

The Design of Web APIs

SECOND EDITION

Arnaud Lauret



MANNING

The Design of Web APIs

SECOND EDITION

Arnaud Lauret



MANNING

The Design of Web APIs, Second Edition

1. [welcome](#)
2. [1 What is API design?](#)
3. [2 Analyzing needs](#)
4. [3 Observing operations from the REST angle](#)
5. [4 Representing operations with HTTP](#)
6. [5 Modeling data](#)
7. [index](#)

welcome

Thank you for purchasing *The Design of Web APIs, Second Edition*.

Web APIs are everywhere; we use them all the time, often without even realizing it. Whether sharing a photo on social media or hailing a ride through an app, web APIs are crucial in making it happen. For developers, APIs are essential as most modern systems rely on multiple software components communicating with each other. We need them to build simple web applications to complex distributed systems. APIs are also products in their own right, as exemplified by Stripe or Twilio. Even government agencies rely on APIs to power their digital services.

The design quality is crucial for web APIs, whether seen as technical interfaces or products, used by a single application or multiple, or created for internal use or third-party. Poorly designed public or private APIs can harm developers' productivity, system performance and integrity, end-users experience, and organization's revenue.

This book aims to help you develop an API designer's mindset and design exceptional web APIs, specifically REST APIs. In these chapters, we will explore the true nature of API design as both a result and a process. We will learn how to analyze and evaluate requirements to identify the API capabilities, discover HTTP and REST, and understand how to use them to represent these capabilities. We will discuss how to create interoperable and user-friendly APIs, ensuring that anyone can instantly use the API's data and operations. We will also learn how to integrate various constraints, especially security, into our design. Additionally, we will focus on handling modifications and preventing breaking an API design unintentionally or breaking it intentionally when it makes sense. Furthermore, we will learn how to become efficient API designers by learning various principles and recipes for making design decisions when faced with new problems. We will learn to convince others (and ourselves) that our design decisions are correct.

I am writing the second edition of this book to address what was not working

well in the first edition and expand it. I also reorganize the content to make it easier to follow and integrate new ideas and feedback received from readers. I am keeping the spirit of the first edition but rewriting everything. In a way, this is almost a new book. Your feedback is essential to make this new edition the best companion on your journey of API design; I hope you'll add your comments to the [Livebook discussion forum](#).

— Arnaud Lauret

In this book

[welcome](#) [1 What is API design?](#) [2 Analyzing needs](#) [3 Observing operations from the REST angle](#) [4 Representing operations with HTTP](#) [5 Modeling data](#)

1 What is API design?

This chapter covers

- Explaining what web APIs are
- Realizing the importance of the design of web APIs
- Clarifying who designs web APIs, which ones should be designed, and when
- Comprehending the purpose of web API design
- Overviewing the web API design process

What magic allows us to share pictures on social media, check bank balances, and hail cabs from our phones? How can developers quickly add telecom or payment services to their apps without expertise? What do basic web or mobile apps need to display data to users? Web application programming interfaces or APIs.

Web APIs are essential in our connected world as they serve as technical interfaces or products for various systems, organizations, and companies, from small startups to large corporations and government entities. API design is crucial for the success of any system, whether its APIs are visible or hidden. Poorly designed APIs can negatively impact developers' productivity, the system's performance and integrity, end-users experience, and revenue.

Learning web API design requires a shared understanding of API and design concepts. This chapter covers the different aspects of web APIs, the importance of their design, and who designs them. It also provides an overview of the API design process and our approach that separates concerns to facilitate learning and execution.

1.1 What is a web API?

Web APIs are software interfaces that allow remote communication between applications. They are invaluable because they don't require knowledge of

the underlying code; anyone can use them, not just their creators.

1.1.1 A remote web interface for applications

A web API enables one application (the server, backend, or provider) to expose functions or operations that other applications (the client or consumer) can use, call, or consume remotely over a network using web technologies. For example, many mobile applications rely on a remote server application accessible via the internet and its web API to retrieve, send, or process data.

Figure 1.1 Using the Socnet mobile application

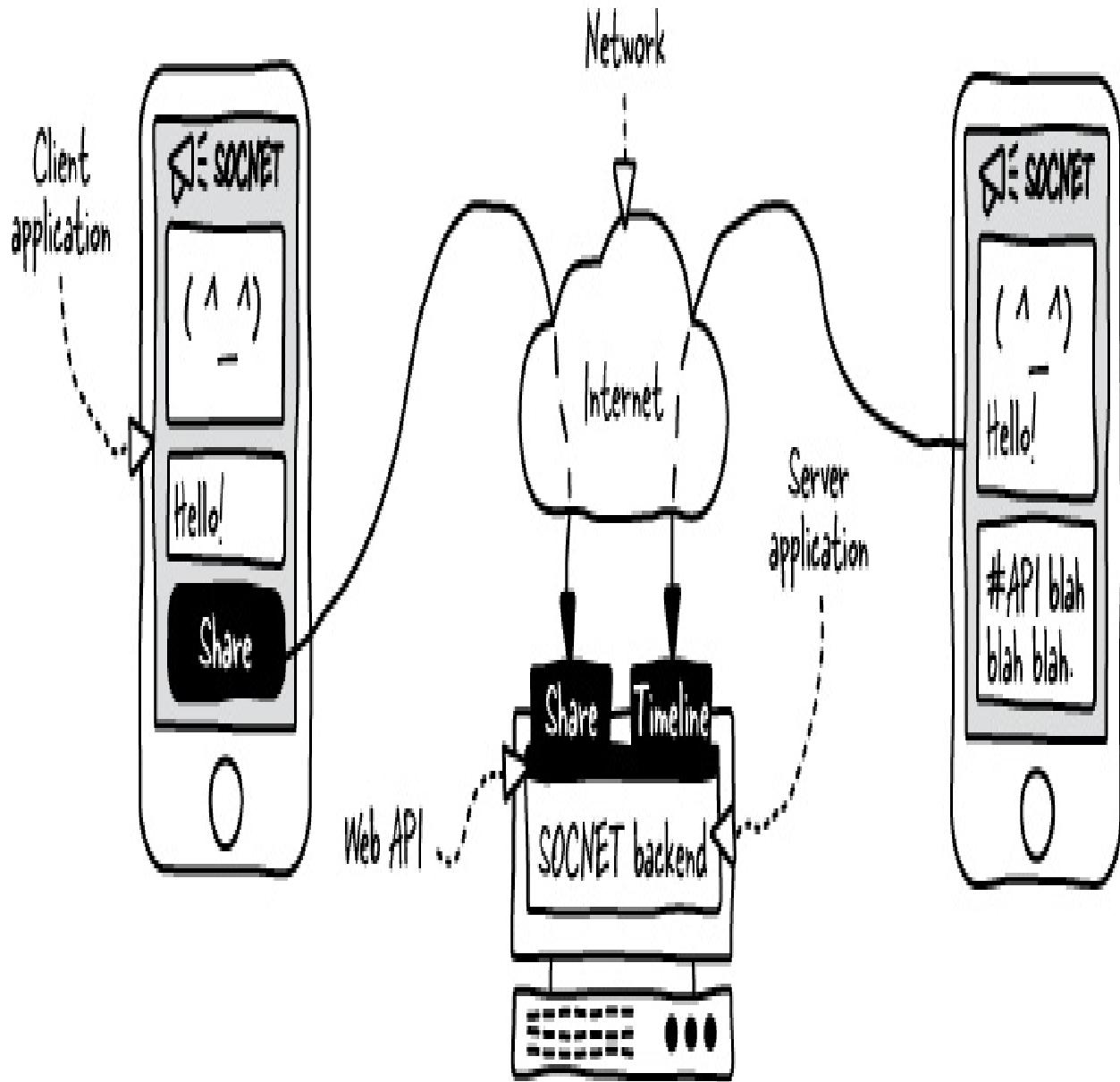


Figure 1.1 shows what happens when a user of the Socnet social network shares a photo via its mobile application. The user takes a photo, types a message, and taps the "Share" button. The mobile application calls the "Share" operation of the server application's web API to send the message and the photo via the internet network. The server application identifies the user's friends in the photo and stores the message, the identified friends, and the photo. Other users of the Socnet mobile application can see the shared messages, photos, and identified people on their timelines thanks to a call to the "Timeline" operation of the same web API.

When a mobile application communicates with a server application via its web API, it leverages the same mechanism as when a web browser retrieves and displays an HTML page of a basic website. That's the origin of the "web" in web APIs.

Note

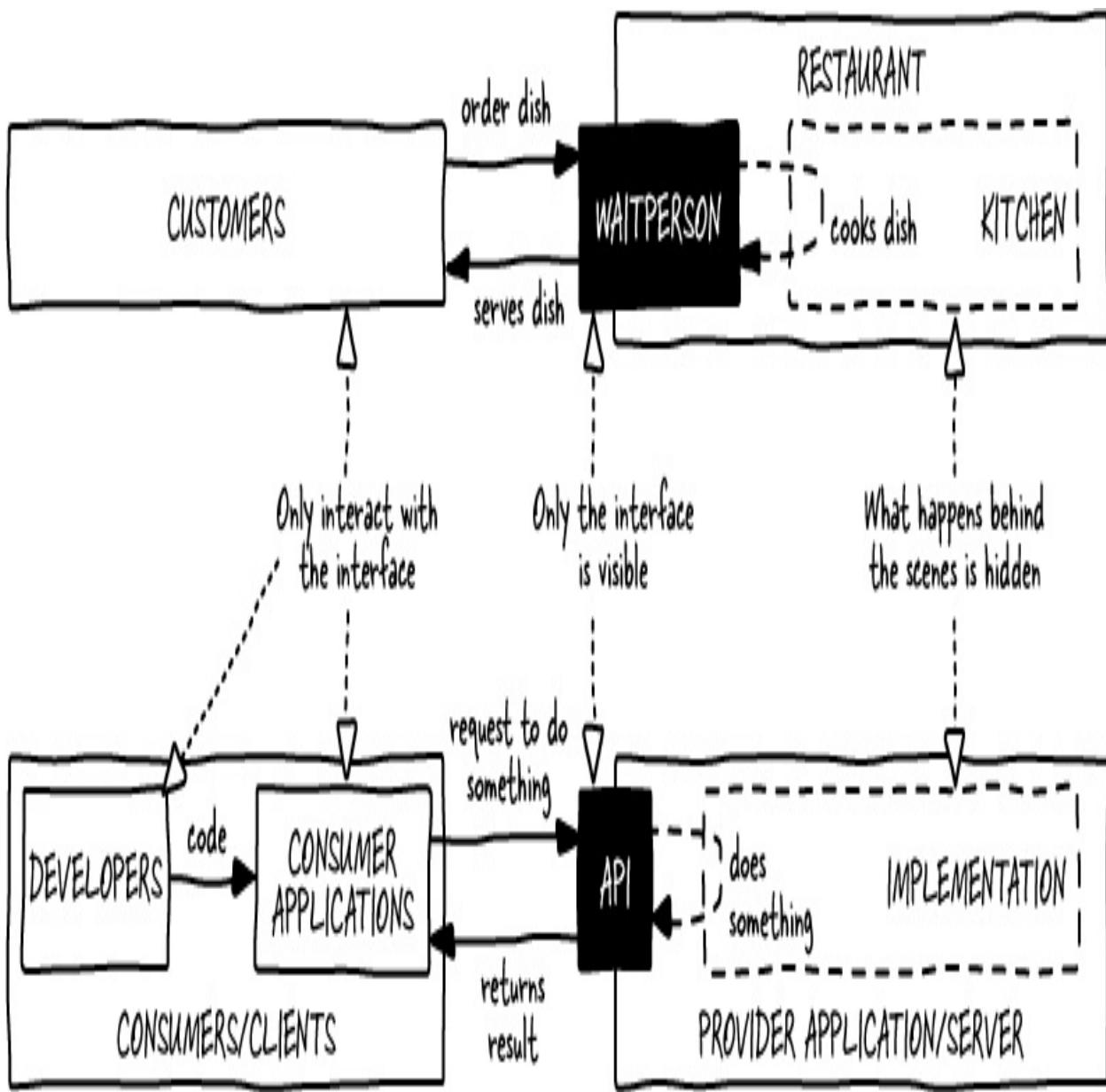
There are different types of web APIs, such as REST, SOAP, GraphQL, and gRPC. No worries if those names mean nothing to you; this book focuses on REST web APIs, but many principles presented here apply to other web APIs. What is a REST API and why we focus on it will be explained in later chapters.

Web APIs can be exposed on any network and consumed by any application. Socnet can use the same API in their web application, create a batch server to regularly call the timeline operation, and expose the next API version on their local network for quality checks before deploying to production.

1.1.2 An interface to an implementation

We often use "web API" as a convenient shortcut to designate an application exposing an API. However, the "web API" is only a part of it. Reading this book, we should not confound the web API with the application's implementation or actual code. A web API is an interface to an implementation and usually hides implementation details.

Figure 1.2 Comparing a restaurant and an application exposing a web API



As shown in figure 1.2, we can compare an API and its implementation to a restaurant. The API is the waitperson who takes your order and brings it back to you. The implementation is what happens in the kitchen. You don't need to know who is in the kitchen, the recipe, its ingredients, how it's cooked, and even if it's cooked there. You just need to know that you'll get the dish you ordered.

The Socnet mobile developers code what happens on a tap of the "share" button without knowing what happens behind the API. Knowing which database is used, how data is organized, or which face detection algorithm is

useless to them. They only need to code the API call, providing the expected data, and the API implementation takes care of the rest.

1.1.3 An interface for others

The fact that someone coding an application using a web API doesn't need to know how it is implemented has exciting consequences: it enables collaboration within an organization and allows people outside the organization to use and even pay for APIs.

At Socnet, three teams efficiently develop the backend API, mobile app, and website with minimal need for synchronization or shared knowledge. The mobile and web teams only need to know the services or features provided by the backend API, not how it is coded. To deliver a robust face detection feature, the backend API team leveraged the "Face Detection API" from "Image Processing As A Service" (for which they pay a subscription). Additionally, Socnet set up a "Search API" for selected partners willing to pay for access to more data.

Figure 1.3 What is an internal, external, private, partner, or public API?

WHO CONSUMES THE API?	THEN IT IS A ...	AND ALSO AN ...	WHERE IS THE API EXPOSED?	THEN IT IS AN ...
Only the provider	Private API	Internal API	Local network, intranet	Internal API
Selected third parties	Partner API	External API	Internet	External API
Anyone*	Public API			

(* Anyone respecting the conditions of use)

The APIs involved here represent three levels of API openness (or closeness): private (Backend API), partner (Search API), and public (Face Detection API). Internal and external may be used to qualify private and partner/public APIs, respectively. These terms may also indicate that an API is exposed on a local network or the internet. Figure 1.3 shows two tables disambiguating all these terms.

Most web APIs are private, and there are millions of them, as any company with an IT system will need them. While many are consumed by those who create them, others across the organization often use them.

Many organizations use partner or public APIs from others, which may be qualified as products, and their access is often paid. These APIs could be exposed by commercial or open-source software installed on their infrastructure. Increasingly, companies offer "as-a-service" products accessible via websites, mobile apps, and APIs, and sometimes the API is the only channel. Examples include payroll, retail, financial services, payments, telecommunications, project management, development, and cloud infrastructure. Even not-so-digital companies and government agencies offer partner or public APIs. Whatever we need, "There's an API for that."

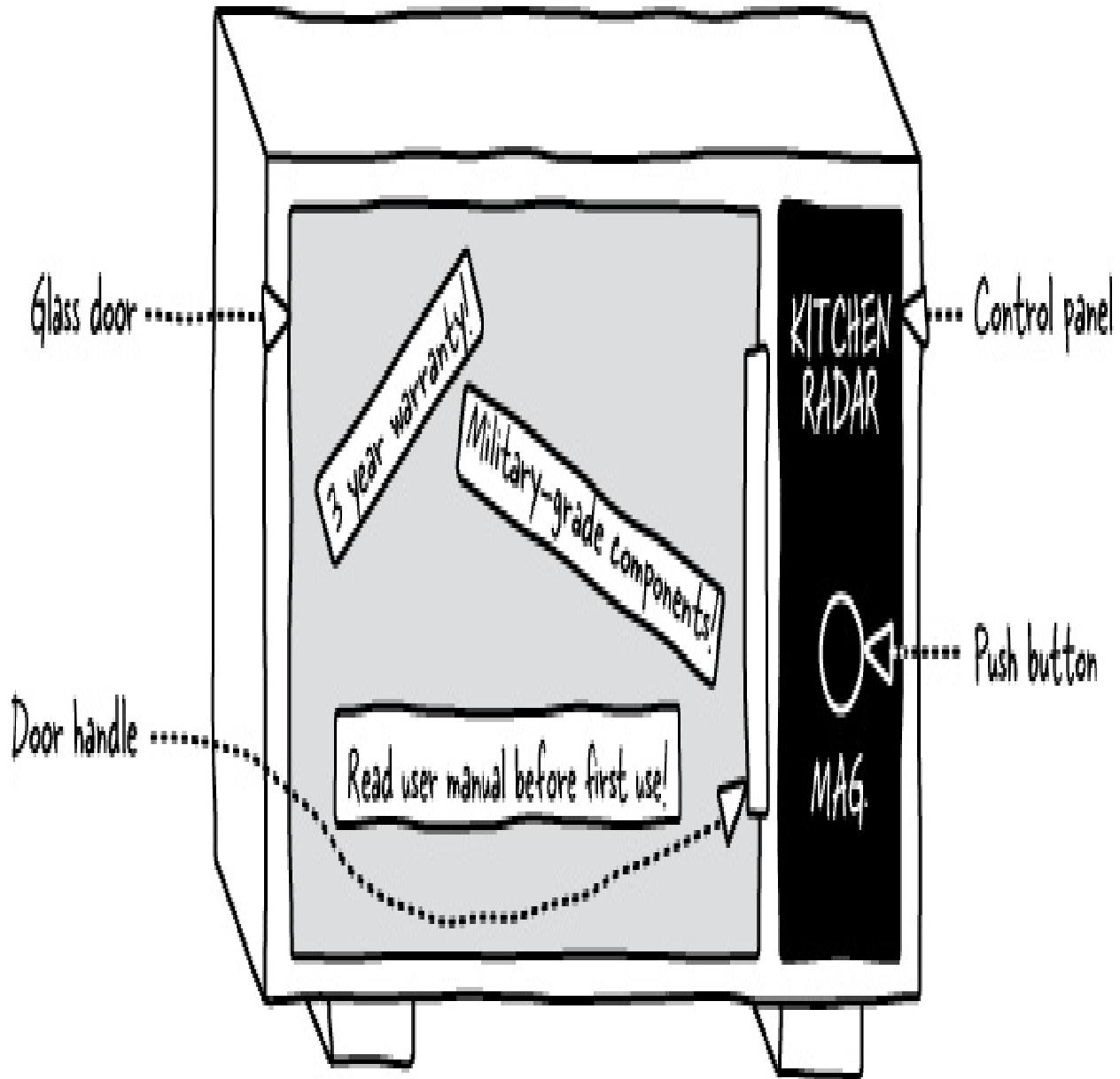
1.2 Why does API design matter?

The term "design" can designate both the process we go through to decide what an API does, how it does it, and how it looks like and the final result, the API itself. As a result, the design of a web API matters because it affects its consumers and provider. This section leverages a real-world analogy to demonstrate how. Then, it explains how this applies to web APIs and why designing them well is essential.

1.2.1 The design of any interface affects its users and creator

Figure 1.4 shows a Kitchen Radar. What is it, and how to operate it? Its interface doesn't help us figure this out. Pushing the "MAG." button seems to start it, but it stops when we release it.

Figure 1.4 The Kitchen Radar

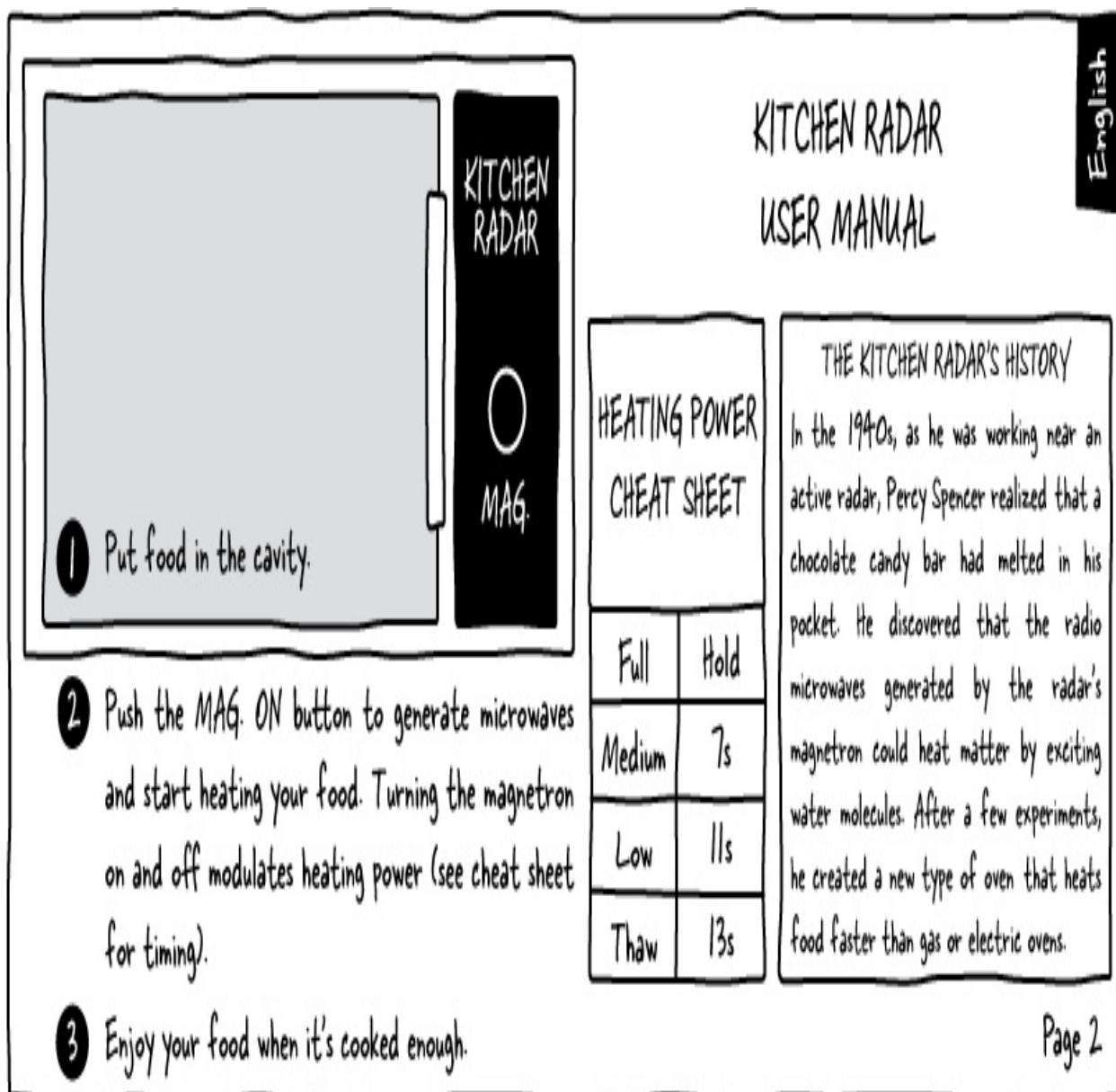


According to the user manual, shown in figure 1.5, The Kitchen Radar, named "Radar" for historical reasons, can heat food. The "MAG." button turns the magnetron on and off. When on, microwaves heat the food in the oven's cavity. To modulate the heating power, users must hold the button for a given time (e.g., 13 seconds) and release it for the same time, as indicated in the "Heating power cheat sheet."

This Kitchen Radar is a microwave oven proposing a terrible user experience despite the user manual, thanks to its cryptic, inside-out, and absurd interface. It requires users to become experts in magnetrons and time themselves

pushing and releasing the button. It's complicated and annoying to use. And there may be reliability and safety issues. How will the magnetron and overall circuitry react to being turned on and off at a random or too high-speed rate? Ultimately, who would buy this product?

Figure 1.5 The Kitchen Radar's user manual



1.2.2 The design of any web API affects both consumers and provider

The design of the "Kitchen Radar" interface is an exaggeration of the worst possible flaws you could find in an everyday object. And yet, I've come across private, partner, or public APIs whose designs look like this; they were hard to understand or use or were exposing inner workings. Such API designs negatively impact consumers and providers.

If an API is hard to understand, its purpose and features unclear, developers may spend extra time integrating it, making more errors, and asking many questions. Potential users may pass by if the API is partner or public, leading to less revenue for the provider.

APIs that are easy to understand can still be hard to use. Developers may spend extra time writing code to orchestrate complex API call flows, resulting in more errors. Due to this complexity, they may make many API calls, leading to unexpected extra load and costly cloud infrastructure billing. If the API is public or partner-based, users may cancel their subscription when they realize how complicated it is.

An API design exposing the implementation's inner workings is often hard to use and understand. But more than that, it creates tight coupling between applications, prolongs modification, and augments the risk of errors or crashes.

1.2.3 Taking care of design unleashes the power of APIs

Taking care of web API design prevents the previous section's issues and unleashes their power. Here are some benefits you may witness:

- *Better developer productivity:* Well-designed APIs are much easier to understand and use. Additionally, they offer greater flexibility and interoperability, requiring less effort to interpret their data. Developers can integrate them into their applications quickly and with minimal code.
- *More modular and efficient systems:* Well-designed APIs allow the creation of decoupled systems where the applications consuming and providing APIs have limited dependencies. They may also contribute to reducing infrastructure usage.

- *Faster time to value*: Developers can achieve their goals quickly by using well-designed APIs that are versatile and reusable in various contexts, allowing for creating innovative solutions without rebuilding everything.
- *Better end-user experience*: Well-designed APIs provide features efficiently and flexibly that can contribute to creating an outstanding experience for the end-users of the applications using them.
- *More API-generated value*: Well-designed APIs can increase indirect and direct value by reducing development costs and increasing revenue through customer satisfaction.

1.3 Who designs web API?

API design matters, but who can design web APIs? You! But probably not alone. API design requires diverse skills and knowledge, often resulting from different people's work. Those with API design skills, IT and subject-matter knowledge, and influence can all contribute directly or indirectly to the design of an API.

1.3.1 Those with API design skills

API designers lead the discussions and hold the pen when designing web APIs. They come from various backgrounds and have different skill sets. I've worked with, advised, and trained API designers from various profiles, including developers, tech leads, architects, business analysts, tech writers, QA engineers, product managers, and product owners. Some started as juniors, while others began after long careers.

While having a thorough understanding of the subject matter and software is undoubtedly an advantage, it is not a requirement for designing APIs. As an API designer, you only need to know the fundamental principles of API design and how to get the necessary information. This book will teach you that.

1.3.2 Those with subject-matter knowledge

APIs solve specific problems like social networking, banking, product catalogs, or database as a service system administration. As an API designer, you don't need to be an expert in the subject matter. Like when creating any application, by interviewing SMEs (subject matter experts), you can ensure your API accurately represents the business domain and its problems.

1.3.3 Those with software knowledge

Web APIs are programming interfaces for software. Designing one requires familiarity with web API-based systems and specific knowledge about the system that will expose the API.

Knowing the principles, practices, and limitations of web API-based software in general and web and mobile applications, in particular, will significantly help design realistic, implementable, and technically usable APIs. For example, some API design patterns may kill a smartphone battery, and others can make an API more effortless to use across many systems. We'll uncover the typical general software concerns you should consider when designing an API and how to avoid or mitigate them in your designs.

Knowing the system behind a web API can be helpful. For instance, if the application responsible for detecting faces takes a minute to identify people in a photo, this should be considered when designing the API. We'll learn the right questions to ask technical leads and architects to uncover such details and how to leverage this knowledge sensibly to avoid revealing internal workings.

1.3.4 Those with influence

An API will be the result of a sum of various influences. As an API designer, you'll have some control over some of these, while others are out of your hands. Even the most knowledgeable API designers must consider feedback from peers, reviewers, security teams, and, most importantly, consumers. Knowing how to integrate feedback is vital to creating an API that satisfies all parties involved.

1.4 When designing web APIs?

API design is an essential task that requires careful consideration. When to undertake this task is a question that often arises. Should we design all APIs? Should modifications to APIs also be designed? When is the best time to design an API? The answer is simple: we should design all types of APIs, such as internal, external, private, partner, or public APIs, as well as any modifications to web APIs. Moreover, we must prioritize API design before developing the implementation.

1.4.1 When creating any API

Should we only design partner or public APIs because they are more visible than private ones? No. They all are essential, and we must design them all, whoever consumes them and wherever they are exposed.

It is now common for organizations to provide partner or public APIs to third parties. Neglecting their design or brutally turning non-designed private APIs into partner or public APIs will lead to terrible APIs and serious consequences, as seen in section 1.2.2. For example, ignoring partner API design can lead to lengthy and costly integration projects, while a poorly designed public API may have no customers.

Neglecting the design of private APIs can also lead to the issues we've uncovered in section 1.2.2. Even if we're the only creators and consumers, poor design impacts code and productivity, leading to missed milestones, fewer features, bugs, and revenue loss.

Finally, using private APIs to improve your skills and prepare for designing APIs for others is crucial. You will likely create more APIs for personal use than those you provide to other teams; most will be private rather than public or partner ones. Practicing with private APIs will help you make better decisions with confidence.

Jeff Bezos' mandate

Around 2002, Jeff Bezos, former CEO of Amazon, issued a mandate stating that all teams must communicate through "service interfaces" (they were not called API then). And that all those "service interfaces" must be designed

from the ground with externalization in mind, as each could be put in customers' hands anytime after its creation. This strategy was key to Amazon's success.

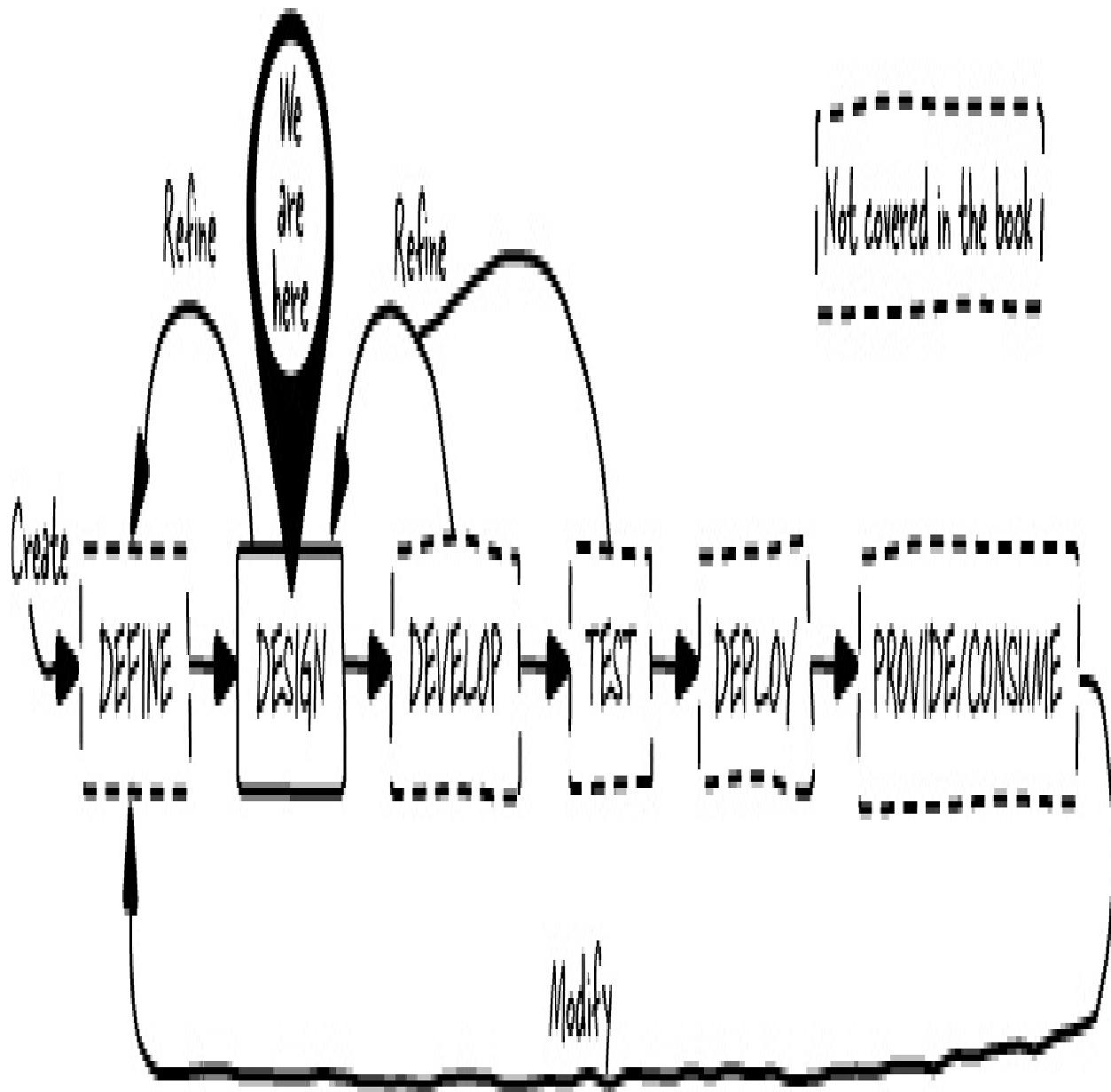
1.4.2 When making any modification

It's essential to design APIs during initial creation and for any modifications, even the most minor ones. The reason for this is the same as when creating them, but the stakes are higher. Carelessly modifying an existing API that applications already use can lead to disastrous consequences. It may break their code, causing crashes or data corruption. And even if, under certain circumstances, it is possible to introduce modifications that will break consumer code without much consequence, we must still consciously design them to evaluate the impacts.

1.4.3 Before developing the implementation

API design precedes implementation and occurs repeatedly throughout the API lifecycle, the various stages an API will go through, from its inception to its consumption.

Figure 1.6 The lifecycle of an API



There are different versions of this lifecycle, usually based on the software development lifecycle and having more or fewer stages. Figure 1.6 shows an API lifecycle that includes six stages:

- *Define*: Stakeholders define vague or precise needs.
- *Design*: An API designer collaborates with colleagues to design the API that fulfills the needs. This book focuses on this stage.
- *Develop*: Developers implement the API.
- *Test*: Developers, QA engineers, and security experts ensure the API's implementation works as intended and is free of bugs or security

breaches.

- *Deploy*: Automated system or people deploy the implementation. It is often accompanied by exposing the API on an API gateway, a proxy facilitating API securitization, consumption, or monitoring.
- *Provide/consume*: Make the API visible and available for targeted consumers. It is often done by adding it to an API catalog or an API developer portal.

The lifecycle simplifies reality; there are back-and-forths between steps, and some actions are not sequential. For example, you'll deploy the API while developing and testing it before deploying it into the production environment.

There may be multiple iterations of design, as discussions and analysis can lead to refining the needs to fulfill, and discoveries made during development or testing can lead to refining the design. Once the API is deployed and consumed, the design stage may be revisited to add or modify features.

1.5 What does designing web APIs mean?

Designing an API involves creating a versatile product in the form of a programming interface that addresses users' needs while also considering contextual constraints. It requires making numerous deliberate decisions ranging from simple to complex, flawless to half-satisfying, and from certain to uncertain, even resulting in mistakes.

1.5.1 Designing a versatile software product

Web APIs, even private ones, are versatile software products that deliver services related to many subjects. We must design them using general, product, or software design principles and techniques. This section illustrates this by reusing the "Kitchen Radar" of section 1.2.2.

