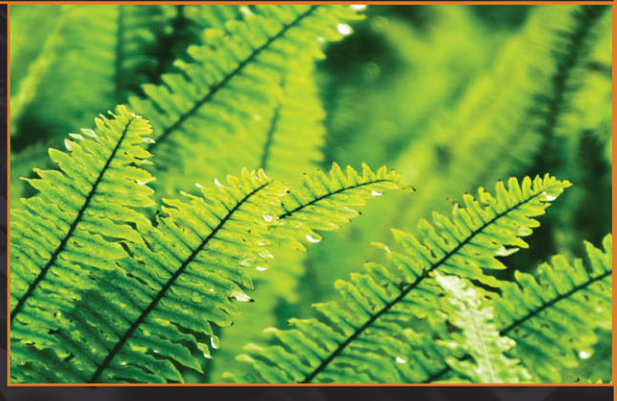
The Addison-Wesley Signature Series

A Vaughn Vernon Signature Book

Patterns for API Design

Simplifying Integration with Loosely Coupled Message Exchanges

Olaf Zimmermann
Mirko Stocker
Daniel Lübke
Uwe Zdun
Cesare Pautasso



Foreword by
Frank Leymann

FREE SAMPLE CHAPTER

"APIs are eating the world. Organizations and collaborations are depending more and more on APIs. For all these APIs to be designed, using patterns is a well-established way of tackling design challenges. *Patterns for API Design* helps practitioners to design their APIs more effectively: They can focus on designing their application domain while standard design issues are solved with patterns. If you're working in the API space, this book will change how you design APIs and how you look at APIs."

—Erik Wilde, Catalyst at Axway

"The authors have captured design patterns across the API lifecycle, from definition to design, in an approachable way. Whether you have designed dozens of web APIs or you are just starting out, this book is a valuable resource to drive consistency and overcome any design challenge you may face. I highly recommend this book!"

—James Higginbotham
Author of Principles of Web API Design: Delivering value with APIs and Microservices *and Executive API Consultant, LaunchAny*

"APIs are everywhere in today's software development landscape. API design looks easy but, as anyone who has suffered a poorly designed API will attest, it is a difficult skill to master and much subtler and more complex than it initially appears. In this book, the authors have used their long experience and years of research work to create a structured body of knowledge about API design. It will help you to understand the underlying concepts needed to create great APIs and provides a practical set of patterns that you can use when creating your own APIs. It is recommended for anyone involved in the design, building, or testing of modern software systems."

—Eoin Woods, CTO, Endava

Application programming interfaces (API) are among the top priority elements to help manage many of the trade-offs involved in system design, in particular distributed systems, which increasingly dominate our software ecosystem. In my experience, this book removes the complexities in understanding and designing APIs with concepts accessible to both practicing engineers and those who are just starting their software engineering and architecting journey. All who aspire to play a key role in system design should understand the API design concepts and patterns presented in this book."

—Ipek Ozkaya
Technical Director, Engineering Intelligence Software System
Software Solutions Division
Carnegie Mellon University Software Engineering Institute
Editor-in-Chief 2019–2023 IEEE Software Magazine

"It is my belief that we are entering into an era where API-first design will become the dominant form of design in large, complex systems. For this reason, *Patterns for API Design* is perfectly timed and should be considered essential reading for any architect."

—Rick Kazman, University of Hawaii

"Finally, the important topic of API design is addressed systematically! I wish I would have had this great pattern collection a few years earlier."

—Dr. Gernot Starke, INNOQ Fellow

"I observed software projects fail because middleware technology hid a system's distributed nature from programmers. They designed problematic APIs of a non-distributed gestalt exercised remotely. This book embraces the required dispersal of software in an interdependent world and provides timeless advice on designing interfaces between its separated parts. The Patterns guide beyond specific middleware technology and will not only help with creation and understanding but also with necessary evolution of the interconnected software systems we grow today and in the future. Those systems not only span the globe for international business, but also work within our cars, houses, and almost any technology our daily lives depend on."

—Peter Sommerlad, independent consultant, author of Pattern-Oriented Software Architecture: A System of Patterns *and* Security Patterns

"The book *Patterns for API Design* is the Swiss army knife for software engineers and architects when it comes to designing, evolving, and documenting APIs. What I particularly like about the book is that it does not just throw the patterns at the reader; instead, the authors use realistic examples, provide hands-on architecture decision support, and exemplify patterns and decisions using a case study. As a result, their pattern language is very accessible. You can use the book to find solutions for specific problems or browse entire chapters to get an overview of the problem and solution spaces related to API design. All patterns are well-crafted, well-named, and peer-reviewed by the practitioner community. It's a joy."

—Dr. Uwe van Heesch, Practicing Software Architect and Former Vice President Hillside Europe

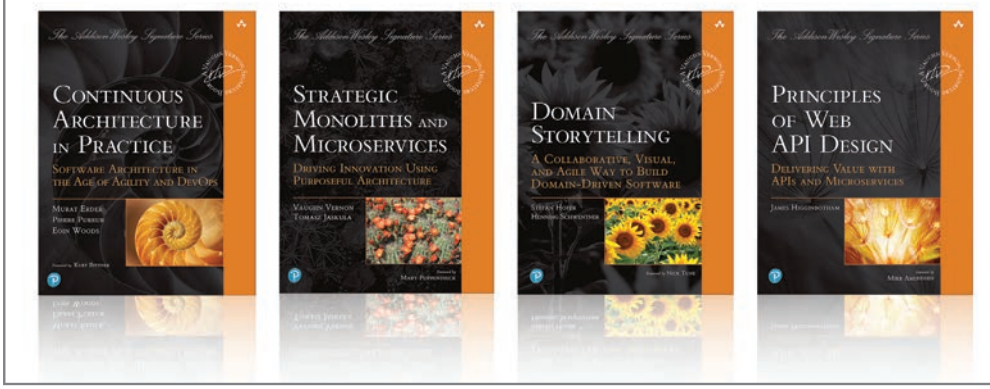
"This comprehensive collection of API patterns is an invaluable resource for software engineers and architects designing interoperable software systems. The introduction into API fundamentals and numerous case study examples make it excellent teaching material for future software engineers. Many of the patterns discussed in this book are extremely useful in practice and were applied to design the APIs of integrated, mission-critical rail operations centre systems."

—Andrei Furda, Senior Software Engineer at Hitachi Rail STS Australia

This page intentionally left blank

Patterns for API Design

Pearson Addison-Wesley Signature Series



Visit informit.com/awss/vernon for a complete list of available publications.

The **Pearson Addison-Wesley Signature Series** provides readers with practical and authoritative information on the latest trends in modern technology for computer professionals. The series is based on one simple premise: great books come from great authors.

Vaughn Vernon is a champion of simplifying software architecture and development, with an emphasis on reactive methods. He has a unique ability to teach and lead with Domain-Driven Design using lightweight tools to unveil unimagined value. He helps organizations achieve competitive advantages using enduring tools such as architectures, patterns, and approaches, and through partnerships between business stakeholders and software developers.

Vaughn's Signature Series guides readers toward advances in software development maturity and greater success with business-centric practices. The series emphasizes organic refinement with a variety of approaches—reactive, object, and functional architecture and programming; domain modeling; right-sized services; patterns; and APIs—and covers best uses of the associated underlying technologies.

Patterns for API Design

Simplifying Integration with Loosely Coupled Message Exchanges

Olaf Zimmermann
Mirko Stocker
Daniel Lübke
Uwe Zdun
Cesare Pautasso

Addison-Wesley

Boston • Columbus • New York • San Francisco • Amsterdam • Cape Town
Dubai • London • Madrid • Milan • Munich • Paris • Montreal • Toronto • Delhi • Mexico City
São Paulo • Sydney • Hong Kong • Seoul • Singapore • Taipei • Tokyo

Cover image: Joshua Small-Photographer / Shutterstock

Visit us on the Web: informit.com

Library of Congress Control Number: 2022947404

ISBN-13: 978-0-13-767010-9
ISBN-10: 0-13-767010-9

ScoutAutomatedPrintCode

Pearson's Commitment to Diversity, Equity, and Inclusion

Pearson is dedicated to creating bias-free content that reflects the diversity of all learners. We embrace the many dimensions of diversity, including but not limited to race, ethnicity, gender, socioeconomic status, ability, age, sexual orientation, and religious or political beliefs.

Education is a powerful force for equity and change in our world. It has the potential to deliver opportunities that improve lives and enable economic mobility. As we work with authors to create content for every product and service, we acknowledge our responsibility to demonstrate inclusivity and incorporate diverse scholarship so that everyone can achieve their potential through learning. As the world's leading learning company, we have a duty to help drive change and live up to our purpose to help more people create a better life for themselves and to create a better world.

Our ambition is to purposefully contribute to a world where:

- Everyone has an equitable and lifelong opportunity to succeed through learning.
- Our educational products and services are inclusive and represent the rich diversity of learners.
- Our educational content accurately reflects the histories and experiences of the learners we serve.
- Our educational content prompts deeper discussions with learners and motivates them to expand their own learning (and worldview).

While we work hard to present unbiased content, we want to hear from you about any concerns or needs with this Pearson product so that we can investigate and address them.

- Please contact us with concerns about any potential bias at https://www.pearson.com/report-bias.html.

This page intentionally left blank

Contents

Foreword by Vaughn Vernon, Series Editor

My signature series emphasizes organic growth and refinement, which I describe in more detail below. It only makes sense to start off by describing the organic communication that I experienced with the first author of this book, Professor Dr. Olaf Zimmermann.

As I often refer to Conway's Law of system design, communication is a critical factor in software development. Systems designs not only resemble the communication structures of the designers; the structure and assembling of individuals as communicators is just as important. It can lead from interesting conversations to stimulating thoughts and continue to deliver innovative products. Olaf and I met at a Java User Group meeting in Bern, Switzerland, in November 2019. I gave a talk on reactive architecture and programming and how it is used with Domain-Driven Design. Afterward, Olaf introduced himself. I also met his graduate student and later colleague, Stefan Kapferer. Together they had organically designed and built the open-source product Context Mapper (a domain-specific language and tools for Domain-Driven Design). Our chance meeting ultimately led to this book's publication. I'll tell more of this story after I describe the motivation and purpose of my book series.

My Signature Series is designed and curated to guide readers toward advances in software development maturity and greater success with business-centric practices. The series emphasizes organic refinement with a variety of approaches—reactive, object, as well as functional architecture and programming; domain modeling; right-sized services; patterns; and APIs—and covers best uses of the associated underlying technologies.

From here, I am focusing now on only two words: *organic refinement*.

The first word, *organic*, stood out to me recently when a friend and colleague used it to describe software architecture. I have heard and used the word *organic* in connection with software development, but I didn't think about that word as carefully as I did then when I personally consumed the two used together: *organic architecture*.

Think about the word *organic*, and even the word *organism*. For the most part, these are used when referring to living things, but they are also used to describe inanimate things that feature some characteristics that resemble life forms. *Organic*

originates in Greek. Its etymology is with reference to a functioning organ of the body. If you read the etymology of *organ*, it has a broader use, and in fact organic followed suit: body organs; to implement; describes a tool for making or doing; a musical instrument.

We can readily think of numerous organic objects—living organisms—from the very large to the microscopic single-celled life forms. With the second use of organism, though, examples may not as readily pop into our mind. One example is an organization, which includes the prefix of both *organic* and *organism*. In this use of *organism*, I'm describing something that is structured with bidirectional dependencies. An organization is an organism because it has organized parts. This kind of organism cannot survive without the parts, and the parts cannot survive without the organism.

Taking that perspective, we can continue applying this thinking to nonliving things because they exhibit characteristics of living organisms. Consider the atom. Every single atom is a system unto itself, and all living things are composed of atoms. Yet, atoms are inorganic and do not reproduce. Even so, it's not difficult to think of atoms as living things in the sense that they are endlessly moving, functioning. Atoms even bond with other atoms. When this occurs, each atom is not only a single system unto itself, but becomes a subsystem along with other atoms as subsystems, with their combined behaviors yielding a greater whole system.

So then, all kinds of concepts regarding software are quite organic in that nonliving things are still "characterized" by aspects of living organisms. When we discuss software model concepts using concrete scenarios, or draw an architecture diagram, or write a unit test and its corresponding domain model unit, software starts to come alive. It isn't static because we continue to discuss how to make it better, subjecting it to refinement, where one scenario leads to another, and that has an impact on the architecture and the domain model. As we continue to iterate, the increasing value in refinements leads to incremental growth of the organism. As time progresses, so does the software. We wrangle with and tackle complexity through useful abstractions, and the software grows and changes shapes, all with the explicit purpose of making work better for real, living organisms at global scales.

Sadly, software organics tend to grow poorly more often than they grow well. Even if they start out life in good health, they tend to get diseases, become deformed, grow unnatural appendages, atrophy, and deteriorate. Worse still is that these symptoms are caused by efforts to refine the software that go wrong instead of making things better. The worst part is that with every failed refinement, everything that goes wrong with these complexly ill bodies doesn't cause their death. Oh, if they could just die! Instead, we have to kill them and killing them requires nerves, skills, and the intestinal fortitude of a dragon slayer. No, not one, but dozens of vigorous dragon slayers. Actually, make that dozens of dragon slayers who have really big brains.

That's where this series comes into play. I am curating a series designed to help you mature and reach greater success with a variety of approaches—reactive, object, and functional architecture and programming; domain modeling; right-sized services; patterns; and APIs. And along with that, the series covers best uses of the associated underlying technologies. It's not accomplished in one fell swoop. It requires organic refinement with purpose and skill. I and the other authors are here to help. To that end, we've delivered our very best to achieve our goal.

Now, back to my story. When Olaf and I first met, I offered for him and Stefan to attend my IDDD Workshop a few weeks later in Munich, Germany. Although neither were able to break away for all three days, they were open to attend the third and final day. My second offer was for Olaf and Stefan to use time after the workshop to demonstrate the Context Mapper tool. The workshop attendees were impressed, as was I. This led to further collaboration on into 2020. Little did any of us expect what that year would bring. Even so, Olaf and I were able to meet somewhat frequently to continue design discussions about Context Mapper. During one of these meetings, Olaf mentioned his work on API patterns that were provided openly. Olaf showed me a number of patterns and additional tooling he and others had built around them. I offered Olaf the opportunity to author in the series. The result is now in front of you.

I later met on a video call with Olaf and Daniel Lübke to kick off product development. I have not had the chance to spend time with the other authors—Mirko Stocker, Uwe Zdun, Cesare Pautasso—but I was assured of the team's quality given their credentials. Notably, Olaf and James Higginbotham collaborated to ensure the complementary outcome of this book and *Principles of Web API Design*, also in this series. As an overall result, I am very impressed with what these five have contributed to the industry literature. API design is a very important topic. The enthusiasm toward the book's announcement proves that it is right in the topic's sweet spot. I am confident that you will agree.

—Vaughn Vernon, series editor

This page intentionally left blank

Foreword by Frank Leymann

APIs are everywhere. The API economy enables innovation in technology areas, including cloud computing and the Internet of Things (IoT), and is also a key enabler of digitalization of many companies. There hardly is any enterprise application without external interfaces to integrate customers, suppliers, and other business partners; solution-internal interfaces decompose such applications into more manageable parts, such as loosely coupled microservices. Web-based APIs play a prominent role in these distributed settings but are not the only way to integrate remote parties: queue-based messaging channels as well as publish/subscribe-based channels are widely used for backend integration, exposing APIs to message producers and consumers. gRPC and GraphQL have gained a lot of momentum as well. Thus, best practices for designing "good" APIs are desirable. Ideally, API designs persist across technologies and survive when those change.

Patterns establish a vocabulary for a problem-solution domain, finding a balance between being abstract and concrete, which gives them both timelessness and relevance today. Take *Enterprise Integration Patterns* by Gregor Hohpe and Bobby Woolf from the Addison Wesley Signature Series as an example: I have been using it in teaching and industry assignments since my time as lead architect of the IBM MQ family of products. Messaging technologies come and, sometimes, go—but the messaging concepts such as Service Activator and Idempotent Receiver are here to stay. I have written cloud computing patterns, IoT patterns, quantum computing patterns, even patterns for patterns in digital humanities myself. And Martin Fowler's *Patterns of Enterprise Application Architecture,* also from the Addison Wesley Signature Series, gives us the Remote Façade and the Service Layer. Hence, many parts of the overall design space of distributed applications are covered well in this literature—but not all. Therefore, it is great to see that the API design space is now supported by patterns too, the request and response messages that travel between API client and API provider in particular.

