermann, Mirko Stocker, Cesare Pautasso, and Uwe Zdun. 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, 2019.

12 Paul B. Monday. *Web Services Patterns: Java Edition*. Apress, Berkely, CA, USA, 2003.

13 Sam Newman. *Building Microservices: Designing Fine-Grained Systems*. O'Reilly, 2015.

14 Guy Pardon, Cesare Pautasso, and Olaf Zimmermann. Consistent Disaster Recovery for Microservices: the BAC Theorem. *IEEE Cloud Computing*, 5(1):49–59, December 2018. `doi:10.1109/MCC.2018.011791714`.

15 Cesare Pautasso and Olaf Zimmermann. The Web as a Software Connector: Integration Resting on Linked Resources. *IEEE Software*, 35:93–98, 2018. `doi:10.1109/MS.2017.4541049`.

16 Cesare Pautasso, Olaf Zimmermann, Mike Amundsen, James Lewis, and Nicolai M. Josuttis. Microservices in Practice, Part 1: Reality Check and Service Design. *IEEE Software*, 34(1):91–98, 2017. `doi:10.1109/MS.2017.24`.

17 Mirko Stocker, Olaf Zimmermann, Daniel Lübke, Uwe Zdun, and Cesare Pautasso. 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, 2018.

18 Phil Sturgeon. *Build APIs you won't hate*. LeanPub, https://leanpub.com/build-apis-you-wont-hate, 2016.

19 Vaughn Vernon. *Implementing Domain-Driven Design*. Addison-Wesley Professional, 2013.

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

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

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

23 Olaf Zimmermann, Mirko Stocker, Daniel Lübke, and Uwe Zdun. Interface Representation Patterns: Crafting and Consuming Message-Based Remote APIs. In *Proc. of the 22nd European Conference on Pattern Languages of Programs*, EuroPLoP '17, pages 27:1–27:36. ACM, 2017. doi:10.1145/3147704.3147734.