Users of the Kitchen Radar want to heat food, not turn a magnetron on and off. API designers should meet users' needs and enable them to achieve their goals. Users of an API include the applications consuming it, the developers creating them, and, to some extent, their end-users. None of them want to access databases or systems; they want to accomplish tasks that matter to

them. Also, an API should be adaptable enough to serve the needs of all present and future users in various contexts.

To meet users' needs better, we could rename the Kitchen Radar to Microwave Oven and replace "MAG." with "HEAT." However, more is needed to make it user-friendly. We must design APIs that are easy to use and intuitive for developers, regardless of their level of expertise in the subject matter. Moreover, APIs should be interoperable, requiring no complex coding to work with applications or data.

The Kitchen Radar can be more user-friendly if we replace the push button with power and duration knobs. However, we need to consider the 60-minute limit of the chosen magnetron and integrate it into the control panel's design. It's crucial to consider the context when designing APIs. We must consider the API type, subject matter, system, and consumer constraints. It helps us create APIs that are realistic, usable, and implementable.

The Kitchen Radar belongs to a line of products, all sharing characteristics, designed by the same person or numerous ones. We must ensure our APIs form a consistent line of products or landscape when seen as a whole. As we create or modify APIs, each must make sense compared to what we've done previously. This topic has significant implications beyond this book. However, we will learn enough to design consistent APIs easily.

1.5.2 Reasoning, deciding, doubting, failing, and iterating

Designing an API involves making deliberate decisions, such as choosing names and data types when modeling data. The resulting API can be a disaster if done randomly or without enough care, like the "Kitchen Radar" and its "MAG." button seen in section 1.2.2. API designers must make numerous decisions throughout the design process, some straightforward and some complex.

Tradeoffs are sometimes necessary. These half-satisfying decisions dictated by the context may not yield an ideal API design, but there may be no other alternative. Implementation constraints can be hard to solve within a given time and budget, requiring integration into the API design. Design questions

and problems can lead to multiple solutions, affecting user experience, security, and evolutivity. With so many alternatives, doubt is unavoidable. Errors can occur due to incomplete understanding of requirements, overlooking details, or changes in context.

It's essential to understand that dissatisfaction, doubt, and failure are part of API design. Thankfully, there are solutions to help make informed decisions, gain confidence in tradeoffs, overcome doubts, and mitigate the risk of failure.

Backing decisions with principles, recipes, and logic, rationalizing pros and cons, and leveraging past decisions enables efficient problem-solving. Also, it's not necessary to be "right" on the first attempt; we can leverage an iterative approach. We can test and validate the design at any stage by seeking feedback from those who contribute to the design, for example, subject matter experts, implementers, or consumers (see section 1.3).

1.6 How to design web APIs?

The process of designing APIs is similar to any design process; it involves analyzing requirements and converting them into a blueprint for the final product. This book uses a methodology breaking down the design process step-by-step and using a layered approach to focus on one main problem at a time.

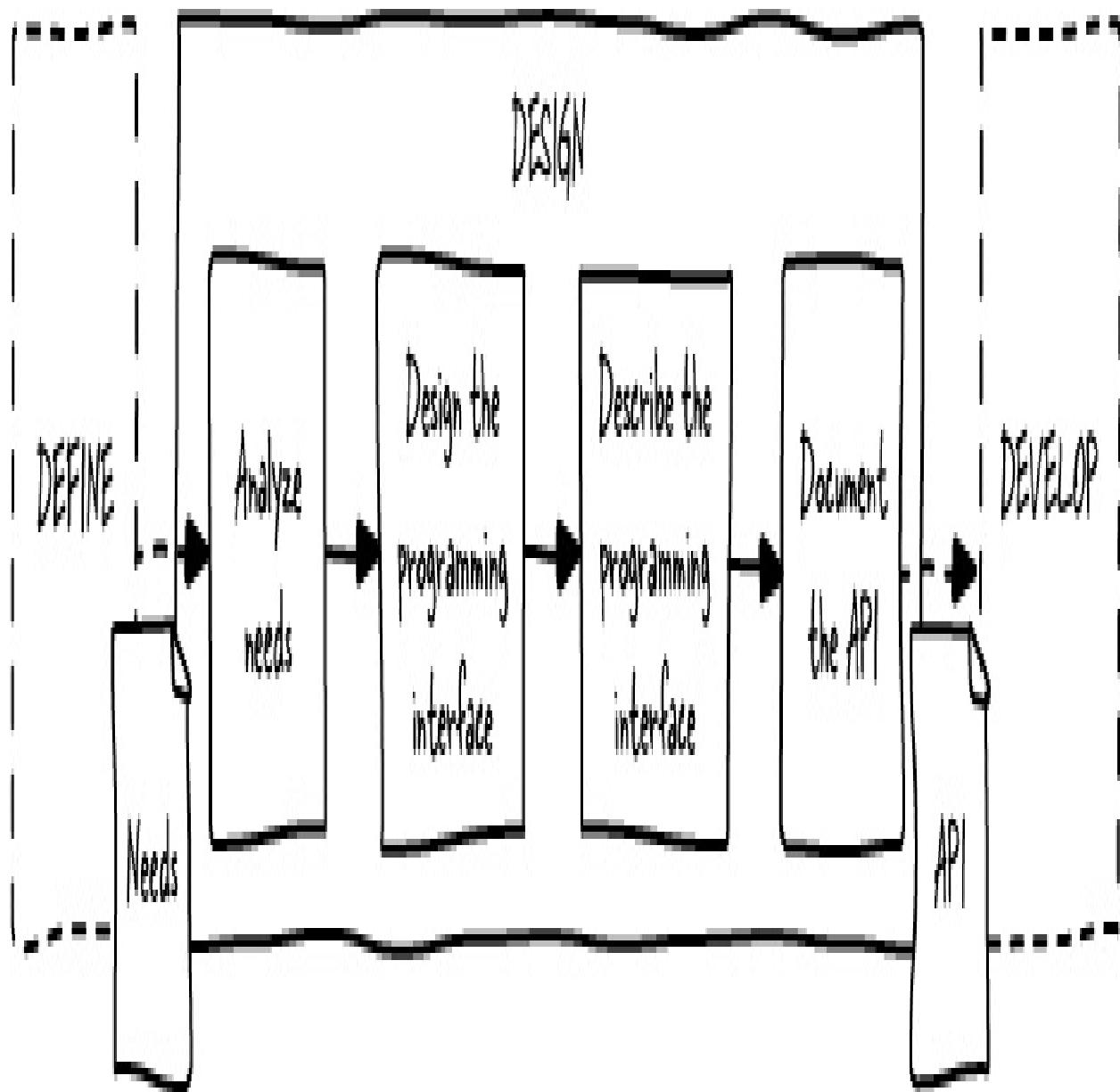
1.6.1 By using a step-by-step approach

To design an API, we use the step-by-step approach separating concerns shown in figure 1.7.

Note

This book focuses on the "Design" phase of the API lifecycle and doesn't cover the "Define" or "Develop" or any other phases. It doesn't discuss business or IT strategy involving APIs or how to code an API implementation.

Figure 1.7 The API design process



Once needs are identified in the Define phase, we can begin designing an API. Needs can range from a general objective or direction ("Social network" or "Database as a service") to a more precise intent ("Enabling tagging friends in a photo on the mobile application and website").

We use the API Capabilities Canvas to analyze these needs and identify API capabilities in plain English, including use cases such as "Sharing a status" and API operations like "Upload a photo" and "Send a message."

Then, we design the programming interface, which includes choosing the type of API, representing operations with a programming interface, and modeling data.

In parallel with the previous step, we describe the programming interface in a blueprint document using a standard API specification format.

Ultimately, we document the API, focusing on what's needed for design validation and implementation. The resulting documentation may suffice for private APIs but may need enhancements for public or partner ones.

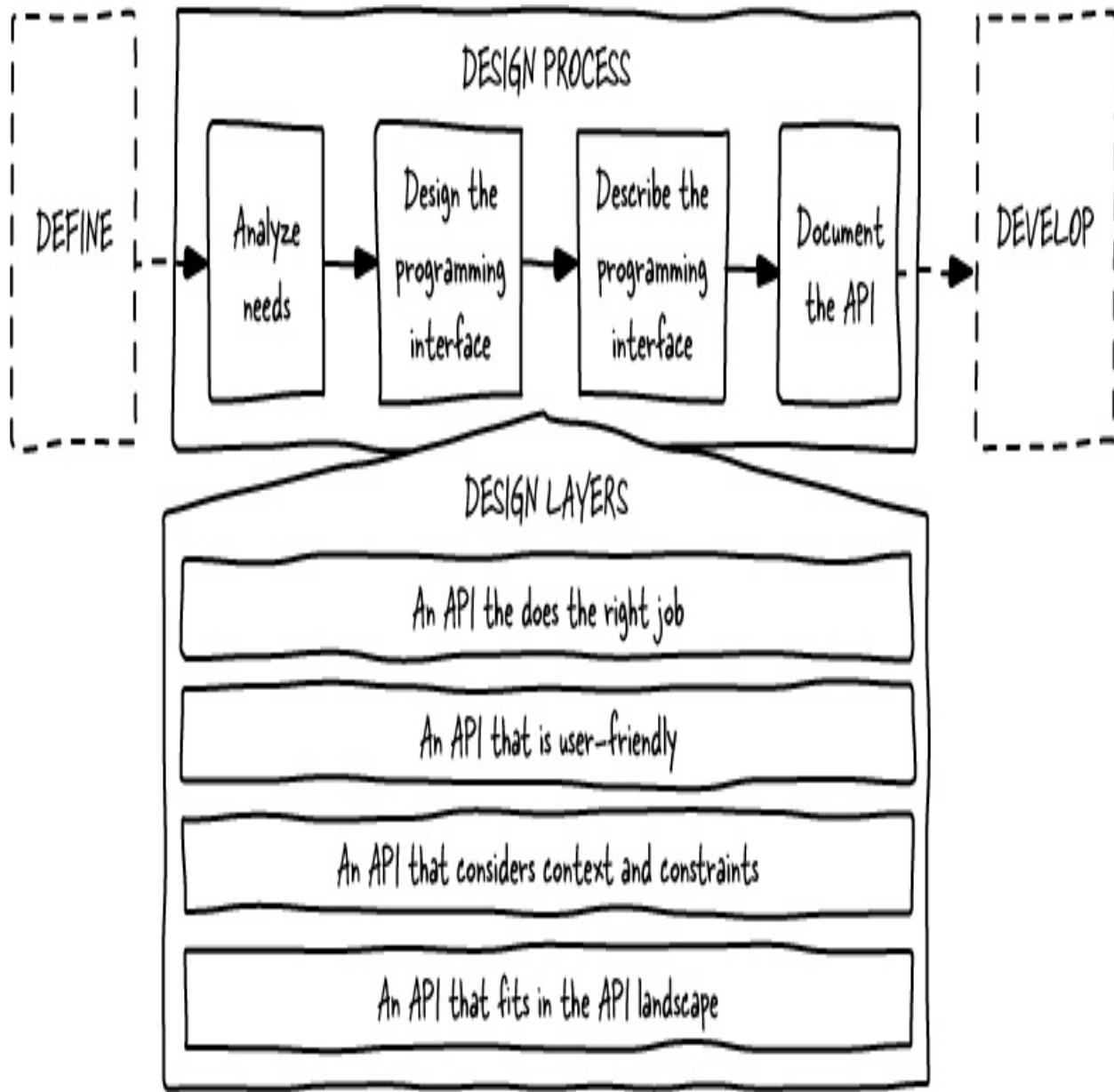
We provide artifacts like API Capabilities Canvas, formal API description, and documentation to implementation developers for the "Develop" phase of the API lifecycle. These artifacts can also be helpful in other stages, like the "Test" phase.

The API design process is iterative. Stakeholders provide feedback at any step. Needs analysis may require refining the needs or validating understanding, while fine-grained programming interface design or documentation may reveal missing elements, leading to design or API capability updates. Issues or imprecisions uncovered in the "Develop" or "Test" phases or later may require design revision.

1.6.2 By having a layered approach

We use the layered approach shown in figure 1.8 to simplify the design process and our learning. This approach ensures the API does the right job, is user-friendly, integrates context and constraints, and fits into the API landscape. The layers match the concerns discussed in section 1.5.1.

Figure 1.8 The four layers of the API design process



We first focus on ensuring the API fulfills the needs identified in the "Define" stage, challenging them if necessary. We carefully avoid exposing inner complexity or being too specific to consumers' needs.

Leveraging standards, common practices, and design principles, we ensure we create a user-friendly API that anyone can understand and use. This aspect covers its capabilities, programming interface operations, and data.

To create an efficient and effective design, we consider the context and identify constraints like security, performance, and system limitations. We

must also ensure we don't break existing functionality when modifying an API.

Ultimately, we ensure our API and its elements fit in the API landscape we are building, contributing to user-friendliness. This concern is related to "API governance" or "API stewardship," which have implications beyond design. This book focuses on avoiding reinventing the wheel and making API design decisions more consistent, confident, efficient, simple, and quick.

1.7 Summary

- A web application programming interface, or web API, is a software interface exposed by an application that allows other applications to interact with it remotely by leveraging web technologies.
- Using a web API can be done without knowing or understanding what its implementation does.
- Web APIs' design can negatively or positively affect developer productivity, system flexibility and efficiency, project delivery, and organizational revenue.
- An API designer doesn't need to know everything; they need to know how to get the information and find the balance when integrating them into their design.
- It's essential to design all internal, external, private, partner, or public APIs, as well as any of their modifications, and to do it before developing them.
- Designing web APIs requires making deliberate decisions with a rationale to create a versatile programming interface that meets users' needs and considers contextual constraints.
- API design involves analyzing needs to identify capabilities, mapping them to a programming interface, formally describing the interface, and documenting the API.
- Designing an API requires focusing on four different layers: ensuring that it does the right job, that it is user-friendly, that it integrates the context and constraints, and that it fits with past decisions.

2 Analyzing needs

This chapter covers

- Why API design starts with needs analysis
- Analyzing needs to identify API capabilities
- Focusing on the proper needs
- Avoiding exposing the provider's inner complexity
- Avoiding integrating overly specific consumer requirements

API design begins by analyzing the needs to identify the API capabilities using plain English or any other natural language rather than programming interface language.

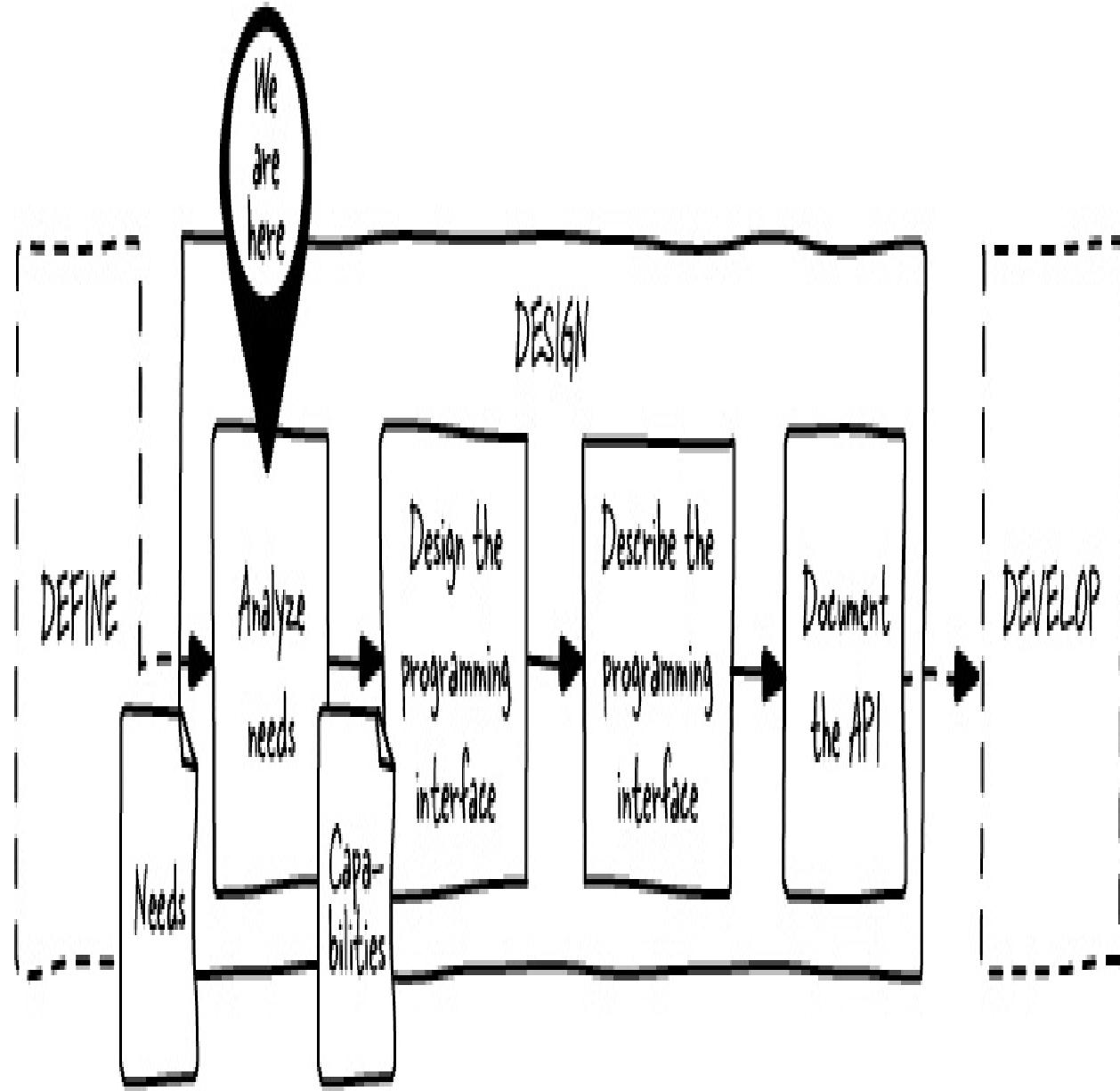
It's because form follows function: this design principle applies to buildings, kitchen appliances, applications, and APIs. Achieving an effective API design requires understanding user needs and identifying the capabilities, functions, or operations fulfilling them before choosing the appropriate programming representations and data modeling. It simplifies discussions, streamlines the design process, and avoids creating complex or incomplete APIs that don't meet user needs.

This chapter first overviews needs analysis, including when, why, and how to analyze needs. It then introduces the API Capabilities Canvas, a methodology for analyzing needs, and applies it to an example.

2.1 Overviewing needs analysis

As shown in figure 2.1, we're at the first step of the design process outlined in section 1.6.1. Needs analysis is preceded by the "Define" stage of the API lifecycle, which this book doesn't cover. Afterward, we'll move on to the next step: "Design the programming interface," covered in chapters 3, 4, and 5.

Figure 2.1 We are here in the API lifecycle and design process



This section describes needs analysis prerequisites. It then discusses why needs analysis comes first and its objectives. Afterward, it overviews how it is done.

2.1.1 Needs analysis prerequisites

Needs analysis starts with an input outlining needs or problems to be solved. It's crucial for designing an API that meets expectations. This input is defined earlier in the "Define" stage of the API lifecycle, which this book

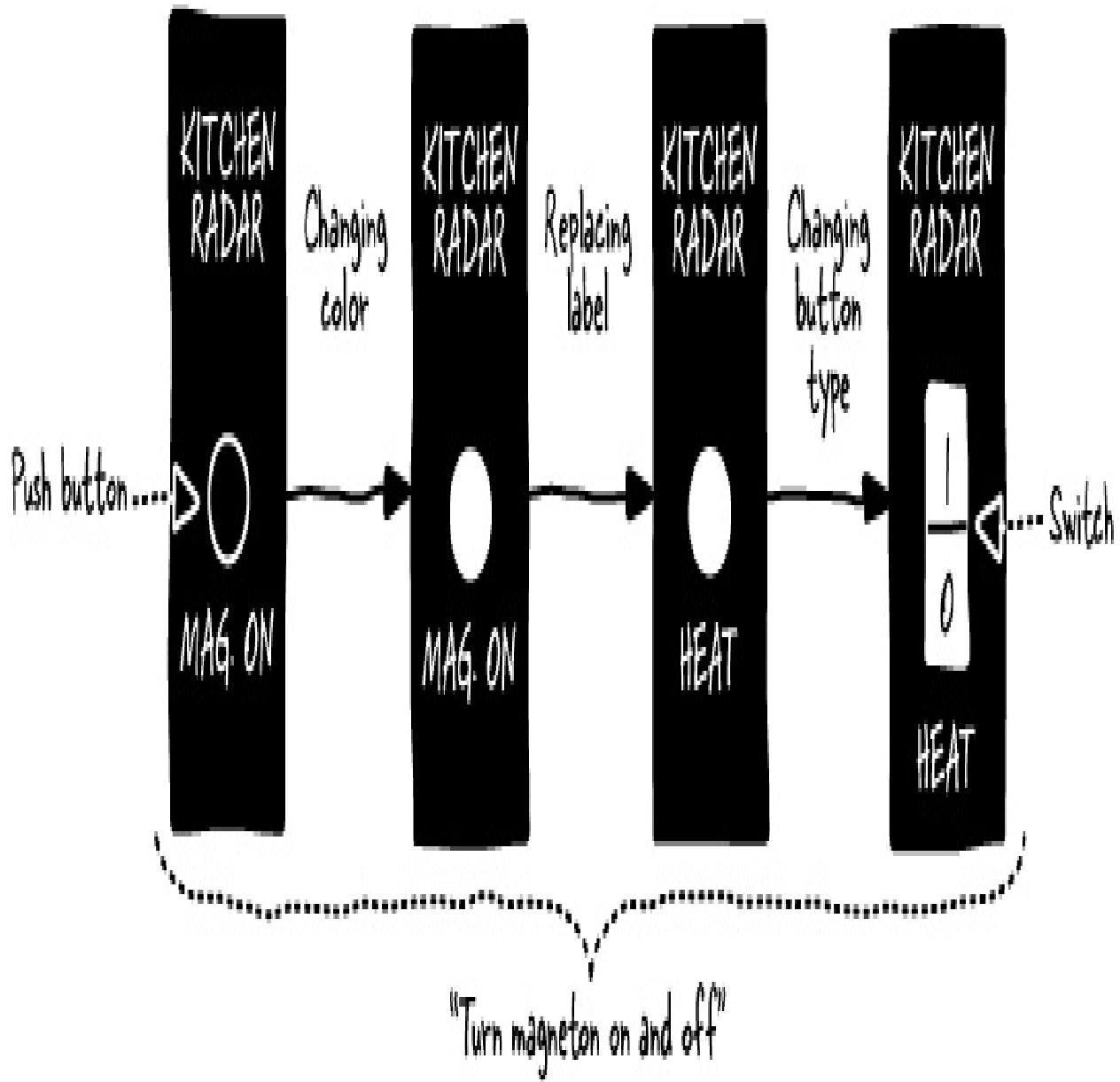
doesn't cover. This prior stage explores the why of the API, possibly including strategic or product concerns, especially for public APIs.

The input can be a sticky note with a vague description of the API's objective, direction, or intent ("Online shopping" or "Database as a service") or a more precise one ("Enabling tracking order status on the customer mobile application and customer care application"). Studies conducted during the "Define" stage, especially when creating a public API, may provide more details (user personas, use cases). But we can work without them, especially for private APIs.

2.1.2 Needs analysis reasons and objectives

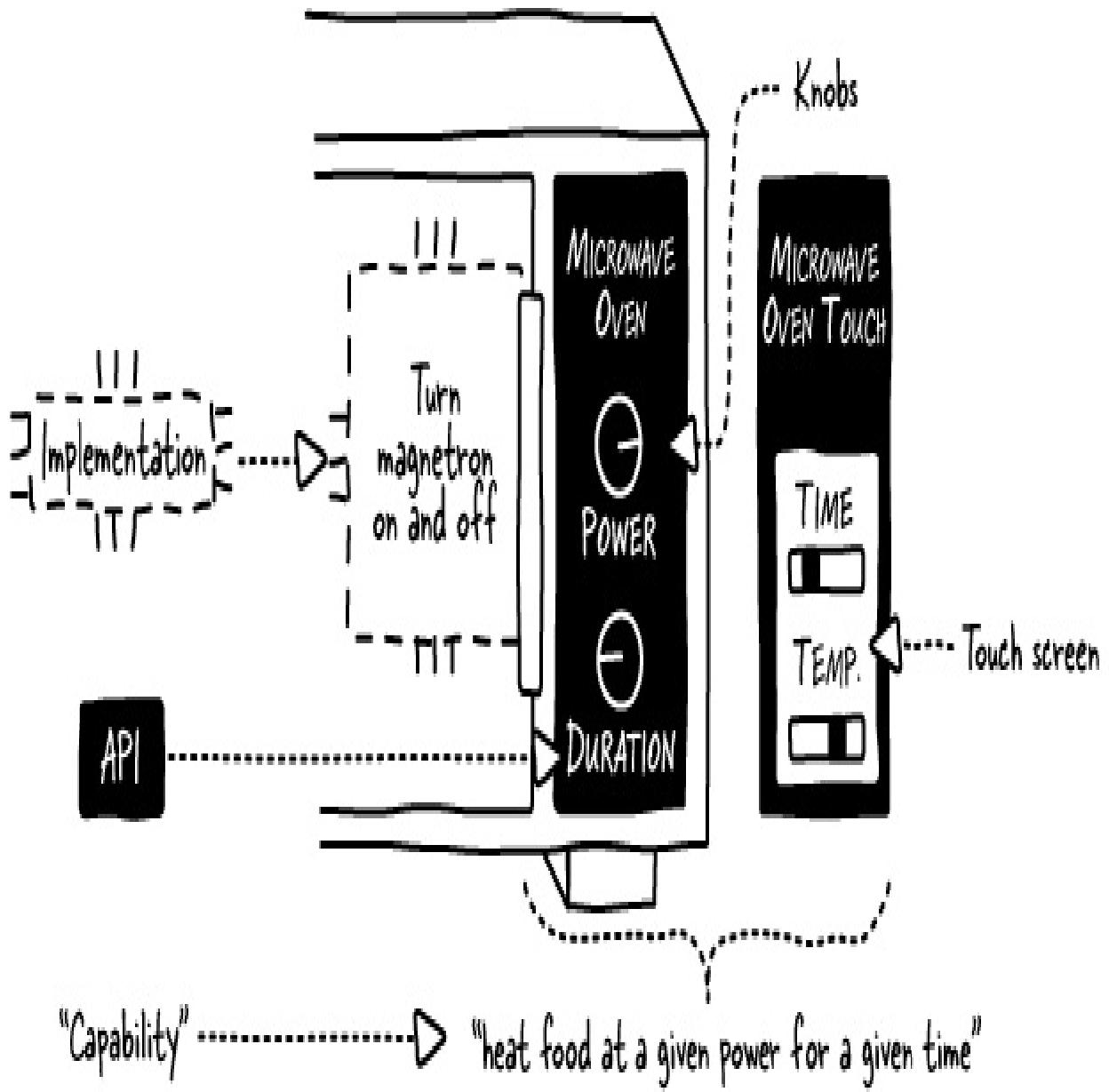
The Kitchen Radar (see section 1.2.1) is an example of what happens when needs analysis is skipped. Its "Turn magnetron on and off" capability exposes its inner complexity, making it hard to use. As shown in figure 2.2, redesigning its interface by changing button label, color, and type won't change that.

Figure 2.2 Redesigning the Kitchen Radar's interface won't improve it



What do users want? They want to heat food. They may need to select the heating power and duration. Analyzing needs helps us identify the appropriate "Heat food at a given power and duration" capability. As shown in figure 2.3, an interface offering this capability may have various designs (labels, control types), but all are easily used without magnetron knowledge. The implementation turns the magnetron on and off based on the interface inputs.

Figure 2.3 Contrasting capabilities, interface (API), and implementation



When designing an API, we must first focus on consumers' needs and identify capabilities (described in plain English) to fulfill them without worrying about the programming interface or its implementation. In the next stage, "Design the programming interface," we'll turn these capabilities into a programming interface.

2.1.3 How to analyze needs

During the needs analysis, we identify API capabilities based on input from

the "Define" stage. The needs-capabilities relationship is rarely one-to-one. With a step-by-step approach separating concerns, we scrutinize use cases, decompose them into smaller steps, and identify what is needed to achieve them, what users get in return, and how they use it to, ultimately, identify unique API operations. The resulting sum of use cases and operations is the API capabilities.

Needs analysis is often collaborative work. We verify the capabilities align with expectations and request clarification if necessary. We don't need to be experts on the topic(s) the API deals with; we discuss with subject matter experts (SMEs). As API designers, we extract and refine knowledge to create a versatile API that serves all consumers while concealing constraints and complexity. This work requires asking many questions, rephrasing answers, and challenging contradictions.

Iterative discussions and thinking are necessary, even for seasoned API designers. We may not achieve perfection on the first try, but through an iterative process, we can refine information into a comprehensive list of capabilities that all parties agree upon.

We only use plain language, English, or any other understood by stakeholders to streamline discussions and ensure the accuracy of API capabilities. We avoid using a "programming interface language;" we'll see why in section 3.1.5.

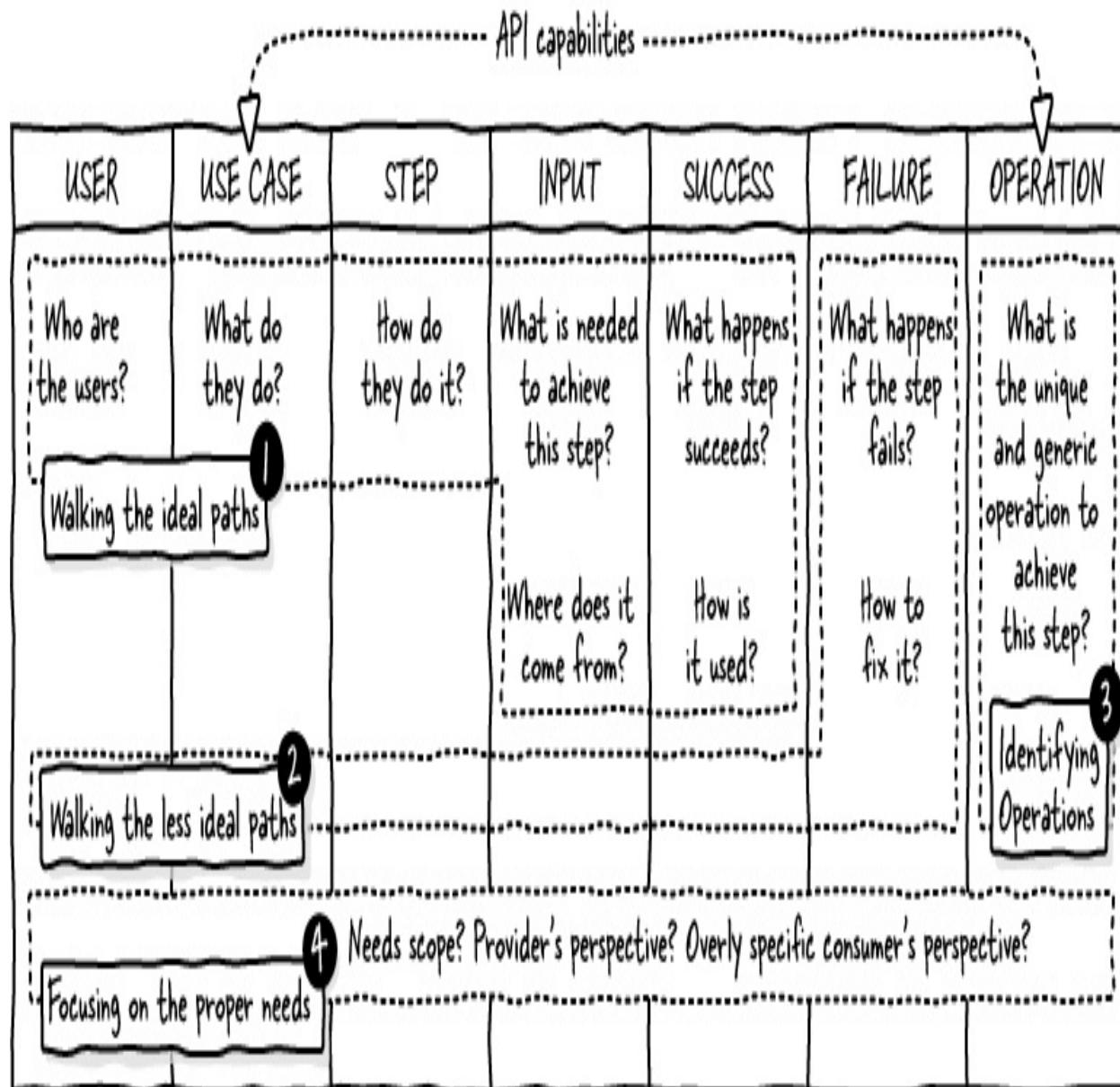
2.2 Introducing the API Capabilities Canvas

The API Capabilities Canvas helps exhaustively and accurately identify versatile API capabilities usable in various contexts. It is both a methodology and a document. It follows principles many instinctively use when designing software: wondering who the users are and what they need to do. Many other software and product design methodologies and templates follow the same process. You can use (and adapt) them once you understand the API Capabilities Canvas concepts. This section overviews how the API Capabilities Canvas works and discusses related tools.

2.2.1 How does the API Capabilities Canvas work?

The API Capabilities Canvas (see figure 2.4) relies on decomposing needs (determined in the Define stage of the API lifecycle) in small steps in two passes (ideal and less ideal), identifying unique operations for all steps, and ensuring focus on the proper needs.

Figure 2.4 The API Capabilities Canvas



To decompose the needs, we identify users (*USER* column), describe their use cases (*USE CASE* column), list steps to achieve the use cases (*STEP* column), inputs to execute each step (*INPUT* column), outcomes and outputs

when the step is successful (*SUCCESS* column), and context, outcomes, and outputs or errors when it fails (*FAILURE* column).

To simplify the decomposition process, we separate concerns and proceed in two passes: walking the ideal paths (see section 2.3) and then the less ideal paths (see section 2.4). In the first pass, we focus on ideal, nominal, common, or happy use cases and paths of steps. In the second pass, we investigate failures. We also analyze sub-paths and use cases set aside during the first pass. During the decomposition, we check each input source, outcome usage, and how to fix failures to ensure nothing is missed.

Afterward, we map each step to a unique, context-agnostic operation (*OPERATION* column). Different steps may share the same operation. See section 2.5).

Ultimately, we check all elements contribute to fulfilling the proper needs. We ensure we stay in the Define stage's needs scope, not to expose internal complexity or integrate overly specific consumer needs. See sections 2.6, 2.7, and 2.8.

Note

Needs analysis requires trial and error to get it right. Even experienced designers rely on an iterative process and need feedback. Finding the right level of detail can be tricky in the beginning, but keep going; it takes practice.

2.2.2 Tools to use along with the API Capabilities Canvas

You can draw and fill the API Capabilities Canvas on a physical or virtual whiteboard. However, it may fall short for bigger APIs or when modifying multiple elements. Also, if physical, you'll need to rewrite everything in a digital document.

A good old spreadsheet is my go-to for a digital API Capabilities Canvas. You can screen share during API design workshops. Adding or moving elements is easy, and searching and filtering features are helpful. Pivot tables provide an overview of unique operations and their use. Examples can be found on my website at <https://apihandyman.io/the-design-of-web-apis>.