The team who wrote this book is a great mix of architects and developers composed of deeply experienced industry professionals, leaders in the patterns community, and academic researchers and lecturers. I have been working with three of the authors of this book for many years and have been following their MAP project since its inception in 2016. They apply the pattern concept faithfully: Each pattern text follows a common template that takes us from a problem context, including design forces, to a conceptual solution. It also comes with a concrete example (often

RESTful HTTP). A critical discussion of pros and cons resolves the initial design forces and closes with pointers to related patterns. Many of the patterns went through shepherding and writers workshops at pattern conferences, which helped to incrementally and iteratively improve and harden them over several years, capturing collective knowledge as a result from this process.

This book provides multiple perspectives on the API design space, from scoping and architecture to message representation structure and quality attribute-driven design to API evolution. Its pattern language can be navigated via different paths, including project phases and structural elements such as API endpoint and operation. As in our *Cloud Computing Patterns* book, a graphical icon for each pattern conveys its essence. These icons serve as mnemonics and can be used to sketch APIs and their elements. The book takes a unique and novel step in providing decision models that collect recurring questions, options, and criteria regarding pattern applications. They provide stepwise, easy-to-follow design guidance without oversimplifying the complexities inherent to API design. A stepwise application to a sample case makes the models and their advices tangible.

In Part 2, the patterns reference, application and integration architects will find the coverage of endpoint roles such as Processing Resource and operation responsibilities such as State Transition Operation useful to size APIs adequately and make (cloud) deployment decisions. State matters, after all, and several patterns make state management behind the API curtain explicit. API developers will benefit from the careful consideration given to identifiers (in patterns such as API Key and Id Element), several options for response shaping (for instance, with Wish Lists and a Wish Template that abstracts from GraphQL), and pragmatic advice on how to expose metadata of different kinds.

I have not seen life-cycle management and versioning strategies captured in pattern form in other books so far. Here, we can learn about Limited Lifetime Guarantees and Two in Production, two patterns very common in enterprise applications. These evolution patterns will be appreciated by API product owners and maintainers.

In summary, this book provides a healthy mix of theory and practice, containing numerous nuggets of deep advice but never losing the big picture. Its 44 patterns, organized in five categories and chapters, are grounded in real-world experience and documented with academic rigor applied and practitioner-community feedback incorporated. I am confident that patterns will serve the community well, today and tomorrow. API designers in industry as well as in research, development, and education related to API design and evolution can benefit from them.

—Prof. Dr. Dr. h. c. Frank Leymann, Managing Director
Institute of Architecture of Application Systems
University of Stuttgart

Preface

This introduction to our book covers the following:

- The context and the purpose of the book—its motivation, goals and scope.
- Who should read the book—our target audience with their use cases and information needs.
- How the book is organized, with patterns serving as knowledge vehicles.

Motivation

Humans communicate in many different languages. The same holds for software. Software not only is written in various programming languages but also communicates via a plethora of protocols (such as HTTP) and message exchange formats (such as JSON). HTTP, JSON, and other technologies operate every time somebody updates their social network profile, orders something in a Web shop, swipes their credit card to purchase something, and so on:

- Application frontends, such as mobile apps on smartphones, place requests for transaction processing at their backends, such as purchase orders in online shops.
- Application parts exchange long-lived data such as customer profiles or product catalogs with each other and with the systems of business partners, customers, and suppliers.
- Application backends provide external services such as payment gateways or cloud storage with data and metadata.

The software components involved in these scenarios—large, small, and in-between—talk to others to achieve their individual goals while jointly serving end users. The software engineer's response to this distribution challenge is application integration via *application programming interfaces (APIs)*. Every integration scenario involves at least two communication parties: API client and API provider.

API clients consume the services exposed by API providers. API documentation governs the client-provider interactions.

Just like humans, software components often struggle to understand each other when they communicate; it is hard for their designers to decide on an adequate size and structure of message content and agree on the best-suited conversation style. Neither party wants to be too quiet or overly talkative when articulating its needs or responding to requests. Some application integration and API designs work very well; the involved parties understand each other and reach their goals. They interoperate effectively and efficiently. Others lack clarity and thereby confuse or stress participants; verbose messages and chatty conversations may overload the communication channels, introduce unnecessary technical risk, and cause extra work in development and operations.

Now, what distinguishes good and poor integration API designs? How can API designers stimulate a positive client developer experience? Ideally, the guidelines for good integration architectures and API designs do not depend on any particular technology or product. Technologies and products come and go, but related design advice should stay relevant for a long time. In our real-world analogy, principles such as those of Cicero's rhetoric and eloquence or Rosenberg's in *Nonviolent Communication: A Language of Life* [Rosenberg 2002] are not specific to English or any other natural language; they will not go out of fashion as natural languages evolve. Our book aims to establish a similar toolbox and vocabulary for integration specialists and API designers. It presents its knowledge bits as *patterns* for API design and evolution that are eligible under different communication paradigms and technologies (with HTTP- and JSON-based Web APIs serving as primary sources of examples).

Goals and Scope

Our mission is to help overcome the complexity of designing and evolving APIs through proven, reusable solution elements:

> How can APIs be engineered understandably and sustainably, starting from stakeholder goals, architecturally significant requirements, and already proven design elements?

While much has been said and written about HTTP, Web APIs, and integration architectures in general (including service-oriented ones), the design of individual API endpoints and message exchanges has received less attention so far:

- How many API operations should be exposed remotely? Which data should be exchanged in request and response messages?

- How is loose coupling of API operations and client-provider interactions ensured?
- What are suitable message representations: flat or hierarchically nested ones? How is agreement reached on the meaning of the representation elements so that these elements are processed correctly and efficiently?
- Should API providers be responsible for processing data provided by their clients, possibly changing the provider-side state and connecting to backend systems? Or should they merely provide shared data stores to their clients?
- How are changes to APIs introduced in a controlled way that balances extensibility and compatibility?

The patterns in this book help answer these questions by sketching proven solutions to specific design problems recurring in certain requirements contexts. Focusing on remote APIs (rather than program-internal ones), they aim at improving the developer experience on both the client side and the provider side.

Target Audience

This book targets intermediate-level software professionals striving to improve their skills and designs. The presented patterns primarily aim at integration architects, API designers, and Web developers interested in platform-independent architectural knowledge. Both backend-to-backend integration specialists and developers of APIs supporting frontend applications can benefit from the knowledge captured in the patterns. As we focus on API endpoint granularity and the data exchanged in messages, additional target roles are API product owner, API reviewer, and cloud tenant and provider.

> This book is for you if you are a medium-experienced software engineer (such as developer, architect, or product owner) already familiar with API fundamentals and want to improve your API design capabilities, including message data contract design and API evolution.

Students, lecturers, and software engineering researchers may find the patterns and their presentation in this book useful as well. We provide an introduction to API fundamentals and a domain model for API design to make the book and its patterns understandable without first having to read a book for beginners.

Knowing about the available patterns and their pros and cons will improve proficiency regarding API design and evolution. APIs and the services they provide will be simpler to develop, consume, and evolve when applying patterns from this book suited for a particular requirements context.

Usage Scenarios

Our objective is to make API design and usage a pleasant experience. To that end, three main use cases for our book and its patterns are as follows:

1. *Facilitate API design discussions and workshops* by establishing a common vocabulary, pointing out required design decisions, and sharing available options and related trade-offs. Empowered by this knowledge, API providers are enabled to expose APIs of quality and style that meet their clients' needs, both short term and long term.
2. *Simplify API design reviews and speed up objective API comparisons* so that APIs can be quality assured—and evolved in a backward-compatible and extensible way.
3. *Enhance API documentation with platform-neutral design information* so that API client developers can grasp the capabilities and constraints of provided APIs with ease. The patterns are designed to be embeddable into API contracts and observable in existing designs.

We provide a fictitious case study and two real-world pattern adoption stories to demonstrate and jumpstart this pattern usage.

We do not expect readers to know any particular modeling approach, design technique, or architectural style already. However, such concepts—for instance, the Align-Define-Design-Refine (ADDR) process, domain-driven design (DDD), and responsibility-driven design (RDD)—have their roles to play. They are reviewed briefly in Appendix A.

Existing Design Heuristics (and Knowledge Gaps)

You can find many excellent books that provide deep advice on specific API technologies and concepts. For instance, the *RESTful Web Services Cookbook* [Allamaraju 2010] explains how to build HTTP resource APIs—for example, which HTTP method such as POST or PUT to pick. Other books explain how asynchronous messaging works in terms of routing, transformation, and guaranteed delivery [Hohpe 2003]. Strategic DDD [Evans 2003; Vernon 2013] can get you started with API

endpoint and service identification. Service-oriented architecture, cloud computing, and microservice infrastructure patterns have been published. Structuring data storages (relational, NoSQL) is also documented comprehensively, and an entire pattern language for distributed systems design is available as well [Buschmann 2007]. Finally, *Release It!* extensively covers design for operations and deployment to production [Nygard 2018a].

The API design process, including goal-driven endpoint identification and operation design, is also covered well in existing books. For instance, *Principles of Web API Design: Delivering Value with APIs and Microservices* [Higginbotham 2021] suggests four process phases with seven steps. *The Design of Web APIs* [Lauret 2019] proposes an API goal canvas, and *Design and Build Great Web APIs: Robust, Reliable, and Resilient* [Amundsen 2020] works with API stories.

Despite these invaluable sources of design advice, the remote API design space still is not covered sufficiently. Specifically, what about the structures of the request and response messages going back and forth between API client and provider? *Enterprise Integration Patterns* [Hohpe 2003] features three patterns representing message types (event, command, and document message) but does not provide further details on their inner workings. However, "data on the outside," exchanged between systems, differs from "data on the inside" that is processed program-internally [Helland 2005]. There are significant differences between the two types of data in terms of their mutability, lifetime, accuracy, consistency, and protection needs. For instance, increasing a local stock-item counter internal to an inventory system probably requires somewhat less architecture design than product pricing and shipment information that is exchanged between manufacturers and logistics companies jointly managing a supply chain via remote APIs and messaging channels.

Message representation design—data on the outside [Helland 2005] or the "Published Language" pattern [Evans 2003] of an API—is the main focus area of this book. It closes the knowledge gaps regarding API endpoint, operation, and message design.

Patterns as Knowledge Sharing Vehicles

Software patterns are sophisticated knowledge-sharing instruments with a track record of more than 25 years. We decided for the pattern format to share API design advice because pattern names aim at forming a domain vocabulary, a "Ubiquitous Language" [Evans 2003]. For instance, the enterprise integration patterns have become the lingua franca of queue-based messaging; these patterns were even implemented in messaging frameworks and tools.

Patterns are not invented but are mined from practical experience and then hardened via peer feedback. The patterns community has developed a set of practices to organize the feedback process; shepherding and writers' workshops are two particularly important ones [Coplien 1997].

At the heart of each pattern is a problem-solution pair. Its forces and the discussion of consequences support informed decision making, for instance, about desired and achieved quality characteristics—but also about the downsides of certain designs. Alternative solutions are discussed, and pointers to related patterns and possible implementation technologies complete the picture.

Note that patterns do not aim at providing complete solutions but serve as sketches to be adopted and tailored for a particular, context-specific API design. In other words, patterns are soft around their edges; they outline possible solutions but do not provide blueprints to be copied blindly. How to adopt and realize a pattern to satisfy project or product requirements remains the responsibility of API designers and owners.

We have been applying and teaching patterns in industry and academia for a long time. Some of us have written patterns for programming, architecting, and integrating distributed application systems and their parts [Voelter 2004; Zimmermann 2009; Pautasso 2016].

We found the pattern concept to be well suited for the usage scenarios stated earlier under "Goals and Scope" and "Target Audience."

Microservice API Patterns

Our pattern language, called *Microservice API Patterns (MAP)*, provides comprehensive views on API design and evolution from the perspective of the messages exchanged when APIs are exposed and consumed. These messages and their payloads are structured as representation elements. The representation elements differ in their *structure* and meaning because API endpoints and their operations have different architectural *responsibilities*. The message structures strongly influence the design time and runtime *qualities* of an API and its underlying implementations; for instance, few large messages cause network and endpoint workloads (such as CPU consumption and network bandwidth usage) that differ from those caused by many small messages. Finally, successful APIs *evolve* over time; the changes over time have to be managed.

We chose the metaphor and acronym MAP because maps provide orientation and guidance, just as pattern languages do; they educate their readers on the options available in an abstract solution space. APIs themselves also have a mapping nature, as they route incoming requests to the underlying service implementations.

We admit that "Microservice API Patterns" might come across as click-bait. In case microservices are no longer fashionable shortly after this book is published, we reserve the right to rename the language and repurpose the acronym. For instance, "Message API Patterns" outlines the scope of the language well too. In the book, we refer to MAP as "the pattern language" or "our patterns" most of the time.

Scope of the Patterns in This Book

This book is the final outcome of a volunteer project focused on the design and evolution of Web APIs and other remote APIs addressing endpoint and message responsibility, structure, and quality concerns as well as service API evolution. The project started in the fall of 2016. The resulting pattern language, presented in this book, helps answer the following questions:

- What is the architectural role played by each API endpoint? How do the endpoint roles and the responsibilities of operations impact service size and granularity?
- What is an adequate number of representation elements in request and response messages? How are these elements structured? How can they be grouped and annotated with supplemental information?
- How can an API provider achieve a certain level of API quality while at the same time using its resources in a cost-effective way? How can quality trade-offs be communicated and accounted for?
- How can API professionals deal with life-cycle management concerns such as support periods and versioning? How can they promote backward compatibility and communicate unavoidable breaking changes?

We collected our patterns by studying numerous Web APIs and API-related specifications and reflecting on our own professional experience (before writing any pattern). We observed many occurrences of the patterns—known uses—both in public Web APIs and in application development and software integration projects in industry. Intermediate versions of many of our patterns went through the shepherding and writers' workshop processes at EuroPLoP[1] from 2017 to 2020; they were later published in the respective conference proceedings.[2]

1. https://europlop.net/content/conference.
2. We decided not to include big collections of known uses in the book; such information is available online and in the EuroPLoP conference proceedings from 2016 to 2020. In some of the supplemental resources, you can find extra implementation hints as well.

Entry Points, Reading Order, and Content Organization

When maneuvering a complex design space to solve wicked problems [Wikipedia 2022a] (and API design certainly qualifies as sometimes wicked), it is often hard to see the forest for the trees. It is neither possible nor desirable to serialize or standardize the problem-solving activities. Therefore, our pattern language has multiple entry points. Each book part can serve as a starting point, and Appendix A suggests even more.

The book has three parts: **Part 1, "Foundations and Narratives," Part 2, "The Patterns,"** and **Part 3, "Our Patterns in Action (Now and Then)."** Figure P.1 shows these parts with their chapters and logical dependencies.

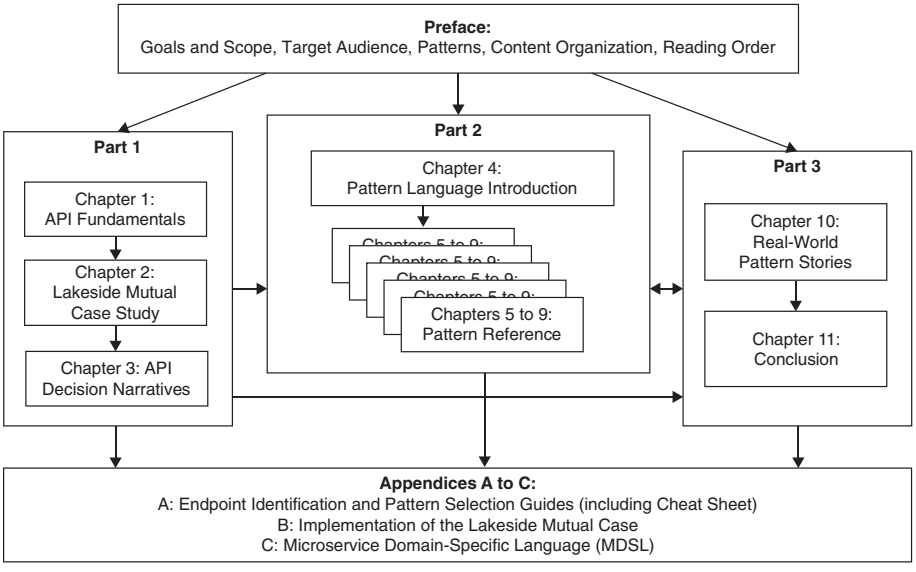


Figure P.1 *Book parts and their dependencies*

Part 1 introduces the domain of API design conceptually, starting with **Chapter 1, "Application Programming Interface (API) Fundamentals."** Lakeside Mutual, our case study and primary source of examples, appears for the first time with its business context, requirements, existing systems, and initial API design in **Chapter 2, "Lakeside Mutual Case Study."** We provide decision models that show how the patterns in our language relate to each other in **Chapter 3, "API Decision Narratives."** Chapter 3 also provides pattern selection criteria and shows how the featured decisions were made in the Lakeside Mutual case. These decision models may serve as navigation aids when reading the book and when applying the patterns in practice.

