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

"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

Chapter 4

Pattern Language Introduction

In Part 1, we learned that remote APIs have become an important feature of modern distributed software systems. APIs provide integration interfaces exposing remote system functionality to end-user applications such as mobile clients, Web applications, and third-party systems. Not only end-user applications consume and rely on APIs—distributed backend systems and microservices within those systems require APIs to be able to work with each other as well.

Lakeside Mutual, a fictitious insurance company, and its microservices-based applications provided us with an example case. We saw that API design and evolution involve many recurring design issues to resolve conflicting requirements and find appropriate trade-offs. Decision models for groups of related issues presented options and criteria to guide us through the required design work. Patterns appeared as alternative options in these decisions.

This chapter takes the next step. It starts with a pattern language overview and then proposes navigation paths through the language. It also introduces a first set of basic scoping and structuring patterns. Having read this chapter, you will be able to explain the scope of our pattern language (in terms of topics covered and architectural concerns) and find patterns you are interested in (for instance, by project phase). You will also be able to characterize the API under construction by its visibility and integration type by way of foundation patterns and know about the basic structure patterns that constitute the syntactic building blocks of request and response messages (and many of the other patterns in our language).

Positioning and Scope

According to our domain model, established in Chapter 1, "Application Programming Interface (API) Fundamentals," API clients and providers exchange request and response messages to call operations in API endpoints. Many of our patterns focus on the payload *content* of such messages that contain one or more representation elements, possibly nested. *Enterprise Integration Patterns* [Hohpe 2003] offers three alternative patterns about this message content: "Document Message," "Command Message," and "Event Message." In messaging systems, such messages travel from the sending endpoint to the receiving endpoint over communication "Channels." These Channels may be offered by queue-based messaging systems but also come as HTTP connections or other use integration technologies, such as GraphQL and gRPC. Protocol capabilities and configuration, as well as message size and content structure, influence the quality properties of an API and its implementation. In this messaging context, APIs can be seen as "Service Activators" [Hohpe 2003]—viewed from the communication channels, they serve as "Adapters" [Gamma 1995] for the application services available in the API implementation.

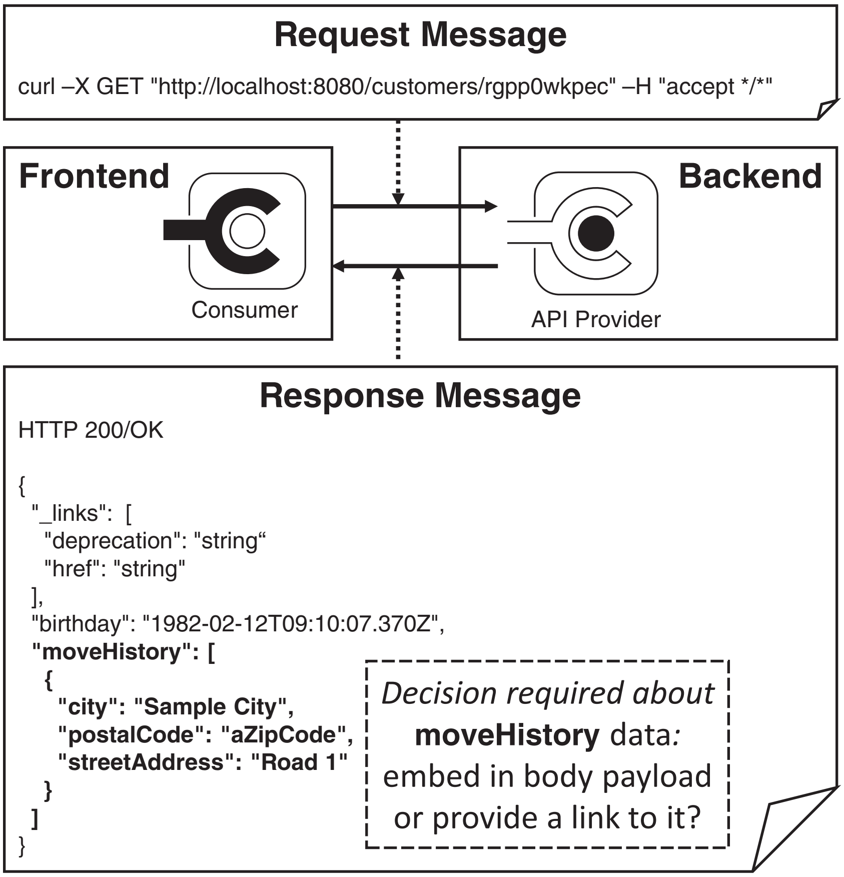
In our pattern language, we look into the command, document, and event messages in terms of their inner structures. We also investigate the roles played by representation elements, operations, and API endpoints—irrespective of the communication protocols used. We discuss how to group messages into endpoints to achieve suitable API granularity and coupling, how to document APIs, and how to manage the evolution of API endpoints and their parts.