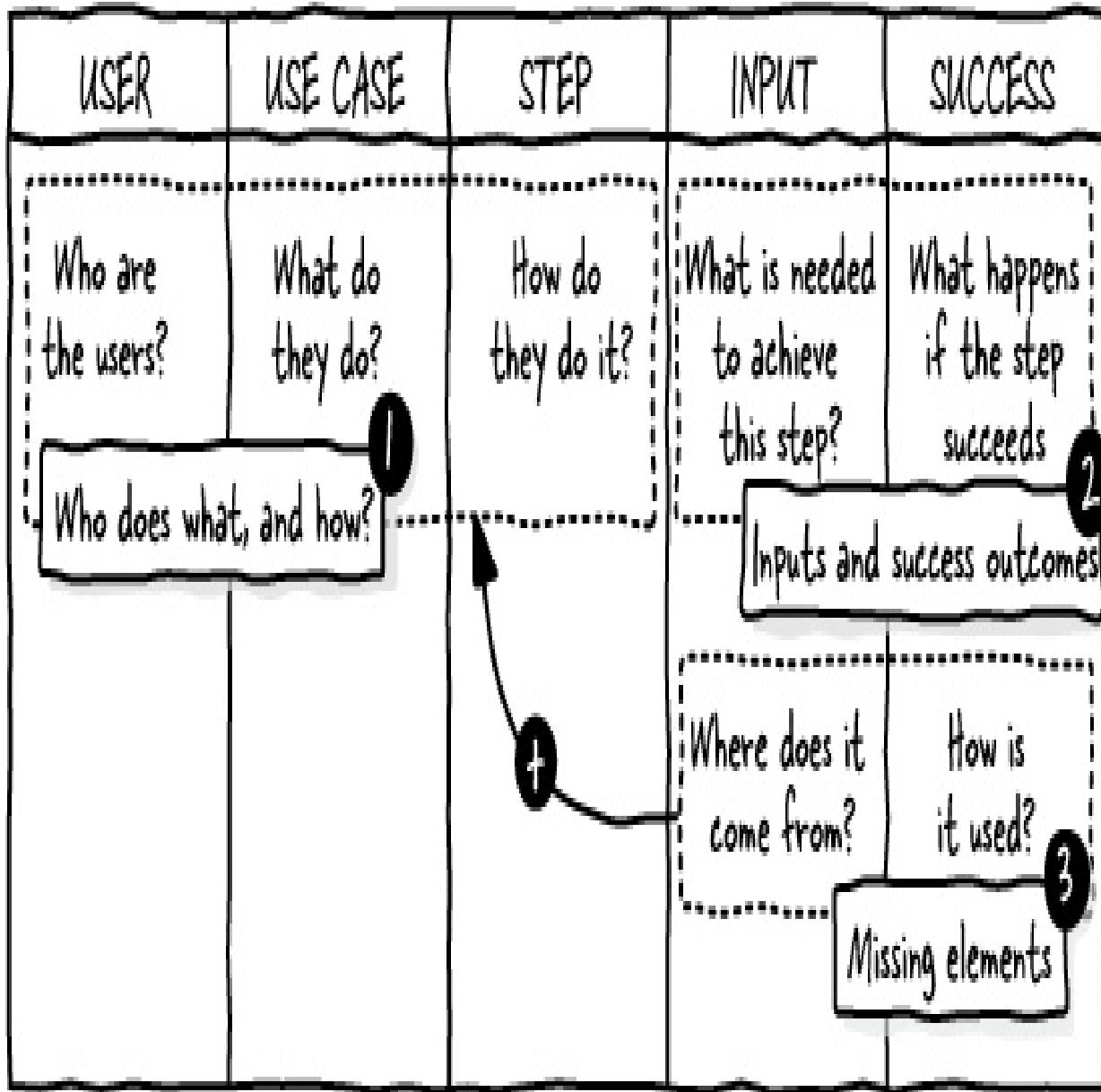
It can be helpful to model step flows with diagrams for complex use cases with optional subbranches and loops. Diagram-as-code tools like PlantUML or MermaidJS allow for easy creation and modification of diagrams.

2.3 Walking the ideal paths

Analyzing needs starts with investigating the most common use cases' ideal paths. We keep secondary use cases, sub-paths, and failures for the next step (see section 2.4). This approach reduces analysis complexity and helps quickly get an overview of the API capabilities.

For this task, we use a subset of the API Capabilities Canvas introduced in section 2.2, shown in figure 2.5. It focuses on identifying who does what and how (users, use cases, steps), identifying steps' inputs and successful outcomes, and spotting missing elements. In this section, we'll learn how to do this using the needs written on a sticky note, "Online Shopping."

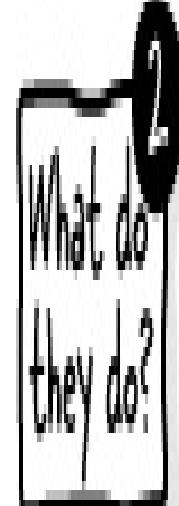
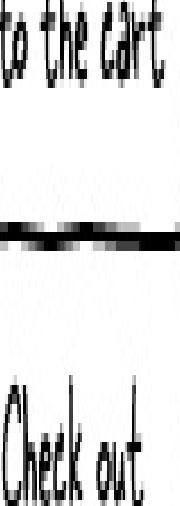
Figure 2.5 Walking the ideal paths with the API Capabilities Canvas



2.3.1 Identifying users, use cases, and steps

Walking the ideal paths starts by investigating who the users are, what they do, and how they do it, as shown in figure 2.6.

Figure 2.6 Identifying users and their use cases and steps in the API Capabilities Canvas

USER	USE CASE	STEP	INPUT	SUCCESS
End-users	Buy products	Add a product to the cart		
 Who are the users?	 What do they do?	 How do they do it?		

Identifying the different users of the API is essential to ensure we identify all capabilities. These could be the consumers (applications, the developers creating them, or the organization they belong to), the end-users of the applications, or the profiles of users or consumers. I recommend starting with the most apparent population, the one that comes to your or SME's mind first, or the one representing 80% of the users. In our "Online Shopping" example, the most prominent users are the end-users who will do online shopping via a mobile application or website consuming the API.

Note

Identifying users also has important implications regarding security. We'll discuss this in chapter 11.

Once we have identified a first who, we list what they do; these are the use cases to be covered by the API, hence high-level actions, processes, or flows performed by the users. Let's again choose the first use case that may pop into our minds or the most common one. What do the end-users of an "Online Shopping" application do? They "Buy products."

Stopping at that level risks missing capabilities; decomposing the use case is crucial. How do the end-users buy products? They repeatedly "Add a product to the cart" and then "Check out." This use case comprises two steps.

Tip

An initial overview of API capabilities can be obtained by working at the user and use case level. It can be helpful to confirm the direction taken with stakeholders. However, to ensure exhaustive and accurate capabilities, it is necessary to investigate the steps.

2.3.2 Determining inputs and success outcomes

We determine inputs and success outcomes for each step and keep failures for later (see section 2.4). The inputs are what users need to achieve the step; they are pieces of information or business concepts. The success outcomes are what happens from the users' perspective when the step is executed without issue. They can describe inputs' states after the step, what has been created or done, or an event. All these elements are only visible from the users' perspective. They are also coarse-grained; for instance, we don't care what the properties of a product are.

What do end-users need to add a product to the cart? As shown in figure 2.7, they need a product and a cart. And what happens when a product is added to the cart? The success outcome description states, "Product added to the cart." Similarly, for "Check out," the input is a cart, and the successful outcome is "User gets an order."

Figure 2.7 Determining steps' inputs and successful outcomes in the API Capabilities Canvas

User	Use case	Step	Input	Success
End-users	Buy products	Add a product to the cart	Product, cart	Product added to cart
		Check out	Cart	User gets an order
		What is needed?	What happens?	

2.3.3 Spotting missing elements with sources and usages

Analyzing inputs and success outcomes helps identify missing steps, use cases, or users. Inputs can be user-known, managed by the API, or from earlier steps. Success outcomes may be used as inputs for other steps. Figure 2.8 illustrates this investigation for the steps of "Buy products."

Figure 2.8 Spotting a new step and a new use case by investigating "Buy products" steps' inputs sources and success outcomes usages

USER	USE CASE	STEP	INPUT	SUCCESS
End-users	Buy products	Search products to buy	Where does it come from?	How it is used?
	Add a product to the cart	Product (Search products to buy), cart (API)	Product added to the cart (Check out)	
	Check out	Cart (API)	User gets an order (Manage orders)	
	Manage orders			

For the "Add a product to the cart" step," we check where the cart and the product come from. The API manages the cart (users don't provide nor get it from another step). Users search for products before adding them to the cart; we missed an essential step. But it's fixed by adding the "Search for products to buy" step at the beginning of the "Buy product" use case. The successful outcome of that step is "Product added to the cart." It's useful for the "check out" step, but nothing new here.

We proceed similarly with the "Check out" step. It needs a cart that is managed by the API. When "User gets an order," they may want to check its

status, modify or cancel it, and even do that with all their orders. We have uncovered a new area to investigate: "Manage orders," we add it to the use case list for the end-users.

2.3.4 Analyzing the spotted elements

To investigate the newly identified elements, we proceed as before (identifying use cases, decomposing them in steps, and spotting missing elements). Figure 2.9 illustrates this analysis for the "Search products to buy" step. We identified a new step, use case, and user.

Figure 2.9 Spotting new elements when investigating "search for products to buy" steps' inputs and success outcomes

USER	USE CASE	STEP	INPUT	SUCCESS
End-users	Buy products	Search products to buy	Catalog (API), Fill catalog, filters (User)	Products matching filters (Add a product to the cart, Check prod. details)
		Check a product detailed info	Selected product (Search products to buy)	Product info (Add a product to the cart)
Catalog admins	Fill the catalog		What is needed? 1	What happens? 2

Diagram illustrating user interactions and system inputs:

- User (End-users):** Initiates the process by performing the **Buy products** use case.
- System (Catalog API):** Handles the **Search products to buy** step, requiring **Catalog (API), Fill catalog, filters (User)** as input.
- User (End-users):** Performs the **Check a product detailed info** step, resulting in **Product info (Add a product to the cart)**.
- System (Catalog API):** Handles the **Selected product (Search products to buy)** step, providing **Product info (Add a product to the cart)** as output.
- User (Catalog admins):** Initiates the **Fill the catalog** use case.
- System (Catalog API):** Handles the **Fill the catalog** step, receiving **What is needed?** (1) as input and producing **What happens?** (2) as output.
- System (Catalog API):** Handles the **Fill the catalog** step, receiving **Where does it come from?** (3) as input and producing **How is it used?** (4) as output.

We identified the step's inputs and success outcomes. Users need a catalog of products to search for products and would benefit from search filters (details aren't needed at this stage). In return, they get the "Products matching filters."

We analyzed input sources and success outcomes usages to spot missing elements. End users provide filter values, while the API manages the catalog. But catalog administrators "fill the catalog" with products. We identified a new type of user and one of their use cases. When end users get the "Products matching filters," they may add them to their cart or check the product's

detailed info beforehand. We added this step to the "Buy products" use case.

2.4 Walking the less ideal paths

Focusing only on the ideal paths would result in an incomplete API design. Once we have walked them, we must explore the less ideal ones: failures, alternative branches, and edge use cases. To do so, we continue using the API Capabilities Canvas introduced in section 2.2; figure 2.10 zooms in on what we study here.

Figure 2.10 Walking the less ideal paths with the API Capabilities Canvas

USER	USE CASE	STEP	INPUT	SUCCESS	FAILURE
		What do they do if ...?	Non-nominal branches	...	What happens in case of failure? How is it used?
	Non-nominal use cases				Steps failures
Who are the users?	What do they do?

The diagram consists of three rectangular boxes connected by arrows. The top box is labeled 'Non-nominal use cases'. An arrow points down to the middle box, which is labeled 'Non-nominal branches'. Another arrow points down to the bottom box, which is labeled 'Steps failures'.

We continue working with the "Online Shopping" example from section 2.3. We investigate failures for each step of the "Buy products" use case, add its non-nominal branches, and then step back to see how to identify and analyze other non-nominal use cases.

2.4.1 Analyzing failures for each step

Analyzing failures helps identify missing steps, use cases, or users. It is also essential for creating a user-friendly API (see chapter 9) and to develop the

implementation. For each step, we list the failures, errors, or problems that may happen from the user's perspective when the step is executed in its use case and individually, why each happens, and how to fix it. Missing or invalid inputs, data state, or business controls can cause failures.

Figure 2.11 illustrates this analysis for the steps of the "Buy products" use case; the FAILURE column is filled, and a new step is identified.

Figure 2.11 Investigating "Buy products" steps failures with the API Capabilities Canvas helps identify a new step

USER	USE CASE	STEP	INPUT	Problems? ①	Why? ②	FAILURE	How to fix? ③
End-users	Buy products	Search products to buy	Catalog			No product matching filters (Retry with diff. filters) No product found when no filters (Fill catalog)	
		Check a product detailed info.	Product			Product doesn't exist (Search products to buy)	
		Add a product to the cart	Product, cart			Product doesn't exist (Search products to buy)	
		Check out	Cart			Cart is empty (Add a product to the cart) A product is unavailable (Remove unavailable product from cart)	
		Remove unavailable product from cart	Product, cart	+	+	Product not in cart (No fix)	

What problems can occur when searching for products? No product can be found due to users providing filters with no corresponding product in the catalog or the catalog being empty. As a fix, end-users may search again with different filters, or administrators may "Fill the catalog" (as identified in section 2.3.3).

What could go wrong if "Check a product details info" is executed without a prior search? No product details may be found because the requested product may not exist in the catalog. To fix that, users can "Search products to buy" to find products existing in the catalog and try again.

"Check out" can fail if the cart is empty, which can be fixed with "Add a product to the cart." The cart may also contain an unavailable product; the user can "Remove unavailable product from cart" to fix this. It's a new step we add after "Check out."

Like for other steps, we investigate what is needed to achieve it and what happens in case of success and failure. Users need the cart managed by the API and the product indicated in the failure of "Check out." On success, the product is removed from the cart. It fails if a user tries to remove a product not in the cart; there's no fix.

2.4.2 Adding non-nominal branches on each use case

To ensure API capabilities exhaustivity, we must explore other courses of action users can take within use cases. We identify non-nominal branches by checking what may happen before or after the identified steps. We analyze new steps as usual. This work is similar to how we decomposed and analyzed use cases so far. This section provides a brief overview of non-nominal path exploration. Fill in the gaps with your knowledge of listing steps, identifying inputs, their source, success outcomes, failure outcomes, and their use and fixes. Figure 2.12 shows the "Buy products" use case completed with the steps of a non-nominal branch.

Figure 2.12 Adding a non-nominal branch on the "Buy products" use case

USER	USE CASE	STEP	What if?	INPUT	SUCCESS	FAILURE
End-users	Buy products	Search products to buy	
		Check a product detailed info	
		Add a product to the cart	
		Verify cart content	2
		Remove unwanted product from cart	
		Check out	
		Remove unavailable product from cart	

What if a user changes their mind about a product on "Check out"? They need to "Remove unwanted product from cart" (step of a non-nominal branch). To do so, they need the cart and the product (its input), which can be obtained by "Verifying cart content" (new step spotted with input source).

We could also have added these two steps thanks to our subject matter expertise. Users usually "Verify cart content" before "Check out" and may "Remove unwanted product from cart" afterward.

2.4.3 Identifying and analyzing the secondary use cases

To ensure API capabilities' exhaustivity, we must analyze the secondary use cases and users we set aside from the first pass we did in 2.3. Secondary use cases can be edge cases rarely happening or specific use cases dealing with problems, such as "Notify a problem with an order." Secondary users are less prominent users, such as the "Catalog administrators." Whatever their nature, we analyze them like the other use cases and users.

2.5 Refining steps to identify operations

After listing and analyzing use cases covering needs, we identify a unique and context-agnostic operation or function for each step. Our objective is to arrive at the fundamentals of the subject matter(s) the API deals with. It's mandatory to achieve the creation of a user-friendly and reusable API. We'll leverage these operations to "Design the programming interface" (next stage of the API design process, see section 2.1). Using the "Online Shopping" example and API Capabilities Canvas, this section explains the difference between steps and operations and shows how to identify operations by refining the steps.

2.5.1 Differentiating steps and operations

Differentiating steps from functions or operations is crucial for a user-friendly, reusable API design (see section 1.2.3). An API bloated with duplicates and highly specific operations is hard to use and reuse.

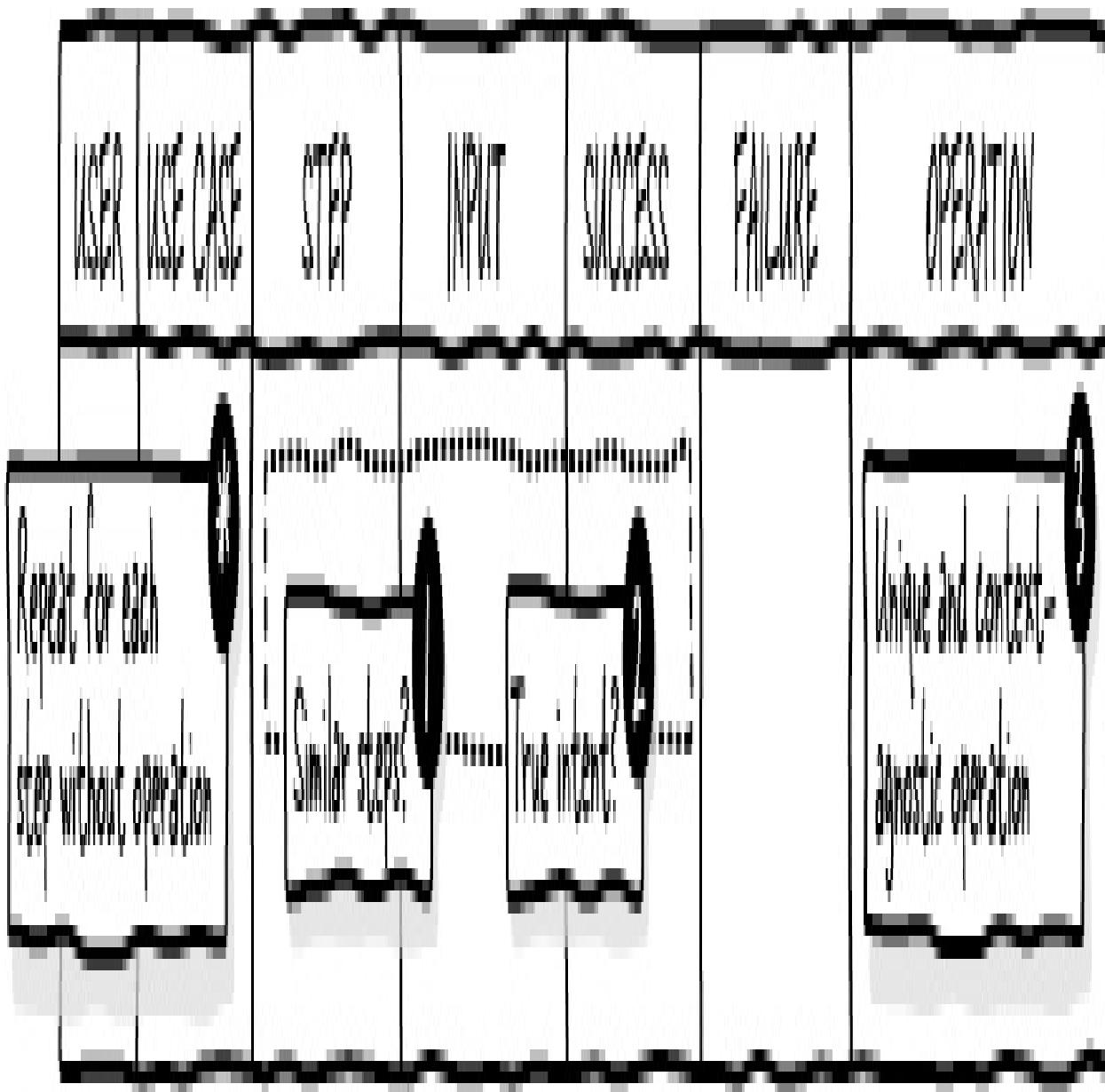
If we turn each step into an API operation as identified, we'll end up with many similar ones. For example, in the "Buy products" use case, the "Remove unwanted product from cart" and "Remove unavailable product from cart" steps are very similar. A unique "Remove product from cart" operation can fulfill them.

Steps may not be reusable in contexts other than the use case they were identified for. The "Search for products to buy" step is specific to the "Buy products" use case. A context-agnostic "Search for products" operation can fulfill it and is reusable in other situations we may encounter long after API deployment.

2.5.2 Identifying unique and versatile operations

As shown in figure 2.13, for each step, we look for similar steps in the API Capabilities Canvas and determine their true intent by leveraging their description, inputs, and success outcomes to describe a unique, context-agnostic operation fulfilling them.

Figure 2.13 Identifying unique and context-agnostic operations with the API Capabilities Canvas



The resulting operations for two "Online Shopping" use cases steps are

shown in figure 2.14. Steps marked with the same letter are similar and share the same operation. We priorly filled the "Fill catalog" use case steps (spotted in section 2.3.3) as we did for "Buy products." It involves looking for products similar to a new one, verifying their information, and adding it if it's not a duplicate.

Figure 2.14 We identified unique and context-agnostic operations for each step of the "Buy products" and "Fill catalog" use cases

USER	USE CASE	STEP	INPUT	SUCCESS	...	OPERATION
End-users	Buy products	A Search products to buy	Catalog, filters	Products matching filters	...	Search for products
		B Check a product detailed info	Selected product	Product info.	...	Get product details
		Add a product to the cart	Selected product, cart	Product is in cart	...	Add a product to the cart
		Verify cart content	Cart	Products in cart	...	List products in cart
		C Remove unwanted product from cart	Unwanted product, cart	Product removed from cart	...	Rem. prod. from cart
		Check out	Cart	Order	...	Check out
		C Remove unavailable product from cart	Unavailable product, cart	Product removed from cart	...	Rem. prod. from cart
Catalog admins	Fill catalog	A Look for similar products	Catalog, filters	Product matching filters	...	Search for products
		B Verify if product is different	Found product	Product info.	...	Get product details
		Add product to catalog	Product, catalog	Product is in catalog	...	Add prod. to catalog

The "Search for products to buy" step is similar to the "Look for similar

"products" step of the "Fill catalog" use case ("A"). Their descriptions resemble each other ("Search products ..." and "Look for ... products"), and they share the same inputs (Catalog, filters) and success outcomes (Products matching filters). Their true intent is to "Search for products;" we use it as their unique and context-agnostic operation description.

The same goes for "Check product details before buying" and "Verify if product is different" ("B"). They have the same fundamental intent and operation, "Get product details."

The "Verify cart content" step has no similar steps. Its description is specific and doesn't clearly express the actual intent. We can find it by looking at the success outcome; it returns the list of "Products in cart." Its operation is "List products in cart."

As for "A" and "B", "Remove unwanted product from cart" and "Remove unavailable product from cart" ("C") are similar. We can remove the context-specific "unwanted" and "unavailable" qualifiers from their descriptions to get their operation: "Remove product from cart."

The "Check out" and "Add a product to the cart" steps have no similar steps, and their descriptions are context-agnostic; we can keep them for their operation.

2.6 Focusing on the proper needs

The API Capabilities Canvas helps us design a versatile API that meets needs. However, certain factors can affect our analysis. We must carefully filter, transform, or accept the elements (users, use cases, steps, inputs, outcomes, operations) for the API's greater good. This section discusses staying on track during the needs analysis by remaining within the scope identified during the Define stage, focusing on the proper perspectives, and leveraging the "Why?" question.

Note

Needs coming from the Define stage can be off-track, too. It's essential to

challenge them if that's the case.

2.6.1 Staying within the Define stage's needs scope

Focusing on the proper needs requires ensuring all elements fit in the scope of the requirements defined during the Define stage. We can request confirmation, check subject matter(s) scope, and verify outcomes usages.

When needs are unclear or coarse-grained, we can request confirmation for unsure elements. For instance, does "Online Shopping" cover the administration of the product catalog?

Elements unrelated to the needs' subject matter(s) are questionable. For instance, "Checking end-user bank account balance" looks distantly related to "Online shopping"; maybe we shouldn't include it in the scope of the API. Still, it can be OK; our analysis may uncover initially unidentified subject matters.

Verifying outcomes usages is a direct follow-up of looking for missing elements (section 2.3.3). Remove any steps whose outcomes have no use for users and aren't inputs for other steps. For example, if the "Buy products" use case has a "List product suppliers" step and users do not use this information, and it's not an input for another step, we should remove it.

Tip

Don't wait for a complete API Capabilities Canvas; take an iterative approach. Start with a list of users and use cases and validate it before further investigation. Validate and confirm newly identified topics before investigating them.

2.6.2 Focusing on the proper perspectives

Focusing on proper needs requires focusing on the correct perspectives.

Our expertise in the subject matter, software architecture, or existing implementation may lead us to expose inner workings and complexity,

making the API hard to understand and use. We must ensure each element is an actual concern for all consumers. It's up to us and SMEs to balance all consumers' needs by staying focused on the subject matter(s). Section 2.7 illustrates typical overly specific consumer needs situations. Chapter 13 discusses when we may need to adapt to consumers' constraints.

Although we design an API from the consumer's perspective to fulfill their needs, integrating overly specific ones or integrating them in an overly specific way leads to less-reusable APIs. We must ensure each element is the consumers' business. Helped by SMEs, architects, tech leads, or implementation developers, we must ensure we're not exposing the provider's perspective. Section 2.8 shows typical examples. Chapter 13 explains we may sometimes have no choice but to adapt gracefully to the provider's constraints.

2.6.3 Asking why to investigate any issue

"Why?" is a powerful question that helps investigate potential issues. For example, why should user deactivate their address when in the process of updating it? Because a user can only have one active address in the database. Which is not the consumer's business but the provider's (see section 2.8.2). Asking why several times can get us to the root of any problem and help us identify unnecessary elements or proper capabilities.

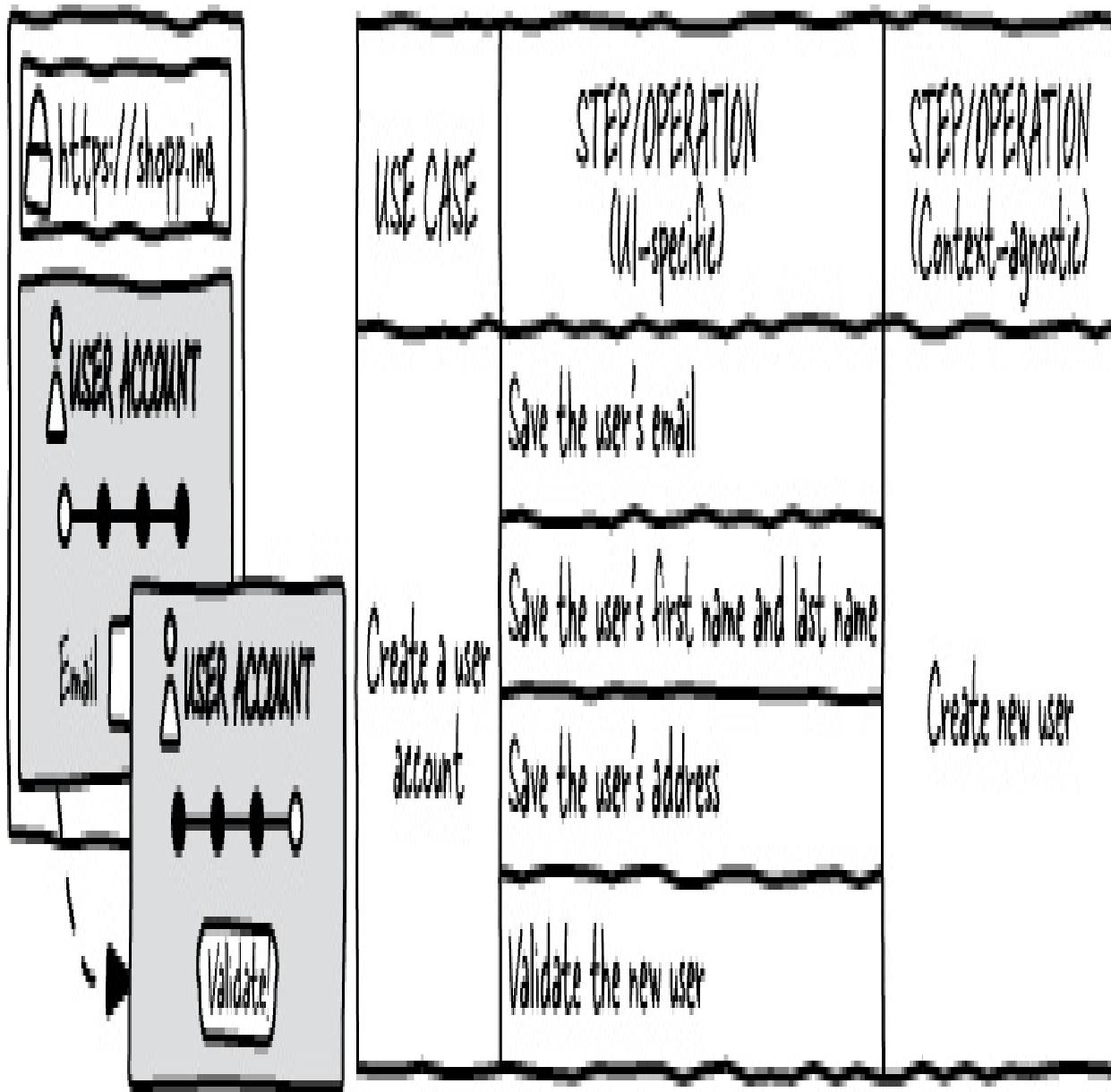
2.7 Avoiding integrating too specific consumers' perspective

Though we design an API from the consumers' perspective, we must be careful not to be too specific, or the resulting API may be usable only by one or a few consumers or may not be reusable in other contexts. Two common ways we can be too specific are mapping our API design to consumers' UI or integrating their business logic. This section illustrates them with the Online Shopping example.

2.7.1 Avoiding mapping consumers' UI

Designing an API based on existing or wireframe UI (user interface) can be helpful. Still, we must be careful not to create use case step flows representing specific UI flows instead of context-agnostic subject matter flows. UI-specific flows make APIs hard to reuse in other contexts, such as a modified UI or another application.

Figure 2.15 Contrasting UI-specific and context-agnostic use case flows



Like the UI it is based on, the "Create a user account" use case (see figure 2.15) comprises four steps/operations: "Save the user's email," "Save the

user's first name and last name," "Save the user's address," and "Validate the new user." Four UI screens to create a user account may make sense. But for an API, that means four calls, making the use case flow unnecessarily complex. Also, if the screen order changes, can we change the step order to match it? Additionally, not executing the final step could result in incomplete user accounts.

Instead, a single "Create new user" subject matter-focused step/operation is easily usable by any consumer (server application or UI). They are free to divide the information gathering into several steps. But ultimately, they will make a single API call to create a user account.

Caution

If a use case's flow mentions fine-grained information or can't stand a UI flow modification, that's a sign of a too-specific consumer's perspective. We must replace it with a generic, context-agnostic flow focusing on the subject matter.

2.7.2 Avoiding integrating consumers' business logic

Consumers may try to delegate a specific job to the API, leading to tight coupling and reduced reusability.

For example, an application leveraging our "Online Shopping" example needs to show a weather forecast pictogram based on the user's address, leading to a "get user's weather forecast" use case. We, and SMEs, can consider this highly specific to this application and unrelated to our primary subject matter, so we won't include it in our "Online Shopping" capabilities. Still, weather forecast-related features may make sense. For instance, the "search for products" operation could have a filter to get products related to weather conditions like "winter," "summer," or "rain." It would be up to the consumers to provide the condition they think is interesting for users.

Caution

An element (user, use case, step, input, outcomes, operation, or later, data

model) implying integrating concerns, business logic, or processing unrelated to or distantly related to the subject matter may be a sign of a too-specific consumer perspective. In case of doubt, check with an SME.

2.8 Avoiding exposing the provider's perspective

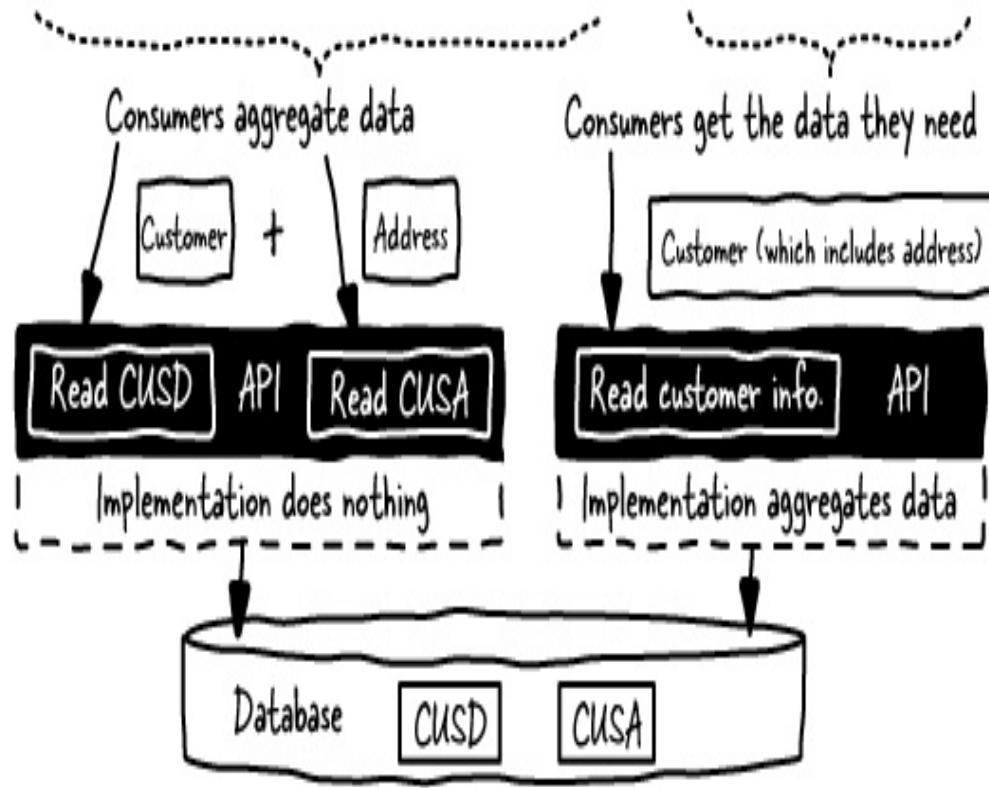
APIs reflecting the provider's perspective expose inner complexity consumers shouldn't be bothered with. They are hard to understand and use and can harm the underlying systems. Three common ways of doing so are exposing data organization, delegating business logic, or exposing software architecture. The Online Shopping example illustrates these ways.

2.8.1 Avoiding exposing the provider's data organization

An API design can mirror underlying data organization, distancing it from the fundamental subject matter and making it complex. At the API level, consumers should view data (such as a customer or product) as cohesive business concept units; the implementation must manage data complexity. Figure 2.16 illustrates how data organization can affect, or not, a use case for retrieving customer info.

Figure 2.16 Contrasting use cases exposing and not exposing underlying data organization

USE CASE	STEP/OPERATION (Provider's perspective)	STEP/OPERATION (Still provider's perspective)	STEP/OPERATION (Consumers' perspective)
A use case implying retrieving customer information	Read CUSD	Read customer data	Read customer information
	Read CUSA	Read customer address	



The first example has two steps/operations, "Read CUSD" and "Read CUSA," mapping the customer data organization in two CUSD (customer data) and CUSA (customer address) tables. Hoping they understand CUSD/CUSA are customer data; it's up to the consumer to aggregate the data to get all customer data.

The second example is similar; it replaces the cryptic names with more meaningful ones, "Read customer data" and "Read customer address." However, consumers still have to aggregate data.

The third example doesn't expose the data organization and focuses on the subject matter with a single step/operation, "Read custom info." The implementation manages data aggregation, and the consumers get the needed data easily.