Part 2 is the pattern reference; it starts with **Chapter 4, "Pattern Language Introduction,"** followed by five chapters full of patterns: **Chapter 5, "Define Endpoint Types and Operations," Chapter 6, "Design Request and Response Message Representations," Chapter 7, "Refine Message Design for Quality," Chapter 8, "Evolve APIs,"** and **Chapter 9, "Document and Communicate API Contracts."** Figure P.2 illustrates these chapters and possible reading paths in this part; for instance, you can learn about basic structure patterns such as ATOMIC PARAMETER and PARAMETER TREE in Chapter 4 and then move on to element stereotypes such as ID ELEMENT and METADATA ELEMENT found in Chapter 6.

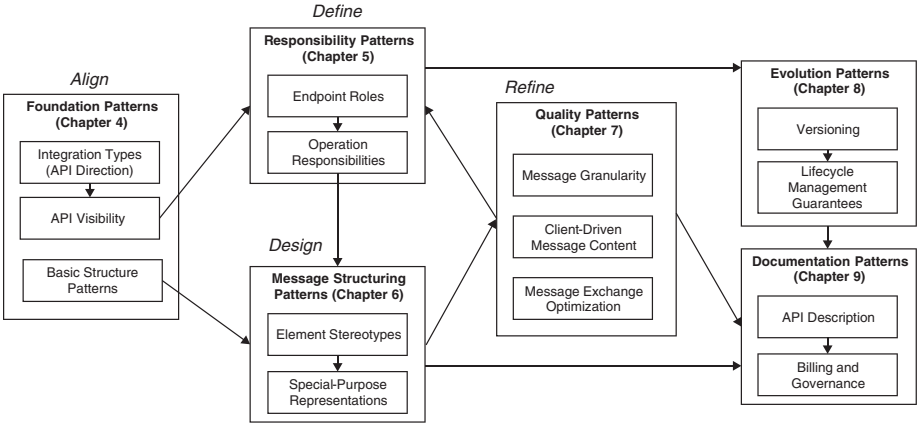


Figure P.2 *Über-pattern map: Chapter flows in Part 2 of the book*

Each pattern description can be seen as a small, specialized article on its own, usually a few pages long. These discussions are structured identically: First, we introduce when and why to apply the pattern. Then we explain how the pattern works and give at least one concrete example. Next, we discuss the consequences of applying the pattern and direct readers to other patterns that become eligible once a particular one has been applied. The names of our patterns are set in SMALL CAPS (example: PROCESSING RESOURCE). This pattern template, introduced in detail in Chapter 4, was derived from the EuroPLoP conference template [Harrison 2003]. We refactored it slightly to take review comments and advice into account (thank you Gregor and Peter!). It puts particular emphasis on quality attributes and their conflicts, as our patterns deal with architecturally significant requirements; consequently, trade-offs are required when making API design and evolution decisions.

Part 3 features the application of the patterns in two real-world projects in rather different domains, e-government and offer/order management in the construction industry. It also reflects, draws some conclusions, and gives an outlook.

Appendix A, "Endpoint Identification and Pattern Selection Guides," provides a problem-oriented cheat sheet as another option to get started. It also discusses how our patterns relate to RDD, DDD, and ADDR. **Appendix B, "Implementation of the Lakeside Mutual Case,"** shares more API design artifacts from the book's case study. **Appendix C, "Microservice Domain-Specific Language (MDSL),"** provides a working knowledge of MDSL, a language for microservices contracts with built-in pattern support via decorators such as `<<Pagination>>`. MDSL provides bindings and generator support for OpenAPI, gRPC protocol buffers, GraphQL, and other interface description and service programming languages.

You will see some (but not much) Java and quite a bit of JSON and HTTP (for instance, in the form of curl commands and responses to them) as you find your way through the book. Very little, if any, gRPC, GraphQL, and SOAP/WSDL might also come your way; if so, it is designed to be simple enough to be understandable without expertise in any of these technologies. Some of our examples are described in MDSL (if you are wondering why we created yet another interface description language: OpenAPI in its YAML or JSON renderings simply does not fit on a single book page when going beyond HelloWorld–ish examples!).

Supplemental information is available through the Web site companion to this book:

https://api-patterns.org

We hope you find the results of our efforts useful so that our patterns have a chance to find their way into the body of knowledge of the global community of integration architects and API developers. We will be glad to hear about your feedback and constructive criticism.

Olaf, Mirko, Daniel, Uwe, Cesare
June 30, 2022

Acknowledgments

We thank Vaughn Vernon for all his feedback and encouragement during our book project. We feel honored to be part of his Addison Wesley Signature Series. Special thanks also go to Haze Humbert, Menka Mehta, Mary Roth, Karthik Orukaimani, and Sandra Schroeder from Pearson for their excellent support and to Frank Leymann for providing the foreword and valuable feedback on our work. Our copy editor, Carol Lallier of Clarity Editing, made this late activity a rewarding, even pleasant experience.

The real-world pattern stories in this book would have not been possible without the cooperation of development projects. Thus, we'd like to thank Walter Berli and Werner Möckli from Terravis and Phillip Ghadir and Willem van Kerkhof from innoQ for their inputs and work on these stories. Nicolas Dipner and Sebnem Kaslack created the initial versions of the patterns icons in their term and bachelor thesis projects. Toni Suter implemented large parts of the Lakeside Mutual case study applications. Stefan Kapferer, developer of Context Mapper, also contributed to the MDSL tools.

We want to thank all the people who provided feedback on the content of this book. Special thanks go to Andrei Furda, who provided input to the introductory material and reviewed many of our patterns; Oliver Kopp and Hans-Peter Hoidn, who applied patterns, provided feedback, and/or organized several informal workshops with peers; James Higginbotham and, again, Hans-Peter Hoidn, who reviewed the book manuscript.

In addition, many colleagues provided helpful feedback, especially the shepherds and writer's workshop participants from EuroPLoP 2017, 2018, 2019, and 2020. We thank the following individuals for their valuable insights: Linus Basig, Luc Bläser, Thomas Brand, Joseph Corneli, Filipe Correia, Dominic Gabriel, Antonio Gámez Díaz, Reto Fankhauser, Hugo Sereno Ferreira, Silvan Gehrig, Alex Gfeller, Gregor Hohpe, Stefan Holtel, Ana Ivanchikj, Stefan Keller, Michael Krisper, Jochen Küster, Fabrizio Lazzaretti, Giacomo De Liberali, Fabrizio Montesi, Frank Müller, Padmalata Nistala, Philipp Oser, Ipek Ozkaya, Boris Pokorny, Stefan Richter, Thomas Ronzon, Andreas Sahlbach, Niels Seidel, Souhaila Serbout, Apitchaka Singjai, Stefan Sobernig, Peter Sommerlad, Markus Stolze, Davide Taibi, Dominic Ullmann, Martin (Uto869), Uwe van Heesch, Timo Verhoeven, Stijn Vermeeren, Tammo van Lessen, Robert Weiser, Erik Wilde, Erik Wittern, Eoin Woods, Rebecca Wirfs-Brock, and Veith Zäch. We also would like to thank the students of several editions of the HSR/OST lectures "Advanced Patterns and Frameworks" and "Application Architecture" and of the USI lecture on "Software Architecture." Their discussion of our patterns and additional feedback are appreciated.

This page intentionally left blank

About the Authors

Olaf Zimmermann is a long-time service orienteer with a PhD in architectural decision modeling. As consultant and professor of software architecture at the Institute for Software at Eastern Switzerland University of Applied Sciences, he focuses on agile architecting, application integration, cloud-nativity, domain-driven design, and service-oriented systems. In his previous life as a software architect at ABB and IBM, he had e-business and enterprise application development clients around the world and worked on systems and network management middleware earlier. Olaf is a Distinguished (Chief/Lead) IT Architect at The Open Group and co-edits the Insights column in *IEEE Software*. He is an author of *Perspectives on Web Services* and the first IBM Redbook on Eclipse. He blogs at ozimmer.ch and medium.com/olzzio.

Mirko Stocker is a programmer by heart who could not decide whether he liked frontend or backend development more, so he stayed in the middle and discovered that APIs hold many interesting challenges as well. He cofounded two startups in the legal tech sector, one of which he still chairs as managing director. This path has led him to become a professor of software engineering at the Eastern Switzerland University of Applied Sciences, where he researches and teaches in the areas of programming languages, software architecture, and Web engineering.

Daniel Lübke is an independent coding and consulting software architect with a focus on business process automation and digitization projects. His interests are software architecture, business process design, and system integration, which inherently require APIs to develop solutions. He received his PhD at the Leibniz Universität Hannover, Germany, in 2007 and has worked in many industry projects in different domains since then. Daniel is author and editor of several books, articles, and research papers; gives training; and regularly presents at conferences on topics of APIs and software architecture.

Uwe Zdun is a full professor of software architecture at the Faculty of Computer Science, University of Vienna. His work focuses on software design and architecture, empirical software engineering, distributed systems engineering (microservices, service-based, cloud, APIs, and blockchain-based systems), DevOps and continuous delivery, software patterns, software modeling, and model-driven development. Uwe has worked on many research and industry projects in these fields, and in

addition to his scientific writing is co-author of the professional books *Remoting Patterns—Foundations of Enterprise, Internet, and Realtime Distributed Object Middleware, Process-Driven SOA—Proven Patterns for Business-IT Alignment,* and *Software-Architektur.*

Cesare Pautasso is a full professor at the Software Institute of the USI Faculty of Informatics, in Lugano, Switzerland, where he leads the Architecture, Design, and Web Information Systems Engineering research group. He chaired the 25th European Conference on Pattern Languages of Programs (EuroPLoP 2022). He was lucky to meet Olaf during a brief stint at the IBM Zurich Research Lab back in 2007, after receiving his PhD from ETH Zurich in 2004. He has co-authored *SOA with REST* (Prentice Hall, 2013) and, self-published the Beautiful APIs series, *RESTful Dictionary,* and *Just Send an Email: Anti-patterns for Email-centric Organizations* on LeanPub.

Chapter 7

Refine Message Design for Quality

This chapter covers seven patterns that address issues with API quality. Arguably, it would be hard to find any API designers and product owners who do not value qualities such as intuitive understandability, splendid performance, and seamless evolvability. That said, any quality improvement comes at a price—a literal cost such as extra development effort but also negative consequences such as an adverse impact on other qualities. This balancing act is caused by the fact that some of the desired qualities conflict with each other—just think about the almost classic performance versus security trade-offs.

We first establish why these issues are relevant in "Introduction to API Quality." The next section presents two patterns dealing with "Message Granularity." Three patterns for "Client-Driven Message Content" follow, and two patterns aim at "Message Exchange Optimization."

These patterns support the third and the fourth phases of the Align-Define-Design-Refine (ADDR) design process for APIs that we introduced at the start of Part 2.

Introduction to API Quality

Modern software systems are distributed systems: mobile and Web clients communicate with backend API services, often hosted by a single or even multiple cloud providers. Multiple backends also exchange information and trigger activities in each other. Independent of the technologies and protocols used, messages travel through one or several APIs in such systems. This places high demands on quality aspects of the API contract and its implementation: API clients expect any provided API to be reliable, responsive, and scalable.

API providers must balance conflicting concerns to guarantee a high service quality while ensuring cost-effectiveness. Hence, all patterns presented in this chapter help resolve the following overarching design issue:

How to achieve a certain level of quality of a published API while at the same time utilizing the available resources in a cost-effective way?

Performance and scalability concerns might not have a high priority when initially developing a new API, especially in agile development—if they arise at all. Usually, there is not enough information on how clients will use the API to make informed decisions. One could also just guess, but that would not be prudent and would violate principles such as making decisions in the most responsible moment [Wirfs-Brock 2011].

Challenges When Improving API Quality

The usage scenarios of API clients differ from each other. Changes that benefit some clients may negatively impact others. For example, a Web application that runs on a mobile device with an unreliable connection might prefer an API that offers just the data that is required to render the current page as quickly as possible. All data that is transmitted, processed, and then not used is a waste, squandering valuable battery time and other resources. Another client running as a backend service might periodically retrieve large amounts of data to generate elaborate reports. Having to do so in multiple client-server interactions introduces a risk of network failures; the reporting has to resume at some point or start from scratch when such failures occur. If the API has been designed with its request/response messages tailored to either use case, the API very likely is not ideally suited for the other one.

Taking a closer look, the following conflicts and design issues arise:

- **Message sizes versus number of requests:** Is it preferable to exchange several small messages or few larger ones? Is it acceptable that some clients might have to send multiple requests to obtain all the data required so that other clients do not have to receive data they do not use?
- **Information needs of individual clients:** Is it valuable and acceptable to prioritize the interests of some customers over those of others?
- **Network bandwidth usage versus computation efforts:** Should bandwidth be preserved at the expense of higher resource usage in API endpoints and their clients? Such resources include computation nodes and data storage.

- **Implementation complexity versus performance:** Are the gained bandwidth savings worth their negative consequences, for instance, a more complex implementation that is harder and more costly to maintain?
- **Statelessness versus performance:** Does it make sense to sacrifice client/provider statelessness to improve performance? Statelessness improves scalability.
- **Ease of use versus latency:** Is it worth speeding up the message exchanges even if doing so results in a harder-to-use API?

Note that the preceding list is nonexhaustive. The answers to these questions depend on the quality goals of the API stakeholders and additional concerns. The patterns in this chapter provide different options to choose from under a given set of requirements; adequate selections differ from API to API. Part 1 of this book provided a decision-oriented overview of these patterns in the "Deciding for API Quality Improvements" section of Chapter 3, "API Decision Narratives." In this chapter, we cover them in depth.

Patterns in This Chapter

The section "Message Granularity" contains two patterns: Embedded Entity and Linked Information Holder. Data Elements offered by API operations frequently reference other elements, for example, using hyperlinks. A client can follow these links to retrieve the additional data; this can become tedious and lead to a higher implementation effort and latency on the client side. Alternatively, clients can retrieve all data at once when providers directly embed the referenced data instead of just linking to it.

"Client-Driven Message Content" features three patterns. API operations sometimes return large sets of data elements (for example, posts on a social media site or products in an e-commerce shop). API clients may be interested in all of these data elements, but not necessarily all at once and not all the time. Pagination divides the data elements into chunks so that only a subset of the sequence is sent and received at once. Clients are no longer overwhelmed with data, and performance and resource usage improve. Providers may offer relatively rich data sets in their response messages. If the problem is that not all clients require all information all the time, then a Wish List allows these clients to request only the attributes in a response data set that they are interested in. Wish Template addresses the same problem but offers clients even more control over possibly nested response data structures. These patterns address concerns such as accuracy of the information, data parsimony, response times, and processing power required to answer a request.

Finally, the "Message Exchange Optimization" section features two patterns, Conditional Request and Request Bundle. The other patterns in this chapter offer

several options to fine-tune message contents to avoid issuing too many requests or transmitting data that is not used; in contrast, Conditional Requests avoid sending data that a client already has. While the number of messages exchanged stays the same, the API implementation can respond with a dedicated status code to inform the client that more recent data is not available. The number of requests sent and responses received can also impair the quality of an API. If clients have to issue many small requests and wait for individual responses, bundling them into a larger message can improve throughput and reduce the client-side implementation effort. The Request Bundle pattern presents this design option.

Figure 7.1 provides an overview of the patterns in this chapter and shows their relations.