We are particularly interested in message payloads that are exchanged as JSON objects—for instance, via HTTP GET, POST, and PUT—and in message queues offered by cloud providers or messaging systems (such as ActiveMQ or RabbitMQ). JSON is a popular message exchange format in Web APIs; our patterns work equally well when XML documents or other text structures are exchanged. They can even be applied to define the content of messages with binary encodings.

Figure 4.1 visualizes the scope of our patterns in a Web API example. An HTTP GET, shown as a curl command, asks for information about a single customer, `rgpp0wkpec`, of Lakeside Mutual (the case introduced in Chapter 2, "Lakeside Mutual Case Study").

The exemplary response message is nested: the customer information contains not only the birthday but also a log of address changes in the form of a `moveHistory`. Indicated by the JSON array notation `[...]`, a collection of relocation moves could be returned (in the example, the array contains only one move destination). Each move destination is characterized by three strings, `"city"`, `"postalCode"`, `"street-Address"`, wrapped in the JSON object notation `{...}` in the figure. This two-level structure raises an important, recurring API design issue:

Should complex data whose parts have containment or other domain-level relations be embedded in message representations, or should links be provided to look up this data with separate calls to other operations in the same (or other) API endpoints?


Request Message
Frontend
Consumer
Backend
API Provider
Response Message
Decision required about **moveHistory** data: embed in body payload or provide a link to it?

```
curl –X GET "http://localhost:8080/customers/rgpp0wkpec" –H "accept */*"
```

```
HTTP 200/OK

{
  "_links":  [
    "deprecation": "string“
    "href": "string"
  ],
  "birthday": "1982-02-12T09:10:07.370Z",
  "moveHistory": [
    {
      "city": "Sample City",
      "postalCode": "aZipCode",
      "streetAddress": "Road 1"
    }
  ]
}
```

Figure 4.1 *Exemplary API call: Exchanged messages and their structure*

Two of our patterns offer alternative answers to this question: Embedded Entity (shown in Figure 4.1) and Linked Information Holder. Embedded Entity injects nested data representations into the payload, whereas a Linked Information Holder places hyperlinks in the payload. In the latter case, the client has to follow these hyperlinks to obtain the referenced data in subsequent requests to the endpoint location found in the links. The chosen combination of these two patterns has a strong impact on the API quality. For instance, message size and number of interactions influence both performance and changeability. Both patterns are valid choices, depending on network and endpoint capabilities, information needs and data access profiles of clients, backend location of the source data, and so on. These criteria, therefore, are pattern selection and adoption forces. We will come back to these patterns and their forces in Chapter 7, "Refine Message Design for Quality."

Patterns: Why and How?

Patterns can help resolve API design issues, presenting proven solutions to problems recurring in a particular context (here, API design and evolution). Patterns are platform-independent by definition, thus avoiding concept, technology, and vendor lock-in. They form a common language for a domain. Adequate pattern usage can make the designs that adopt them easier to understand, port, and evolve.

Each pattern text can be seen as a small, specialized, standalone article. These texts are structured according to a common template:

- *When and Why to Apply* establishes the context and preconditions for pattern eligibility, followed by a problem statement that specifies a design issue to be resolved. Different forces on the design explain why the problem is hard to solve. Architectural decision drivers and conflicting quality attributes are often referenced here; a nonsolution may also be pointed out.
- The *How It Works* section presents a conceptual, generalized solution to the design question from the problem statement that describes how the solution works and which variants (if any) we observed in practice.
- The *Example* section shows how the solution can be implemented in a concrete application context, for instance, when working with a particular technology set such as HTTP and JSON.
- The *Discussion* section explains to what extent the solution resolves the pattern forces; it may also include additional pros and cons and identify alternative solutions.
- The *Related Patterns* section points to the next patterns that become eligible and interesting once a particular one has been applied.
- Finally, additional pointers and references are given under *More Information*.

Coming back to our two exemplary patterns, Linked Information Holder and Embedded Entity are documented in this format in Chapter 7.

Note that using a pattern does not dictate a particular implementation but leaves a lot of flexibility for its project-context-specific adoption. In fact, patterns should never be followed blindly but should be seen as a tool or guide. A product- or project-specific design can satisfy its concrete, actual requirements only if it knows them (which is hard for a generalized artifact such as a pattern).

Navigating through the Patterns

When we decided how to organize our patterns, we looked at two other books for inspiration: *Enterprise Integration Patterns* [Hohpe 2003] is organized by the life cycle of messages traveling through a distributed system, from creation and sending to routing, transforming, and receiving. *Patterns of Enterprise Application Architecture* [Fowler 2002] uses logical layers as a chapter and topic breakdown, progressing from domain layer to persistence layer and presentation layer.