Caution

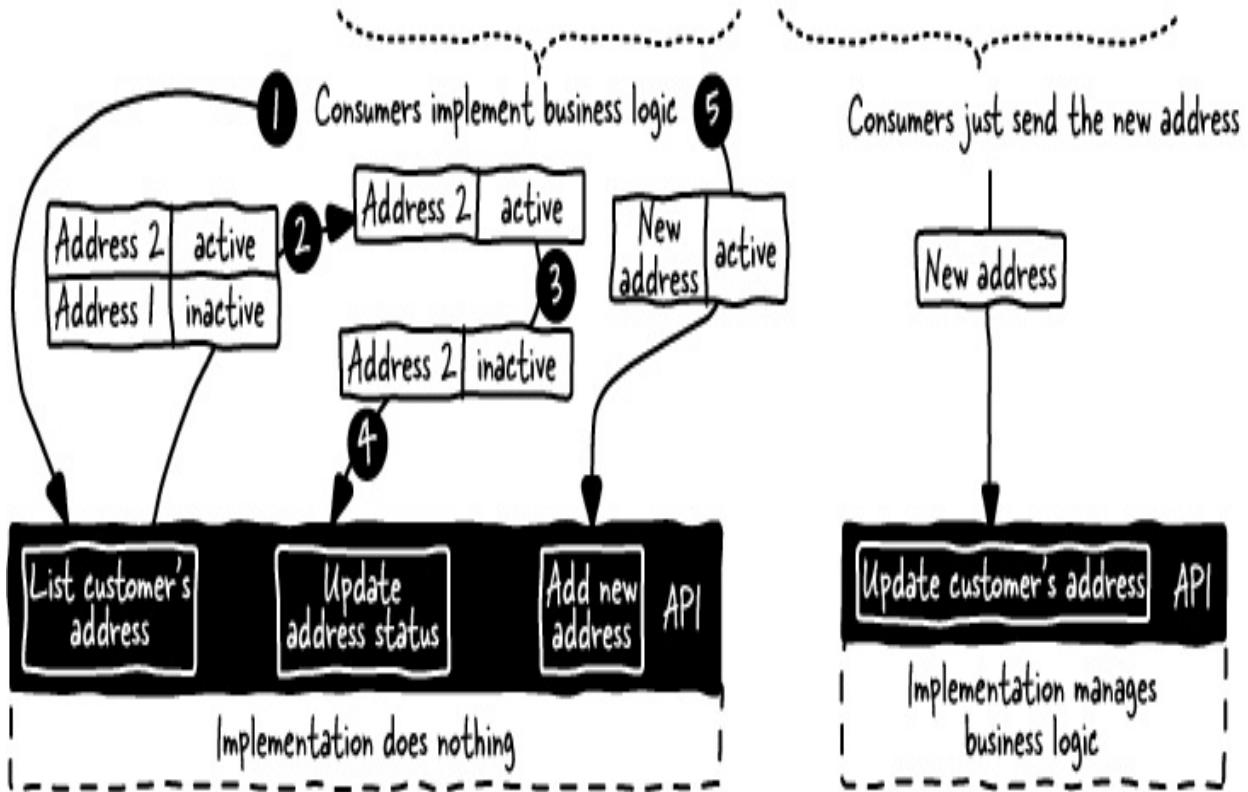
When table names are present in step or operation names or when they indicate how data is structured, it is often a sign of the provider's perspective.

2.8.2 Avoiding exposing the provider's business logic

An API design can mirror internal business logic, making it hard to use and potentially leading to underlying data and system corruption. Figure 2.17 illustrates it with an API relying on a system where older addresses are kept for security purposes, and a customer's address is the one with an "active" status.

Figure 2.17 Contrasting use cases exposing and not exposing underlying business logic

USE CASE	STEP/OPERATION (Provider's perspective)	STEP/OPERATION (Consumers' perspective)
Modifying customer's address	List customer's addresses	Update customer's address
	Update address status	
	Add new address	



Modifying a customer's address from the provider's perspective takes three steps. Users "List customer's addresses" to get the active one, "Update address status" to make this address inactive, and finally, "Add a new address" with an active status. Going through these steps and data manipulation is complex, but more critical issues exist.

These steps will be executed by an uncontrolled consumer (developed by a third party, for example) or in an unsecured environment (a browser, for instance). Due to unexpected crashes, errors in code, or malicious intent, consumers may stop at the second step, leaving a customer without an active

address, or add a new address without deactivating the active one, leading to data integrity issues.

When thought from the consumer's perspective, the use case has a single step, "Update customer's address," which ensures the implementation we control manages the business logic securely and preserves data integrity.

Warning

If incorrect API steps/operations execution can compromise underlying data and systems integrity, we trust API consumers with business logic. It's solely the implementation's responsibility to handle such logic. See chapter 11 for more secure API design considerations.

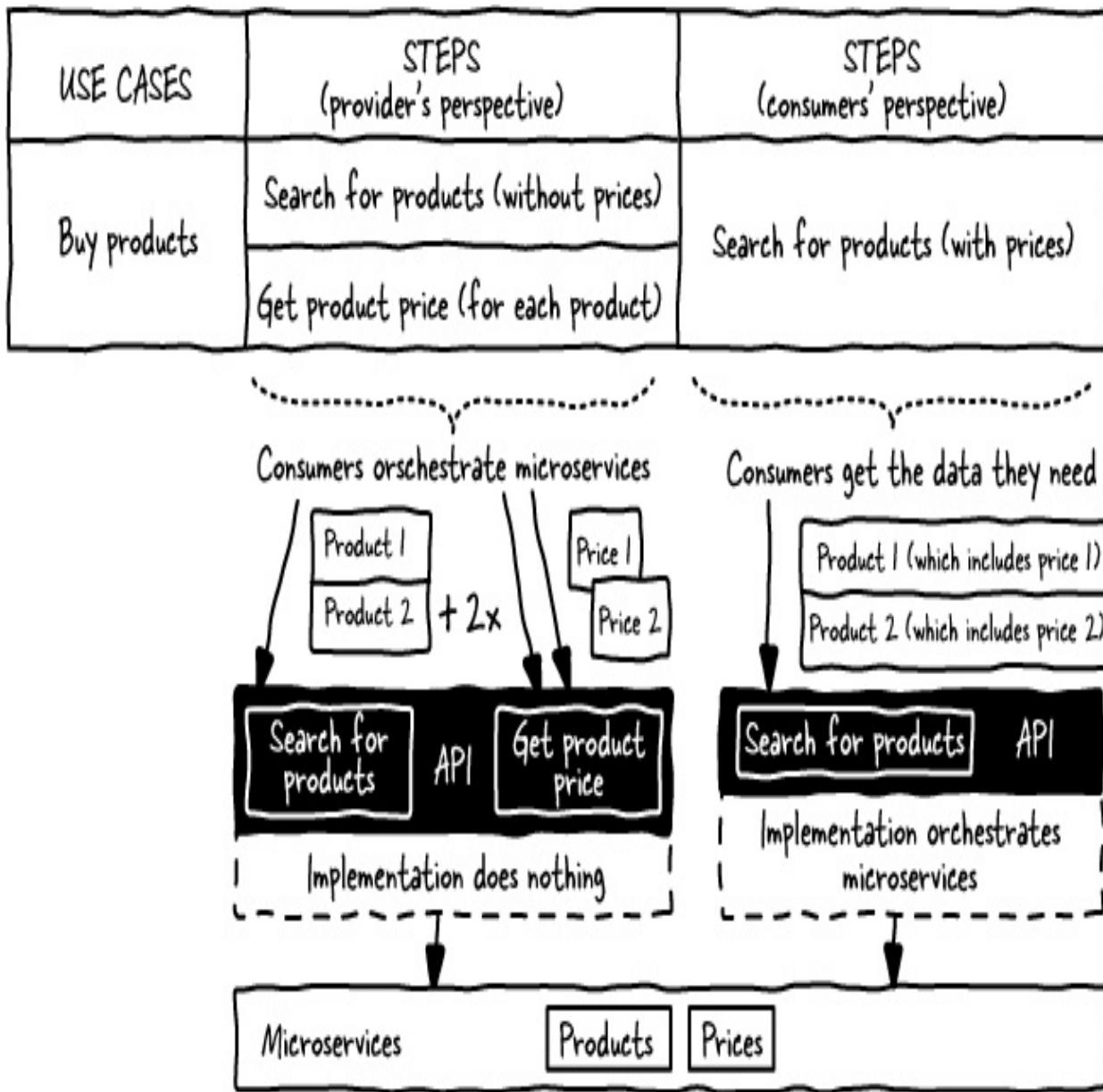
2.8.3 Avoiding exposing the provider's software architecture

APIs enable building systems from various software pieces, but exposing the composition of an API's system can lead to complex and less performant APIs. Figure 2.18 illustrates this issue with an API relying on a system composed of two microservices (or small server applications). One handles most of the products' data, and the other manages their prices.

Conway's law

Conway's law states that any organization's system design will mirror its communication structure. This adage, first published in April 1968 in Datamation magazine, applies to APIs. They are influenced by the organization's communication structure and how it exchanges and processes data across its applications.

Figure 2.18 Contrasting use cases exposing and not exposing underlying system architecture|



When buying products from the provider's perspective, users "Search for products" and loop on all products to "Get product price." A product without a price is irrelevant from the subject matter perspective of our API, so users will always do this inconvenient sequence, which also has performance concerns we'll discuss in chapter 12.

There can be excellent reasons for such an architecture, but that's none of the consumer's business, and it splits an API's business concept across operations. A single "Search for products" whose implementation handles getting their data, including their price, is preferable.

Caution

If application names appear in steps or operations or retrieving data requires complex sequences, it's probably a sign of the provider's perspective. It's up to the API implementation to deal with the complexity of different applications handling a business concept.

2.9 Summary

- The design process starts with analyzing the Define stage needs to identify API capabilities (use cases and operations needed to achieve them).
- Start with the most common or ideal use cases that go well, excluding errors, failures, or complex back and forth.
- Use cases are identified and decomposed into steps by investigating who does what and how.
- Investigating steps' source of inputs and success outcomes' usage helps uncover steps, use cases, or users.
- Explore failures, non-nominal branches, and less common use cases for a comprehensive API design.
- To identify potential failures and spot missing steps, investigate the possible problems, their causes, and solutions.
- Non-nominal subbranches are identified by wondering what other action a user can take.
- Differentiate steps and operations to ensure the API is reusable in other contexts.
- Determine the unique and context-agnostic operations needed to achieve each step by looking for similar steps and describing them in a context-agnostic way.
- Ensure that all identified elements (users, use cases, steps, inputs, outcomes, operations) are in the scope of the needs and focus on the proper perspective (not provider's or overly specific to some consumer).
- Verify outcomes usages to spot unnecessary elements.
- Don't map use case flows to UI flows and integrate consumer-specific business logic.
- Don't expose data organization, trust consumers with business logic, and expose software architecture.

3 Observing operations from the REST angle

This chapter covers

- The basics of HTTP and REST APIs
- Identifying resources and their relations
- Identifying resources' actions and their inputs and outputs

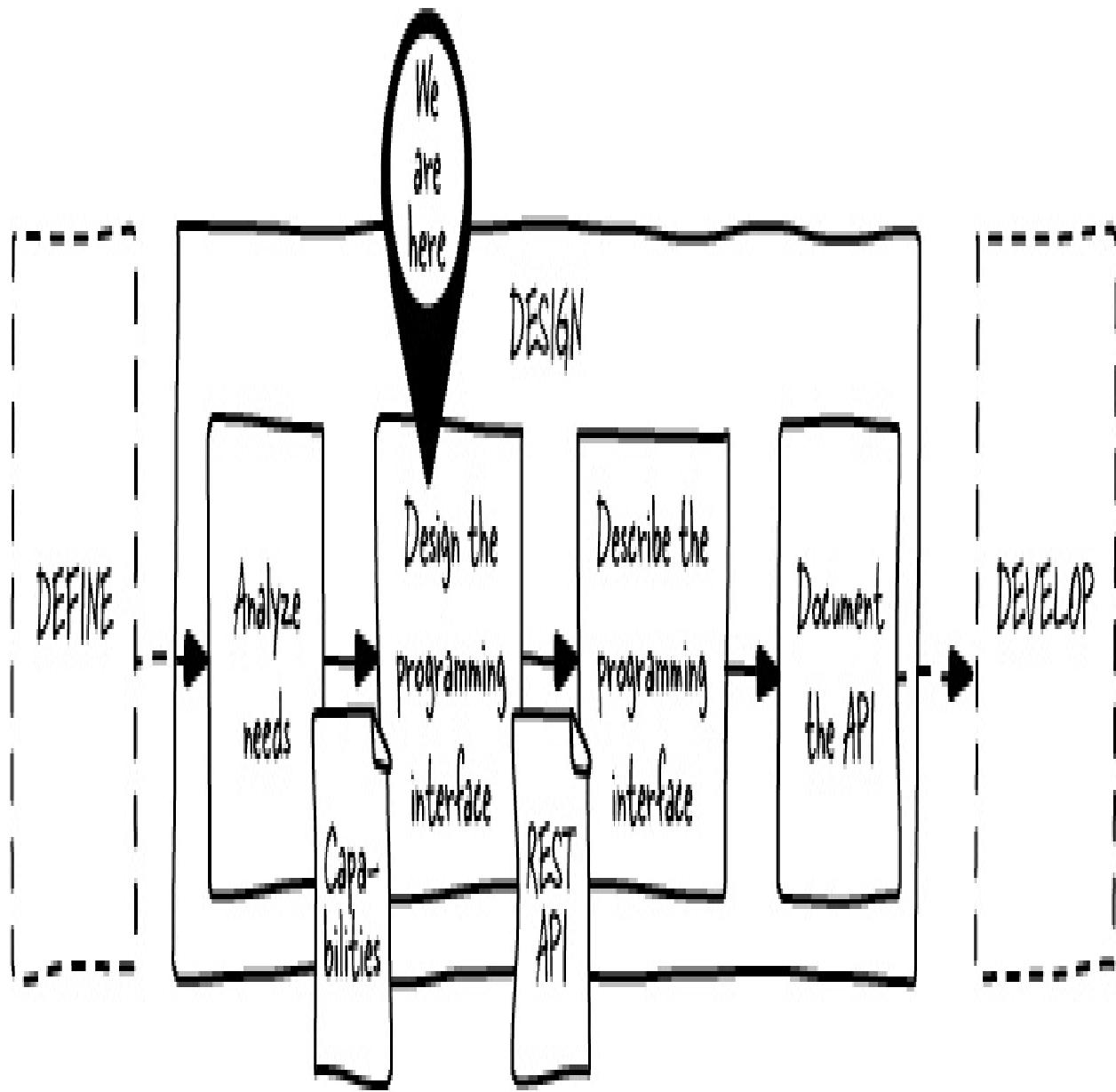
Once we've analyzed the needs and identified the API capabilities, we can design the programming interface by applying the model of a type of API (in this book, a REST API) to turn the identified operations and related elements into their programmable counterpart. Observing the identified operations from the REST angle before designing the REST programming interface facilitates the work and helps ensure accuracy and versatility.

This chapter introduces the "Design the programming interface" stage and REST APIs. Afterward, it explains how to leverage the API Capabilities Canvas to identify resources, relations, and actions. We'll turn these elements into a REST programming interface in chapters 4 and 5. It demonstrates this using the Online Shopping example of chapter 2.

3.1 Overviewing programming interface design

As shown in figure 3.1, we enter the second step of the design process outlined in section 1.6.1, "Design the programming interface," which chapters 3, 4, and 5 cover. It is preceded by needs analysis (see chapter 2). In parallel, we describe the programming interface we design (see chapters 6, 7, and 16).

Figure 3.1 We are here in the API lifecycle and design process



Caution

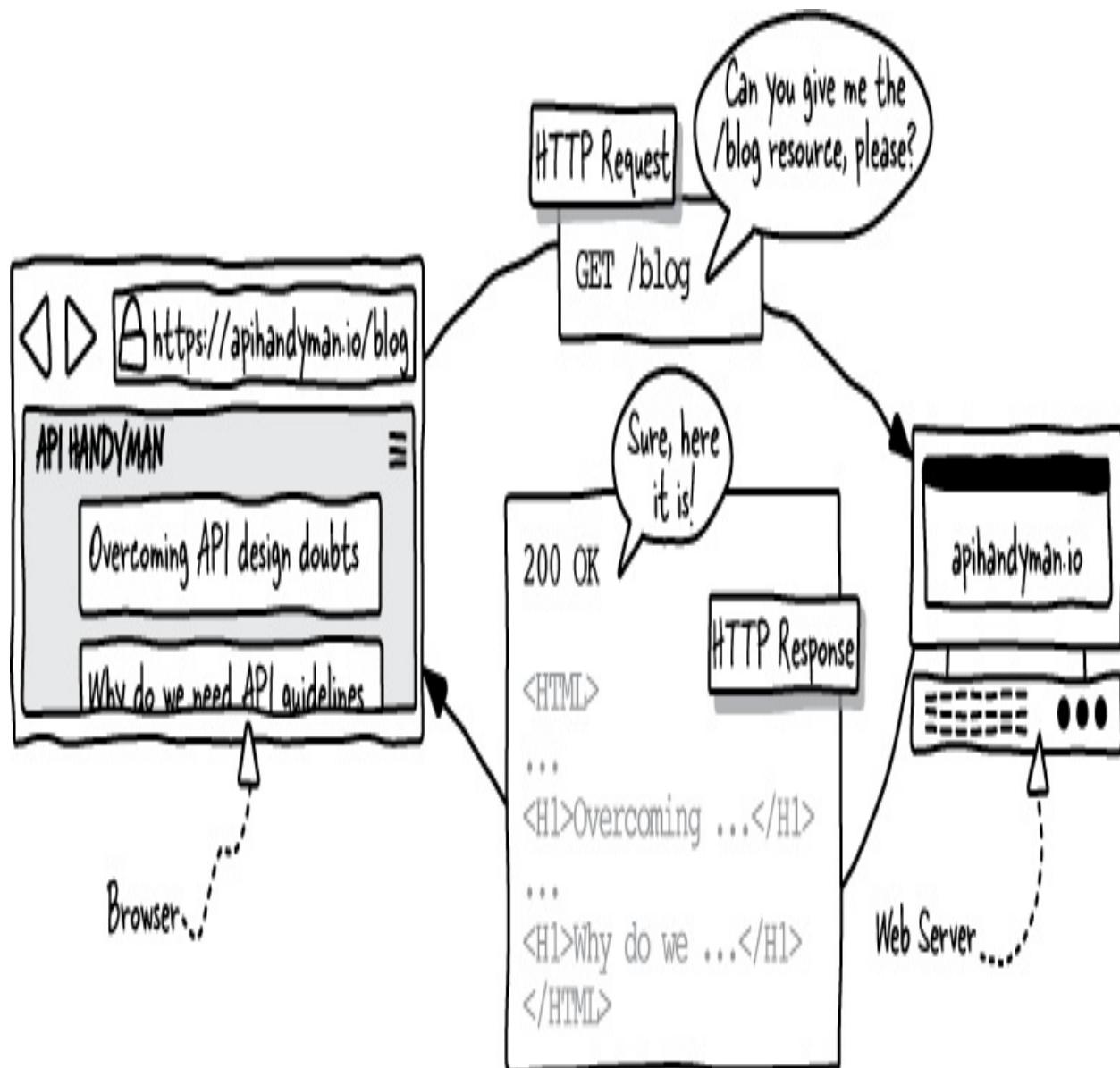
We assume a REST API suits our needs, but choosing an API style based on a rationale is crucial; section 13.5 covers how to make this decision.

This section first introduces the HTTP protocol and REST APIs. Afterward, it overviews how to design a REST programming interface. Ultimately, it discusses why we didn't discuss HTTP or REST during the needs analysis.

3.1.1 Introducing the HTTP protocol

A web browser retrieving a web page or an application calling a web API leverages the same web technology (see section 1.1.1). That is *HTTP* (hypertext transfer protocol), a synchronous, request-response protocol that allows the manipulation of *resources* with standardized *HTTP methods*. Resources can be in any format, such as HTML pages, videos, PDFs, or data in any format. Methods allow basic actions, such as retrieving (`GET`) or sending (`POST`). HTTP enables clients and servers to communicate regardless of technologies and implementation details.

Figure 3.2 A browser loads a web page with HTTP



As shown in figure 3.2, to view the list of posts on my blog, we can enter the <https://apihandyman.io/blog> URL in a web browser, which sends a GET /blog HTTP request to the apihandyman.io server. GET is a standard HTTP method meaning "give me this resource," and /blog is the path to identify the resource. The server returns an HTTP response with a 200 OK HTTP status, indicating the processing of the request went well, along with the requested page's HTML code. The browser parses the HTML and retrieves other resources, such as JS, CSS, or images, referenced in the HTML code using the same mechanism.

Listing 3.1 Retrieving a web page in Python

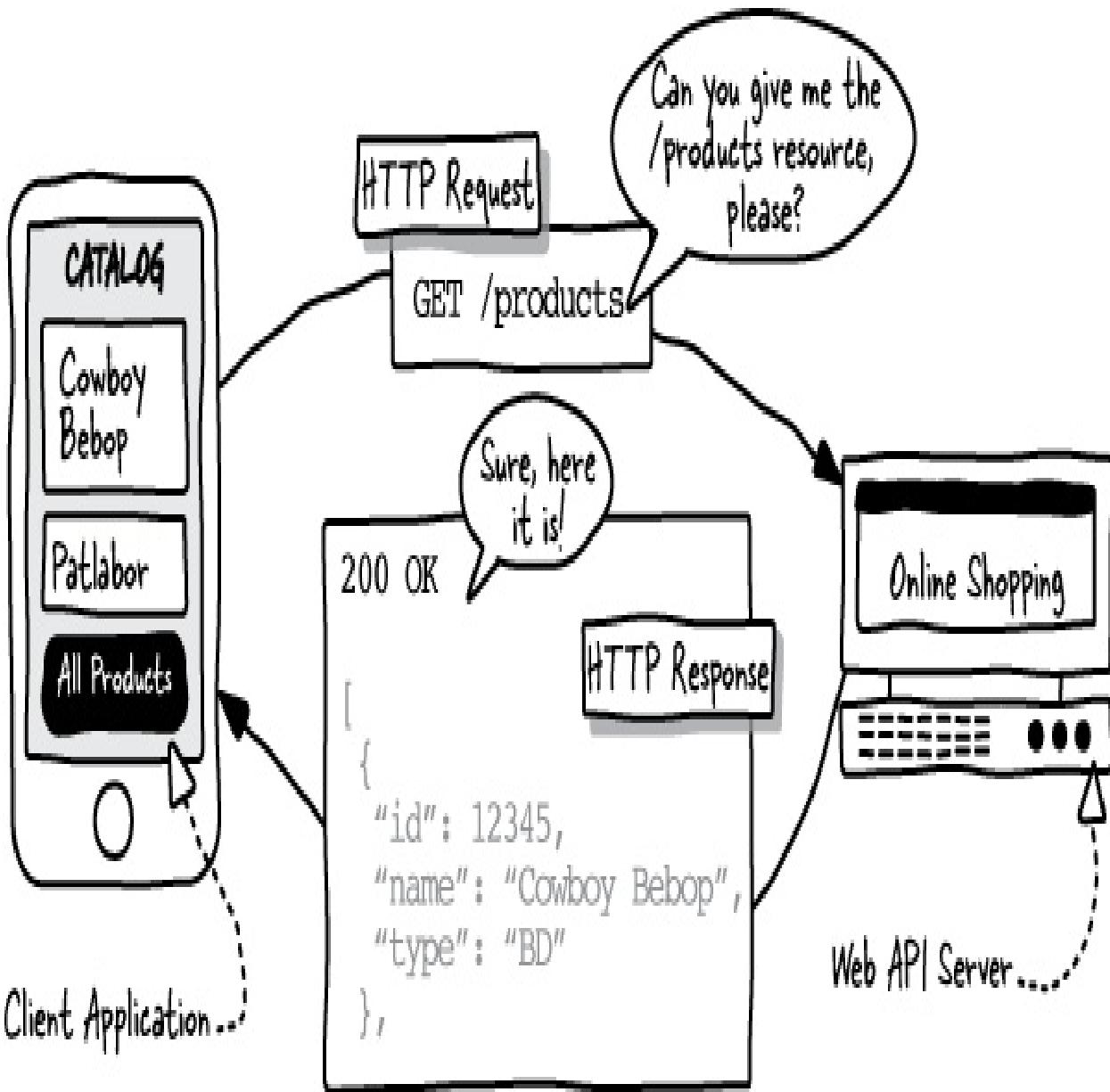
```
import requests
page = requests.get("https://apihandyman.io/blog")
print(page.text)
```

The same GET /blog HTTP request can be sent using the curl <https://apihandyman.io/blog> command line or the Python script of listing 3.1. Whether the server is a WordPress PHP application generating pages from a database or a static server loading files from the file system, it would respond with an HTTP response containing 200 OK and the HTML code for the /blog page.

3.1.2 Introducing REST APIs

Although we'll see they're more than that in section 4.8, we consider for now REST APIs as web APIs that leverage HTTP extensively and respect its semantics.

Figure 3.3 A client application calls a REST API



As shown in figure 3.3, when a client application wants to "Search for products" with the "Online Shopping" API (see chapter 2), it sends a GET /products HTTP request to the web API server. The server responds with a 200 OK status and the requested list of products regardless of how they are stored and retrieved.

From the HTTP perspective, this REST API call is no different from the blog example in section 3.1.1. However, the resource is a "business entity" or "concept" related to the "Online Shopping" subject matter instead of web resources like HTML or CSS files, and the client application gets structured

data instead of HTML.

Note

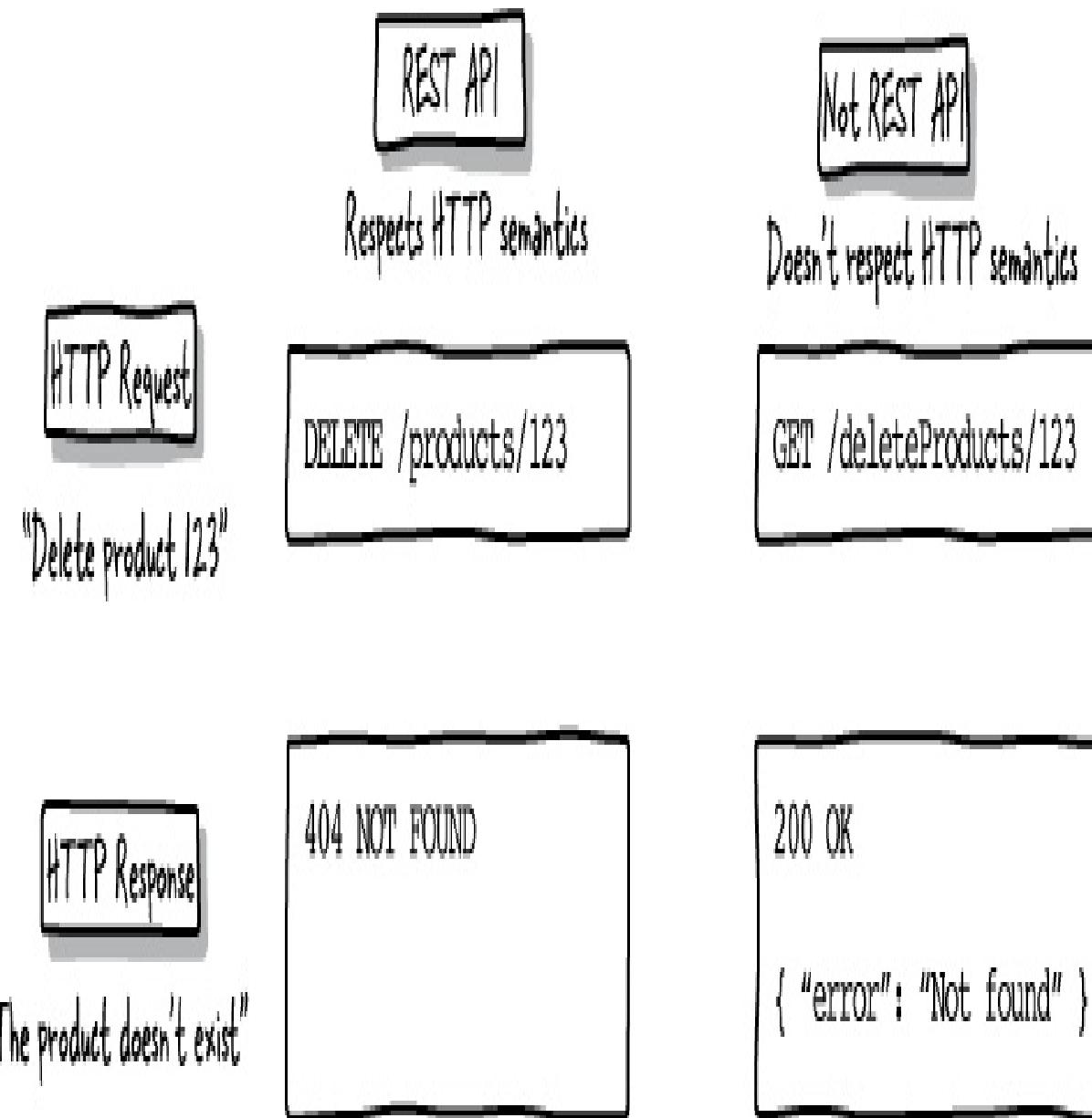
The data is in JSON format, but that doesn't matter now. Section 5.1.2 discusses this format and section 9.7 demonstrates the use of other formats.

The HTTP/REST model may look familiar to those accustomed to object-oriented programming (OOP), as an HTTP resource can be compared to an object or class, and the HTTP methods to the methods of a class or object. However, unlike OOP, HTTP is limited to standardized methods.

3.1.3 Contrasting REST and not-so-REST APIs

Some so-called REST APIs barely leverage HTTP and even go against its semantics. Figure 3.4 contrasts deleting a non-existing product with a REST and not-REST API.

Figure 3.4 Contrasting a REST and not-REST-at-all API call to delete a product that doesn't exist



With a REST API respecting HTTP semantics, the client application can send a `DELETE /products/123` HTTP request. The `DELETE` HTTP method means "delete a resource," the `/products/123` path identifies the resource to delete (a specific product with a `123` reference). The server returns an HTTP `404 Not Found` response if no such product exists (similarly to when you request a non-existing page on a website).

An API not respecting HTTP semantics would instead accept a `GET /deleteProduct/123` HTTP request and return a `200 OK` with an error message indicating the product to delete is not found. HTTP is only used to

transport messages with custom meanings that consumers can't decipher by leveraging the usual HTTP semantics. This example goes against the definition of HTTP: it leverages the `GET` ("read") HTTP method for a "delete" action and a successful HTTP status code (`200 OK`) to signify an error (Not found).

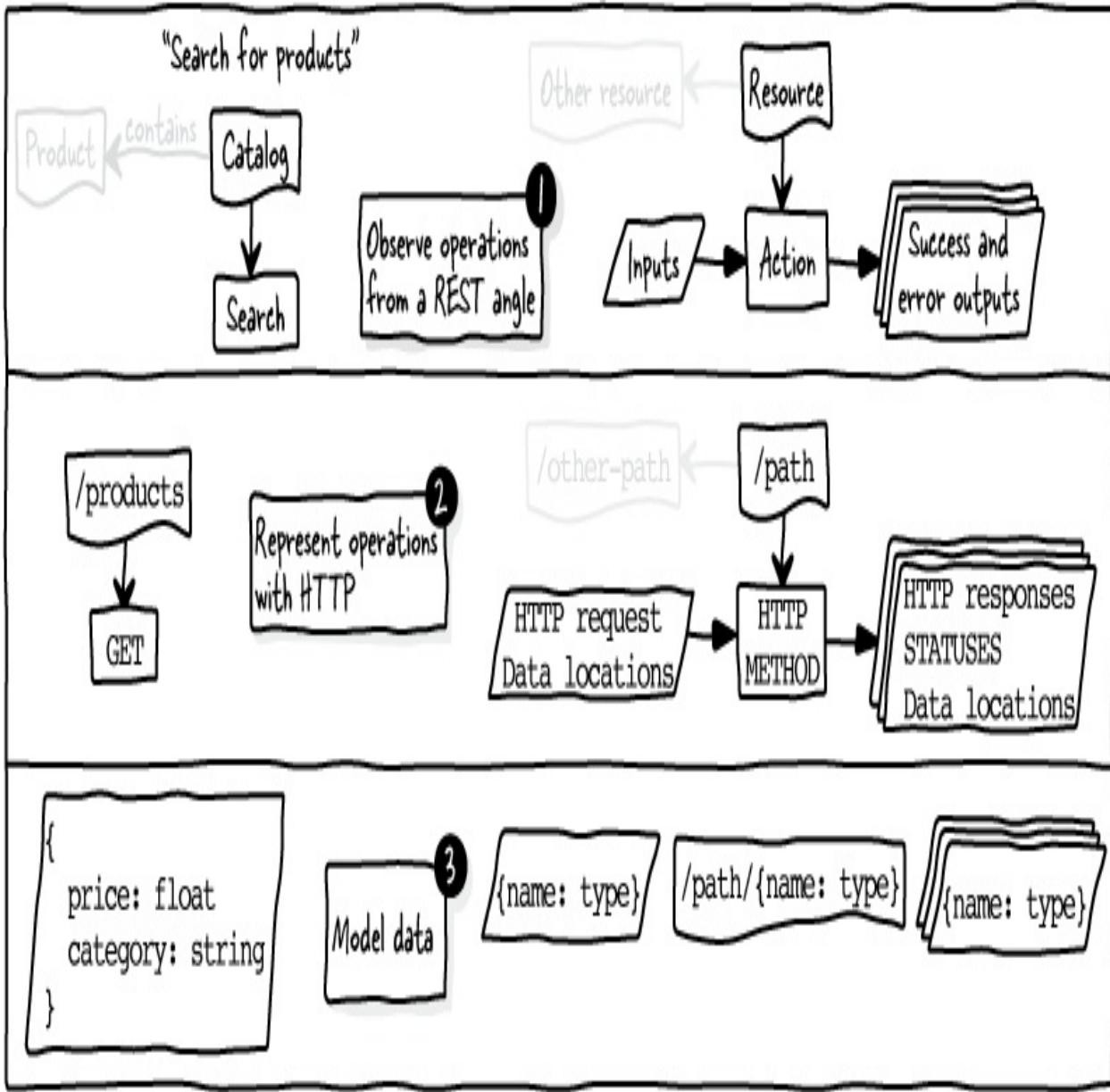
Warning

It may make sense for some web APIs to not extensively leverage the HTTP protocol and use it as a transport layer. Still, it's best to stay within the protocol's definition (discussed in chapter 9).

3.1.4 How to design a REST programming interface?

Designing a REST programming interface efficiently and accurately requires separating concerns; figure 3.5 illustrates the steps.

Figure 3.5 The three steps to design a REST programming interface



We observe operations ("Search for products") from the REST angle (see sections 3.2 and following). We identify resources ("Catalog"), their relations ("contains Product resources"), the actions that apply to them ("Search"), their inputs, and success and error outputs.

Leveraging identified elements, we represent operations with HTTP (see chapter 4). We design paths representing resources (/products for "Catalog"), choose standard HTTP methods to represent actions (GET for "Search"), pick HTTP status codes to indicate success or failure, and locate inputs and outputs data in requests and responses.

Ultimately, we design fine-grained data models (see chapter 5). We design the data of resources, operation inputs, and outputs by identifying, naming, and typing each piece of data, such as a product's category of type string.

3.1.5 Why not talk about HTTP and REST during the needs analysis?

Transforming "Search for products" into GET /products may seem simple, but using the programming interface language during needs analysis can lead to an inadequate API or complicate thinking and discussions.

Designing the programming interface during needs analysis without knowing the problem to solve may lead to a biased design; a REST API may not be the best solution, and even if it is, the usual way of designing operations may not be appropriate due to undiscovered information. The concept of resource and standard HTTP method could also taint the analysis, resulting in an API that fails to do the job.