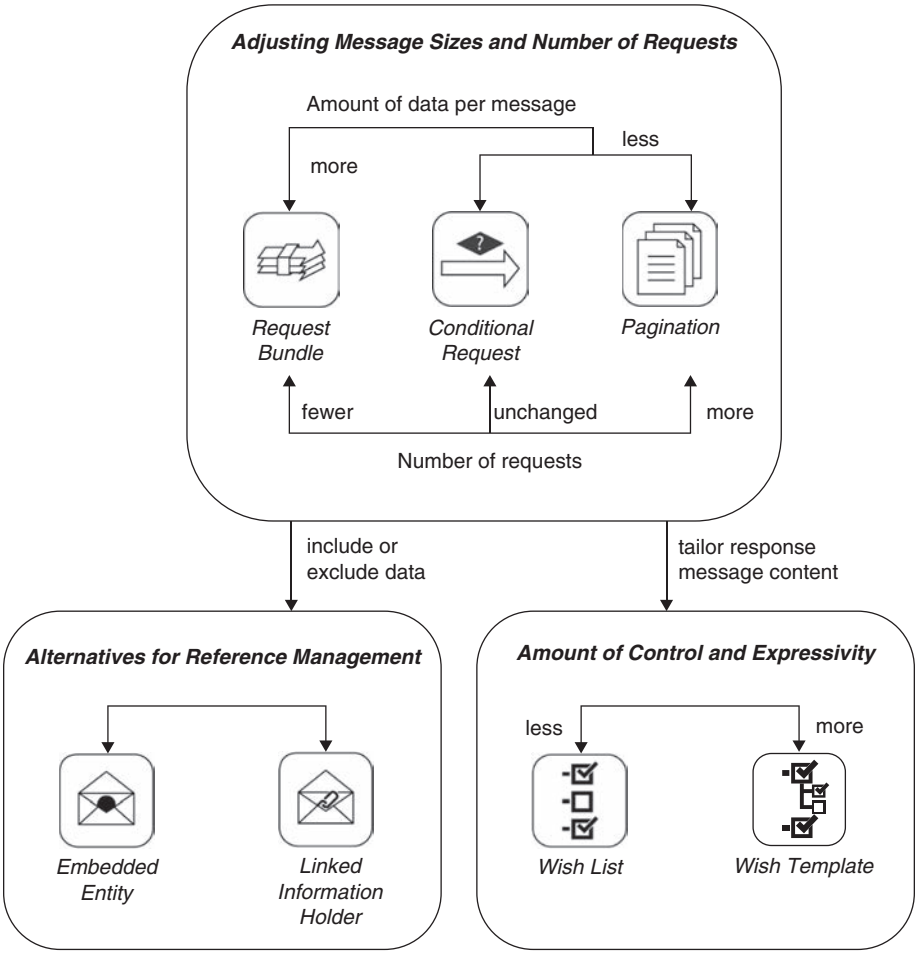


Figure 7.1 *Pattern map for this chapter (API quality)*

Message Granularity

Information elements in request and response message representations, concepts from our API domain model (see Chapter 1, "Application Programming Interface (API) Fundamentals"), often reference other ones to express containment, aggregation, or other relationships. For instance, operational data such as orders and shipments typically is associated with master data such as product and customer records. To expose such references when defining API endpoints and their operations, the two basic options are as follows:

1. Embedded Entity: Embed the referenced data in a possibly nested Data Element (introduced in Chapter 6, "Design Request and Response Message Representations") in the message representation.
2. Linked Information Holder: Place a Link Element (also Chapter 6) in the message representation to look up the referenced data via a separate API call to an Information Holder Resource (Chapter 5, "Define Endpoint Types and Operations").

These message sizing and scoping options have an impact on the API quality:

- **Performance and scalability:** Both message size and number of calls required to cover an entire integration scenario should be kept low. Few messages that transport a lot of data take time to create and process; many small messages are easy to create but cause more work for the communications infrastructure and require receiver-side coordination.
- **Modifiability and flexibility:** Backward compatibility and extensibility are desired in any distributed system whose parts evolve independently of each other. Information elements contained in structured, self-contained representations might be hard to change because any local updates must be coordinated and synchronized with updates to the API operations that work with them and related data structures in the API implementation. Structured representations that contains references to external resources usually is even harder to change than self-contained data because clients have to be aware of such references so that they can follow them correctly.
- **Data quality:** Structured master data such as customer profiles or product details differs from simple unstructured reference data such as country and currency codes (Chapter 5 provides a categorization of domain data by lifetime and mutability). The more data is transported, the more governance is required to make

this data useful. For instance, data ownership might differ for products and customers in an online shop, and the respective data owners usually have different requirements, for example, regarding data protection, data validation, and update frequency. Extra metadata and data management procedures might be required.

- **Data privacy:** In terms of data privacy classifications, the source and the target of data relationships might have different protection needs; an example is a customer record with contact address and credit card information. More fine-grained data retrieval facilitates the enforcement of appropriate controls and rules, lowering the risk of embedded restricted data accidentally slipping through.
- **Data freshness and consistency:** If data is retrieved by competing clients at different times, inconsistent snapshots of and views on data in these clients might materialize. Data references (links) may help clients to retrieve the most recent version of the referenced data. However, such references may break, as their targets may change or disappear after the link referring to it has been sent. By embedding all referenced data in the same message, API providers can deliver an internally consistent snapshot of the content, avoiding the risk of link targets becoming unavailable. Software engineering principles such as single responsibility may lead to challenges regarding data consistency and data integrity when taken to the extreme because data may get fragmented and scattered.

The two message granularity patterns, Embedded Entity and Linked Information Holder in this section address these issues in opposite ways. Combining them on a case-by-case basis leads to adequate message sizes, balancing the number of calls and the amount of data exchanged to meet diverse integration requirements.

Pattern: Embedded Entity

When and Why to Apply

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, master data such as a customer profile may *contain* other elements providing contact information including addresses and phone numbers, or a periodic business results 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 creating request messages or processing response messages.

How can one avoid exchanging multiple messages when their receivers require insights about multiple related information elements?

One could simply define one API endpoint for each basic information element (for instance, an entity defined in an application domain model). This endpoint is accessed whenever API clients require data from that information element, for example, when it is referenced from another one. But if API clients use such data in many situations, this solution causes many subsequent requests when references are followed. 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 than local data to keep consistent.

How It Works

For any data relationship that the receiver wants to follow, embed a Data Element in the request or response message that contains the data of the target end of the relationship. Place this Embedded Entity inside the representation of the source of the relationship.

Analyze the outgoing relationships in the new Data Element and consider embedding them in the message as well. Repeat this analysis until *transitive closure* is reached—that is, until all reachable elements have been either included or excluded (or circles are detected and processing stopped). Review each source-target relationship carefully to assess whether the target data is really needed on the receiver side in enough cases. A yes answer to this question warrants transmitting relationship information as Embedded Entities; otherwise, transmitting references to Linked Information Holders might be sufficient. For instance, if a purchase order has a *uses* relation to product master data and this master data is required to make sense of the purchase order, the purchase order representation in request or response messages should contain a copy of all relevant information stored in the product master data in an Embedded Entity.

Figure 7.2 sketches the solution.

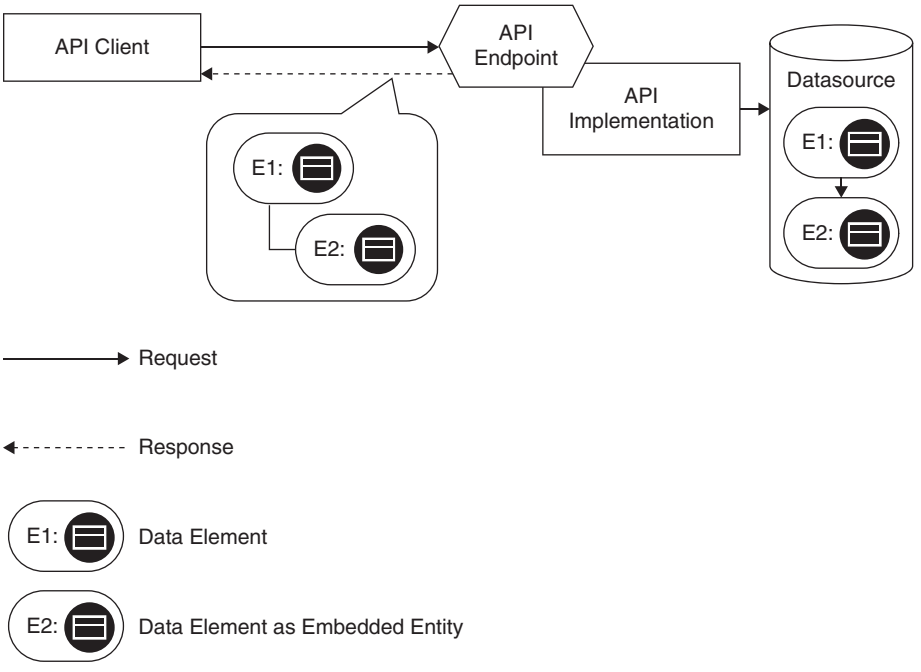


Figure 7.2 *EMBEDDED ENTITY: Single API endpoint and operation, returning structured message content that matches the structure of the source data in the API implementation to follow data relations*

Including an EMBEDDED ENTITY in a message leads to a PARAMETER TREE structure that contains the DATA ELEMENT representing the related data. Additional METADATA ELEMENTS in this tree may denote the relationship type and other supplemental information. There are several options for structuring the tree, corresponding to the contained DATA ELEMENT. It may be nested, for instance, when representing deep containment relationship hierarchies; it may be flat and simply list one or more ATOMIC PARAMETERS. When working with JSON in HTTP resource APIs, JSON objects (possibly including other JSON objects) realize these options. One-to-many relationships (such as a purchase order referring to its order items) cause the EMBEDDED ENTITY to be set-valued. JSON arrays can represent such sets. The options for representing many-to-many relationships are similar to those in the LINKED INFORMATION HOLDERS pattern; for instance, the PARAMETER TREE might contain dedicated nodes for the relationships. Some redundancy might be desired or tolerable, but it also may confuse consumers who expect normalized data. Bidirectional relationships require special attention. One of the directions can be used to create the EMBEDDED ENTITY hierarchy; if the opposite direction should also be made explicit in the message representation, a second instance of this pattern might be required,

causing data duplication. In that case, it might be better to express the second relationship with embedded Id Elements or Link Elements instead.

In any of these cases, the API Description has to explain the existence, structure, and meaning of the Embedded Entity instances.

Example

Lakeside Mutual, our microservices sample application introduced in Chapter 2, "Lakeside Mutual Case Study," contains a service called Customer Core that aggregates several information items (here, entities and value objects from domain-driven design [DDD]) in its operation signatures. API clients such as the Customer Self-Service frontend can access this data via an HTTP resource API. This API contains several instances of the Embedded Entity pattern. Applying the pattern, a response message might look as follows:[1]

```
curl -X GET http://localhost:8080/customers/gktlipwhjr

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

1. Note that the data shown is fictitious, generated by https://www.mockaroo.com.

The referenced information elements are all fully contained in the response message; examples are `customerProfile` and `customerInteractionLog`. No URI links to other resources appear. Note that the `customerProfile` entity actually embeds nested data in this exemplary data set (for example, `currentAddress` and `moveHistory`), while the `customerInteractionLog` does not (but is still included as an empty Embedded Entity).

Discussion

Applying this pattern solves the problem of having to exchange multiple messages when receivers require multiple related information elements. An Embedded Entity reduces the number of calls required: if the required information is included, the client does not have to create a follow-on request to obtain it. Embedding entities can lead to a reduction in the number of endpoints, because no dedicated endpoint to retrieve linked information is required. However, embedding entities leads to larger response messages, which usually take longer to transfer and consume more bandwidth. Care must also be taken to ensure that the included information does not have higher protection needs than the source and that no restricted data slips through.

It can be challenging to anticipate what information different message receivers (that is, API clients for response messages) require to perform their tasks. As a result, there is a tendency to include more data than most clients need. Such design can be found in many Public APIs serving many diverse and possibly unknown clients.

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. 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).

If most or all of the data is actually used, sending many small messages might require more bandwidth than sending one large message (for instance, because protocol header metadata is sent with each small message). If the embedded entities change at different speeds, retransmitting them causes unnecessary overhead because messages with partially changed content can only be cached in their entirety. A fast-changing operational entity might refer to immutable master data, for instance.

The decision to use Embedded Entity might depend on the number of message consumers and the homogeneity of their use cases. For example, if only one consumer with a specific use case is targeted, it is often good to embed all required data straight away. In contrast, different consumers or use cases might not work with the same data. In order to minimize message sizes, it might be advisable not to transfer all data. Both client and provider might be developed by the same organization—for example, when providing "Backends for Frontends" [Newman 2015]. Embedding entities can be a

reasonable strategy to minimize the number of requests in that case. In such a setting, they simplify development by introducing a uniform regular structure.

Combinations of linking and embedding data often make sense, for instance, embedding all data immediately displayed in a user interface and linking the rest for retrieval upon demand. The linked data is then fetched only when the user scrolls or opens the corresponding user interface elements. Atlassian [Atlassian 2022] discusses such a hybrid approach: "Embedded related objects are typically limited in their fields to avoid such object graphs from becoming too deep and noisy. They often exclude their own nested objects in an attempt to strike a balance between performance and utility."

"API Gateways" [Richardson 2016] and messaging middleware [Hohpe 2003] can also help when dealing with different information needs. Gateways 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). Messaging systems may provide transformation capabilities such as filters and enrichers.

Related Patterns

Linked Information Holder describes the complementary, opposite solution for the reference management problem. One reason for switching to the Linked Information Holder might be to mitigate performance problems, for instance, caused by slow or unreliable networks that make it difficult to transfer large messages. Linked Information Holders can help to improve the situation, as they allow caching each entity independently.

If reducing message size is the main design goal, a Wish List or, even more expressive, a Wish Template can also be applied to minimize the data to be transferred by letting consumers dynamically describe which subset of the data they need. Wish List or Wish Template can help to fine-tune the content in an Embedded Entity.

Operational Data Holders reference Master Data Holders by definition (either directly or indirectly); these references often are represented as Linked Information Holders. References between data holders of the same type are more likely to be included with the Embedded Entity pattern. Both Information Holder Resources and Processing Resources might deal with structured data that needs to be linked or embedded; in particular, Retrieval Operations either embed or link related information.

More Information

Phil Sturgeon features this pattern as "Embedded Document (Nesting)" in [Sturgeon 2016b]. See Section 7.5 of *Build APIs You Won't Hate* for additional advice and examples.

Pattern: Linked Information Holder

When and Why to Apply

An API exposes structured data to meet the information needs of its clients. This data contains elements that relate to each other (for example, product master data may *contain* other information elements providing detailed information, or a performance report for a period of time may *aggregate* raw data such as individual measurements). API clients work with several of the related information elements when preparing request messages or processing response messages. Not all of this information is always useful for the clients in its entirety.[2]

How can messages be kept small even when an API deals with multiple information elements that reference each other?

A rule of thumb for distributed system design states that exchanged messages should be small because large messages may overutilize the network and the endpoint processing resources. However, not all of what communication participants want to share with each other might fit into such small messages; for instance, they might want to follow many or all of the relationships within information elements. If relationship sources and targets are not combined into a single message, participants have to inform each other how to locate and access the individual pieces. This distributed information set has to be designed, implemented, and evolved; the resulting dependencies between the participants and the information they share have to be managed. For instance, insurance policies typically refer to customer and product master data; each of these related information elements might, in turn, consist of several parts (see Chapter 2 for deeper coverage of the data and domain entities in this example).

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 lead to large messages containing data not required by some clients and harm the performance of individual API calls. It couples the stakeholders of this data.

2. This pattern context is similar to that of Embedded Entity but emphasizes the diversity of client wants and needs.

How It Works

Add a Link Element to messages that pertain to multiple related information elements. Let the resulting Linked Information Holder reference another API endpoint that exposes the linked element.

The referenced API endpoint often is an Information Holder Resource representing the linked information element. This element might be an entity from the domain model that is exposed by the API (possibly wrapped and mapped); it can also be the result of a computation in the API implementation.

Linked Information Holders might appear in request and response messages; the latter case is more common. Typically, a Parameter Tree is used in the representation structure, combining collections of Link Elements and, optionally, Metadata Elements explaining the link semantics; in simple cases, a set of Atomic Parameters or a single Atomic Parameter might suffice as link carriers.

Figure 7.3 illustrates the two-step conversation realizing the pattern.

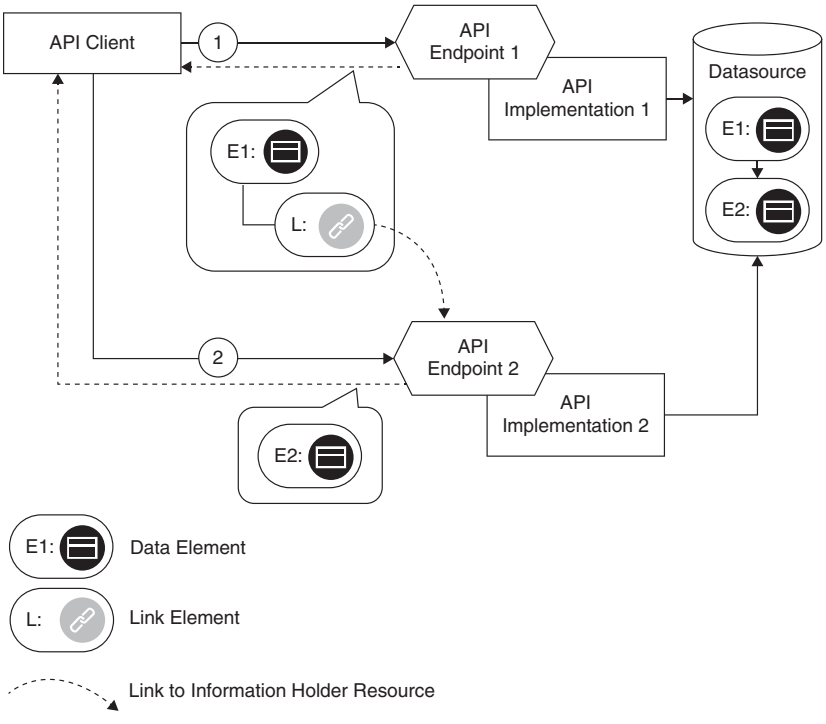


Figure 7.3 *Linked Information Holder: Two API endpoints are involved. The first response contains a link rather than data from the data source; the data is retrieved from it in a follow-on request to the second endpoint*

The Link Element that constitutes the Linked Information Holder provides location information, for instance, a URL (with domain/hostname and port number when using HTTP over TCP/IP). The Link Element also has a local name to be able to identify it within the message representation (such as a JSON object). If more information about the relation should be sent to clients, this Link Element can be annotated with details about the corresponding relationship, for instance, a Metadata Element specifying its type and semantics. In any case, API clients and providers must agree on the meaning of the link relationships and be aware of coupling and side effects introduced. The existence and the meaning of the Linked Information Holder, including cardinalities on both ends of the relation, has to be documented in the API Description.

One-to-many relationships can be modeled as collections, for instance, by transmitting multiple Link Elements as Atomic Parameter Lists. Many-to-many relationships (such as that between books and their readers in a library management system) can be modeled as two one-to-many relationships, with one collection linking the source data to the targets and a second one linking the target data to the sources (assuming that the message recipient wants to follow the relation in both directions). Such design may require the introduction of an additional API endpoint, a *relationship holder resource,* representing the relation rather than its source or target. This endpoint then exposes operations to retrieve all relationships with their sources and targets; it may also allow clients to find the other end of a relationship they already know about. Different types of Link Elements identify these ends in messages sent to and from the relationship holder resource. Unlike in the Embedded Entity pattern, circular dependencies in the data are less of an issue when working with Linked Information Holders (but still should be handled); the responsibility to avoid endless loops in the data processing shifts from the message sender to the recipient.

Example

Our Lakeside Mutual sample application for Customer Management utilizes 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 DDD. API clients can access this data through a Customer Information Holder, implemented as a REST controller in Spring Boot.

The Customer Information Holder, called `customers`, realizes the Information Holder Resource pattern. When applying Linked Information Holder for its `customerProfile` and its `moveHistory` a response message may look as follows:

```
curl -X GET http://localhost:8080/customers/gktlipwhjr

{
  "customer": {
    "id": "gktlipwhjr"
  },
  "links": [{
    "rel": "customerProfile",
    "href": "/customers/gktlipwhjr/profile"
  }, {
    "rel": "moveHistory",
    "href": "/customers/gktlipwhjr/moveHistory"
  }],
  "email": "rdavenhall0@example.com",
  "phoneNumber": "491 103 8336",
  "customerInteractionLog": {
    "contactHistory": [],
    "classification": "??"
  }
}
```

Both `profile` and `moveHistory` are implemented as sub-resources of the Customer Information Holder. The `customerProfile` can be retrieved by a subsequent GET request to the URI `/customers/gktlipwhjr/profile`. How does the client know that a GET request must be used? This information could have been included in a Metadata Element. In this example, the designers of the API decided not to include it. Instead, their API Description specifies that GET requests are used by default to retrieve information.

Discussion

Linking instead of embedding related data results in smaller messages and uses fewer resources in the communications infrastructure when exchanging individual messages. However, this has to be contrasted with the possibly higher resource use caused by the extra messages required to follow the links: Additional requests are required to dereference the linked information. Linking instead of embedding might demand more resources in the communications infrastructure. Additional Information Holder Resource endpoints have to be provided for the linked data, causing development and operations effort and cost, but allowing to enforce additional access restrictions.

When introducing Linked Information Holders 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 a promise infinitely. Even if a long lifetime of the linked endpoint is guaranteed, links still may break, for instance, when the data organization or deployment location changes. Clients should expect this and be able to follow redirects or referrals to the updated links. To minimize breaking links, the API provider should invest in maintaining link consistency; a Link Lookup Resource can be used to do so.

Sometimes the data distribution reduces the number of messages exchanged. Different Linked Information Holders may be defined for data that changes at a different velocity. Clients then can request frequently changing data whenever they require the latest snapshot of it; they do not have to re-request slower changing data that is embedded with it (and therefore tightly coupled).

The pattern leads to modular API designs but also adds a dependency that must be managed. It potentially has performance, workload, and maintenance costs attached. The Embedded Entity pattern can be used instead if justified from a performance point of view. This makes sense if a few large calls turn out to perform better than many small ones due to network and endpoint processing capabilities or constraints (this should be measured and not guessed). It might be required to switch back and forth between Embedded Entity and Linked Information Holder during API evolution; with Two In Production, both designs can be offered at the same time, for instance, for experimentation with a potential change. The API refactorings "Inline Information Holder" and "Extract Information Holder" of the Interface Refactoring Catalog [Stocker 2021b] provide further guidance and step-by-step instructions.

Linked Information Holder is well suited when referencing rich information holders serving multiple usage scenarios: usually, not all message recipients require the full set of referenced data, for instance, when Master Data Holders such as customer profiles or product records are referenced from Operational Data Holders such as customer inquiries or orders. Following links to Linked Information Holders, message recipients can obtain the required subsets on demand.

The decision to use Linked Information Holder and/or to include an Embedded Entity might depend on the number of API clients and the level of similarity of their use cases. Another decision driver is the complexity of the domain model and the application scenarios it represents. For example, if one client with a specific use case is targeted, it usually makes sense to embed all data. However, if there are several clients, not all of them might appreciate the same comprehensive data. In such situations, Linked Information Holders pointing at the data used only by a fraction of the clients reduces the message sizes.

Related Patterns

Linked Information Holders typically reference Information Holder Resources. The referenced Information Holder Resources can be combined with Link Lookup Resource to cope with potentially broken links. By definition, Operational Data Holders reference Master Data Holders; these references can either be included and flattened as Embedded Entities or structured and then progressively followed using Linked Information Holders.

Other patterns that help reduce the amount of data exchanged can be used alternatively. For instance, Conditional Request, Wish List, and Wish Template are eligible; Pagination is an option too.

More Information

"Linked Service" [Daigneau 2011] is a similar pattern but is less focused on data. "Web Service Patterns" [Monday 2003] has a "Partial DTO Population" pattern that solves a similar problem; DTO stands for Data Transfer Object.

See *Build APIs You Won't Hate,* Section 7.4 [Sturgeon 2016b], for additional advice and examples, to be found under "Compound Document (Sideloading)."

The backup, availability, consistency (BAC) theorem investigates data management issues further [Pardon 2018].

Client-Driven Message Content (aka Response Shaping)

In the previous section, we presented two patterns to handle references between data elements in messages. An API provider can choose between embedding or linking related data elements, and also combine these two options to achieve suitable message sizes. Depending on the clients and their API usage, their best usage may be clear. But usage scenarios of clients might be so different that an even better solution would be to let clients themselves decide at runtime which data they are interested in.

The patterns in this section offer two different approaches to optimize this facet of API quality further, *response slicing* and *response shaping.* They address the following challenges:

- **Performance, scalability, and resource use:** Providing all clients with all data every time, even to those that only have a limited or minimal information need, comes at a price. From a performance and workload point of view, it therefore makes sense to transmit only the relevant parts of a data set. However, the pre- and postprocessing required to rightsize the message exchanges also require resources and might harm performance. These costs have to be balanced against the expected reduction of the response message size and the capabilities of the underlying transport network.

- **Information needs of individual clients:** An API provider might have to serve multiple clients with different information needs. Usually, providers do not want to implement custom APIs or client-specific operations but let the clients share a set of common operations. However, certain clients might be interested in just a subset of the data made available via an API. The common operations might be too limited or too powerful in such cases. Other clients might be overwhelmed if a large set if data arrives at once. Delivering too little or too much data to a client is also known as *underfetching* and *overfetching*.
- **Loose coupling and interoperability:** The message structures are important elements of the API contract between API provider and API client; they contribute to the shared knowledge of the communication participants, which impacts the format autonomy aspect of loose coupling. Metadata to control data set sizing and sequencing becomes part of this shared knowledge and has to be evolved along with the payload.
- **Developer convenience and experience:** The developer experience, including learning effort and programming convenience, is closely related to understandability and complexity considerations. For instance, a compact format optimized for transfer might be difficult to document and understand, and to prepare and digest. Elaborate structures enhanced with metadata that simplify and optimize processing cause extra effort during construction (both at design time and at runtime).
- **Security and data privacy:** Security requirements (data integrity and confidentiality in particular) and data privacy concerns are relevant in any message design; security measures might require additional message payloads such as API Keys or security tokens. An important consideration is which payload can and should actually be sent; data that is not sent cannot be tampered with (at least not on the wire). The need for certain, data-specific security measures might actually lead to different message designs (for instance, credit card information might be factored out into a dedicated API endpoint with specifically secured operations). In the context of slicing and sequencing large data sets, all parts can be treated equally unless they have different protection needs. The heavy load caused by assembling and transmitting large data sets can expose the provider to denial-of-service attacks.
- **Test and maintenance effort:** Enabling clients to select which data to receive (and when) creates options and flexibility with regards to what the provider has to expect (and accept) in incoming requests. Therefore, the testing and maintenance effort increases.

The patterns in this section, Pagination, Wish List, and Wish Template, address these challenges in different ways.

Pattern: Pagination

When and Why to Apply

Clients query an API, fetching collections of data items to be displayed to the user or processed in other applications. In at least one of these queries, the API provider responds by sending a large number of items. The size of this response may be larger than what the client needs or is ready to consume at once.

The data set may consist of identically structured elements (for example, rows fetched from a relational database or line items in a batch job executed by an enterprise information system in the backend) or of heterogeneous data items not adhering to a common schema (for example, parts of a document from a document-oriented NoSQL database such as MongoDB).

How can an API provider deliver large sequences of structured data without overwhelming clients?

In addition to the forces already presented in the introduction to this section, Pagination balances the following ones:

- **Session awareness and isolation:** Slicing read-only data is relatively simple. But what if the underlying data set changes while being retrieved? Does the API guarantee that once a client retrieves the first page, the subsequent pages (which may or may not be retrieved later) will contain a data set that is consistent with the subset initially retrieved? How about multiple concurrent requests for partial data?
- **Data set size and data access profile:** Some data sets are large and repetitive, and not all transmitted data is accessed all the time. This offers optimization potential, especially for sequential access over data items ordered from the most recent to the oldest, which may no longer be relevant for the client. Moreover, clients may not be ready to digest data sets of arbitrary sizes.

One could think of sending the entire large response data set in a single response message, but such a simple approach might waste endpoint and network capacity; it also does not scale well. The size of the response to a query may be unknown in advance, or the result set may be too large to be processed at once on the client side (or on the provider side). Without mechanisms to limit such queries, processing errors

such as out-of-memory exceptions may occur, and the client or the endpoint implementation may crash. Developers and API designers often underestimate the memory requirements imposed by unlimited query contracts. These problems usually go unnoticed until concurrent workload is placed on the system or the data set size increases. In shared environments, it is possible that unlimited queries cannot be processed efficiently in parallel, which leads to similar performance, scalability, and consistency issues—only combined with concurrent requests, which are hard to debug and analyze anyway.

How It Works

Divide large response data sets into manageable and easy-to-transmit chunks (also known as pages). Send one chunk of partial results per response message and inform the client about the total and/or remaining number of chunks. Provide optional filtering capabilities to allow clients to request a particular selection of results. For extra convenience, include a reference to the next chunk/page from the current one.

The number of data elements in a chunk can be fixed (its size then is part of the API contract) or can be specified by the client dynamically as a request parameter. Metadata Elements and Link Elements inform the API client how to retrieve additional chunks subsequently.

API clients then process some or all partial responses iteratively as needed; they request the result data page by page. Hence, subsequent requests for additional chunks might have to be correlated. It might make sense to define a policy that governs how clients can terminate the processing of the result set and the preparation of partial responses (possibly requiring session state management).

Figure 7.4 visualizes a sequence of requests that use Pagination to retrieve three pages of data.

Variants The pattern comes in four variants that navigate the data set in different ways: page based, offset based, cursor or token based, and time based.

Page-Based Pagination (a somewhat tautological name) and *Offset-Based Pagination* refer to the elements of the data set differently. The page-based variant divides the data set into same-sized pages; the client or the provider specify the page size. Clients then request pages by their index (like page numbers in a book). With Offset-Based Pagination, a client selects an offset into the whole data set (that is, how many single elements to skip) and the number of elements to return in the next chunk (often referred to as *limit*). Both approaches may be used interchangeably (the offset

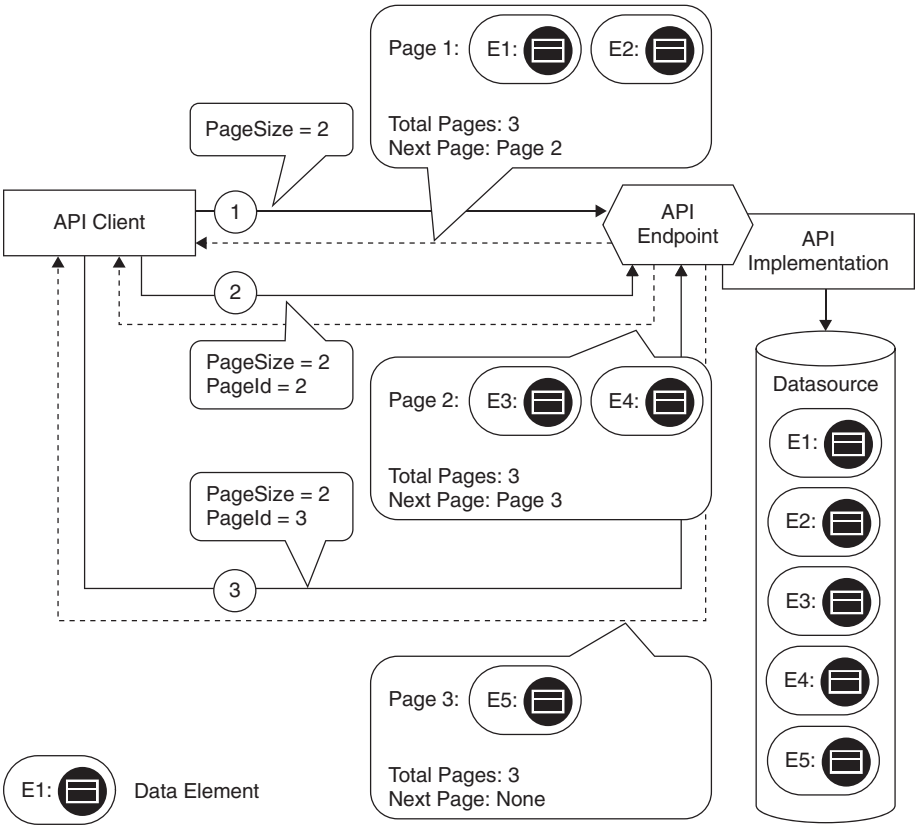


Figure 7.4 *Pagination: Query and follow-on requests for pages, response messages with partial results*

can be calculated by multiplying the page size with the page number); they address the problem and resolve the forces in similar ways. Page-Based Pagination and Offset-Based Pagination do not differ much with respect to developer experience and other qualities. Whether entries are requested with an offset and limit or all entries are divided into pages of a particular size and then requested by an index is a minor difference. Either case requires two integer parameters.

These variants are not well suited for data that changes in between requests and therefore invalidates the index or offset calculations. For example, given a data set ordered by creation time from most recent to oldest, let us assume that a client has retrieved the first page and now requests the second one. In between these requests, the element at the front of the data set is removed, causing an element to move from the second to the first page without the client ever seeing it.

The *Cursor-Based Pagination* variant solves this problem: it does not rely on the absolute position of an element in the data set. Instead, clients send an identifier that the provider can use to locate a specific item in the data set, along with the number of elements to retrieve. The resulting chunk does not change even if new elements have been added since the last request.

The remaining fourth variant, *Time-Based Pagination*, is similar to Cursor-Based Pagination but uses timestamps instead of element IDs. It is used in practice less frequently but could be applied to scroll through a time plot by gradually requesting older or newer data points.

Example

The Lakeside Mutual Customer Core backend API illustrates Offset-Based Pagination in its `customer` endpoint:

```
curl -X GET http://localhost:8080/customers?limit=2&offset=0
```

This call returns the first chunk of two entities and several control Metadata Elements. Besides the link relation [Allamaraju 2010] that points at the next chunk, the response also contains the corresponding `offset`, `limit`, and total `size` values. Note that `size` is not required to implement Pagination on the provider side but allows API clients to show end users or other consumers how many more data elements (or pages) may be requested subsequently.