Regrettably, neither layers nor life cycles alone seemed to work well for the API domain. Hence, we could not decide on one best way to organize but offer multiple ones to guide you through the patterns: architectural scope (as defined by the API domain model from Chapter 1), topic categories, and refinement phases.[1]

Structural Organization: Find Patterns by Scope

Most of our patterns focus on API building blocks at different levels of abstraction and detail; some concern the API as a whole and its documentation, both technical and commercial. The resulting architectural scopes are API as a whole, endpoint, operation, and message. We introduced these basic concepts in the API domain model in Chapter 1. Figure 4.2 calls out patterns for these five scopes.

1. This "ask for one, get three" tactic is an exception to our general rule, "if in doubt, leave it out" [Zimmermann 2021b], fortunately only on the meta-level. Hopefully, standards committees and API designers stick to this rule better than we do ;-).

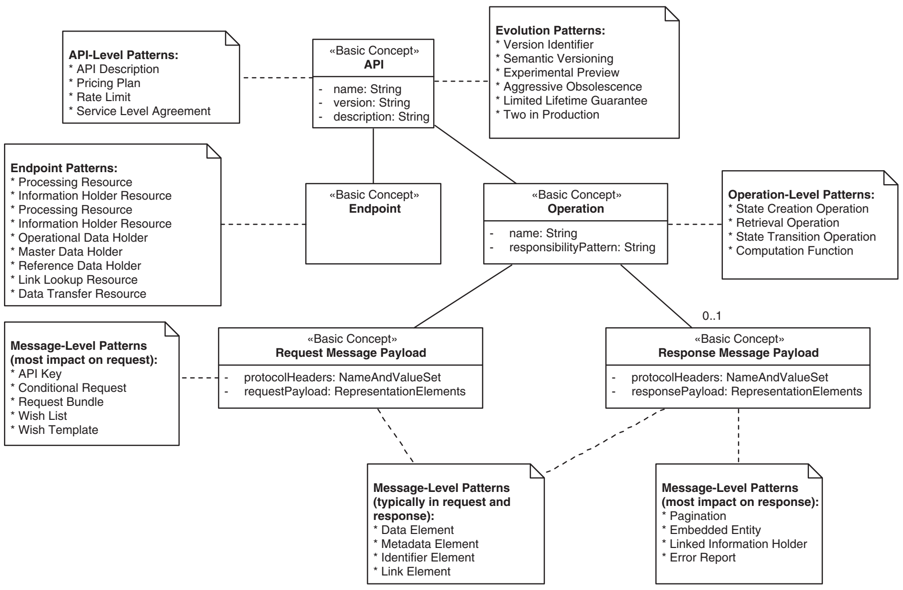

API-Level Patterns:
* API Description
* Pricing Plan
* Rate Limit
* Service Level Agreement
«Basic Concept» API
- name: String
- version: String
- description: String
Evolution Patterns:
* Version Identifier
* Semantic Versioning
* Experimental Preview
* Aggressive Obsolescence
* Limited Lifetime Guarantee
* Two in Production
Endpoint Patterns:
* Processing Resource
* Information Holder Resource
* Processing Resource
* Information Holder Resource
* Operational Data Holder
* Master Data Holder
* Reference Data Holder
* Link Lookup Resource
* Data Transfer Resource
«Basic Concept» Endpoint
«Basic Concept» Operation
- name: String
- responsibilityPattern: String
Operation-Level Patterns:
* State Creation Operation
* Retrieval Operation
* State Transition Operation
* Computation Function
0..1
Message-Level Patterns (most impact on request):
* API Key
* Conditional Request
* Request Bundle
* Wish List
* Wish Template
«Basic Concept» Request Message Payload
- protocolHeaders: NameAndValueSet
- requestPayload: RepresentationElements
«Basic Concept» Response Message Payload
- protocolHeaders: NameAndValueSet
- responsePayload: RepresentationElements
Message-Level Patterns (typically in request and response):
* Data Element
* Metadata Element
* Identifier Element
* Link Element
Message-Level Patterns (most impact on response):
* Pagination
* Embedded Entity
* Linked Information Holder
* Error Report

Figure 4.2 *Patterns by domain model element and architectural scope*

Patterns such as API Description and Service Level Agreement concern the API as a whole. Others, such as Processing Resource and Data Transfer Resource, operate on single endpoints. Many patterns deal with operation or message design; some of these primarily target request messages (API Key, Wish List), and others are more focused on response messages (Pagination, Error Report). Element stereotypes may appear both in requests and responses (Identifier Element, Metadata Element).

Call to action: When being confronted with an API design task, ask yourself which of these scopes you are about to deal with and refer to Figure 4.2 to find patterns of interest for this task.

Theme Categorization: Search for Topics

We grouped the patterns into five categories. Each category answers several related topical questions:

- **Foundation patterns:** Which types of systems and components are integrated? From where should an API be accessible? How should it be documented?