Including programming interface considerations at this stage may lead to prolonged and unnecessary discussions. Even if REST is the adapted solution, identifying resources, designing paths, or choosing HTTP methods can be tricky. A limited view of the problem can make it more challenging. Not everyone is fluent in programming interface language and HTTP. When working with SMEs, prioritize discussions on the subject matter to avoid misunderstandings. However, they may still need to assist in identifying business concepts, related actions, or choosing names during data modeling.

3.2 Observing the API Capabilities Canvas from the REST angle

The rest of this chapter focuses on observing the API Capabilities Canvas from the REST angle to spot the REST elements needed to represent operations with HTTP (see chapter 4).

We use the API Capabilities Canvas in figure 3.6 to learn how to perform this task. It contains a subset of elements from chapter 2's "Online Shopping"

example, representing the five most typical API operations: searching, reading, creating, updating, and deleting things. These are often called CRUD operations; CRUD stands for create, read (also applies to search), update, and delete.

Figure 3.6 API Capabilities Canvas filled with typical API operations

USER	USE CASE	STEP	INPUT	SUCCESS	FAILURE	OPERATION
End-users	Buy products	Search for products to buy	Catalog, filters	Products matching filters	No product found	Search for products
		Check a product detailed info.	Selected product	Product info.	Product doesn't exist	Get product details
Catalog admins	Fill catalog	Look for similar products	Catalog, characteristics	No product found	Products matching characteristics	Search for products
		Verify if product is different	Found product	Product info.	Product doesn't exist	Get product details
		Add a product to the catalog	Product, catalog	Product is in catalog	Wrong product info.	Add a product to the catalog C
		Modify product info.	Selected product, modified info	Product is updated	Product doesn't exist	Modify a product U
		Remove a product from the catalog	Selected product	Product is removed	Product doesn't exist	Remove a product from the catalog D

This section reorganizes the canvas' information around operations and expands it to save findings. Then, it overviews how to observe operations from the REST angle to uncover what we seek.

3.2.1 Reorganizing and expanding the API goals canvas

To facilitate the observation of operations from the REST angle, we reorganize the API Capabilities Canvas around operations and expand it to save our findings.

The API Capabilities Canvas is currently organized around use cases, and the same operation may appear on multiple steps (see figure 3.6), which complexifies our task. Having the canvas in a spreadsheet makes its reorganization around operations simple. We can create a pivot table having OPERATION as the main column followed by INPUT, SUCCESS, FAILURE, STEP, USE CASE, and USER, as shown in figure 3.7. Alternatively, we can filter on the OPERATION column to select all steps mentioning a specific operation. Avoid sorting data by operations as this may disrupt the use cases' steps order.

Figure 3.7 Pivoted API Capabilities Canvas with typical API operations

OPERATION	INPUT	SUCCESS	FAILURE	STEP	USE CASE	USER
C Add a product to the catalog	Product, catalog	Product is in catalog	Wrong product info.	Add a product to the catalog	Fill catalog	Catalog admins
R Search for products	Catalog, filters	Products matching filters	No product found	Search for products to buy	Buy products	End-users
	Catalog, characteristics	No product found	Products matching characteristics	Look for similar products	Fill catalog	Catalog admins
G Get product details	Selected product	Product info.	Product doesn't exist	Check a product detailed info.	Buy products	End-users
	Found product	Product info.	Product doesn't exist	Verify if product is different	Fill catalog	Catalog admins
M Modify a product	Selected product, modified info	Product is updated	Product doesn't exist	Modify product info.	Fill catalog	Catalog admins
D Remove a product from the catalog	Selected product	Product is removed	Product doesn't exist	Remove a product from the catalog	Fill catalog	Catalog admins

We'll save resource- and operation-related findings in the "Operations" and "Resources" tables of figure 3.8. We can add them as new sheets in our API spreadsheet. Thanks to the pivot table, we already filled the OPERATION column with the unique operation names we found.

Figure 3.8 Expanding the API Capabilities Canvas with the Operations and Resources tables

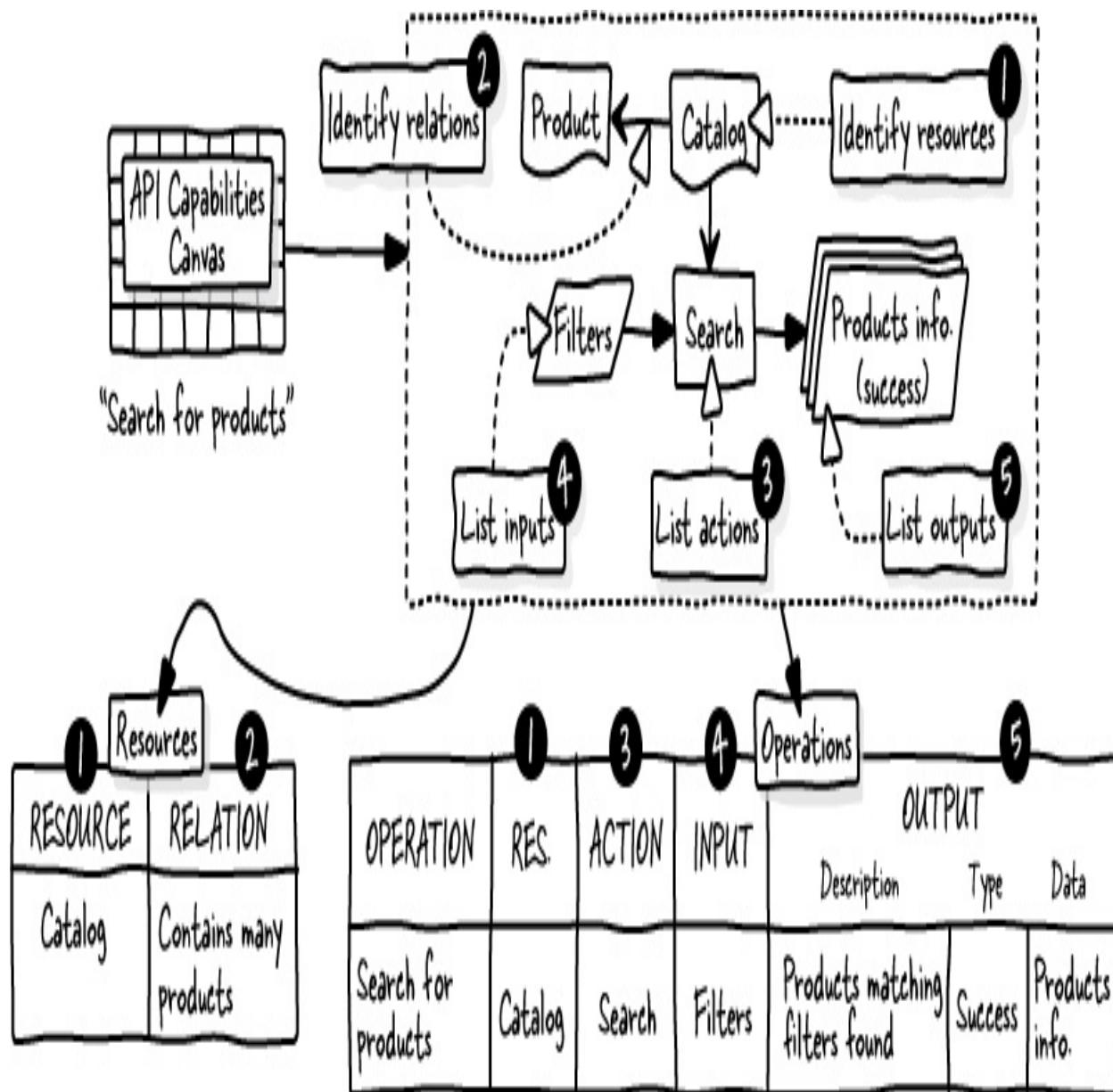
OPERATION	RESOURCE	ACTION	INPUT	OUTPUT		
				Description	Type	Data
Add a product to the catalog						
Search for products						
Get product details						
Modify a product						
Remove a product from the catalog						

The diagram illustrates the mapping of operations to resources. A vertical dashed line connects the 'Add a product to the catalog' operation to a box labeled 'Resources'. Another vertical dashed line connects the 'Remove a product from the catalog' operation to a box labeled 'Operations'.

3.2.2 How to observe operations from the REST angle

SMEs can significantly contribute to observing operations from the REST angle; this task relies on plain language and subject matter vocabulary. As shown in figure 3.9, we identify resources (or business concepts) manipulated by the operations and how they are related (section 3.3), which actions apply to them, and their inputs and outputs (section 3.4). Once done, we can move to the next step, representing these elements with HTTP (see chapter 4).

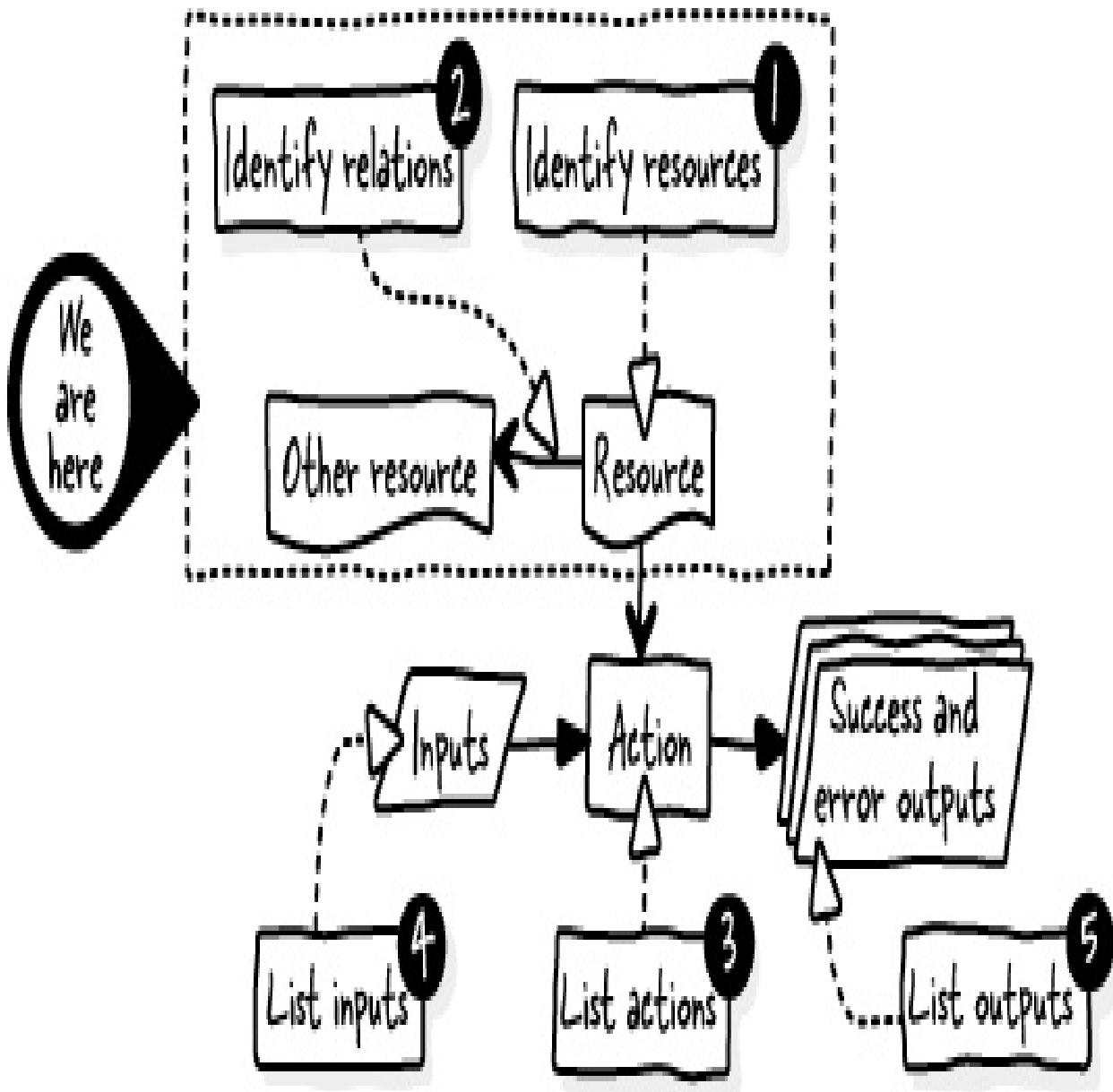
Figure 3.9 Observing operations from the REST angle and filling the Resources and Operations tables



3.3 Identifying resources and their relations

The observation from the REST angle of the API Capabilities Canvas starts with identifying resources manipulated by operations and how they are related.

Figure 3.10 Observation from the REST angle requires identifying resources and their relations



This section first discusses what a resource is. Then, using the five typical operations in figure 3.7 of section 3.2.1, it demonstrates how to identify resources and their relations. Afterward, it uncovers patterns and recipes to simplify this task.

3.3.1 What is a resource?

We've uncovered the concept of resource when introducing HTTP and REST in sections 3.1.1 and 3.1.2. In HTTP, a resource is virtually anything that a

path can represent. It will be manipulated with standard HTTP methods, hence operations. It's the same for a REST API, but before being represented by a path, a resource is a "business entity" or "subject-matter concept."

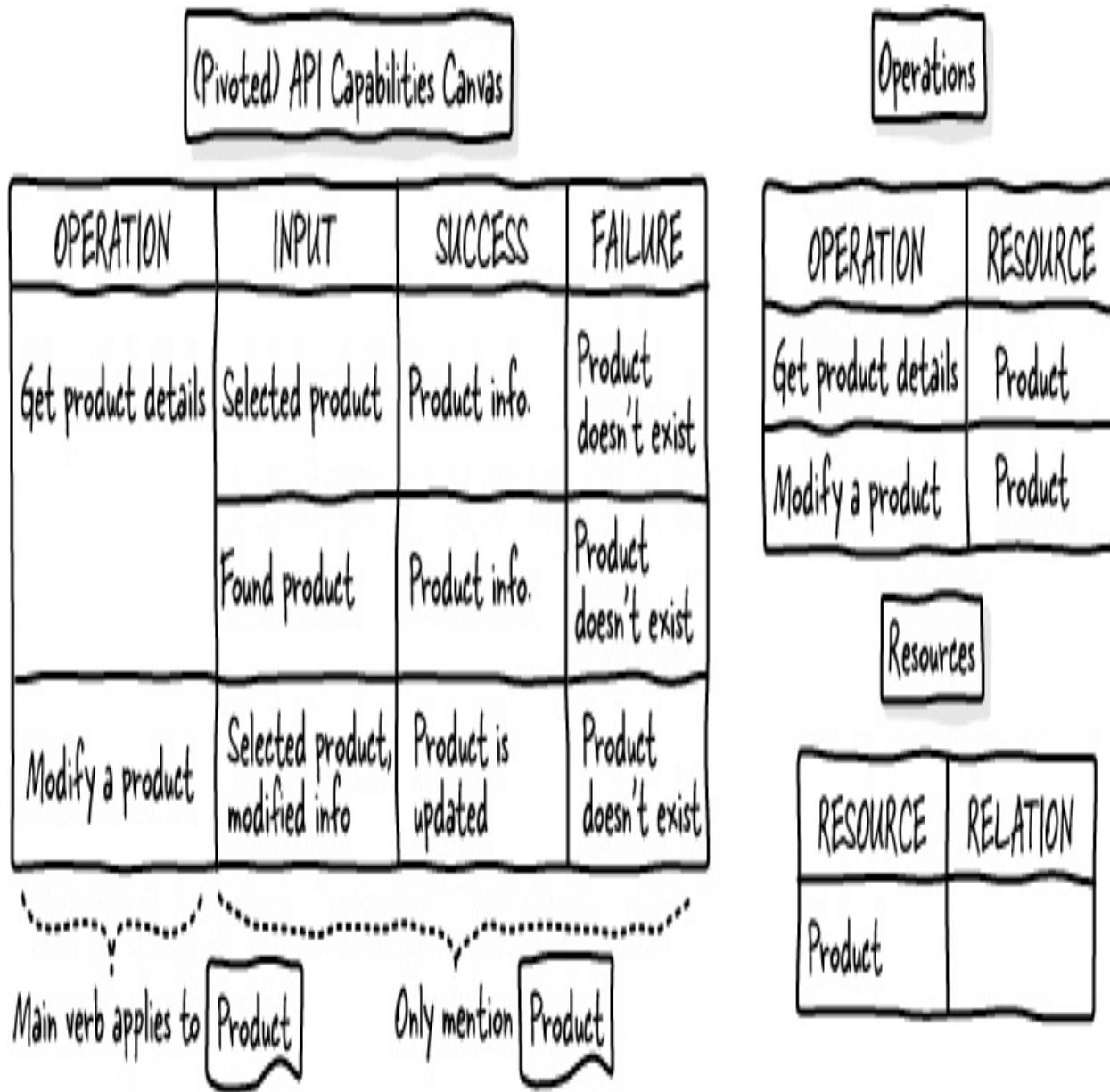
A resource is a high-level business concept or entity related to the API's subject matter(s), designated with a noun or short description using its domain's terminology (and not yet a path; see section 4.2.2). It can exist independently and be manipulated stand-alone. It should not be confused with its properties, the tiny pieces of data composing it (see chapter 5). A resource would typically be a class for those familiar with object-oriented programming.

In our "Online Shopping" example, the "Product" is a high-level subject matter concept that appeared much during the needs analysis (see chapter 2): this is a resource. On the contrary, the "name" of the product is not a resource; it is a piece of information belonging to it that can't exist on its own. A counter-example is the price of a product. It is a property of a product, but if we had pursued the needs analysis, we might have discovered that we must also manipulate prices independently as resources to get historical data.

3.3.2 Identifying an operation's resource

We leverage an operation's description, input, success, and failure to identify the resource it manipulates.

Figure 3.11 Leveraging description, input, success, and failure to identify the resource manipulated by an operation



The operation's resource is often the target of its description's main verb (see figure 3.11). In "Modify a product," the verb "modify" applies to "product." We can assume the resource is "Product." Looking at this operation's input, success, and failure confirm it; they all focus on the concept of "Product." Similarly, in "Get product details," the verb "get" applies to "product details," and the inputs, success, and failure focus on "Product." Both operations manipulate the same resource.

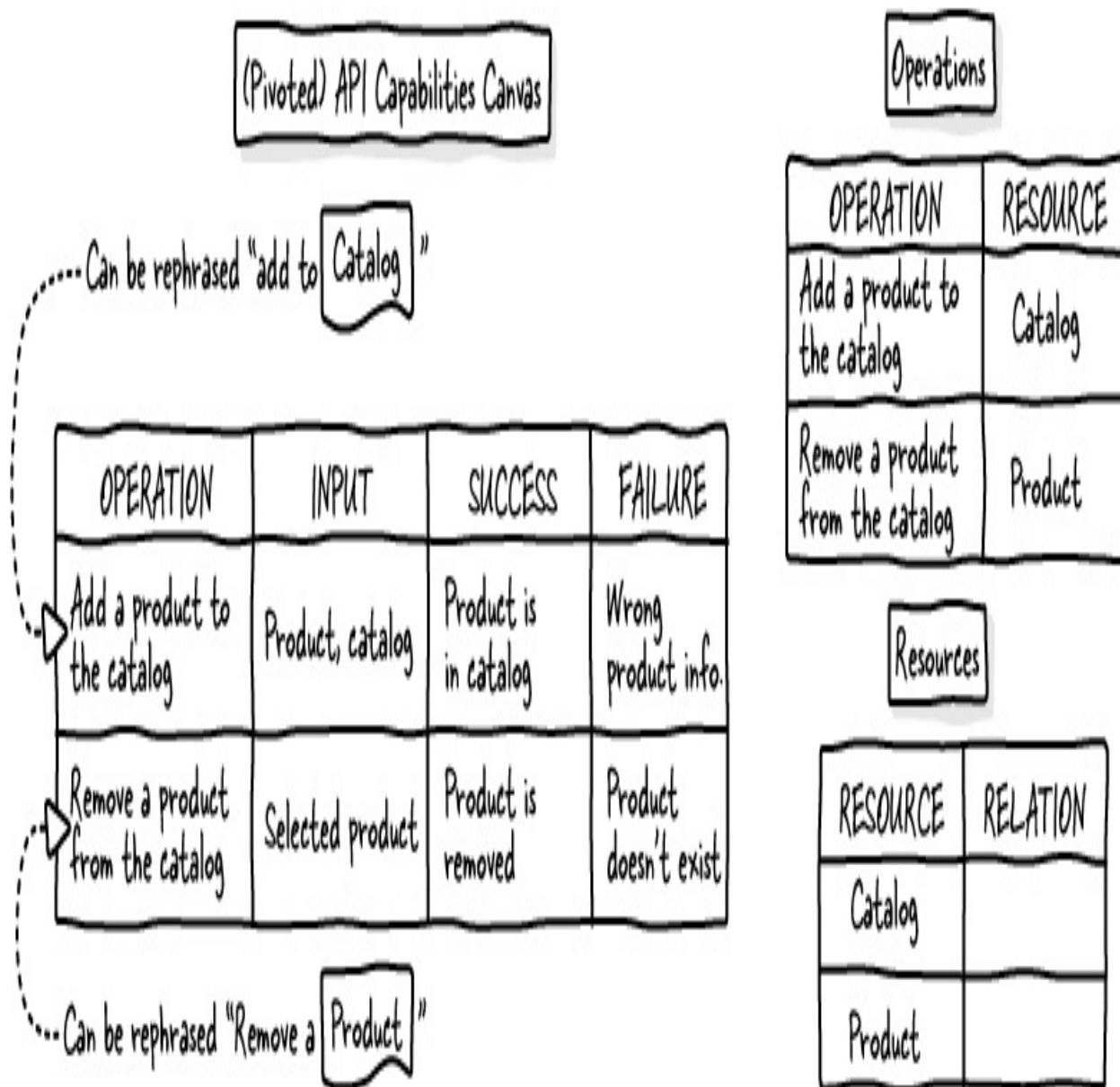
Note

An operation manipulates only one resource, and a resource can be manipulated by different operations.

3.3.3 Tweaking an operation's description to identify resource

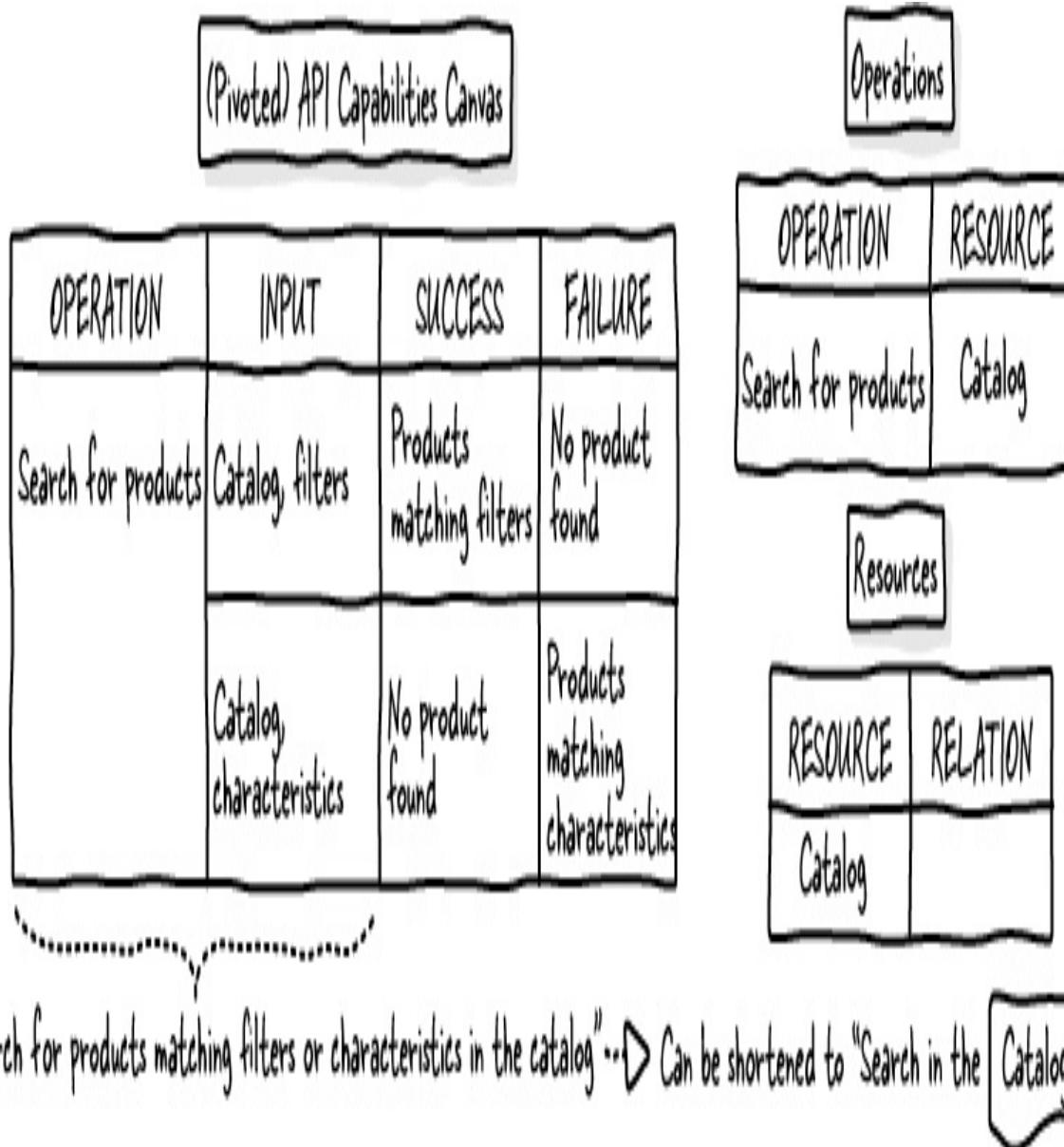
Sometimes, shortening or expanding descriptions help better identify an operation's resource.

Figure 3.12 Shortening the description to identify an operation's resource



We can shorten the operation's description to see better its resource (see figure 3.12). In "Add a product to the catalog," which of "Product" or "Catalog" is the resource? We can shorten it to "Add to the catalog" and conclude it's the "Catalog." And, though input mentions both "Product" and "Catalog," the success and failure state the product is/isn't added to the catalog. That confirms the main concept this operation manipulates is the "Catalog" and not the "Product." For "Remove a product from the catalog," we can also shorten the description to "Remove a product" and conclude this operation manipulates a "Product" (like "Modify a product" did). The input, success, and failure confirm it; they only focus on the "Product" without mentioning the "Catalog."

Figure 3.13 Expanding the description to identify an operation's resource

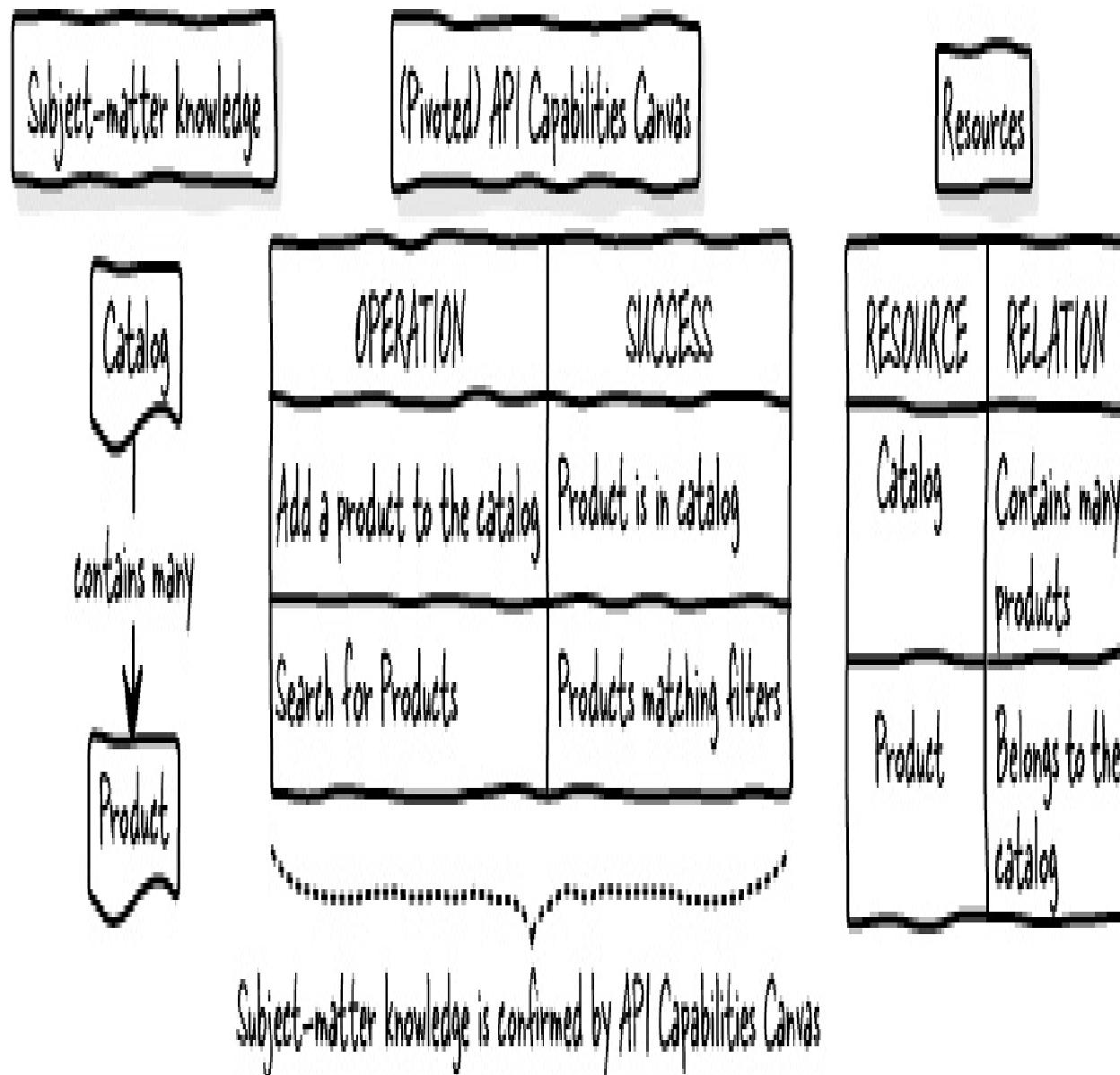


We can also expand the operation's description to identify its resource better (see figure 3.13). For example, "Search for products" requires a "Catalog" and "Filters" or "Characteristics". We can expand its description to "Search for products matching filters or characteristics in the catalog" and shorten it to "Search in the catalog." The "Catalog" is the resource we search into, just like the previous two operations.

3.3.4 Identifying resources relations

Once we've analyzed all operations and determined their resources, we identify their relations using our subject-matter knowledge and leveraging the API Capabilities Canvas information. Note that resources may not have relations depending on the subject matter.

Figure 3.14 Leveraging subject-matter knowledge and API Capabilities Canvas to identify resources relations



As shown in figure 3.14, from the "Online Shopping" subject matter perspective, it's pretty evident that a "Catalog" (of products) contains many

elements of type "Product," and a "Product" belongs to a "Catalog." In the API Capabilities Canvas, the "Search for products" and "Add a product to the catalog" operations or "Products matching filters" and "Product in the catalog" successes confirm this relationship.

3.3.5 Leveraging patterns and recipes to identify resources and relations

Analyzing descriptions and leveraging inputs and outcomes are fundamental for identifying resources and their relations. Still, we've discovered *recipes* applicable anytime we encounter typical *patterns*, such as create, search, read, update, and delete (or CRUD) operations.

Note

Along the whole design process, recognizing typical patterns and applying proven recipes facilitates the design work, helps us be more confident in design decisions, and contributes to creating excellent APIs.

The resource is the *element* when reading, updating, or deleting an *element*. The resource is the *container* of the element when creating or adding an element to a *container* or when listing or searching for elements belonging to a *container*. Also, how we describe relations between resources may depend on the subject-matter terminology, but we'll usually end with "X belongs to Y" or "Y contains X" relations.

Caution

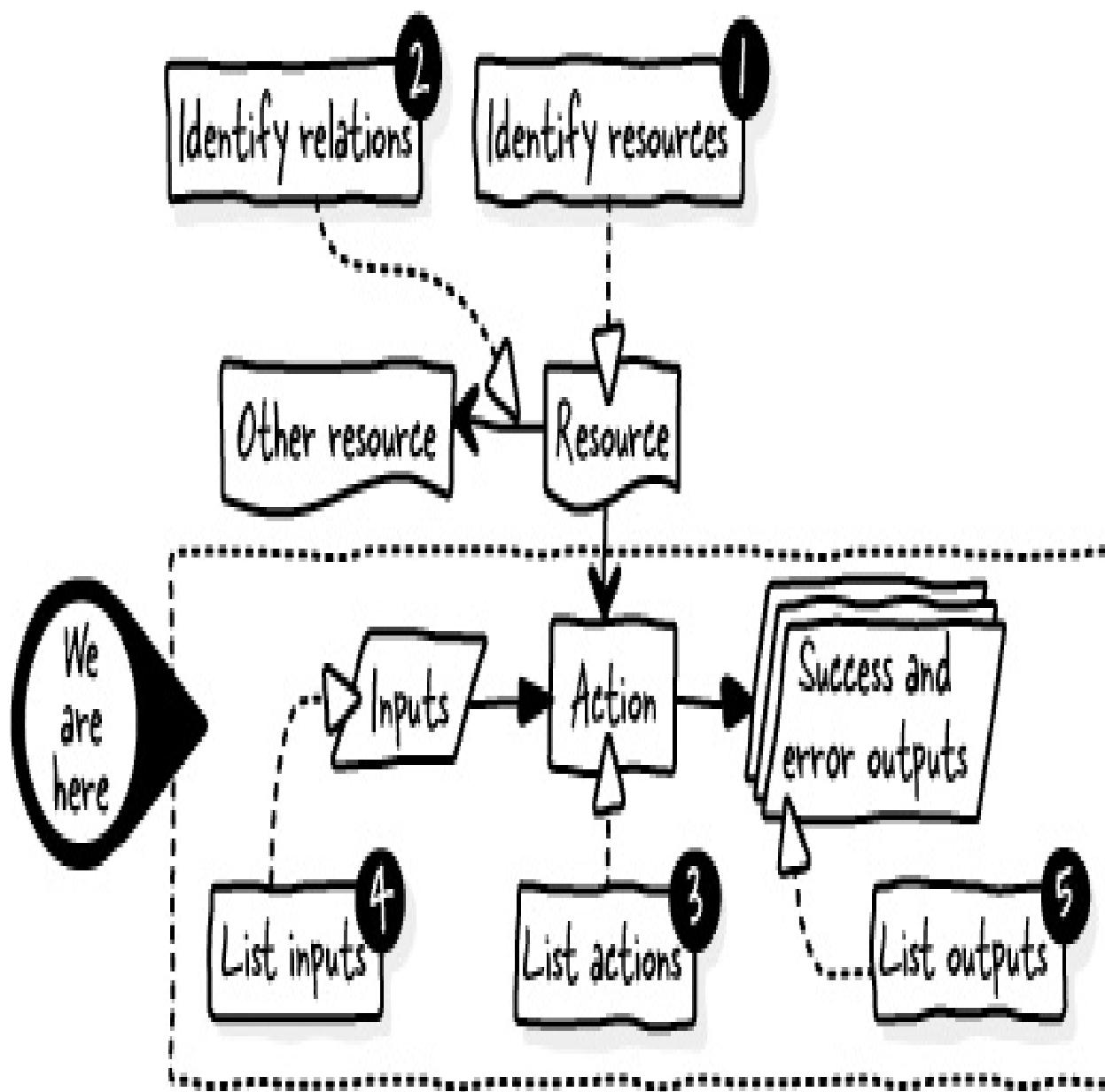
Identifying resources and their relations is similar to defining classes or tables. Use your preferred methods but avoid being influenced by pre-existing code or databases (see section 2.6).