```
{
  "offset": 0,
  "limit": 2,
  "size": 50,
  "customers": [
    ...
  ,
    ...
  ],
  "_links": {
    "next": {
      "href": "/customers?limit=2&offset=2"
    }
  }
}
```

The preceding example can easily be mapped to the corresponding SQL query `LIMIT 2 OFFSET 0`. Instead of talking about offsets and limits, the API could also use the page metaphor in its message vocabulary, as shown here:

```
curl -X GET http://localhost:8080/customers?page-size=2&page=0

{
  "page": 0,
  "pageSize": 2,
  "totalPages": 25,
  "customers": [
    ...
  ,
    ...
  ],
  "_links": {
    "next": {
      "href": "/customers?page-size=2&page=1"
    }
  }
}
```

Using Cursor-Based Pagination, the client first requests an initial page of the desired size 2:

```
curl -X GET http://localhost:8080/customers?page-size=2

{
  "pageSize": 2,
  "customers": [
    ...
  ,
    ...
  ],
  "_links": {
    "next": {
      "href": "/customers?page-size=2&cursor=mfn834fj"
    }
  }
}
```

The response contains a link to the next chunk of data, represented by the cursor value `mfn834fj`. The cursor could be as simple as the primary key of the database or contain more information, such as a query filter.

Discussion

Pagination aims to substantially improve resource consumption and performance by sending only the data presently required and doing so just in time.

A single large response message might be inefficient to exchange and process. In this context, data set size and data access profile (that is, the user needs), especially the number of data records required to be available to an API client (immediately and over time), require particular attention. Especially when returning data for human consumption, not all data may be needed immediately; then Pagination has the potential to improve the response times for data access significantly.

From a security standpoint, retrieving and encoding large data sets may incur high effort and cost on the provider side and therefore lead to a denial-of-service attack. Moreover, transferring large data sets across a network can lead to interruptions, as most networks are not guaranteed to be reliable, especially cellular networks. This aspect is improved with Pagination because attackers can only request pages with small portions of data instead of an entire data set (assuming that the maximum value for the page size is limited). Note that in a rather subtle attack, it could still be enough to request the first page; if a poorly designed API implementation loads a vast data set as a whole, expecting to feed the data to the client page by page, an attacker still is able to fill up the server memory.

If the structure of the desired responses is not set oriented, so that a collection of data items can be partitioned into chunks, Pagination cannot be applied. Compared to response messages using the Parameter Tree pattern without Pagination, the pattern is substantially more complex to understand and thus might be less convenient to use, as it turns a single call into a longer conversation. Pagination requires more programming effort than does exchanging all data with one message.

Pagination leads to tighter coupling between API client and provider than single message transfers because additional representation elements are required to manage the slicing of the result sets into chunks. This can be mitigated by standardizing the required Metadata Elements. For example, with hypermedia, one just follows a Web link to fetch the next page. A remaining coupling issue is the session that may have to be established with each client while the pages are being scanned.

If API clients want to go beyond sequential access, complex parameter representations may be required to perform random access by seeking specific pages (or to allow clients to compute the page index themselves). The Cursor-Based Pagination variant with its—from a client perspective, opaque—cursor or token usually does not allow random access.

Delivering one page at a time allows the API client to process a digestible amount of data; a specification of which page to return facilitates remote navigation directly within the data set. Less endpoint memory and network capacity are required to handle individual pages, although some overhead is introduced because Pagination management is required (discussed shortly).

The application of Pagination leads to additional design concerns:

- Where, when, and how to define the page size (the number of data elements per page). This influences the chattiness of the API (retrieving the data in many small pages requires a large number of messages).
- How to order results—that is, how to assign them to pages and how to arrange the partial results on these pages. This order typically cannot change after the paginated retrieval begins. Changing the order as an API evolves over its life cycle might make a new API version incompatible with previous ones, which might go unnoticed if not communicated properly and tested thoroughly.
- Where and how to store intermediate results, and for how long (deletion policy, timeouts).
- How to deal with request repetition. For instance, do the initial and the subsequent requests have to be idempotent to prevent errors and inconsistencies?
- How to correlate pages/chunks (with the original, the previous, and the next requests).

Further design issues for the API implementation include the caching policy (if any), the liveness (currentness) of results, filtering, as well as query pre- and post-processing (for example, aggregations, counts, sums). Typical data access layer concerns (for instance, isolation level and locking in relational databases) come into play here as well [Fowler 2002]. Consistency requirements differ by client type and use case: Is the client developer aware of the Pagination? The resolution of these concerns is context-specific; for instance, frontend representations of search results in Web applications differ from batch master data replication in Backend Integrations of enterprise information systems.

With respect to behind-the-scenes changes to mutable collections, two cases have to be distinguished. One issue that has to be dealt with is that new items might be added while the client walks through the pages. The second issue concerns updates to (or removal of) items on a page that has already been seen by the client. Pagination can deal with new items but will usually miss changes to already downloaded items that happened while a Pagination "session" was ongoing.

If the page size was set too small, sometimes the result of Pagination can be annoying for users (especially developers using the API), as they have to click through and wait to retrieve the next page even if there are only a few results. Also, human users may expect client-side searches to filter an entire data set; introducing Pagination may incorrectly result in empty search results because the matching data items are found in pages that have not yet been retrieved.

Not all functions requiring an entire record set, such as searching, work (well) with Pagination, or they require extra effort (such as intermediate data structures on the API client side). Paginating after searching/filtering (and not vice versa) reduces workload.

This pattern covers the download of large data sets, but what about upload? Such *Request Pagination* can be seen as a complementary pattern. It would gradually upload the data and fire off a processing job only once all data is there. Incremental State Build-up, one of the Conversation Patterns [Hohpe 2017], has this inverse nature. It describes a solution similar to Pagination to deliver the data from the client to the provider in multiple steps.

Related Patterns

Pagination can be seen as the opposite of Request Bundle: whereas Pagination is concerned with reducing the individual message size by splitting one large message into many smaller pages, Request Bundle combines several messages into a single large one.

A paginated query typically defines an Atomic Parameter List for its input parameters containing the query parameters and a Parameter Tree for its output parameters (that is, the pages).

A request-response correlation scheme might be required so that the client can distinguish the partial results of multiple queries in arriving response messages; the pattern "Correlation Identifier" [Hohpe 2003] might be eligible in such cases.

A "Message Sequence" [Hohpe 2003] also can be used when a single large data element has to be split up.

More Information

Chapter 10 of *Build APIs You Won't Hate* covers Pagination types, discusses implementation approaches, and presents examples in PHP [Sturgeon 2016b]. Chapter 8 in the *RESTful Web Services Cookbook* deals with queries in a RESTful HTTP context [Allamaraju 2010]. *Web API Design: The Missing Link* covers Pagination under "More on Representation Design" [Apigee 2018].

In a broader context, the user interface (UI) and Web design communities have captured Pagination patterns in different contexts (not API design and management, but interaction design and information visualization). See coverage of the topic at the Interaction Design Foundation Web site [Foundation 2021] and the UI Patterns Web site [UI Patterns 2021].

Chapter 8 of *Implementing Domain-Driven Design* features stepwise retrieval of a notification log/archive, which can be seen as Offset-Based Pagination [Vernon 2013]. RFC 5005 covers feed paging and archiving for Atom [Nottingham 2007].

Pattern: Wish List

When and Why to Apply

API providers serve multiple different clients that invoke the same operations. Not all clients have the same information needs: some might use just a subset of the data offered by an endpoint and its operations; other clients might need rich data sets.

How can an API client inform the API provider at runtime about the data it is interested in?

When addressing this problem, API designers balance performance aspects such as response time and throughput with factors influencing the developer experience such as learning effort and evolvability. They strive for data parsimony (or *Datensparsamkeit*).

These forces could be resolved by introducing infrastructure components such as network- and application-level gateways and caches to reduce the load on the server, but such components add to the complexity of the deployment model and network topology of the API ecosystem and increase related infrastructure testing, operations management, and maintenance efforts.

How It Works

As an API client, provide a Wish List in the request that enumerates all desired data elements of the requested resource. As an API provider, deliver only those data elements in the response message that are enumerated in the Wish List ("response shaping").

Specify the Wish List as an Atomic Parameter List or flat Parameter Tree. As a special case, a simple Atomic Parameter may be included that indicates a verbosity level (or level of detail) such as `minimal`, `medium`, or `full`.

Figure 7.5 sketches the request and response messages used when introducing a WISH LIST:

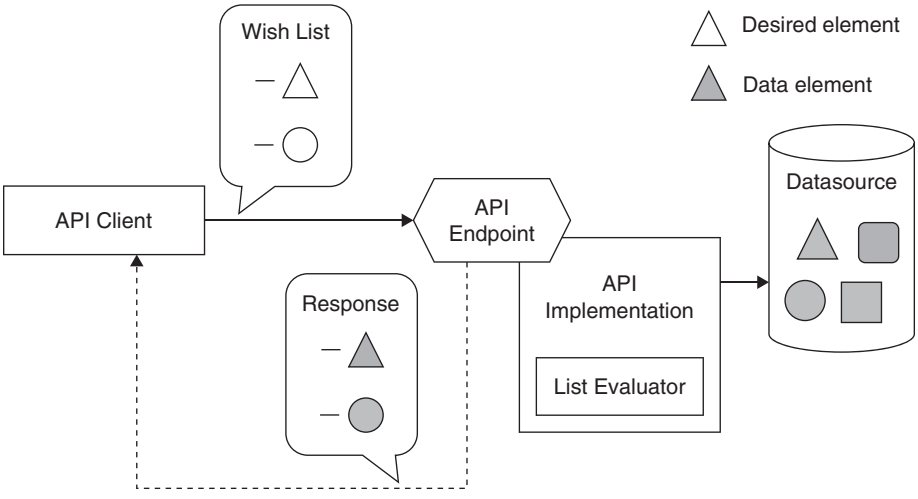


Figure 7.5 *WISH LIST: A client enumerates the desired data elements of the resource*

The List Evaluator in the figure has two implementation options. It often is translated to a filter for the data source so that only relevant data is loaded. Alternatively, the API implementation can fetch a full result set from the data source and select the entities that appear in the client wish when assembling the response data. Note that the data source can be any kind of backend system, possibly remote, or database. For instance, the wish translates into a WHERE clause of a SQL query when the data source is a relational database. If a remote system is accessed via an API, the WISH LIST might simply be passed on after having been validated (assuming that the downstream API also supports the pattern).

Variants A common variant is to provide options for *expansion* in responses. The response to the first request provides only a terse result with a list of parameters that can be expanded in subsequent requests. To expand the request results, the client can select one or more of these parameters in the WISH LIST of a follow-on request.

Another variant is to define and support a wildcard mechanism, as known from SQL and other query languages. For instance, a star `*` might request all data elements of a particular resource (which could then be the default if no wishes are specified). Even more complex schemes are possible, such as cascaded specifications (for example, `customer.*` fetching all data about the customer).

Example

In the Lakeside Mutual Customer Core application, a request for a customer returns all of its available attributes.

```
curl -X GET http://localhost:8080/customers/gktlipwhjr
```

For customer ID `gktlipwhjr`, this would return the following:

```
{
  "customerId": "gktlipwhjr",
  "firstname": "Max",
  "lastname": "Mustermann",
  "birthday": "1989-12-31T23:00:00.000+0000",
  "streetAddress": "Oberseestrasse 10",
  "postalCode": "8640",
  "city": "Rapperswil",
  "email": "admin@example.com",
  "phoneNumber": "055 222 4111",
  "moveHistory": [ ],
  "customerInteractionLog": {
    "contactHistory": [ ],
    "classification": {
      "priority": "gold"
    }
  }
}
```

To improve this design, a Wish List in the query string can restrict the result to the fields included in the wish. In the example, an API client might be interested in only the `customerId`, `birthday`, and `postalCode`:

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

The returned response now contains only the requested fields:

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

This response is much smaller; only the information required by the client is transmitted.

Discussion

Wish List helps manage the different information needs of API clients. It is well suited if the network has limited capacity and there is a certain amount of confidence that clients usually require only a subset of the available data. The potential negative consequences include additional security threats, additional complexity, as well as test and maintenance efforts. Before introducing a Wish List mechanism, these negative consequences must be considered carefully. Often, they are treated as an afterthought, and mitigating them can lead to maintenance and evolution problems once the API is in production.

By adding or not adding attribute values in the Wish List instance, the API client expresses its wishes to the provider; hence, the desire for data parsimony (or *Datensparsamkeit*) is met. The provider does not have to supply specialized and optimized versions of operations for certain clients or to guess data required for client use cases. Clients can specify the data they require, thereby enhancing performance by creating less database and network load.

Providers have to implement more logic in their service layers, possibly affecting other layers down to data access as well. Providers risk exposing their data model to clients, increasing coupling. Clients have to create the Wish List, the network has to transport this metadata, and the provider has to process it.

A comma-separated list of attribute names can lead to problems when mapped to programming language elements. For instance, misspelling an attribute name might lead to an error (if the API client is lucky), or the expressed wish might be ignored (which might lead the API client to the impression that the attribute does not exist). Furthermore, API changes might have unexpected consequences; for instance, a renamed attribute might no longer be found if clients do not modify their wishes accordingly.

Solutions using the more complex variants introduced earlier (such as cascaded specifications, wildcards, or expansion) might be harder to understand and build than simpler alternatives. Sometimes existing provider-internal search-and-filter capabilities such as wildcards or regular expressions can be reused.

This pattern (or, more generally speaking, all patterns and practices sharing this common goal and theme of client-driven message content) is also known as *response shaping*.

Related Patterns

Wish Template addresses the same problem as Wish List but uses a possibly nested structure to express the wishes rather than a flat list of element names. Both Wish List and Wish Template usually deal with Parameter Trees in response messages because patterns to reduce message sizes are particularly useful when dealing with complex response data structures.

Using a Wish List has a positive influence on sticking to a Rate Limit, as less data is transferred when the pattern is used. To reduce the transferred data further, it can be combined with Conditional Request.

The Pagination pattern also reduces response message sizes by splitting large repetitive responses into parts. The two patterns can be combined.

More Information

Regular expression syntax or query languages such as XPath (for XML payloads) can be seen as an advanced variant of this pattern. GraphQL [GraphQL 2021] offers a declarative query language to describe the representation to be retrieved against an agreed-upon schema found in the API documentation. We cover GraphQL in more detail in the Wish Template pattern.

Web API Design: The Missing Link [Apigee 2018] recommends comma-separated Wish Lists in its chapter "More on Representation Design." James Higginbotham features this pattern as "Zoom-Embed" [Higginbotham 2018].

"Practical API Design at Netflix, Part 1: Using Protobuf FieldMask" in the Netflix Technology Blog [Borysov 2021] mentions GraphQL field selectors and sparse fieldsets in JSON:API [JSON API 2022]. It then features Protocol Buffer `FieldMask` as a solution for gRPC APIs within the Netflix Studio Engineering. The authors suggest that API providers may ship client libraries with prebuilt `FieldMask` for the most frequently used combinations of fields. This makes sense if multiple consumers are interested in the same subset of fields.

Pattern: Wish Template

When and Why to Apply

An API provider has to serve multiple different clients that invoke the same operations. Not all clients have the same information needs: some might need just a subset of the data offered by the endpoint; other clients might need rich, deeply structured data sets.

How can an API client inform the API provider about nested data that it is interested in? How can such preferences be expressed flexibly and dynamically?[3]

3. Note that this problem is very similar to the problem of the pattern Wish List but adds the theme of response data nesting.

An API provider that has multiple clients with different information might simply expose a complex data structure that represents the superset (or union) of what the client community wants (for example, all attributes of master data such as product or customer information or collections of operational data entities such as purchase order items). Very likely, this structure becomes increasingly complex as the API evolves. Such a one-size-fits-all approach also costs performance (response time, throughput) and introduces security threats.

Alternatively, one could use a flat WISH LIST that simply enumerates desired attributes, but such a simple approach has limited expressiveness when dealing with nested data structures.

Network-level and application-level gateways and proxies can be introduced to improve performance, for instance, by caching. Such responses to performance issues add to the complexity of the deployment model and network topology and come with design and configuration effort.

How It Works

Add one or more additional parameters to the request message that mirror the hierarchical structure of the parameters in the corresponding response message. Make these parameters optional or use Boolean as their types so that their values indicate whether or not a parameter should be included.

The structure of the wish that mirrors the response message often is a PARAMETER TREE. API clients can populate instances of this WISH TEMPLATE parameter with empty, sample, or dummy values when sending a request message or set its Boolean value to true to indicate their interest in it. The API provider then uses the mirrored structure of the wish as a template for the response and substitutes the requested values with actual response data. Figure 7.6 illustrates this design.

The Template Processor in the figure has two implementation options, depending on the chosen template format. If a mirror object is already received from the wire and structured as a PARAMETER TREE, this data structure can be traversed to prepare the data source retrieval (or to extract relevant parts from the result set). Alternatively, the templates may come in the form of a declarative query, which must be evaluated first and then translated to a database query or a filter to be applied to the fetched data (these two options are similar to those in the List Evaluator component of a WISH LIST processor shown in Figure 7.5). The evaluation of the template instance can be straightforward and supported by libraries or language concepts in the API implementation (for instance, navigating through nested JSON objects with JSONPath, XML documents with XPath, or matching a regular expression).

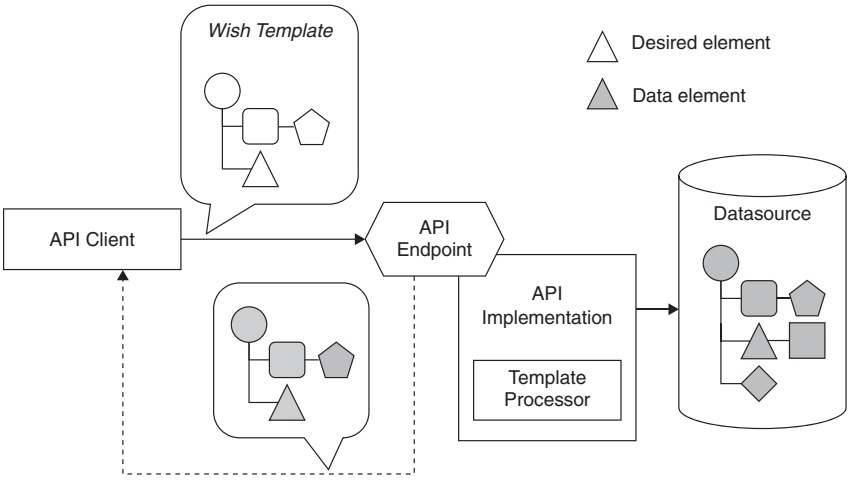


Figure 7.6 *Wish Template components and processing steps*

For complex template syntaxes constituting a domain-specific language, the introduction of compiler concepts such as scanning and parsing might be necessary.

Figure 7.7 shows the matching input and output parameter structure for two top-level fields, aValue and aString, and a nested child object that also has two fields, aFlag and aSecondString. The output parameters (or response message elements) have integer and string types, and the mirror in the request message specifies matching Boolean values. Setting the Boolean to true indicates interest in the data.

Example

The following MDSL service contract sketch introduces a `<<Wish_Template>>` highlighted with a stereotype:

```
data type PersonalData P // unspecified, placeholder
data type Address P // unspecified, placeholder
data type CustomerEntity <<Entity>> {PersonalData?, Address?}

endpoint type CustomerInformationHolderService
  exposes
    operation getCustomerAttributes
      expecting payload {
        "customerId":ID, // the customer ID
        <<Wish_Template>>"mockObject":CustomerEntity
        // has same structure as desired result set
      }
      delivering payload CustomerEntity
```

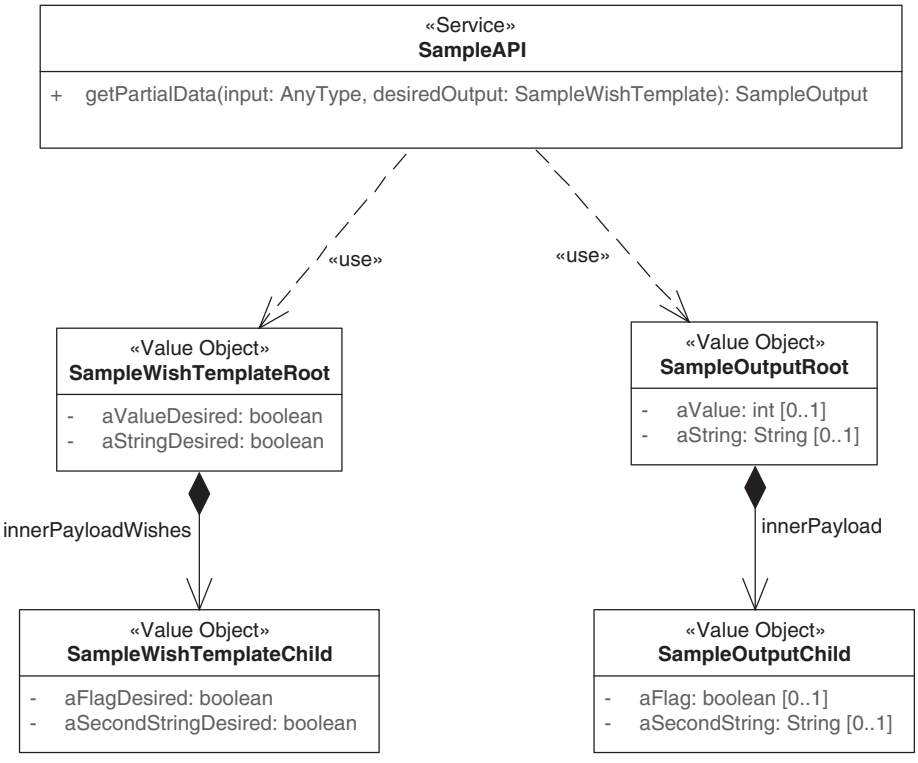


Figure 7.7 *Possible structure of mock/mirror object (WISH TEMPLATE)*

In this example of an API, the client can send a `CustomerEntity` mirror (or mock) object that may include `PersonalData` and/or `Address` attributes (this is defined in the `data type` definition `CustomerEntity`). The provider can then check which attributes were sent (ignoring the dummy values in the wish) and respond with a filled-out `CustomerEntity` instance containing `PersonalData` and/or `Address`.

Discussion

Data parsimony (or *Datensparsamkeit*) is an important general design principle in distributed systems that are performance- and security-critical. However, this principle is not always applied when iteratively and incrementally defining an API endpoint: it is typically easier to add things (here, information items or attributes) than to remove them. That is, once something is added to an API, it is often hard to determine whether it can be safely removed in a backward-compatible way (without breaking changes, that is) as many (maybe even unknown) clients might depend on it. By specifying selected attribute values in the WISH TEMPLATE instance and filling it

with marker values or Boolean flags, the consumer expresses its wishes to the provider; thus, the desire for data parsimony and flexibility is met.

When implementing this pattern, several decisions have to be made, including how to represent and populate the template. The sibling pattern Wish List mentions a comma-separated list of wishes as one approach, but the Parameter Trees forming the Wish Template are more elaborate and therefore require encoding and syntactic analysis. While highly sophisticated template notations might improve the developer experience on the client side and performance significantly, they also run the risk of turning into a larger, rather complex piece of middleware embedded into the API implementation (which comes with development, test, and maintenance effort as well as technical risk).

Another issue is how to handle errors for wishes that cannot be fulfilled, for example, because the client specified an invalid parameter. One approach could be to ignore the parameter silently, but this might hide real problems, for instance, if there was a typo or the name of a parameter changed.

The pattern is applicable not only when designing APIs around business capabilities but also when working with more IT-infrastructure-related domains such as software-defined networks, virtualization containers, or big data analysis. Such domains and software solutions for them typically have rich domain models and many configuration options. Dealing with the resulting variability justifies a flexible approach to API design and information retrieval.

GraphQL, with its type system, introspection, and validation capabilities, as well as its resolver concept can be seen as an advanced realization of this pattern [GraphQL 2021]. The Wish Templates of GraphQL are the query and mutation schemas providing declarative descriptions of the client wants and needs. Note that the adoption of GraphQL requires the implementation of a GraphQL server (effectively realizing the Template Processor in Figure 7.6). This server is a particular type of API endpoint located on top of the actual API endpoints (which become resolvers in GraphQL terms). This server has to parse the declarative description of queries and mutations and then call one or more resolvers, which in turn may call additional ones when following the data structure hierarchy.

Related Patterns

Wish List addresses the same problem but uses a flat enumeration rather than a mock/template object; both these patterns deal with instances of Parameter Tree in response messages. The Wish Template becomes part of a Parameter Tree that appears in the request message.

Wish Template shares many characteristics with its sibling pattern Wish List. For instance, without client- and provider-side data contract validation against a

schema (XSD, JSON Schema), Wish Template has the same drawbacks as the simple enumeration approach described in the Wish List pattern. Wish Templates can become more complex to specify and understand than simple lists of wishes; schemas and validators are usually not required for simple lists of wishes. Provider developers must be aware that complex wishes with deep nesting can strain and stress the communication infrastructure.[4] Processing can then also get more complex. Accepting the additional effort, as well as the complexity added to the parameter data definitions and their processing, only makes sense if simpler structures like Wish Lists cannot express the wish adequately.

Using a Wish Template has a positive influence on a Rate Limit, as less data is transferred when the pattern is used and fewer requests are required.

More Information

In "You Might Not Need GraphQL," Phil Sturgeon shows several APIs that implement response shaping and how they correspond to related GraphQL concepts [Sturgeon 2017].

Message Exchange Optimization (aka Conversation Efficiency)

The previous section offered patterns that allow clients to specify the partition of large data sets and which individual data points they are interested in. This lets API providers and clients avoid unnecessary data transfers and requests. But maybe the client already has a copy of the data and does not want to receive the same data again. Or they might have to send many individual requests that cause transmission and processing overhead. The patterns described here provide solutions to these two issues and try to balance the following common forces:

- **Complexity of endpoint, client, and message payload design and programming:** The additional effort needed to implement and operate a more complex API endpoint that takes data update frequency characteristics into account needs to be balanced against the expected reduction in endpoint processing and bandwidth usage. Reducing the number of requests does not imply that

4. Olaf Hartig and Jorge Pérez analyzed the performance of the GitHub GraphQL API and found an "exponential increase in result sizes" as they increased the query level depth. The API timed out on queries with nesting levels higher than 5 [Hartig 2018].

less information is exchanged. Hence, the remaining messages have to carry more complex payloads.

- **Accuracy of reporting and billing:** Reporting and billing of API usage must be accurate and should be perceived as being fair. A solution that burdens the client with additional work (for instance, keeping track of which version of data it has) to reduce the provider's workload might require some incentive from the provider. This additional complexity in the client-provider conversation might also have an impact on the accounting of API calls.

The two patterns responding to these forces are Conditional Request and Request Bundle.

Pattern: Conditional Request

When and Why to Apply

Some clients keep on requesting the same server-side data repeatedly. This data does not change between requests.

How can unnecessary server-side processing and bandwidth usage be avoided when invoking API operations that return rarely changing data?

In addition to the challenges introduced at the beginning of this section, the following forces apply:

- **Size of messages:** If network bandwidth or endpoint processing power is limited, retransmitting large responses that the client already has received is wasteful.
- **Client workload:** Clients may want to learn whether the result of an operation has changed since their last invocation in order to avoid reprocessing the same results. This reduces their workload.
- **Provider workload:** Some requests are rather inexpensive to answer, such as those not involving complex processing, external database queries, or other backend calls. Any additional runtime complexity of the API endpoint, for instance, any decision logic introduced to avoid unnecessary calls, might negate the possible savings in such cases.

- **Data currentness versus correctness:** API clients might want to cache a local copy of the data to reduce the number of API calls. As copy holders, they must decide when to refresh their caches to avoid stale data. The same considerations apply to metadata. On the one hand, when data changes, chances are that metadata about it has to change too. On the other hand, the data could remain the same, and only the metadata might change. Attempts to make conversations more efficient must take these considerations into account.

One might consider scaling up or scaling out on the physical deployment level to achieve the desired performance, but such an approach has its limits and is costly. The API provider or an intermediary API gateway might cache previously requested data to serve them quickly without having to recreate or fetch them from the database or a backend service. Such dedicated caches have to be kept current and invalidated at times, which leads to a complex set of design problems.[5]

In an alternative design, the client could send a "preflight" or "look before you leap" request asking the provider if anything has changed before sending the actual request. But this design doubles the number of requests, makes the client implementation more complex, and might reduce client performance when the network has a high latency.

How It Works

Make requests conditional by adding Metadata Elements to their message representations (or protocol headers) and processing these requests only if the condition specified by the metadata is met.

If the condition is not met, the provider does not reply with a full response but returns a special status code instead. Clients can then use the previously cached value. In the simplest case, the conditions represented by Metadata Elements could be transferred in an Atomic Parameter. Application-specific data version numbers or timestamps can be used if already present in the request.

5. As Phil Karlton (quoted by Martin Fowler) notes, "There are only two hard things in Computer Science: cache invalidation and naming things" [Fowler 2009]. Fowler provides tongue-in-cheek evidence for this claim.

Figure 7.8 illustrates the solution elements.

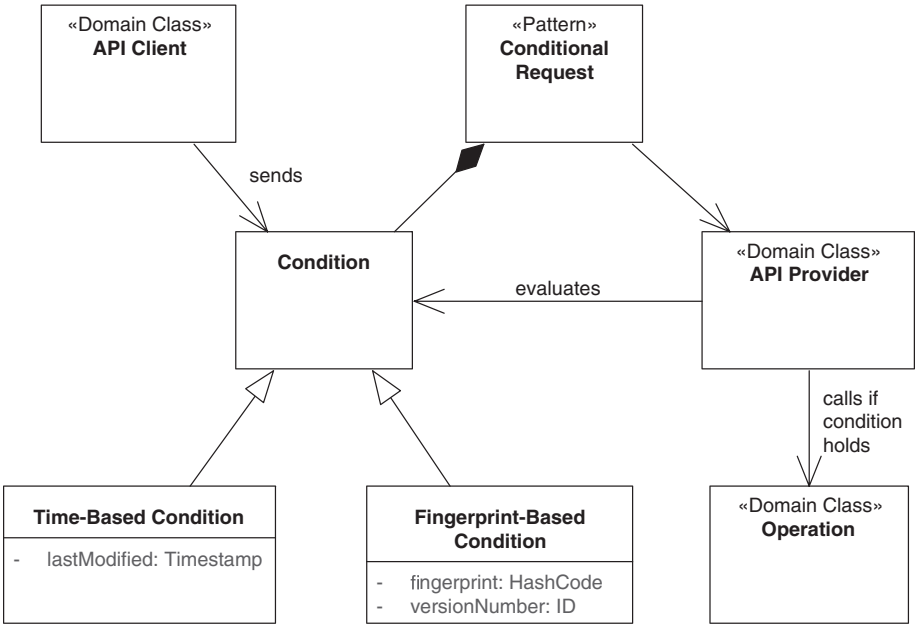


Figure 7.8 *Conditional Request*

It is also possible to implement the Conditional Request pattern within the communication infrastructure, orthogonal and complementary to the application-specific content. To do so, the provider may include a hash of the data served. The client can then include this hash in subsequent requests to indicate which version of the data it already has and for which data it wishes to receive only newer versions. A special `condition violated` response is returned instead of the complete response if the condition is not met. This approach implements a "virtual caching" strategy, allowing clients to recycle previously retrieved responses (assuming they have kept a copy).

Variants Request conditions can take different forms, leading to different variants of this pattern:

- *Time-Based Conditional Request:* Resources are timestamped with a `last-modified` date. A client can use this timestamp in the subsequent requests so that the server will reply with a resource representation only if it is newer than the copy the client already has. Note that this approach requires some clock synchronization between clients and servers if it is supposed to work accurately (which might not always be required). In HTTP, the `If-Modified-Since` request header carries such a timestamp, and the `304 Not Modified` status code is used to indicate that a newer version is not available.

- *Fingerprint-Based Conditional Request:* Resources are tagged, that is, fingerprinted, by the provider, using, for example, a hash function applied to the response body or some version number. Clients can then include the fingerprint to indicate the version of the data they already have. In HTTP, the entity tag (`ETag`), as described in RFC 7232 [Fielding 2014a], serves that purpose together with the `If-None-Match` request header and the previously mentioned `304 Not Modified` status code.

Example

Many Web application frameworks, such as Spring, support Conditional Requests natively. The Spring-based Customer Core backend application in the Lakeside Mutual scenario includes `ETags`—implementing the fingerprint-based Conditional Request variant—in all responses. For example, consider retrieving a customer:

```
curl -X GET --include \
http:://localhost:8080/customers/gktlipwhjr
```

A response containing an `ETag` header could start with:

```
HTTP/1.1 200
ETag: "0c2c09ecd1ed498aa7d07a516a0e56ebc"
Content-Type: application/hal+json;charset=UTF-8
Content-Length: 801
Date: Wed, 20 Jun 2018 05:36:39 GMT
{
  "customerId": "gktlipwhjr",
...
```

Subsequent requests can then include the `ETag` received from the provider previously to make the request conditional:

```
curl -X GET --include --header \
'If-None-Match: "0c2c09ecd1ed498aa7d07a516a0e56ebc"' \
http://localhost:8080/customers/gktlipwhjr
```

If the entity has not changed, that is, `If-None-Match` occurs, the provider answers with a `304 Not Modified` response including the same `ETag`:

```
HTTP/1.1 304
ETag: "0c2c09ecd1ed498aa7d07a516a0e56ebc"
Date: Wed, 20 Jun 2018 05:47:11 GMT
```

If the customer has changed, the client will get the full response, including a new `ETag`, as shown in Figure 7.9.

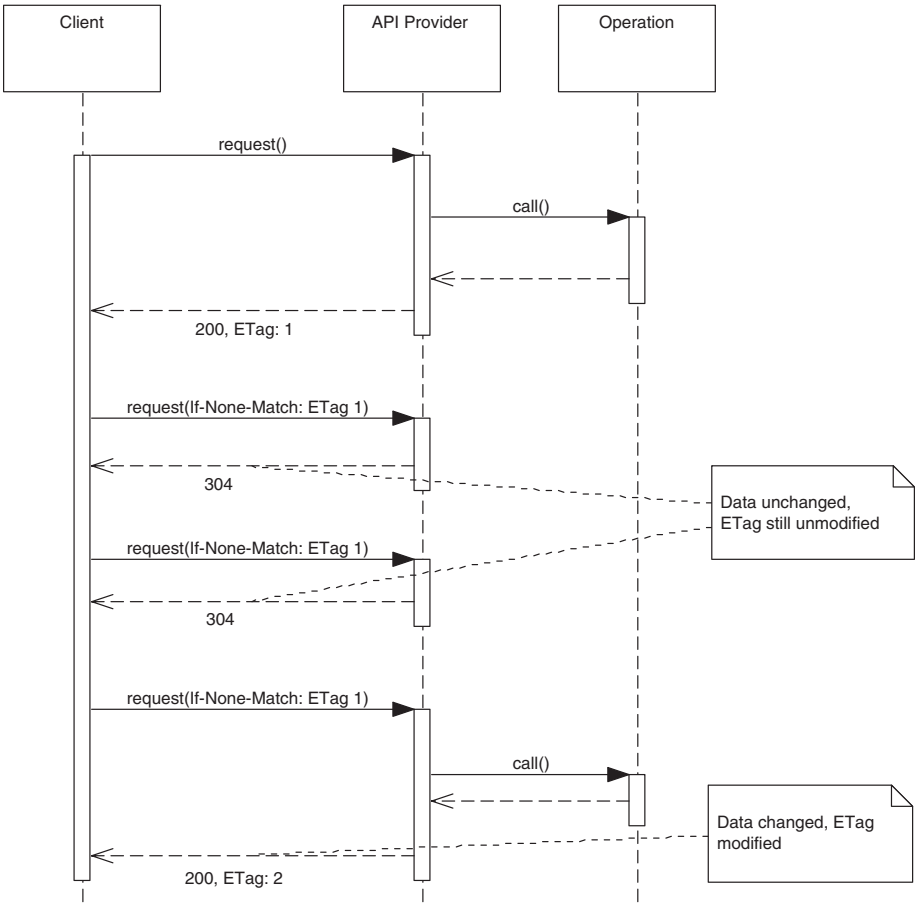


Figure 7.9 *Conditional Request example*

Note that the Customer Core microservice implements Conditional Request as a filter applied to the response. Using a filter means that the response is still computed but then is discarded by the filter and replaced with the `304 Not Modified` status code. This approach has the benefit of being transparent to the endpoint implementation; however, it only saves bandwidth and not computation time. A server-side cache could be used to minimize the computational time as well.

Discussion

Conditional Requests allow both clients and API providers to save bandwidth without assuming that providers remember whether a given client has already seen the latest version of the requested data. It is up to the clients to remind the server about their latest known version of the data. They cache previous responses and are

responsible for keeping track of their timestamp or fingerprint and resending this information along with their next requests. This simplifies the configuration of the *data currentness interval*. Timestamps, as one way to specify the data currentness interval, are simple to implement even in distributed systems as long as only one system writes the data. The time of this system is the master time in that case.

The complexity of the provider-side API endpoint does not increase if the pattern is implemented with a filter, as shown in the preceding example. Further improvements, such as additional caching of responses, can be realized for specific endpoints to reduce provider workload. This increases the complexity of the endpoint, as they have to evaluate the conditions, filters, and exceptions, including errors that might occur because of the condition handling or filtering.

Providers also have to decide how Conditional Requests affect other quality measures such as a Rate Limit and whether such requests require special treatment in a Pricing Plan.

Clients can choose whether or not to make use of Conditional Requests, depending on their performance requirements. Another selection criterion is whether clients can afford to rely on the server to detect whether the state of the API resources has changed. The number of messages transmitted does not change with Conditional Requests, but the payload size can be reduced significantly. Rereading an old response from the client cache is usually much faster than reloading it from the API provider.

Related Patterns

Using a Conditional Request may have a positive influence on a Rate Limit that includes response data volumes in the definition of the limit, as less data is transferred when this pattern is used.

The pattern can be carefully combined with either Wish List or Wish Template. This combination can be rather useful to indicate the subset of data that is to be returned if the condition evaluates to `true` and the data needs to be sent (again).

A combination of Conditional Requests with Pagination is possible, but there are edge cases to be considered. For example, the data of a particular page might not have changed, but more data was added, and the total number of pages has increased. Such a change in metadata should also be included when evaluating the condition.

More Information

Chapter 10 in the *RESTful Web Services Cookbook* [Allamaraju 2010] is dedicated to conditional requests. Some of the nine recipes in this chapter even deal with requests that modify data.

Pattern: Request Bundle

When and Why to Apply

An API endpoint that exposes one or more operations has been specified. The API provider observes that clients make many small, independent requests; individual responses are returned for these requests. These chatty interaction sequences harm scalability and throughput.

How can the number of requests and responses be reduced to increase communication efficiency?

In addition to the general desire for efficient messaging and data parsimony (as discussed in the introduction to this chapter), the goal of this pattern is to improve performance:

- **Latency:** Reducing the number of API calls may improve client and provider performance, for instance, when the network has high latency or overhead is incurred by sending multiple requests and responses.
- **Throughput:** Exchanging the same information through fewer messages may lead to a higher throughput. However, the client has to wait longer until it can start working with the data.

One might consider using more or better hardware to meet the performance demands of the API clients, but such an approach has its physical limits and is costly.

How It Works

Define a Request Bundle as a data container that assembles multiple independent requests in a single request message. Add metadata such as identifiers of individual requests (bundle elements) and a bundle element counter.

There are two options to design the response messages:

1. One request with one response: Request Bundle with a *single bundled response*.
2. One request with multiple responses: Request Bundle with *multiple responses*.

The Request Bundle container message can, for instance, be structured as a Parameter Tree or a Parameter Forest. In the first option, a message structure for the response container that mirrors the request assembly and corresponds to the bundled requests has to be defined. The second option can be implemented with support from the underlying network protocols to support suitable message exchange and conversation patterns. For example, with HTTP, the provider can delay the response until a bundle item has been processed. RFC 6202 [Saint-Andre 2011] presents details on this technique, which is called *long polling*.

Errors have to be handled both individually and on the container level. Different options exist; for instance, an Error Report for the entire batch can be combined with an associative array of individual Error Reports for bundle elements accessible via Id Elements.

Figure 7.10 shows a Request Bundle of three individual requests, A, B, and C, assembled into a single remote API call. Here, a single bundled response is used (Option 1).

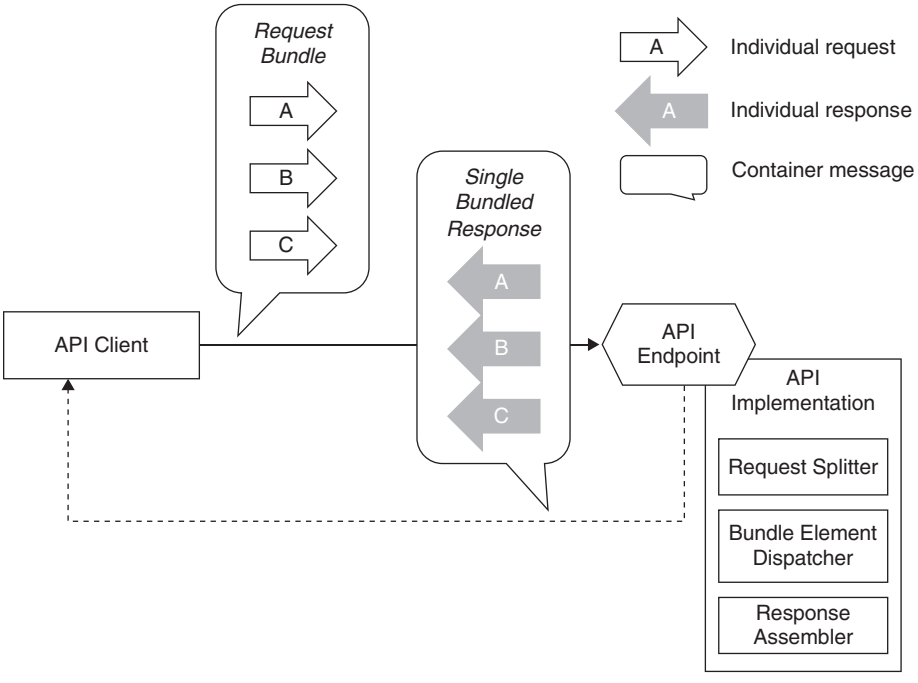


Figure 7.10 *Request Bundle: Three independent requests, A, B, and C, are assembled in a container message. The provider processes the requests and replies with a Single Bundled Response*

The API implementation has to split the request bundle and assemble the response bundle. This can be as straightforward as iterating through an array that the provider-side endpoint hands over, but it also may require some additional decision and dispatch logic, for instance, using a control Metadata Element in the request to decide where in the API implementation to route the bundle elements to. The API client has to split a bundled response in a similar way if the provider returns a single bundled response.

Example

In the Lakeside Mutual Customer Core service, clients can request multiple customers from the customer's Information Holder Resource by specifying an Atomic Parameter List of customer Id Elements. A path parameter serves as a bundle container. A comma (`,`) separates the bundle elements:

```
curl -X GET http://localhost:8080/customers/ce4btlyluu,rgpp0wkpec
```

This will return the two requested customers as Data Elements, represented as JSON objects in a bundle-level array (using the single bundled response option):

```
{
  "customers": [
    {
      "customerId": "ce4btlyluu",
      "firstname": "Robbie",
      "lastname": "Davenhall",
      "birthday": "1961-08-11T23:00:00.000+0000",
      ...
      "_links": { ... }
    },
    {
      "customerId": "rgpp0wkpec",
      "firstname": "Max",
      "lastname": "Mustermann",
      "birthday": "1989-12-31T23:00:00.000+0000",
      ...
      "_links": { ... }
    }
  ],
  "_links": { ... }
}
```

This example implements the pattern option Request Bundle with single bundled response, introduced earlier.

Discussion

By transmitting a bundle of requests at once, the number of messages can be reduced significantly if the client-side usage scenarios include batch or bulk processing (for instance, periodic updates to customer master data). As a consequence, the communication is sped up because less network communication is required. Depending on the actual use case, client implementation effort might also decrease because the client does not have to keep track of multiple ongoing requests. It can process all logically independent bundle elements found in a single response one by one.

The pattern adds to endpoint processing effort and complexity. Providers have to split the request messages and, when realizing Request Bundle with multiple responses, coordinate multiple individual responses. Client processing effort and complexity can increase as well because clients must deal with the Request Bundle and its independent elements, again requiring a splitting strategy. Finally, the message payload design and processing get more complex, as data from multiple sources has to be merged into one message.

Being independent of each other, individual requests in the Request Bundle might be executed concurrently by the endpoint. Hence, the client should not make any assumptions about the order of evaluation of the requests. API providers should document this container property in the API Description. Guaranteeing a particular order of bundle elements causes extra work, for instance ordering a single bundled response in the same way as the incoming Request Bundle.

The pattern is eligible if the underlying communication protocol cannot handle multiple requests at once. It assumes that data access controls are sufficiently defined and presented so that all bundle elements are allowed to be processed. If not, the provider must compose partial responses indicating to the client which commands/requests in the bundle failed and how to correct the corresponding input so that invocation can be retried. Such element-level access control can be challenging to handle on the client side.

Clients must wait until all messages in the bundle have been processed, increasing the overall latency until a first response is received; however, compared to many consecutive calls, the total communication time typically speeds up, as less network communication is required. The coordination effort might make the service provider stateful, which is considered harmful in microservices and cloud environments due to its negative impact on scalability. That is, it becomes more difficult to scale out horizontally when workload increases because the microservices middleware or the cloud provider infrastructure may contain load balancers that now have to make sure that subsequent requests reach the right instances and that failover procedures recreate state in a suited fashion. It is not obvious whether the bundle or its elements should be the units of scaling.

Related Patterns

The request and response messages of a Request Bundle form Parameter Forests or Parameter Trees. Additional information about the structure and information that identifies individual requests comes as one or more Id Elements or Metadata Elements. Such identifiers might realize the "Correlation Identifier" pattern [Hohpe 2003] to trace responses back to requests.

A Request Bundle can be delivered as a Conditional Request. The pattern can also be combined with a Wish List or a Wish Template. It must be carefully analyzed if enough gains can be realized to warrant the complexity of a combination of two or even three of those patterns. If the requested entities are of the same kind (for instance, several people in an address book are requested), Pagination and its variants can be applied instead of Request Bundle.

Using a Request Bundle has a positive influence on staying within a Rate Limit that counts operation invocations because fewer messages are exchanged when the pattern is used. This pattern goes well with explicit Error Reports because it is often desirable to report the error status or success per bundle element and not only for the entire Request Bundle.

Request Bundle can be seen as an extension of the general "Command" design pattern: each individual request is a command according to terminology from [Gamma 1995]. "Message Sequence" [Hohpe 2003] solves the opposite problem: to reduce the message size, messages are split into smaller ones and tagged with a sequence ID. The price for this is a higher number of messages.

More Information

Recipe 13 in Chapter 11 of the *RESTful Web Services Cookbook* [Allamaraju 2010] advises against providing a generalized endpoint to tunnel multiple individual requests.

Coroutines can improve performance when applying the Request Bundle pattern in the context of batch processing (aka chunking). "Improving Batch Performance when Migrating to Microservices with Chunking and Coroutines" discusses this option in detail [Knoche 2019].

Summary

This chapter presented patterns concerned with API quality, specifically, finding the sweet spot between API design granularity, runtime performance, and the ability to support many diverse clients. It investigated whether many small or few large messages should be exchanged.

Applying the Embedded Entity pattern makes the API exchange self-contained. Linked Information Holder leads to smaller messages that can refer to other API endpoints and, therefore, will lead to multiple round-trips to retrieve the same information.

Pagination lets clients retrieve data sets piecewise, depending on their information needs. If the exact selection of details to be fetched is not known at design time, and clients would like the API to satisfy all of their desires, then Wish Lists and Wish Templates offer the required flexibility.

Bulk messages in a Request Bundle require only one interaction. While performance can be carefully optimized by sending and receiving payloads with the right granularity, it also helps to introduce Conditional Requests and avoid resending the same information to clients who already have it.

Note that performance is hard to predict in general and in distributed systems in particular. Typically, it is measured under steady conditions as a system landscape evolves; if a performance control shows a negative trend that runs the risk of violating one or more formally specified Service Level Agreements or other specified runtime quality policies, the API design and its implementation should be revised. This is a broad set of important issues for all distributed systems; it becomes even more severe when a system is decomposed into small parts, such as microservices to be scaled and evolved independently of each other. Even when services are loosely coupled, the performance budget for meeting the response-time requirements of an end user performing a particular business-level function can be evaluated only as a whole and end-to-end. Commercial products and open-source software for load/performance testing and monitoring exist. Challenges include the effort required to set up an environment that has the potential to produce meaningful, reproducible results as well as the ability to cope with change (of requirements, system architectures, and their implementations). Simulating performance is another option. There is a large body of academic work on predictive performance modeling of software systems and software architectures (for example, "The Palladio-Bench for Modeling and Simulating Software Architectures" [Heinrich 2018]).

Next up is API evolution, including approaches to versioning and life-cycle management (Chapter 8, "Evolve APIs").

This page intentionally left blank

Index

C

D

M-N

O

P

Q

R

S

T

U

V

W

X-Y-Z