3.4 Identifying resources' actions

Once we identify a resource (or all resources and relations), we can identify the action an operation applies to it. This section explains what an action is, demonstrates how to identify it, and lists its inputs and outputs. It leverages

the resources identified in section 3.2.2.

Figure 3.15 Observation from the REST angle requires identifying actions applied to resources and their inputs and outputs



3.4.1 What is an action, and how to identify it?

Each operation applies an action to its resource, described by the main verb from the operation's description, the same we used to identify the resource (see section 3.3). That's why we can identify an operation's resource and

action simultaneously. Figure 3.16 shows the enhanced operation descriptions we used when identifying resources so we connect the two tasks.

Figure 3.16 Identifying actions from operations descriptions

Operations		
OPERATIONS	RESOURCES	ACTIONS
Add a product to the catalog	Catalog	Add
Search for products (in the catalog)	Catalog	Search
Get product details	Product	Get
Modify a product	Product	Modify
Remove a product from the catalog	Product	Remove

...!

The main verb is the action

In "Add (a product) to the catalog," the main verb is "Add;" it is the action applied to the "Catalog" resource by this operation. Similarly, with "Search (for products) in the catalog," the main verb/action is "Search." The same goes for the three other operations, "Get product details," "Modify a product," and "Remove a product (from the catalog)," their action is, respectively,

"Get," "Modify," and "Remove."

Caution

Don't jump ahead when identifying actions (especially once you've learned how to map them to HTTP methods in section 4.3). Use raw verbs from operation descriptions and avoid replacing them with CRUD verbs or HTTP methods.

3.4.2 Listing an action's inputs

Each operation's action inputs merge the inputs of all steps leveraging the operation. We describe them in a context-agnostic way to avoid duplicates, as when identifying operations in section 2.5.2.

Figure 3.17 Merging multiple context-specific step inputs into a unique and context-agnostic action input

(Pivoted) API Capabilities Canvas

OPERATION	INPUT (source)	STEP
Get product details	Selected product (Search for prod.)	Check a product detailed info.
	Found product (Search for prod.)	Verify if product is different

OPERATION	RESOURCE	ACTION	INPUT
Get product details	Product	Get	Product reference

Steps inputs are refined into context agnostic and unique operation/action inputs...

When different steps use an operation/action, we merge inputs, as shown in figure 3.17. The "Get product details" operation is used by two steps whose inputs are "Selected product" and "Found product." They both identify a specific product found with "Search for products" regardless of its use. We discuss with SMEs what they usually use to identify a particular product; it's a "Product reference." We add it to the action's inputs.

Note

Naming can be challenging; if uncertainties exist, we can change our minds

later during fine-grained data modeling. We'll discuss naming and choosing identifiers in chapter 8.

Figure 3.18 No merging is needed when a single step uses the operation, but we may improve descriptions

The diagram illustrates the mapping between the (Pivoted) API Capabilities Canvas and the Operations table. At the top left is a rounded rectangle labeled '(Pivoted) API Capabilities Canvas'. To its right is a smaller rounded rectangle labeled 'Operations'. Below these are two tables. The left table has columns 'OPERATION' (containing 'Modify a product' and 'Remove a product from the catalog'), 'INPUT' (containing 'Selected product, modified info' and 'Selected product'), and 'STEP' (containing 'Modify product info' and 'Remove a product from the catalog'). The right table has columns 'OPERATION' (containing 'Modify a Product' and 'Remove a Product from...'), 'RESOURCE' (containing 'Product' and 'Product'), 'ACTION' (containing 'Modify' and 'Remove'), and 'INPUT' (containing 'Product reference' and 'Product reference').

OPERATION	INPUT	STEP
Modify a product	Selected product, modified info	Modify product info
Remove a product from the catalog	Selected product	Remove a product from the catalog

OPERATION	RESOURCE	ACTION	INPUT
Modify a Product	Product	Modify	Product reference
Remove a Product from...	Product	Remove	Product reference

The task is more straightforward when the operation/action is used on a single step, as shown in figure 3.18. Similarly to the previous example, both "Modify a product" and "Remove a product from the catalog" expect a "Selected product," so we add a "Product reference" to their action's inputs.

Modifying a product also requires "Modified information," which we can make more explicit with the "Modified product information" description at the action level.

3.4.3 Dealing with the operation resource when listing an action inputs

The action inputs usually exclude the operation's resource, but exceptions may exist. They may help us spot elements we missed during the needs analysis.

Figure 3.19 Uncovering a new use case when including the operation resource in the inputs

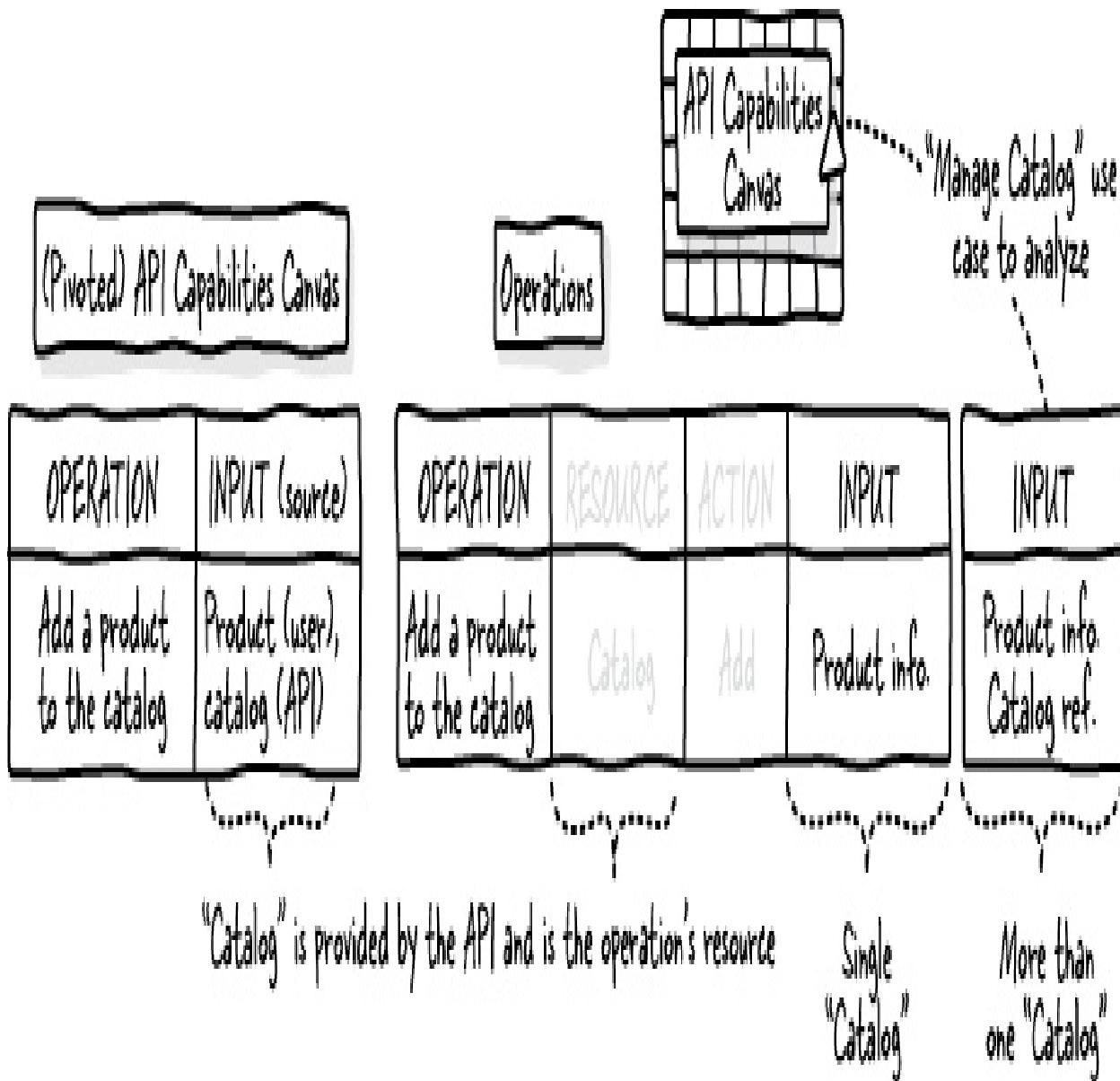


Figure 3.19 shows the "Add a product to the catalog" operation has a "Product" input, which we turn into "Product information" in action inputs to avoid confusion with the product resource. The "Catalog" operation input source is the API, and it's the operation's resource. If there are multiple catalogs, we add a "Catalog reference" to the action inputs to identify the catalog to work with and investigate a new use case, "Manage catalogs." If there's only one catalog, we don't add it to the action's input. After discussing with SMEs, we keep the one catalog option.

Figure 3.20 Merging steps inputs and excluding the operation resource from inputs

(Pivoted) API Capabilities Canvas

OPERATION	INPUT (source)	STEP
Search for products	Catalog (API), Filters (user)	Search for products to buy
	Catalog (API), characteristics (user)	Look for similar products

OPERATION	RESOURCE	ACTION	INPUT
Search for products	Catalog	Search	Filters

The "Search for products" operation is used in two steps: "Search for product to buy" and "Look for similar products." It also has the "Catalog" input, so we exclude it as it is the operation resource, and there's only one catalog. After discussing with SMEs, we merge "Filters" and "Characteristics" as "Filters," both being search criteria allowing users to find specific products.

Note

Listing an action's inputs may reveal inputs with different names that are the same. If unsure, keep them separate and re-evaluate during data modeling.

3.4.4 Listing an action's outputs

For each operation/action, we build a single outputs list containing all successes and failures cases. For each case, we have a description, type (success or error), and data (if any). As for inputs, we merge elements coming from different steps and describe them in a context-agnostic way.

Figure 3.21 Merging different steps outcomes and rephrasing descriptions

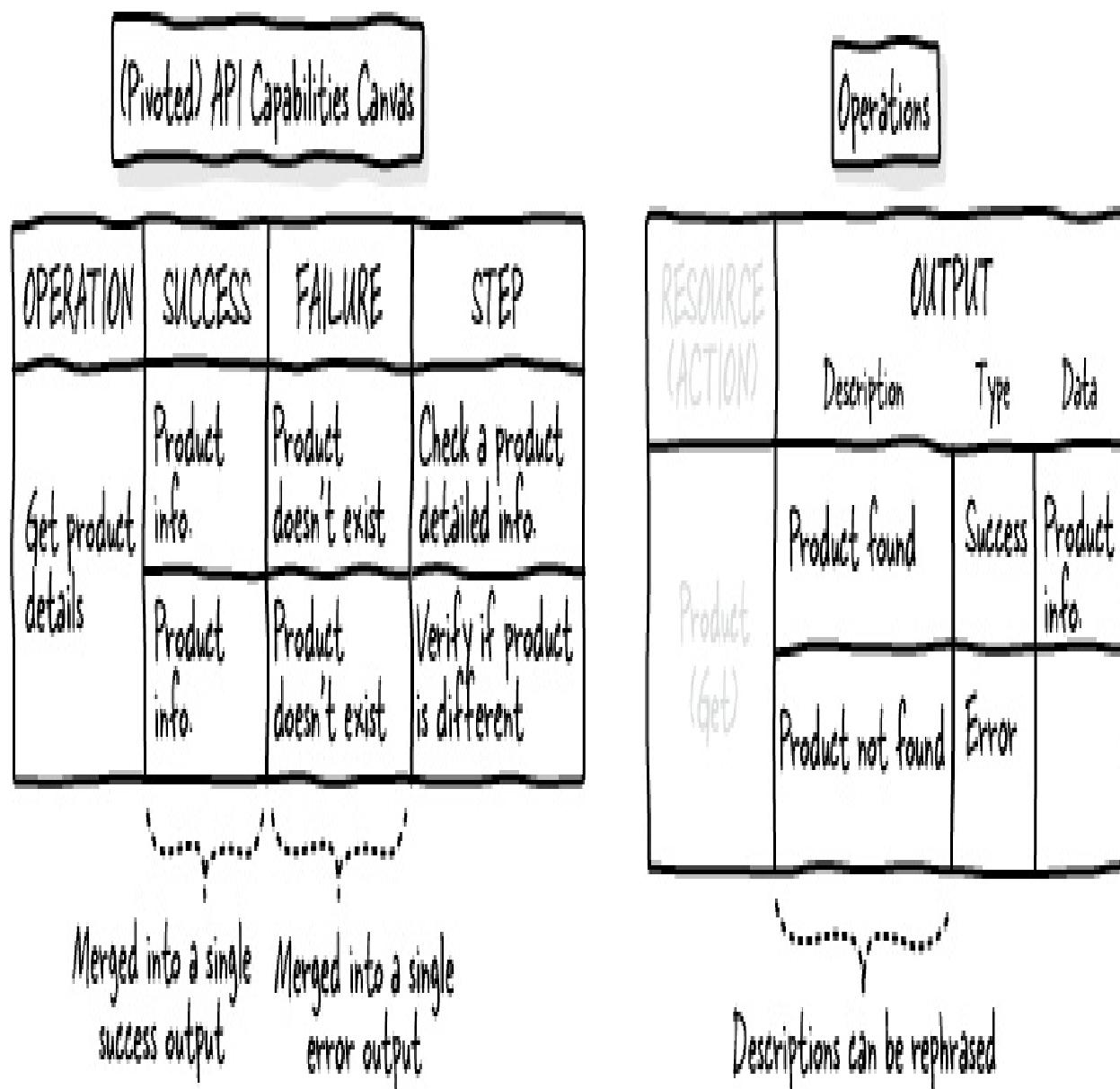


Figure 3.21 shows "Get product details" is used in steps "Check product

detailed info" and "Verify if product is different," with the same success and failure outcomes. The success outcome is "Product information," added to the actions outputs list with the description "Product found," type "Success," and data "Product information." The failure outcome is "Product doesn't exist," added to the list with the description "Product not found," type "Error," and no data.

Caution

Don't mix apples and oranges! An operation returning heterogeneous content doesn't make sense and is often a sign of wrong operation or data identification. For instance, if an operation returns "Books" and "Toothbrushes," they could be replaced by "Products." If it returns "Products" and "Providers," it should probably be split into two operations.

Figure 3.22 No merge is necessary when a single step uses the operation

(Pivoted) API Capabilities Canvas

OPERATION	SUCCESS	FAILURE	STEP
Add a product to the catalog	Product is in catalog	Wrong product info.	Add a product to the catalog
Modify a product	Product is updated	Product doesn't exist	Modify product info.
Remove a product from the catalog	Product is removed	Product doesn't exist	Remove a product from the catalog

Operations

RESOURCE (ACTION)	OUTPUTS		
	Description	Type	Data
Catalog (Add)	Product added to the catalog	Success	
	Wrong product information	Error	
Product (Modify)	Product modified	Success	
	No product found	Error	
Product (Remove)	Product removed	Success	
	No product found	Error	

We proceed similarly for "Add a product to the catalog," "Modify a product," and "Remove a product from catalog" operations, all of which are used by a single step (figure 3.21). The "Add" action's outputs are "Product added to the catalog" (Success, no data) and "Wrong product information" (Error, no data). The "Modify" action's outputs are "Product modified" (Success, no data) and "Product not found" (Error, no data). The "Remove" action's outputs are "Product removed" (Success, no data) and "Product not found" (Error, no data).

3.4.5 Dealing with contradictory successes and failures when listing outputs

Having a consumer interpret an operation's/action's success or error output as the opposite can happen depending on the context. However, determining what is considered success or error for an operation/action must be done from a context-agnostic perspective. It depends on its nature, the data being manipulated and returned, and the subject matter.

Figure 3.23 Turning context-specific steps' outcomes into context-agnostic operation/action outputs

(Pivoted) API Capabilities Canvas

OPERATION	SUCCESS	FAILURE	STEP
Search for products	Products matching filters	No product found	Search for products to buy
	No product found	Products matching characteristics	Look for similar products

RESOURCE (ACTION)	OUTPUT	Description	Type	Data
Catalog (Search)	Products matching filters found	Success	Products info.	
	No products matching filters	Success		

Interpretation of an operation output may depend on context

Context agnostic output types

In the "Search for products" operation, shown in figure 3.23, the two steps' success and failure outcomes contradict each other. When searching for products to buy, finding products is a success, and finding none is an error. When looking for similar products to avoid having duplicates in the catalog, it's the opposite.

We add both "Product matching filters found" (with "Products information" data) and "No products matching filters" (with no data) to the operation/action outputs. Then, we choose their type from a context-agnostic perspective. We set the "Product matching filters found" type to "Success"

because that is how search operations usually behave. For "No products matching filters," though both "Success" and "Error" options exist, we choose "Success" for reasons we'll uncover in section 9.6 when working on errors. That's a pattern to remember: search or list operations do not error when they find nothing.

Caution

Design decisions like making a search operation that finds nothing a success should be applied consistently to all future designs to make APIs user-friendly. See chapters 2 and 4 for more.

3.5 Summary

- HTTP is a synchronous, request-response protocol that allows the manipulation of resources with standardized HTTP methods.
- A resource can be anything, such as an HTML file or data in any format.
- REST APIs leverage HTTP extensively and respect its semantics.
- There are three steps to designing a REST API: observe operations from the REST perspective, represent them with HTTP, and model data.
- Only relying on a plain language, such as English, observe operations from a REST perspective to identify resources, actions, inputs, and outputs.
- The five typical API operations are searching for elements and creating, reading, updating, and deleting an element. Also called CRUD operations.
- A resource is a standalone business concept distinct from properties.
- An operation uses a single resource, which several operations can use.
- The resource is the target of the main verb in the operation description.
- The resource is the container for creating/adding and listing/searching elements; it is the element itself for reading/updating/deleting.
- An action is the main verb that applies to the resource manipulated by an operation.
- An action inputs list merges the inputs of the steps using it; use context-agnostic description to avoid duplicates.
- The operation resource should be removed from the action's input list unless multiple instances exist.

- An action outputs list merges the success and failure of the steps using it. Each output has a description, type (success or error), and optional data.
- Choose an action's output type from a context-agnostic perspective. Consumers may interpret success and error differently based on their context.

4 Representing operations with HTTP

This chapter covers

- Designing resources' paths
- Mapping resources' actions to HTTP methods
- Representing actions' successes and failures with HTTP status codes
- Selecting actions' input and output data locations in HTTP requests and responses
- Representing not-so-CRUD operations with HTTP
- Introducing the REST architectural style and its principles

Once we've observed the operations listed in the API Capabilities Canvas from the REST angle, we can design the HTTP-based programming interface, mapping the identified elements (resources, actions, inputs, and outputs) to HTTP but excluding data modeling (covered in chapter 5).

This chapter overviews when and how to represent operations with HTTP. Then, it demonstrates how to turn resources, actions, inputs, and outputs into their HTTP counterparts, using the Online Shopping example elements identified in chapter 3. Afterward, it explains how to represent not-so-CRUD operations. Ultimately, it discusses REST as an architectural style and its principles.

4.1 Overviewing representing operations with HTTP

As shown in figure 4.1, we're still in the second stage of the API design process outlined in section 1.6.1, "Design the programming interface." We enter the second step of this stage, "Represent operations with HTTP," preceded by "Observe operations from a REST angle" (chapter 3) and followed by "Model data" (chapter 5). Figures 4.2 and 4.3 show the

information gathered during observation from the REST angle for the "Online Shopping" example. This section quickly re-explains REST and HTTP before outlining the steps to represent operations with HTTP.

Figure 4.1 We are here in the API lifecycle and design process

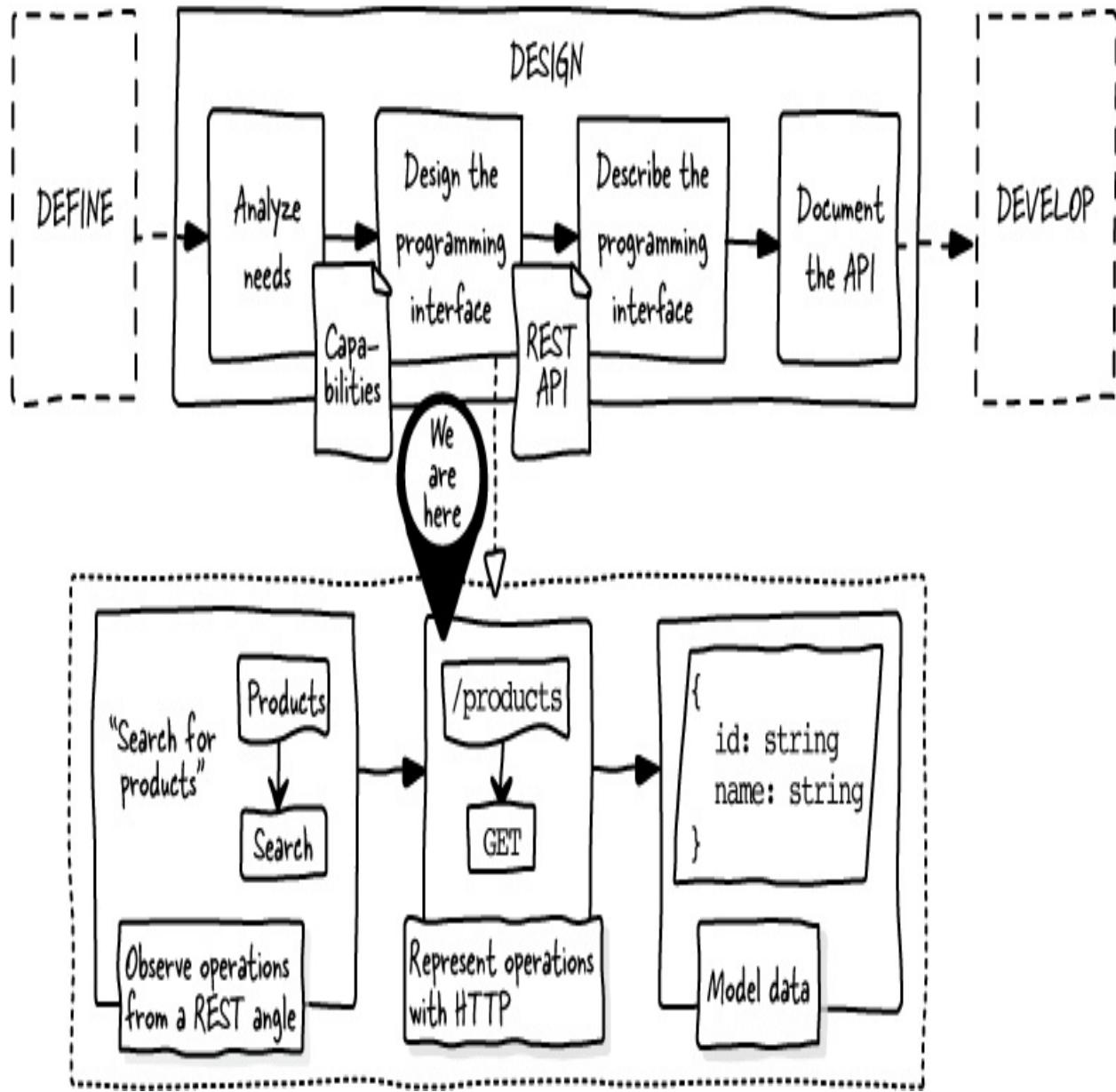


Figure 4.2 Resources identified while observing operations from a REST angle for the "Online Shopping" example

RESOURCE	RELATION
Catalog	Contains many Products
Product	Belongs to the catalog

Figure 4.3 Operations, resources, actions, inputs, and outputs identified while observing operations from a REST angle for the "Online Shopping" example

OPERATION	RESOURCE	ACTION	INPUT	OUTPUT		
				Description	Type	Data
Add a product to the catalog	Catalog	Add	Product info	Product added to the catalog	Success	
				Wrong product information	Error	
Search for products	Catalog	Search	Filters	Products matching filters found	Success	Products info.
				No products matching filters	Success	
Get product details	Product	Get	Product reference	Product found	Success	Product info.
				No product found	Error	
Modify a product	Product	Modify	Product reference, modified product info.	Product modified	Success	
				No product found	Error	
Remove a product from the catalog	Product	Remove	Product reference	Product removed	Success	
				No product found	Error	

4.1.1 Refreshing our memories about HTTP and REST

We've discovered in section 3.1.2 that REST APIs heavily leverage HTTP and respect its semantics (though we'll realize in section 4.8.1 they're more than that).

To search for products with a REST API, the consumer can send a GET /products HTTP request, where GET is a standard HTTP method representing a "Read" action and /products is a path representing the

"Catalog (of products)" resource. The API server returns an HTTP response with a standard HTTP status, such as `200 OK` to signify success, and optional data, such as a list of products. It's only a simple example; this chapter introduces more HTTP methods and statuses and different places to put data in both requests and responses.

4.1.2 How do we represent operations with HTTP?

Figure 4.4 shows how to represent with HTTP the elements, such as resources, actions that apply to them, and their inputs and outputs, identified when observing the operations from a REST angle (see chapter 3). Figures 4.5 and 4.6 show the information used and the result (highlighted with numbered bullets matching steps).

Figure 4.4 The steps to represent an operation with HTTP

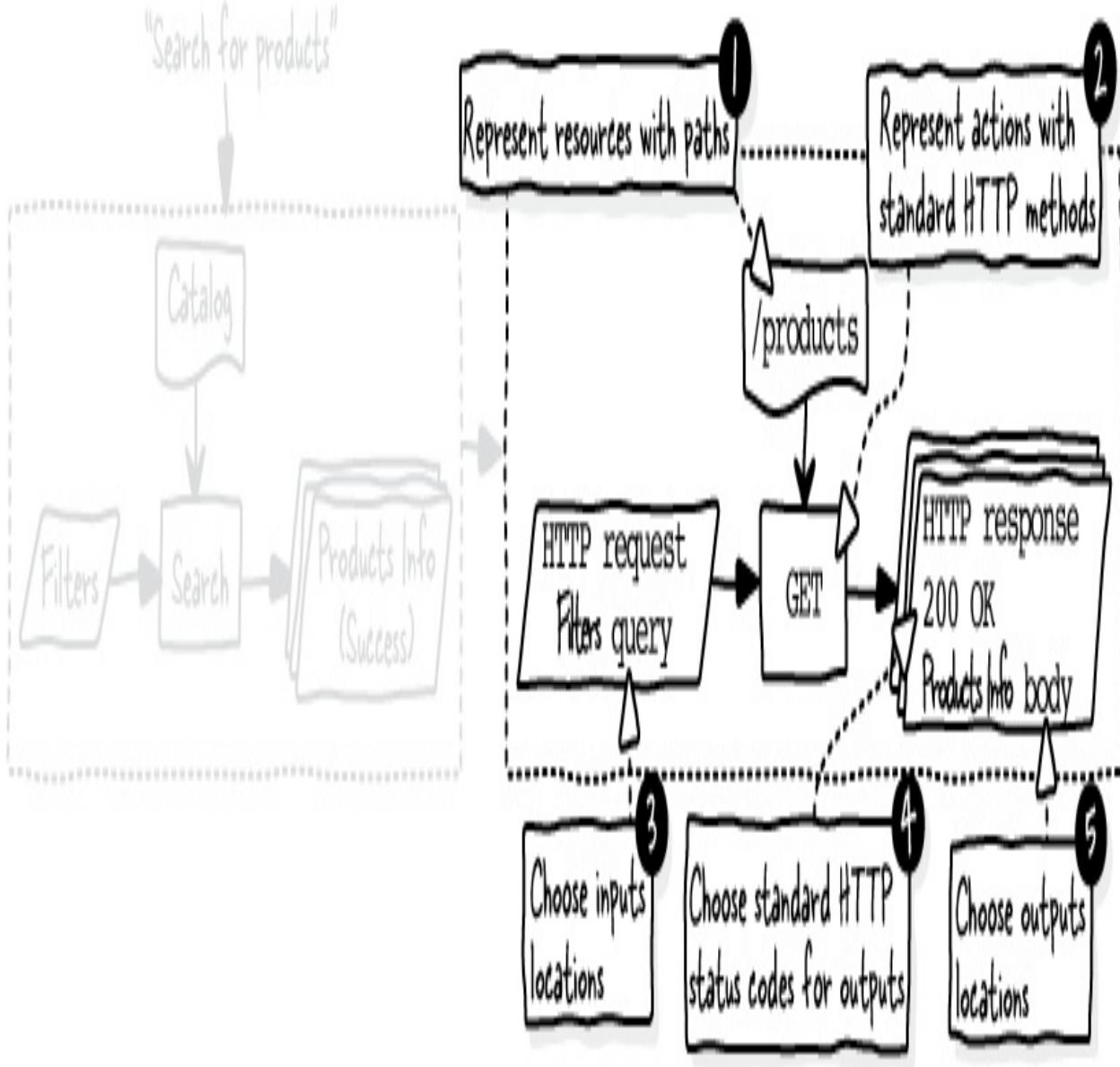


Figure 4.5 The "Catalog" resource and its HTTP counterpart (Resource table in API spreadsheet)

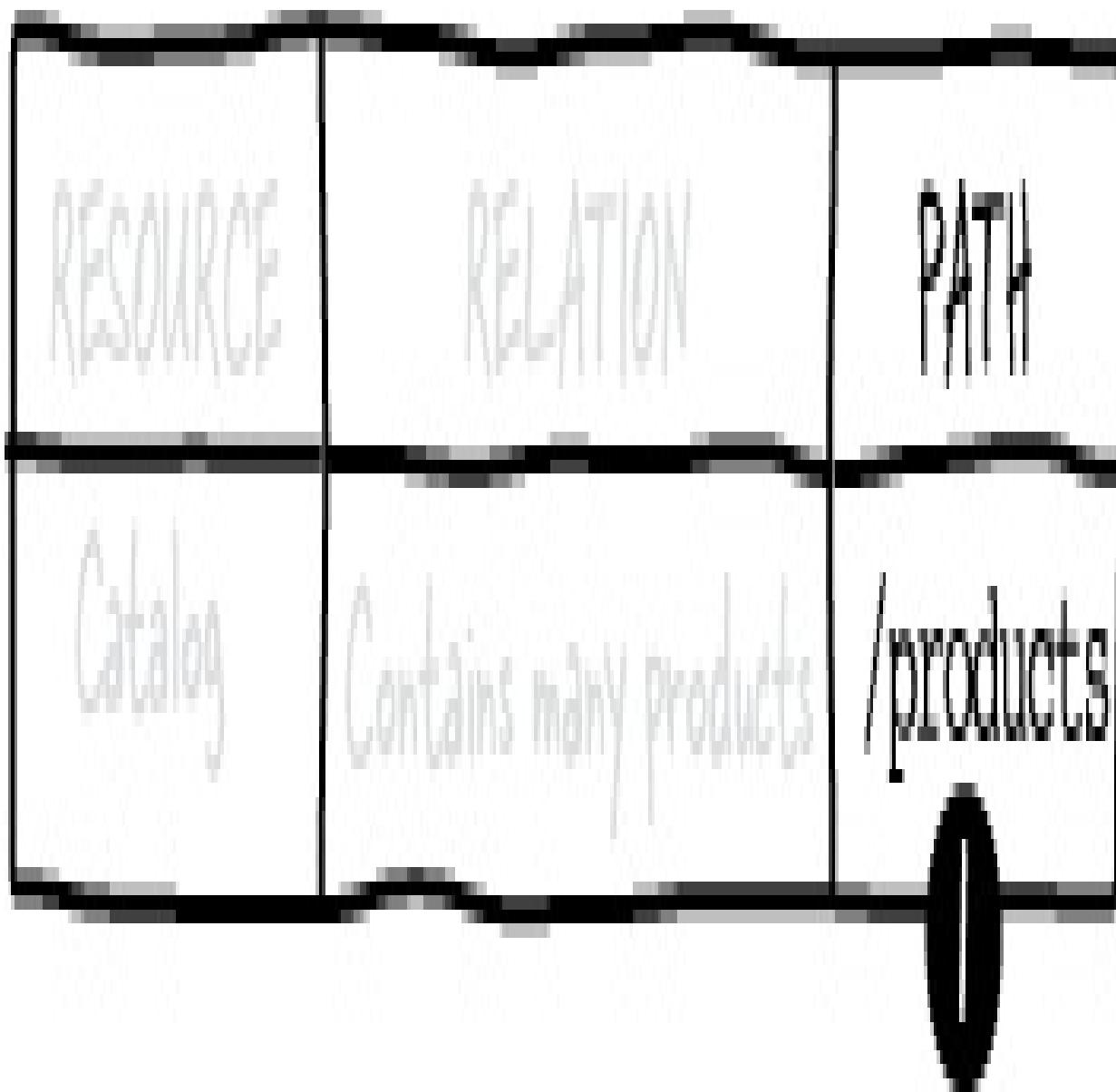
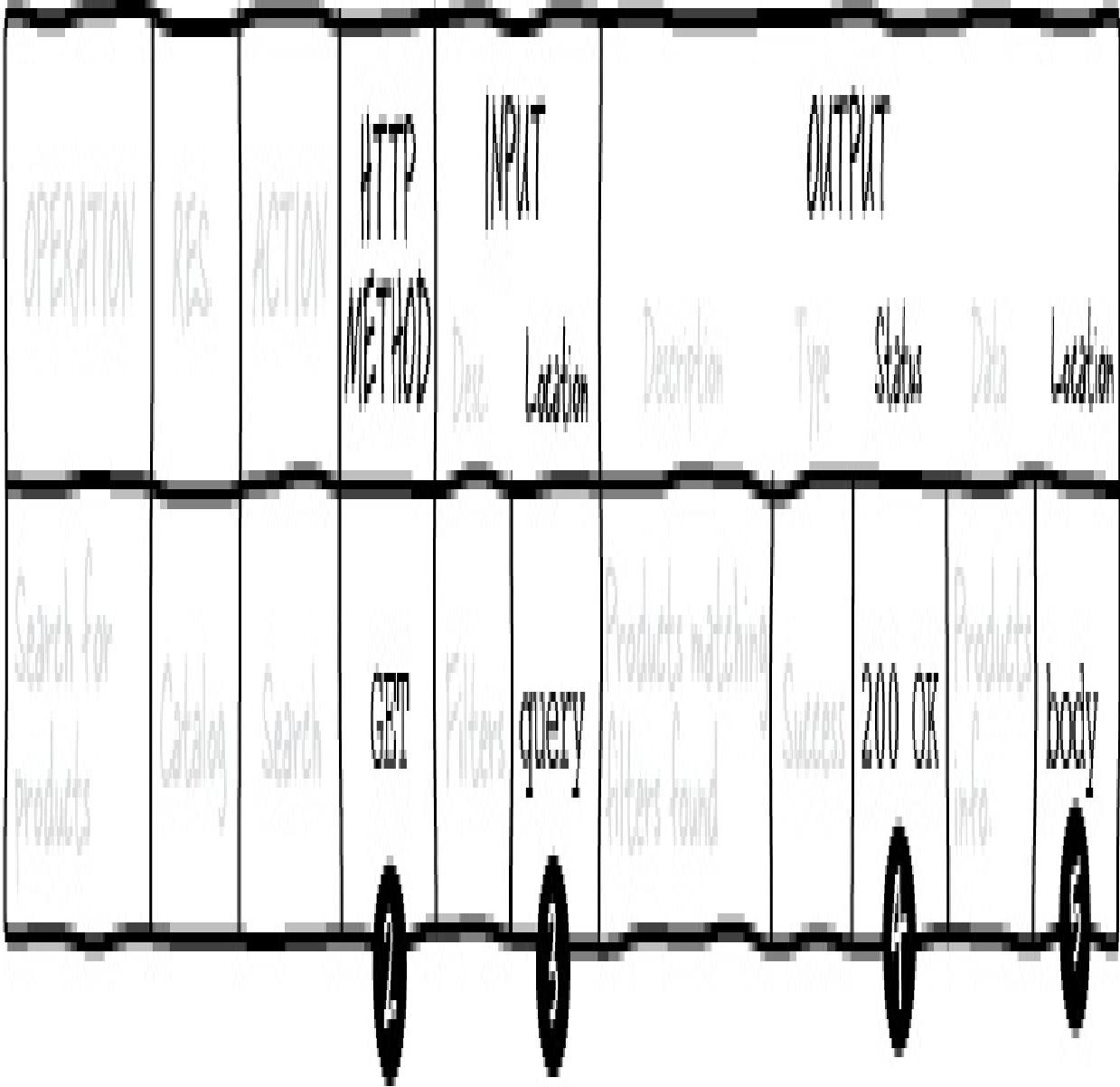


Figure 4.6 The "Search for products" operation and its HTTP counterpart (Operation table in API spreadsheet)



We focus first on HTTP requests. We represent the resource ("Catalog") with a path (/products), select an HTTP method (GET) to represent the action ("Search"), and choose a location for each input data (query for "Filters"). Afterward, we work on HTP responses. We pick HTTP status codes to represent each output (200 OK for "Products matching filters found") and choose the location of each output data (body for "Products info").

Note

We don't investigate the specifics of "Filters" input or "Products info" output;

we keep fine-grained data modeling for the next step (chapter 5). Also, we store our findings temporarily in our API spreadsheet's tables; chapter 6 will show us a better way to handle this.

4.2 Representing resources with paths

As seen in section 3.1.2, REST APIs manipulate resources, which are business concepts like a "Product," and are identified by a path. This section discusses the fundamentals of resource path design and demonstrates how to design paths, leveraging the "Catalog" and "Product" resources identified in section 3.3.2.

4.2.1 Fundamentals of resource path design

We'll discuss various constraints and better practices for resource path design throughout this book. At this stage, a path identifies a unique resource, may have multiple segments and parameters, is hierarchical, doesn't reflect the underlying organization, and doesn't end with a slash.

In the <https://example.com/path> URL, the resource path is /path and identifies a unique resource. As two files can't share the same path on a file system, two resources can't share the same path on an API server. However, different paths may point to the same resource.

A path may contain multiple segments (this, is, a, or path in /this/is/a/path) separated by slashes (/). It may also include variable parts called path parameters represented between {} in API documentation or code. For instance, /segment/{parameter} contains a path parameter named parameter. In an HTTP request, {parameter} is replaced by a value, like /segment/123456.

HTTP documentation states resource paths are *usually* hierarchical; in /p/c, p contains or is a parent of c (and c is an element or child of p). Following this recommendation contributes to creating user-friendly APIs (discussed in chapter 9).

A path doesn't have to reflect underlying data or implementation

organization. For example, `/mangas` may have its data stored in a `BOOK` table (remember the provider's perspective of section 2.8).

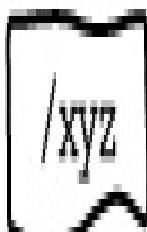
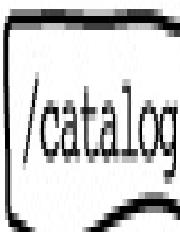
Tip

A resource path with a trailing slash, such as `/path/`, can cause routing bugs at implementation code or network infrastructure levels, leading to hours of debugging; use `'/path'` instead.

4.2.2 Designing a meaningful path

Figure 4.7 shows different paths identifying the "Catalog" resource, valid from a REST perspective, but some choices are better than others.

Figure 4.7 Options for the Catalog resource's path

Resource	Paths			
A collection of products 	Random 	Abbreviation 	Resource name 	Resource content 

Unique but not meaningful

Unique and meaningful

We can randomly choose the first path in our API, /xyz, for example, but it's better to make decisions guided by what we want to convey than choosing a random name that no one will understand (including our future selves).

We usually call a spade a spade; /catalog is a good option as it's unique and meaningful. Abbreviated names are common in programming, but /cat is less significant. A name based on the resource's content is OK, too; a "Catalog" is a list or collection of products so that we can use the /products path.

Note

This book discusses the art of choosing names in part 2, specifically diving into designing user-friendly paths in section 9.3.

Before choosing one of the meaningful options, we discuss the possible paths for the "Product" resource that the "Catalog" contains.

4.2.3 Targeting a specific element with a path parameter

Figure 4.8 shows various paths for the "Product." This section focuses on the first three, aiming to identify a product uniquely.

Figure 4.8 The product resource's paths

Resource	Paths (123456 is a {Product reference} path parameter)	Unique	Meaningful	Hierarchical
A single product Product	/product Resource name	X	✓	n/a
	/123456 Resource identifier	X	X	n/a
	/product-123456 Resource name and identifier	✓	✓	n/a
	/123456/catalog Identifier and parent name	✓	✓	X
	/catalog/123456 Parent name and identifier	✓	✓	✓
	/products/123456 Resource type and identifier	✓	✓	✓
 / {Product reference}		may collide with		/ {Supplier reference}

The /product path is meaningful but can't uniquely identify a product. Fortunately, we know that all operations leveraging this resource have a unique "Product reference" input (see figure 4.3 of section 4.1). The /{Product reference} (/123456) path uses it as a path parameter, allowing product identification.

But it's not meaningful, and a future /{Supplier reference} identifying a "Supplier" may collide with this path (learn more about future-proof design in chapter 14). Concatenating the resource name and the reference can solve this. The /product-{product reference} (/product-123456) path is a

unique and meaningful path saying, "I'm product 123456". However, it doesn't convey the relationship between "Catalog" and "Product" resources.

Tip

An ID, reference, number, or code that identifies the resource and appears as input on all of its operations is a *resource identifier* that must be present in the resource path to make it unique.

4.2.4 Materializing relationship between resources

The last three paths of figure 4.8 materialize that a "Product" is an element of the "Catalog." The `/product reference/catalog` (`/123456/catalog`) path reads as "Product 123456 belongs to the catalog," but its hierarchy is reversed (child comes before parent). The `/catalog/product reference` (`/catalog/123456`) path fixes this, saying, "The catalog contains product 123456." We can also use `/products/product reference` (`/products/123456`), which states, "Product 123456 belongs to the list of products."

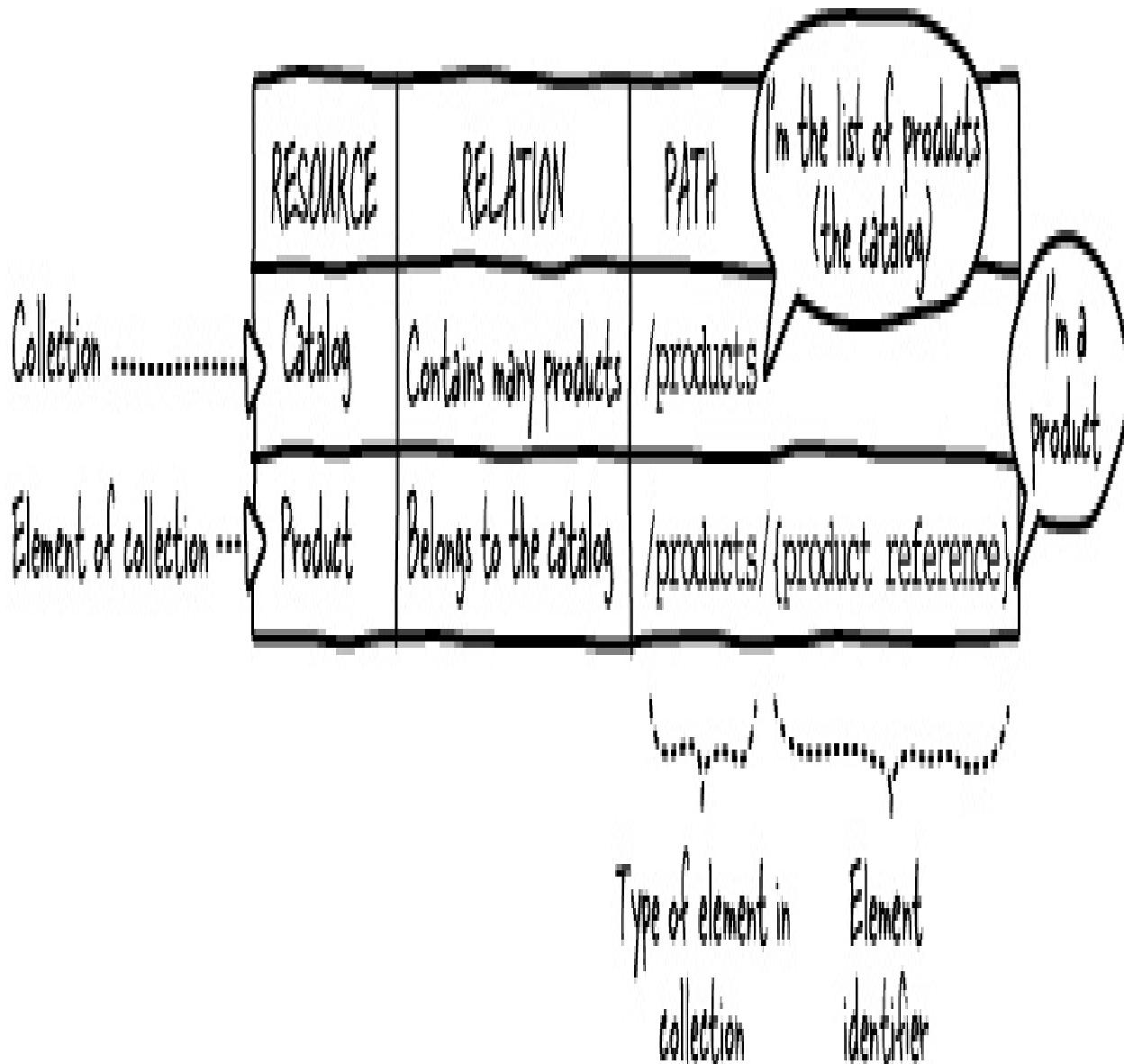
4.2.5 Designing paths for a collection and its elements

All combinations of "Catalog" and "Product" unique paths are valid from a REST perspective. Still, we choose the ones shown in figure 4.9, relying on a design pattern and recipe section 9.3 discusses.

Tip

A collection resource, which is a list of element resources, is often represented by the `/elements` path (plural noun) and its children by `/elements/element resource identifier`.

Figure 4.9 Typical paths representing a collection resource and its element



The "Catalog" is a collection resource containing "Product" resources, so we represent them as /products and /products/{Product reference}. These paths uniquely identify each resource (thanks to the Product reference path parameter for "Product"), describe the relationship between the two (thanks to being hierarchical), and are meaningful (each clearly states what the resource is).

4.3 Representing actions with HTTP methods

As seen in section 3.1.2, REST APIs represent operations with standardized HTTP methods applied on resource paths. This section introduces the HTTP methods usually used, then explains and demonstrates how to choose one for each operation's action identified in section 3.4.1, and ultimately generalizes learnings.

Note

Chapters 9, 12, and 13 discuss other concerns that may impact how we choose HTTP methods. Section 4.8.2 and chapter 9 discuss how standard HTTP methods benefit API usability.

4.3.1 Which HTTP methods use

Though there are more, most APIs leverage only five HTTP methods to represent actions: POST, GET, PUT, PATCH, and DELETE (chapter 9 discusses why). Figure 4.10 summarizes their meaning and usage.

Figure 4.10 How to use the five HTTP methods typically used in REST APIs

HTTP METHOD	MEANING (HTTP)	MEANING (CRUD)	ACTION EXAMPLE
POST	Create	C	Create, add, start, save, send
	Process	non-CRUD	Do, execute
GET	Read	R	Read, get, search, filter, select, retrieve, show, download
PUT	Replace	U	Replace, modify, update, change, edit
	Create	C	Create, add, start, save, send
	Upsert	CU	Create if not present, update otherwise
PATCH	Update (partial)	U	Partially replace, modify, update, change, edit
DELETE	Delete	D	Delete, cancel, close, finish, stop

POST usually represents a creation (C of CRUD), adding an element to the targeted resource. Use it for actions such as "create," "add," "start," "save," or "send." Its real meaning is broader; it means "process (according to resource's signification)" and can be used as a fallback when no other method fits.

GET reads the resource (R of CRUD). Use it for actions like "read," "get," "search," "filter," "select," "retrieve," "show," or "download."

PUT is for complete resource replacement or update (U of CRUD), creation (C

of CRUD), and upsert, which updates an existing or creates a new resource (CU of CRUD). Use it for actions like "modify," "update," "change," "replace," or "edit," or the same actions as POST for creation.

PATCH is similar to PUT as it updates (U of CRUD) a resource, but it can do it partially. Use it for the same action as PUT for updating, with the possibility of being "partial."

DELETE represents a deletion (D of CRUD). Use it for actions such as "delete," "cancel," "close," "finish," or "stop."

Note

Remember that HTTP is "disconnected" from the implementation; what happens when an HTTP request is processed depends on the subject matter and implementation choices. For example, on a DELETE /something request, the implementation may delete a line from a table (hard delete) or update a flag (soft delete).

4.3.2 Choosing HTTP methods to represent actions

We select the HTTP method representing each action best (the summary of figure 4.10 can help us) by working on resources one by one. This approach helps detect conflicts: an HTTP method can only be defined once on each resource. Conflicts are rare and usually indicate wrong resource or operation identification; reevaluate them in such a case. A resource can have any number of HTTP methods and methods with similar intent defined.

Figure 4.11 shows the HTTP methods for the five typical REST API operations of the Online Shopping example (we designed paths in section 4.2). The following sections explain these results from simple to complex cases. In practice, you'll work on a resource-by-resource and action-by-action basis.

Figure 4.11 Operation table of the API spreadsheet with actions mapped to HTTP methods (completed with resource paths)

OPERATION	RESOURCE	ACTION	HTTP METHOD	RESOURCE PATH
Add a product to the catalog	Catalog	Add	POST	/products
Search for products		Search	GET	
Get product details		Get	GET	
Modify a product	Product	Modify	PUT or PATCH	/products/{Product reference}
Remove a product from the catalog		Remove	DELETE	

4.3.3 Representing search, read, and delete actions

The "Catalog" resource has a "Search" action; it's fundamentally a "Read" action we can map to the GET HTTP method. To "Search for products," consumers will send a GET /products HTTP request.

The "Product" resource has a "Get" action, which we can easily map to the GET HTTP method. To "Get product details," consumers will send a GET /products/{Product reference} HTTP request.

The "Product" resource has a "Remove" action of type "Delete," so we choose the `DELETE` HTTP method. To "Remove a product from the catalog," consumers will send a `DELETE /products/{Product reference}` HTTP request.

4.3.4 Representing update actions

The "Modify a product" operation of the "Product" resource has a "Modify" action, which is an update; we can map it to `PUT` or `PATCH` `/products/{Product reference}`. Choose `PUT` at this learning stage; chapters 5, 9, 12, and 13 discuss further considerations. It is possible to have both methods defined. We keep both to demonstrate the five usual HTTP methods.

4.3.5 Representing create actions

The "Catalog" resource has an "Add" action aiming to create a product, which we can represent with `POST` or `PUT`. While `POST /products` creates a product, `PUT /products` replaces the entire catalog. To use `PUT`, we must target the resource we want to create, a "Product," and so do a `PUT /products/{Product reference}`. It leads to merging "Add a product to the catalog" and "Modify a product" operations into "Adding or modifying a product." Also, it requires consumers to be able to provide the product reference upon creation. These consequences may or may not be an issue. Choose `POST` at this learning stage; chapters 5, 9, 12, and 13 discuss further considerations.

4.3.6 Mapping typical operations to HTTP

We've uncovered new patterns and recipes applicable anytime we need to map one of the typical create, search, read, update, and delete operations to HTTP.

Given that `/elements` represents a list of elements and `/elements/{element identifier}` represents an element of that list, the five typical REST API operations can be mapped to HTTP as follows:

- Create an element: POST /elements (default choice at this stage) or /PUT /elements/{element identifier}.
- List or search for elements: GET /elements.
- Read an element: GET /elements/{element identifier}.
- Update an element: PUT (default choice at this stage) or PATCH /elements/{element identifier}.
- Delete an element: DELETE /elements/{element identifier}.

4.4 Choosing input data locations in HTTP requests

Knowing the HTTP method and path for each operation, we can locate their inputs in the HTTP request. This location and the HTTP method impact data modeling (chapter 5). This section discusses locations for data in an HTTP request, choosing one for each input identified in section 3.4.2, and generalizes learnings.

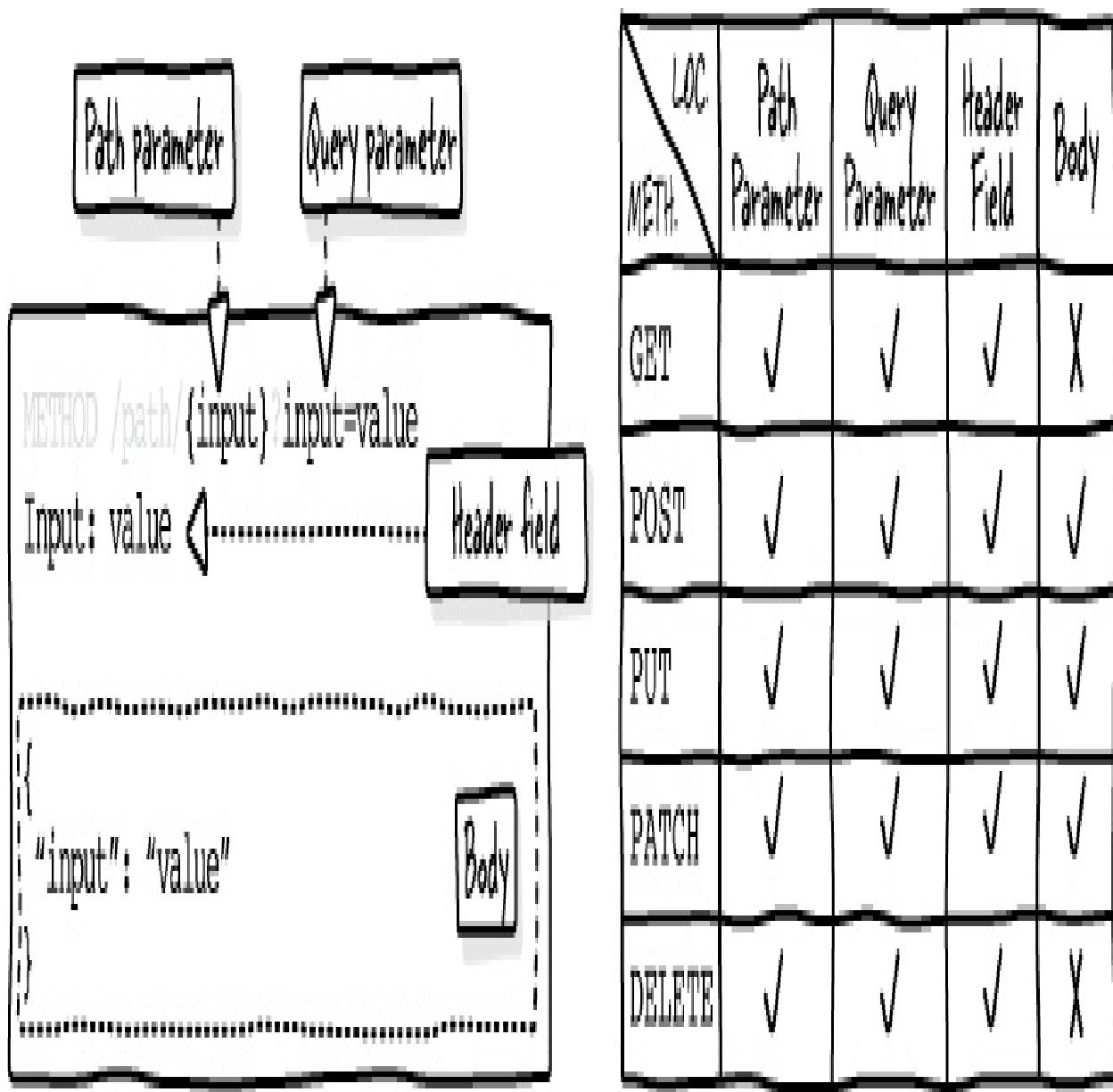
Note

Parts 2 and 3 discuss other considerations when choosing locations and values of inputs in HTTP requests, such as data formats, security, and provider or consumer constraints.

4.4.1 Where to put input data in an HTTP request?

As shown in figure 4.12, an HTTP request has four data locations: path and query parameters, header fields, and the body. Only specific methods can have a body. For more information on data modeling concerns, refer to chapter 5.

Figure 4.12 Input data locations in an HTTP request



As seen in section 4.2.1, path parameters are located in the resource path (on the first line of the HTTP request). Their value can be anything that fits into a (usually short) string. For example, in the path `/resources/{Resource identifier}/sub-resources/{Sub-resource identifier}`, the `{}` indicates two path parameters. In an HTTP request, such a path could be `/resources/12/sub-resources/ab`.

The resource's path can be completed with query parameters containing non-hierarchical data participating in resource identification (discussed in chapter 9). They are added after a `?`, are in a `name=value` form, and are separated by `&`

if there is more than one. We're free to choose their name as we like, and their values can be anything that fits into a (usually short) string. For example, `/resources?a=1&b=no` has two query parameters: a (value 1) and b (value no).

After the first line, HTTP requests include header fields that contain metadata about the request's origin, target, content, and other details. HTTP and its extensions define over 200 standard headers (see *IANA HTTP Field Registry* at <https://www.iana.org/assignments/http-fields/http-fields.xhtml>). Their format is `name: value`. Though defining custom headers is possible, we'll use standard ones for now. Their values can be anything that fits into a (usually short) string. For example, `Content-length: 345` is a standard HTTP header indicating the request body size in bytes.

The request body follows the headers and is used in `POST`, `PUT`, and `PATCH`; we can't use it on `GET` or `DELETE`. It can contain anything, such as text or binary data, like JSON, XML, or an image. The body is a "representation" of the resource to create or update; the implementation may process it and not store it exactly as the consumer sent. The "representation" concept is discussed further in chapter 9.

4.4.2 Choosing input data locations

HTTP method and data nature influence the location of input data. As some resource operations may share parameters, proceeding on a resource-by-resource basis is helpful. Figure 4.13 shows the input data locations for the typical operations of the Online Shopping example. The following sections explaining these results are organized by the data nature; in practice, you'll work on a resource-by-resource basis.

Figure 4.13 The operations table expanded with input data location based on HTTP methods and data nature

OPERATION	RESOURCE	ACTION	HTTP METHOD	Description	INPUT	Location
Add a product to the catalog	Catalog	Add	POST	Product information	body	
Search for products		Search	GET	Filters		query
Get product details		Get	GET	Product reference		path
Modify a product		Modify	PUT or PATCH	Product reference	Modified product information	body
Remove a product from the catalog		Remove	DELETE	Product reference		path

Location is influenced by HTTP method and input data nature

4.4.3 Choosing a location for resource identifiers

The three operations of the "Product" resource (GET, PUT or PATCH, and DELETE /products/{product reference}) share the same "Product reference" input identified as a resource identifier and path parameter in section 4.2.2. However, we investigate it for the sake of our learning.

This input can't be a header as it's not a request's metadata matching any of IANA's 200 standard headers. It can't be in the body, as GET or DELETE can't

have one. If it was only `PUT` (or `PATCH`), we could put it in the body, but it doesn't represent the resource to update; it identifies it. Therefore, it could be a path or query parameter. Both are OK for HTTP; `/products?reference={product reference}` and `/products/{product references}` allow us to identify the "Product" resource uniquely, still, for reasons explained in chapter 9, we put the product reference in a path parameter as we initially did.

4.4.4 Choosing a location for resource representations

To modify a product (`PUT` or `PATCH`), we need "Modified product information." It can't fit into a header, path parameter, or query parameter, as it's structured data unfit for a short string, and it's not request metadata matching any standard header. According to HTTP, it must be in the body, as it represents the new state of the resource. The same applies to "Add a product to the catalog" (`POST`), which has "Product information" input.

4.4.5 Choosing a location for resource modifiers

The "Search for products" (`GET /products`) operation has a single input "Filters." The HTTP method is `GET`, so we can't put it into the request body. It also doesn't fit into a standard HTTP header. That leaves path and query parameter locations. Both options are valid for HTTP. If these filters are "type" and "description," we could have `/products/{type}/{description}` (path parameters) and `/products?type={type}&description={description}` (query parameters).

Still, we chose the query parameter option for "Filters." While the product reference path parameter was a resource identifier, these are *resource modifiers* allowing the selection of a subset of all products. Setting them as path parameters makes them mandatory, which is not convenient; we need to be able to `GET /products` without them and get all products. Chapters 9, and 14 explains further the reasons behind this choice.

4.4.6 Choosing input data locations for typical operations

We've uncovered new patterns and recipes applicable anytime we need to identify input locations of the typical create, search, read, update, and delete

operations to HTTP:

- The data needed to create or update a resource goes into the body.
- The resource identifiers go into path parameters.
- The resource modifiers (like search filters) that do not fit into a standard HTTP header go into query parameters.
- HTTP headers must only be used for standard HTTP request metadata (at this stage).

4.5 Representing output types with HTTP statuses

Having designed resource paths, chosen HTTP methods, and decided on input locations, we can move on to HTTP response, starting with HTTP statuses. This section covers HTTP statuses, how to choose them, selecting statuses for success and error outputs identified in section 3.4.4, and ensuring exhaustive error handling with HTTP. Ultimately, it summarizes our learnings.

Note

Section 4.8.2 and chapter 9 discuss how standard HTTP statuses benefit API usability.

4.5.1 What is an HTTP status?

In an HTTP response, the HTTP status indicates how the processing of the HTTP request went. You might have seen statuses like 404 Not Found while accessing a non-existing web page. A status comprises a three-digit code (404) and a human-readable reason describing it (Not Found), as shown in figure 4.14.

Figure 4.14 The structure of an HTTP status code and the main HTTP status classes

HTTP status	CLASS	SIGNIFICATION	MAIN VALUE
404 Not Found	2XX	Success	200 OK
Code Reason	4XX	Error (client)	400 Bad request
Class 4XX	5XX	Error (server)	500 Internal Server Error

The codes ranging from 100 to 599 are organized into five classes from 1XX to 5XX, having a specific signification. Most APIs leverage the 2XX (success), 4XX(client error), and 5XX(server error) classes.

2XX class codes between 200 and 299 indicate the server has successfully processed the request.

4XX class codes between 400 and 499 indicate client/consumer errors. The server can't process the request due to, for example, unparsable data, unhandled HTTP method, missing mandatory property, business logic

checks, or insufficient rights.

5xx class codes between 500 and 599 indicate server/implementation errors caused by unexpected problems (a bugged implementation throwing a null pointer exception or an inaccessible database server, for example) or planned unavailability.

The class system makes interpreting unknown code easy. You may not know the meaning of the 413 HTTP status code, but you can tell it indicates an error caused by the consumer.

4.5.2 How to choose HTTP statuses for outputs

We can proceed operation by operation to choose the HTTP statuses best representing each output listed in section 3.4.4, leveraging what we discover from one to another. At this stage, selecting an HTTP status code depends on the following:

- The type of output: success (2xx) or error (4xx, 5xx).
- In case of error, who caused it: consumer (4xx) or provider (5xx).
- The HTTP method of the request.

The book showcases commonly used HTTP statuses in REST APIs, covering most cases. HTTP methods documentation may provide recommendations (see *IANA HTTP Method Registry* at <https://www.iana.org/assignments/http-methods/http-methods.xhtml>). For more codes, refer to the *IANA HTTP Status Code Registry* (<https://www.iana.org/assignments/http-status-codes/http-status-codes.xhtml>). Use caution when using codes not referenced in this book, and avoid using unassigned codes to create custom statuses (see Chapter 9 for more details). Use x00 main value of a class when in doubt.

In the following sections, we'll first treat successes (2xx), then errors (4xx or 5xx) to facilitate learning. In reality, proceed operation by operation and output by output.

4.5.3 Choosing successful HTTP statuses for read operations

Figure 4.15 shows the five typical operations of the Online Shopping example and their success outputs with corresponding HTTP statuses of the 2xx (success) class. This section and the following ones explain the results for each typical operation.

Figure 4.15 Operation table focusing on success outputs and completed with HTTP statuses

OPERATION	RES.	ACTION	HTTP METHOD	OUTPUT			
				Description	Type	Data	HTTP Status
Add a product to the catalog	Catalog	Add	POST	Product added to the catalog	Success	Missing data!	201 Created
Search for products	Catalog	Search	GET	Products matching filters found	Success	Products info.	200 OK
				No products matching filters	Success		200 OK
Get product details	Product	Get	GET	Product found	Success	Product info.	200 OK
Modify a product	Product	Modify	PUT OR PATCH	Product modified	Success	Missing data!	200 OK
Remove a product from the catalog	Product	Remove	DELETE	Product removed	Success		204 No Content

Consumers can "Get product details" by sending a GET /products/{Product reference} HTTP request. We could look at each 2xx code documentation to find the best one to represent this success. Still, it is faster to check the GET

HTTP method, which says a `GET` request usually returns a `200 OK` response (chapter 9 discusses other options).

4.5.4 Choosing successful HTTP statuses for delete operations

To "Remove a product from the catalog," consumers use the `DELETE` HTTP method, whose documentation gives three options: `200 OK`, `202 Accepted`, and `204 No Content`. We can use `200 OK` if the action has been executed and the response contains data describing its status. We can use `202 Accepted` if the action will likely succeed but has not yet been executed. `204 No Content` is similar to `200 OK`, but the response contains no data. For our case, where deletion is instantaneous and doesn't return data, `204 No Content` is the best option. We'll get back to `200 OK` and `202 Accepted` in chapters 9 and 13, respectively.

4.5.5 Choosing successful HTTP statuses for update operations

The "Modify a product" operation uses `PUT` or `PATCH` methods. Navigating in documentation, we finally found the HTTP status options are the same as for `DELETE`. As the update is instantaneous, we have to choose between `200 OK` and `204 No Content`. The output description, which says "Product modified," isn't clear about returning data (`200`) or not (`204`). Both options are OK; we choose the most usual one, `200` (chapters 9 and 13 discuss the pros and cons of these options). That means we need to add output data; see section 4.6.2.

4.5.6 Choosing successful HTTP statuses for search operations

Based on what we've seen before with `GET` and `DELETE`, "Search for products" (`GET /products`) can return a `200 OK` when "Product matching filters found" and `204 No Content` when "No products matching filters ." We may also use `200 OK` for both; we choose this option for reasons chapter 9 will explain.

4.5.7 Choosing successful HTTP statuses for create operations

Following `POST` HTTP method documentation, "Add a product to the catalog" (`POST /products`) returns a `201 Created` as we create a resource. Same if

using `PUT /products/{product reference}`, which allows the consumer to differentiate between an update (`200 OK`) and a creation (`201 Created`).

The documentation reveals a missing piece: a creation should return the created resource's information, or at least minimal data to retrieve it later. Section 4.6.2 shows how to detect and fill such a gap.

4.5.8 Choosing error HTTP statuses

Section 4.5.1 taught us error HTTP statuses fall into two classes: `4xx` (client) and `5xx` (server); determining who is responsible for the error allows us to identify the correct one. As shown in figure 4.16, we identified two errors across all operations of the Online Shopping example: "No product found" and "Wrong product information."

Figure 4.16 Operation table focusing on error outputs and completed with HTTP statuses

OPERATION	RES.	ACTION	HTTP METHOD	INPUT		OUTPUT		
				Desc.	Location	Description	Type	HTTP Status
Add a product to the catalog	Catalog	Add	POST	Prod. info	body	Wrong product information	Error	400 Bad Request
Search for products	Catalog	Search	GET	Filters	query			
Get product details	Product	Get	GET	Prod. ref.	path	No product found	Error	404 Not Found
Modify a product	Product	Modify	PUT or PATCH	Prod. ref.	path	No product found	Error	404 Not Found
				Mod. prod. info.	body			
Remove a product from the catalog	Product	Remove	DELETE	Prod. ref.	path	No product found	Error	404 Not Found

The "No product found" error is caused by a wrong product reference provided by a consumer, resulting in a 4xx HTTP status class. Based on our browsing experience and documentation, 404 Not Found is the relevant code to indicate that the resource/path doesn't exist.

The "Wrong product information" error occurs when a consumer provides incorrect or incomplete data while adding a product, resulting in a 4xx HTTP status class. Scanning the IANA list, we can find several options we'll discuss in section 9.6. At this stage of our learning, we take for granted we can use the most usual one, 400 Bad Request.

4.5.9 Ensuring exhaustive error handling

Errors found during needs analysis may not be exhaustive; this is normal as the focus is on business needs. Choosing HTTP statuses is the perfect moment to identify and fill the gaps. Figure 4.17 shows what we can detect at this stage of our learning; we'll learn to discover more gaps throughout the book.

Figure 4.17 Errors missed during needs analysis and their HTTP statuses

OPERATION	RES.	ACTION	HTTP METHOD	INPUT		OUTPUT		
				Desc.	Location	Description	Type	HTTP Status
Add a product to the catalog	Catalog	Add	POST	Prod. info	body	Wrong product information	Error	400 Bad Request
Search for products	Catalog	Search	GET	Filters	query	Wrong filters	Error	400 Bad Request
Get product details	Product	Get	GET	Prod. ref.	path	No product found	Error	404 Not Found
Modify a product	Product	Modify	PUT or PATCH	Prod. ref.	path	No product found	Error	404 Not Found
				Mod. prod. info.	body	Wrong product information	Error	400 Bad Request
Remove a product from the catalog	Product	Remove	DELETE	Prod. ref.	path	No product found	Error	404 Not Found
All operations						Unexpected server error	Error	500 Internal Server Error

Some operations expecting inputs are missing errors related to improper and missing inputs, and so `400 Bad Request`. That is the case for "Search for products" and its "Filters" and "Modify a product" and its "Modified product information." We can check that any operation with path parameters has a "Resource not found" error and `404 Not Found`; no missing errors here.

Ultimately, even though the implementation is thoroughly tested and the infrastructure is reliable, an improbable but possible server failure must be described. All operations must have an "Unexpected server error" output and return a `500 Internal Server Error`.

4.5.10 Choosing HTTP statuses for typical operations

We've uncovered new patterns and recipes applicable anytime we choose the HTTP status codes representing the outputs of the typical create, search, read, update, and delete operations and helping us detect gaps.

Choosing HTTP statuses for successful outputs:

- A successful creation returns a `201 Created`.
- A successful read returns a `200 OK`.
- A successful search returns a `200 OK`.
- A successful update returns a `200 OK` when the updated resource is returned and `204 No Content` when it's not.
- A successful delete returns a `204 No Content` if no status data is returned and `200 OK` if there is.

Choosing HTTP statuses for error outputs and spotting missing errors:

- An operation expecting input query or body data must handle missing or invalid data errors and return a `400 Bad Request`.
- An operation whose resource path contains one or more path parameters must handle resource not found errors and return a `404 Not Found`.
- Each operation must handle unexpected server errors and return a `500 Internal Server Error`.

Note

Chapters 9 and 13 discuss other options, pros and cons.

4.6 Choosing outputs locations in HTTP responses

Our last task is determining locations for output data in the HTTP response as it impacts data modeling (chapter 5). This section covers data locations in HTTP responses, filling output data gaps with HTTP, and choosing locations for data identified in section section 3.4.4, and generalizes learnings.

4.6.1 Where to put data in an HTTP response?

As shown in figure 4.18, an HTTP response has a similar structure to an HTTP request, with the first line containing the HTTP status instead of a method and path. Headers follow, containing metadata about the response, followed by an empty line and the body. As for the HTTP request, headers contain anything that fits into a (usually) short string, and we use only standard ones (see IANA Header Field Registry at <https://www.iana.org/assignments/http-fields/http-fields.xhtml>), and the body can contain any data.

Figure 4.18 Output data locations in an HTTP response

CODE REASON

Output: value <-----> Header field

```
{  
  "output": "value"  
}
```

Body

4.6.2 Filling the output data gaps

As seen in section 4.4, we may have missed some output elements when listing them in section 3.4.4 and HTTP can help us fill the gaps (see figure 4.19 for the final output list).

The "Add a product to the catalog" operation uses POST, whose documentation recommends returning the representation/data of the created resource or minimal information (the product reference) to retrieve it later

(with "Get product details"). It also uses `201 Created`, which requires the response to contain the URL of the created resource in a `Location` header, which allows for later retrieval with a `GET {created resource URL}`, even when no data is returned (see chapter 10 for more details).

We chose to return `200 OK` (with data) instead of `204 No Content` (without data) on the "Modify a product" operation, so we should add the updated product data as output. Similarly, for the "No product matching filters" `200 OK` output of "Search for products," we must return data like "Empty products list" (see chapter 9 for more details).

HTTP status codes can hint at what is happening, but they often fall short in case of error; thus, HTTP documentation recommends returning "Error information" data on all errors (see section 9.6 for more details).

4.6.3 Choosing outputs locations

Once we have an exhaustive output data list, we choose their locations (header or body) similarly as we did for input data in section 4.4. Figure 4.19 shows the final HTTP representation of the typical operations of the Online Shopping example, including output data location.

Figure 4.19 Finalized operation table completed with all output data and their locations

OPERATION	RES.	ACTION	HTTP METHOD	INPUT		Description	OUTPUT			
				Desc.	Location		Type	Status*	Data	Location
Add a product to the catalog	Catalog	Add	POST	Prod. info	body	Product added to the catalog	Success	201	Product info.	body
						Wrong product information	Error	400	Product URL	header
Search for products	Catalog	Search	GET	Filters	query	Products matching filters found	Success	200	Products info.	body
						No products matching filters	Success	200	Empty products info.	body
						Wrong filters	Error	400	Error info.	body
Get product details	Product	Get	GET	Prod. ref.	path	Product found	Success	200	Product info.	body
						No product found	Error	404	Error info.	body
Modify a product	Product	Modify	PUT or PATCH	Prod. ref.	path	Product modified	Success	200	Product info.	body
						No product found	Error	404	Error info.	body
				Mod. prod. info.	body	Wrong product information	Error	400	Error info.	body
Remove a product from the catalog	Product	Remove	DELETE	Prod. ref.	path	Product removed	Success	204		
						No product found	Error	404	Error info.	body
All operations						Unexpected server error	Error	500	Error info.	body

Status* 200 OK, 201 Created, 204 No Content, 400 Bad Request,
404 Not Found, 500 Internal Server Error

The "Product information" returned upon creating, reading, or updating a product, as well as the "Products information" or "Empty products list," is structured data unfit for a header. More importantly, they represent the resource the operation manipulates, so, as HTTP documentation says, they go into the response body. Similarly, "Error information" goes in the body as HTTP recommends.

Strictly following the 201 Created status documentation, we return the "Product URL" of the "Add a product to the catalog" operation as a standard Location header.

4.6.4 Choosing output data locations for typical operations

We've uncovered new patterns and recipes applicable anytime we need to identify output locations of the typical create, search, read, update, and delete operations to HTTP:

- A "read" operation (`GET`) returns the requested resource in the body.
- A "search" operation (`GET`) returns found elements or an empty list in the body.
- A "create" operation (`POST` or `PUT`) returns the created resource in the body and its URL in a standard `Location` header.
- An "update" operation (`PUT` or `PATCH`) returns the modified resource in the body.
- A "delete" operation (`DELETE`) returns nothing
- All errors return error data in the body.
- HTTP headers are only for standard HTTP response metadata (at this stage).

Note

Chapter 9 discusses when returning data on a "delete" makes sense. Chapter 13 discusses when returning created or modified resources should be avoided.

4.7 Representing a "Do" operation with HTTP

Not all operations fit the typical create, search, read, update, and delete operations. For instance, mapping the "Check out" step of the "Buy products" use case (see section 2.3.1) to an HTTP operation is not straightforward. The challenge is figuring out the resource and the corresponding HTTP method.

Figure 4.20 Three HTTP representations for a "Do" operation, such as "Check out"

OPERATION	RESOURCE	ACTION	HTTP METHOD AND PATH
Check out cart	Cart's check out	Action resource Check out	POST /cart/check-out
Create a checkout (from cart)	Checkouts	Action to business concept Create	POST /checkouts
Create an order (from cart)	Orders	Action's result as business concept Create	POST /cart/orders

As shown in figure 4.20, this section uncovers three ways to handle such a "Do" or non-CRUD operation: using an action resource, turning the action into a business concept, and focusing on the action's result.

4.7.1 Using an action resource

The POST HTTP method means "Process according to resource's signification" (see section 4.3.1), so a "Do" operation can be represented by POST /do, where /do is the path of an action resource. The request body may

contain data needed to execute the action, and the response body, the resulting data, possibly along with the input data. When it makes sense, the path should show a relationship between the action resource and a business matter resource, similar to how a method is related to a class.

We can represent the "Check out" operation with a `POST /cart/check-out` (or `POST /carts/{Cart identifier}/check-out` if multiple carts exist). The path materializes the relationship between the cart and "Check out" resources. It takes no input and returns a `201 Created` with the created order information (body) and URL (Location header, `/orders/123456`, for example).

Creating something is not mandatory; we can use action resources for volatile processing, for example, summing two numbers with a `POST /sum` expecting two numbers in its body and returning the `200 OK` with the result.

Note

Action resources are often wrongly considered non-REST, but according to REST, a resource can be anything, and `POST` handles such cases.

Though an action resource is an acceptable REST API pattern I keep in my toolbox, I mostly use noun-based business concept resources in my API designs as they may offer more possibilities, as shown in the following sections.

4.7.2 Turning the action into a business concept

If it makes sense from a subject matter perspective, we can turn an action resource into a business concept by nominalizing its verb. Additionally, it allows for adding more features than just "Do."

We can consider "Checkouts" ("check out" nominalization) as an essential business concept independent from the cart (though it leverages it under the hood). `POST /checkouts` takes no data and returns checkout-related data (including an order reference) along with the `/checkouts/{Checkout ID}` URL to retrieve it later ("Get checkout details" operation). Also, we can "Search for checkouts (with filters)" with `GET /checkouts`.

Fundamentally, we could have similar operations represented with `POST` and `GET /cart/check-out`, and `GET /cart/check-out/{Checkout ID}`, but that is unusual and inconsistent with what we did with the Catalog and Product resources, for example (see part 2 for more details about consistency).

When the nominalization is not as simple as "check out" to "checkout," you can leverage suffixes such as -ing, -ance, -ence, -ment, -tion, or -sion. For instance, "do" will become "doings," and execute" will become "executions."

Note

Turning an action into a business concept is perfect for long processes or operations that chapter 13 covers.

4.7.3 Focusing on the result

If the action is not interesting from a subject matter perspective, we can work directly with its result and create a resource based on it.

We can decide the resulting "Order" is the crucial business concept and so represent the "Check out" operation with `POST /cart/orders` ("Create an order from the cart"). It would return the created order (body) along with its URL (Location header, `/orders/{Order reference}`).

The difference between `/cart/orders` and `/orders/{Order reference}` paths comes from the fact that we could create orders from the cart's content (`POST /cart/orders`) and an ad-hoc list of products (`POST /orders`). Ultimately, no matter how an order was created, it is an "Order" resource instance, so `/orders/{Order reference}` (see chapter 10 for more details).

4.8 Leveraging the REST architectural style principles for API design

To simplify our learning, we've considered REST APIs as mapping capabilities to HTTP, but it's a common oversight to reduce them to just that and miss essential principles. REST APIs are based on the REST

architectural style, providing a foundation for efficient, scalable, and reliable remote API-based systems. This section discusses the REST architecture style, its principles, and how they relate to API design.

4.8.1 Introducing the REST architectural style

Designing a web API involves working on a distributed system composed of software communicating over a network. A mobile application and its backend, microservices working together, or the Internet are distributed systems.

The REST architectural style enables building distributed systems that are efficient (fast network communication and request processing), scalable (capable of handling more and more requests), reliable (resistant to failure), simple, portable (reusable), and modifiable. Roy Fielding developed it in his 2000 dissertation, "Architectural Styles and the Design of Network-based Software Architectures," while working on HTTP 1.1. A REST software architecture needs to conform to the six following constraints:

- *Client/server separation* - Mobile apps (client) and API servers must have separate and balanced responsibilities.
- *Statelessness* - Requests contain all necessary information; no client context (session) is stored between them.
- *Cache* - Responses to requests specify if they can be reused and for how long (to avoid repeating the same call)
- *Layered system* - Clients only see and interact with servers, unaware of the underlying infrastructure.
- *Uniform interface* - Interactions are performed via the manipulation of resources through representations of their state/data with standard methods (it's the origin of the REST acronym: Representational State Transfer) and the help of metadata, enabling representation interpretation and knowing resource capabilities.
- *Code on demand (Optional)* - A server can transfer executable code to the client (JavaScript, for example).

[More about the REST architectural style](#)

Fielding's dissertation, "Architectural Styles and the Design of Network-based Software Architectures," is available at <https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm>; REST is defined in chapter 5. It gives no guidance on API and API design

The world has evolved, and REST has been used, misused, and abused since 2000. "Reflections on the REST Architectural Style and 'Principled Design of the Modern Web Architecture,'" by Fielding et al., <https://research.google.com/pubs/pub46310.html>, describes the history, evolution, and shortcomings of REST as well as several architectural styles derived from it.

4.8.2 Applying REST principles to API design

This book already leveraged or uncovered some REST constraints and will continue to do so. Note that you can leverage these principles for other types of remote APIs.

Section 2.6 discusses the provider and consumer perspectives discussed, which are related to the *client/server separation* constraints. Part 2 and chapter 14 dive deeper into this topic.

The *statelessness constraint* is hidden behind the context-agnostic operation of section 2.5.2. Chapter 10 dives deeper into stateless API call flows.

Chapter 12 that teaches how to create efficient APIs uncovers the *cache constraint*.

Section 41.3 mentions that consumers are only aware of the API, which is the first layer of the system; it is an example of the *layered system constraint*.

In this chapter, we represented API capabilities with resources and HTTP methods, following the *uniform interface constraint*. Its importance and benefits will be better understood in parts 2 and 3.

Code-on-demand constraint is mainly used in HTML and JavaScript apps, not often in REST APIs; chapter 10 leverages its spirit for API call flows.

What is or is not REST (often sterile) debates

Many REST or RESTful APIs do not follow REST principles and barely leverage HTTP correctly. The "What is or is not REST/RESTful?" question has sparked many (heated and sterile) debates, often unrelated to REST or due to REST and HTTP misunderstanding.

Some people argue that ``POST /do` is not RESTful because /do is not a resource;'' /do can be a resource, and this usage is valid according to HTTP. Similarly, others declare, "A collection must have a plural (or singular noun) in a REST API;" naming conventions do not determine whether an API is RESTful.

This book has your back to help you understand the principles, apply them seamlessly, and know when not to apply them. For arguments, chapter 15 teaches how to avoid them.

4.9 Summary

- A resource path must uniquely identify each resource (using path parameters when needed, /resources/{identifier}), clearly state the resource, and indicate any parent-child relations (/parent/child).
- A collection (list) resource path indicates the element it contains (/elements, for example), and its children's resources concatenate their unique identifier and parent's path (/elements/{identifier}).
- Map the five typical REST API operations to HTTP as follows: create element: POST /elements or /PUT /elements/{element reference}, search for elements: GET /elements, read an element: GET /elements/{element identifier}, update an element: PUT or PATCH /elements/{element identifier}, and delete an element: DELETE /elements/{element identifier}.
- The data needed to create or update a resource goes into the body.
- The resource identifiers always go into path parameters.
- The resource modifiers (like search filters) that do not fit into a standard HTTP header go into query parameters.
- Only standard HTTP headers defined in the IANA registry should be used (at this stage).

- The success of an operation is represented by a 2XX class HTTP status code, an error caused by the consumers, a 4XX, and an error caused by the provider, a 5XX.
- A successful creation returns 201 Created, and other operations may return 200 OK if data is returned (search, read, update), and 204 No Content otherwise (delete).
- Operations expecting input query or body data must return 400 Bad Request, operations with path parameter(s) must return 404 Not Found, and all operations must handle unexpected server errors and return 500 Internal Server Error.
- Output data go into the response body unless it fits into a standard header defined in the IANA registry.
- A "Do" non-CRUD operation can be mapped to POST /do (action resource), /doings (nominalization of action), or /results (resource based on result). The two latter are to be preferred.
- An API adhering to the REST architecture style is efficient, scalable, reliable, simple, portable, and modifiable. It must respect client/server separation, statelessness, cache, uniform interface, and optional code-on-demand constraints to adhere to.

5 Modeling data

This chapter covers

- Designing resource data models
- Designing operations inputs and outputs data based on resource models
- Leveraging fine-grained data models to ensure completeness and proper focus of the API

Once we've given an HTTP representation to operations and identified locations of coarse-grained inputs and outputs in HTTP requests and responses, we can design their fine-grained data models.

Data modeling involves selecting appropriate data, names, types, and organization. At this stage of our learning, we aim to efficiently model data that meets identified needs. Though a good first draft, we must polish our design afterward; user-friendliness, performance, security, and implementation constraints are essential factors to consider in data modeling. Parts 2, and 3 cover them in depth.

The chapter overviews data modeling, clarifying which data we model and introducing the JSON portable data format. Using the Online Shopping example of previous chapters it demonstrates how to model resources and derive them into inputs and outputs based on their locations and types of operation. It also shows how to leverage fine-grained data to ensure completeness and proper focus of the API.

5.1 Overviewing data modeling

As shown in figure 5.1, we're still in the second stage of the API design process outlined in section 1.6.1, "Design the programming interface." We enter the last step of this stage, "Model data," which is preceded by "Represent operations with HTTP" (chapter 4) and "Observe operations from a REST angle" (chapter 3). Figures 5.2 and 5.3 show the information

gathered during the prior steps for the "Online Shopping" example. This section clarifies which data we model and introduces the JSON data format before outlining how to finalize the API design with data modeling.

Figure 5.1 We are here in the API lifecycle and design process

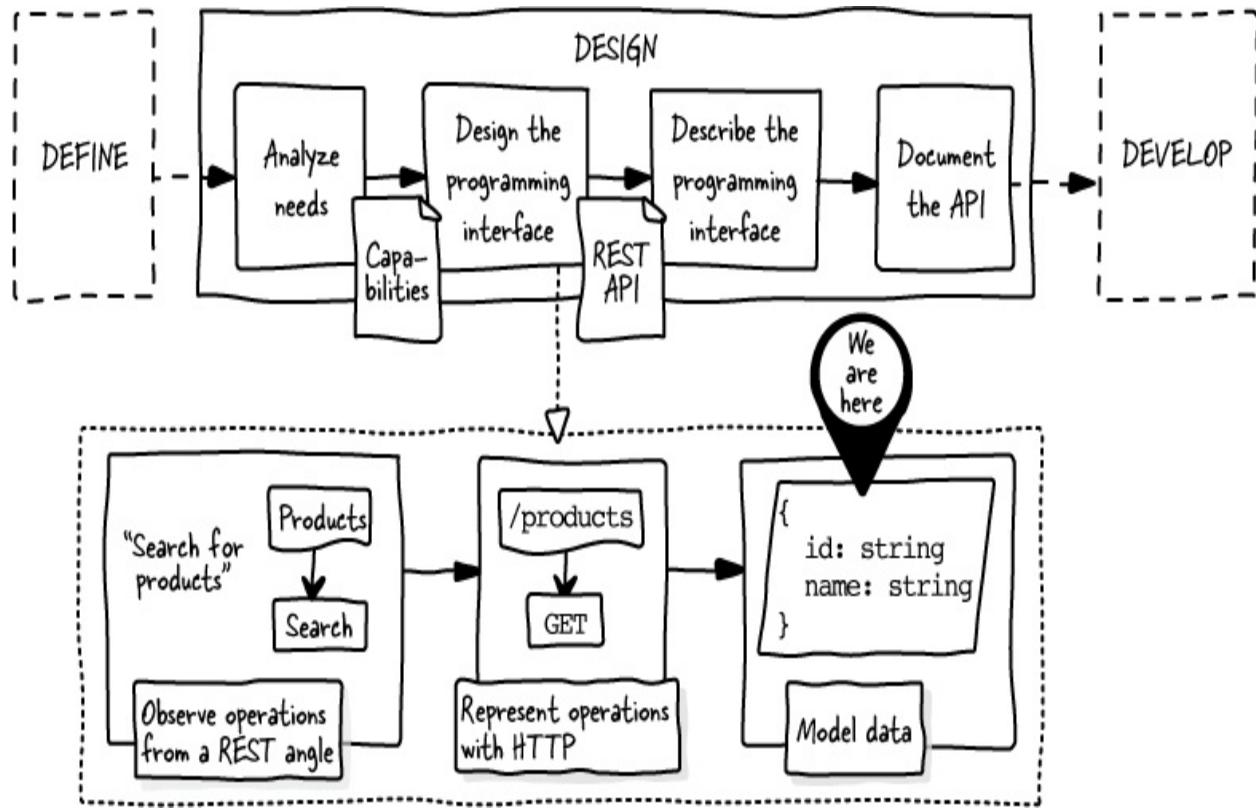


Figure 5.2 The resources of the Online Shopping example and their paths

RESOURCE	RELATION	PATH
Catalog	Contains many products	/products
Product	Belongs to the catalog	/products/{product reference}

Figure 5.3 The Online Shopping example's operations and their HTTP methods, inputs and outputs location, and HTTP status codes

OPERATION	RES.	ACTION	HTTP METHOD	INPUT		OUTPUT				
				Desc.	Location	Description	Type	Status*	Data	Location
Add a product to the catalog	Catalog	Add	POST	Prod. info	body	Product added to the catalog	Success	201	Product info.	body
						Wrong product information	Error	400	Product URL	header
						Wrong product information	Error	400	Error info.	body
Search for products	Catalog	Search	GET	Filters	query	Products matching filters found	Success	200	Products info.	body
						No products matching filters	Success	200	Empty products info.	body
						Wrong filters	Error	400	Error info.	body
Get product details	Product	Get	GET	Prod. ref.	path	Product found	Success	200	Product info.	body
						No product found	Error	404	Error info.	body
Modify a product	Product	Modify	PUT or PATCH	Prod. ref.	path	Product modified	Success	200	Product info.	body
						No product found	Error	404	Error info.	body
				Mod. prod. info.	body	Wrong product information	Error	400	Error info.	body
Remove a product from the catalog	Product	Remove	DELETE	Prod. ref.	path	Product removed	Success	204		
						No product found	Error	404	Error info.	body
All operations						Unexpected server error	Error	500	Error info.	body

Status* 200 OK, 201 Created, 204 No Content, 400 Bad Request,
404 Not Found, 500 Internal Server Error

5.1.1 Which data are we modeling?

When representing operations with HTTP, we discovered the possible data locations in requests and responses (see sections 4.4.1 and 4.6.1). They are the path parameters, query parameters, header fields, and bodies shown in

figure 5.4. The data models we design in this chapter describe the data we put in those locations; figure 5.5 show examples.

Figure 5.4 HTTP request and response data locations

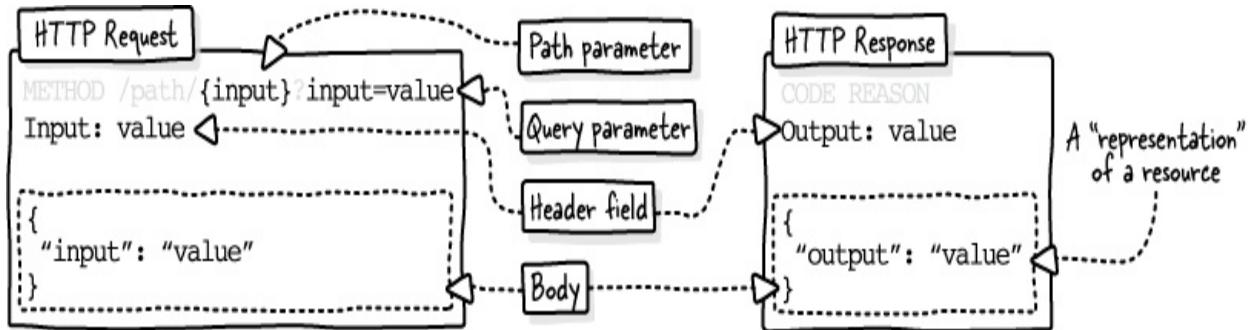
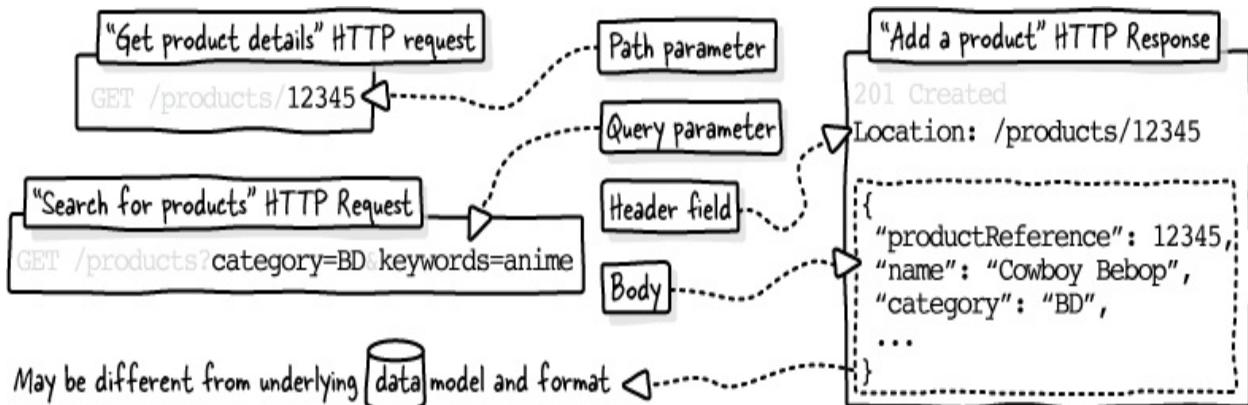


Figure 5.5 HTTP request and response data examples



Path parameters appear in HTTP request paths and are resource identifiers pinpointing a unique resource; an example is the 12345 product reference necessary to get a product's details.

Query parameters are resource modifiers that may appear at the end of the path after a ? and separated by & in the name=value form. The category and keywords product search filters are examples.

HTTP header fields contain metadata about requests and responses. We use only standard ones from the "HTTP header fields" IANA registry. For instance, the `Location` header indicates the created product URL when a product is added to the catalog.

HTTP request and response bodies contain a representation of a resource's desired or current state. For example, when adding a product to the catalog, the request body contains a representation of a product to create, and the response body has a representation of the created product.

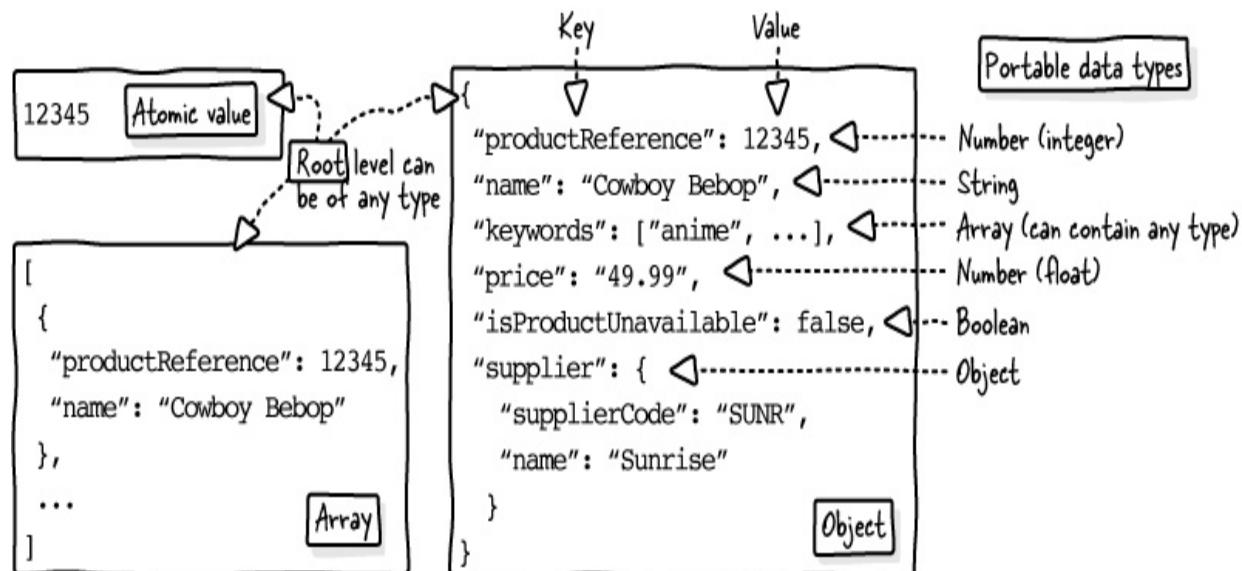
Caution

When designing web APIs, we must focus on modeling the data exchanged between the consumers and the provider from the subject matter perspective rather than from the data storage one (see section 2.8.1).

5.1.2 Introducing the JSON portable data format

HTTP allows for the use of any data format in request and response bodies. JSON (JavaScript Object Notation, <https://www.json.org/>) is the most common format in REST APIs. Though based on JavaScript, it is programming language-independent. JSON is widely adopted for various uses like database storage, configuration files, or web APIs.

Figure 5.6 JSON portable data format examples



As shown in figure 5.6, JSON can describe atomic values (strings, numbers, or booleans), arrays or lists containing ordered values, and objects containing

unordered key/value pairs. Brackets ([]) delimit an array, and commas separate its values (,). Curly braces ({}) delimit an object, and commas separate its properties. An object's property key or name is a quoted string ("price") and is separated from its value by a colon (:). A value can be of any type: a string like "Cowboy Bebop", a number like 12345 or 49.99, a boolean (true or false), an object, or an array. It can also be the null value to indicate it's not set.

To ensure maximum compatibility between the provider and consumers, we model all data going into bodies and all other locations with JSON in mind. Path parameters, query parameters, or header data are usually based on atomic values.

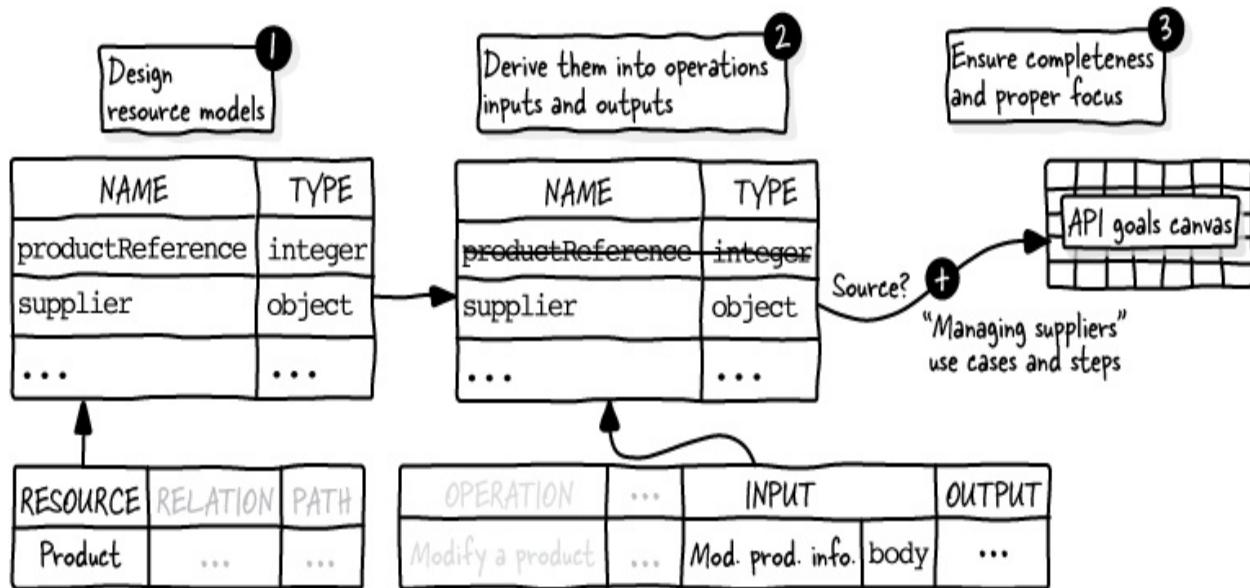
Note

JSON is a common data format in web APIs, but we can use others, such as XML or CSV. Chapter 9 shows what they look like and how to handle different data formats with the same operation.

5.1.3 Finalizing the API design with data modeling

As shown in figure 5.7, we proceed in three phases to model data and achieve the design of an API.

Figure 5.7 Modeling data and ensuring completeness and proper focus



We design the theoretical or complete resource models containing all possible data a business concept can have. We choose names and data types and decide how to structure data. The "Product" resource is an object with properties such as `productReference` (an integer) and `supplier` (an object).

Then, we derive the resource models into inputs and outputs of each operation using them, picking all or a subset of their elements. When adding a product, the body shouldn't contain the `productReference` generated by the server.

Note

We temporarily record data models in spreadsheet tables; chapter 7 shows a more efficient way to describe them. Our data models could be better; it's essential to consider factors such as user-friendliness, performance, security, and implementation constraints, which parts 2 and 3 cover in depth.

Afterward, we leverage the operation inputs and outputs data models to ensure our API design is complete and has the proper focus by checking each property source and usage (as we did during the needs analysis in sections 2.3.3, 2.6, and 2.7). For instance, if adding a product to the catalog requires supplier information, we may need to add supplier-related use cases and steps to our API Capabilities Canvas.

5.2 Designing theoretical resource data models

We first design theoretical resource data models containing all possible data of the business concept they represent (see section 3.3.1) before actual inputs or successful outputs for all operations.

As data is often connected between requests and responses, locations, and operations, designing an API's resource models first simplifies and accelerates the design, reduces errors, and fosters consistency, leading to easy-to-use-and-maintain APIs. For example, the data needed to create a product or the data returned when reading a product are similar.

Leveraging the Online Shopping example resources, this section discusses determining a resource's structure, choosing properties, their name and types, and if they are required, and streamlining this process.

5.2.1 Determining a resource's structure

All elements we design must be of any portable introduced in section 5.1.2, including resources. Chapters 9 and 14 will show us that all resources should be an object. Still, for simplicity at this stage of our learning, we will consider a collection resource as an array and an individual resource as an object.

Figure 5.8 Resources and their models

RESOURCE	RELATION	PATH	RESOURCE MODEL
Catalog	Contains many products	/products	array of Product
Product	Belongs to the catalog	/products/{product reference}	Product (object)

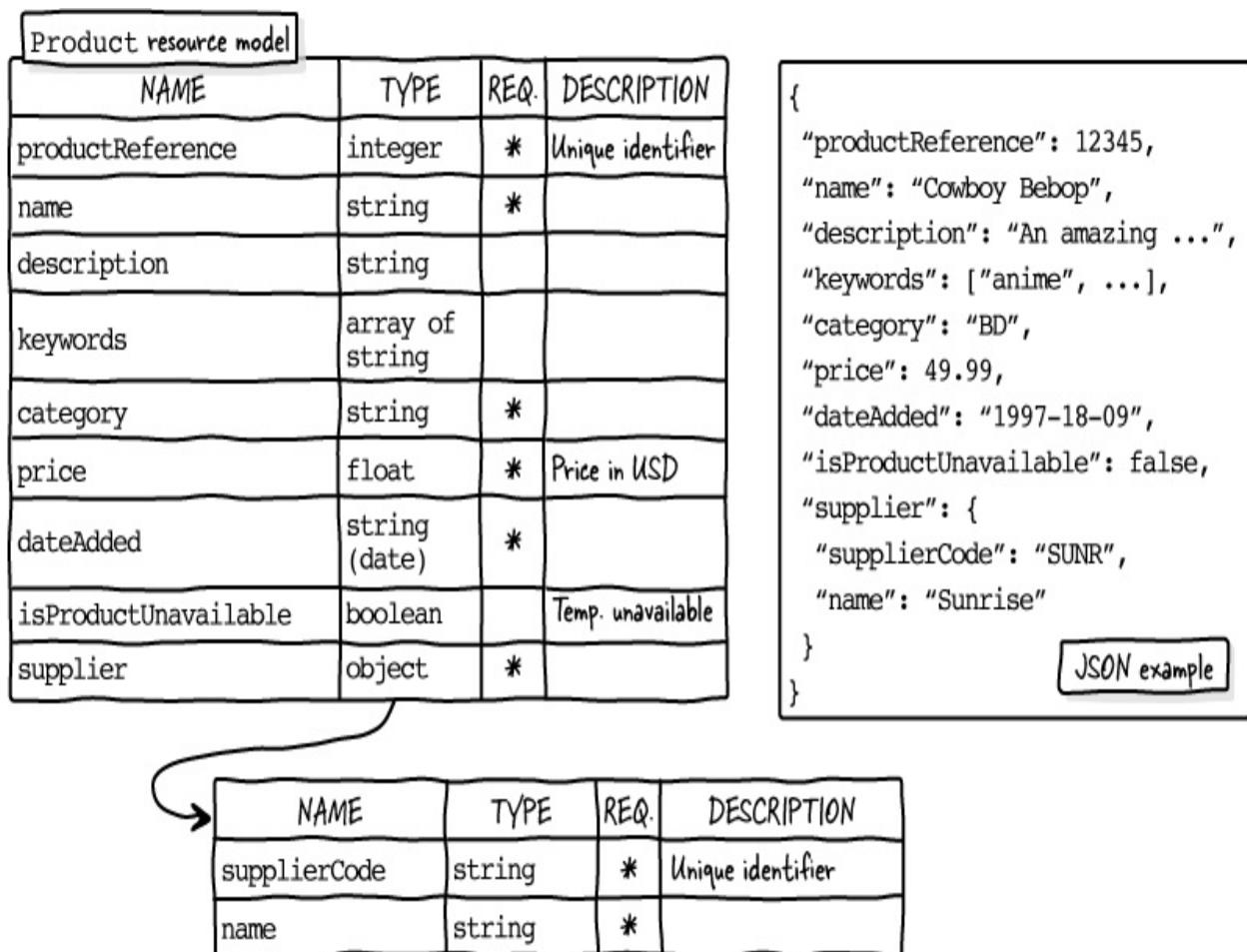
As shown in 5.8, "Product" is an element of the "Catalog", so a Product of type object, while "Catalog" is a collection or list of "Product", so an array of Product.

5.2.2 Choosing an object resource's properties

To design an object data model like Product, we identify what information is necessary to fully represent the business concept, hence its properties, guided by our knowledge or that of our SMEs, supplemented with information from existing applications, implementation code, or databases. We must ensure the result fulfills the needs, makes sense from the consumers' perspective, and isn't tainted by unwanted influences (see section 2.6); section 5.5 will get back to these concerns.

Figure 5.9 shows and the following sections discuss the result of our discussions with SMEs about "What goes into a Product?".

Figure 5.9 The Product resource theoretical model and a JSON example



5.2.3 Choosing a property name and type

Choosing names and types is an art that parts 2 and 3 cover in depth. At this stage, we choose meaningful names and write them as we would write variable names in code or query parameters in URLs. For instance, the reference uniquely identifying a product is `productReference`, `product_reference`, or any other variation depending on our preferences.

We also pick portable types (see section 5.1.2) that seem appropriate for the data and indicate a format when necessary. The `Product` model of figure 5.9 showcases all the portable data types. The `productReference` is an `integer`, `name` is a `string`, `price` is a `float`, and `isProductAvailable` is a `boolean`. Note that `dateAdded` is a `string` with a (date) format (`YYYY-MM-DD`); see chapter 8 for more about formats and dates. Not all properties are atomic values; the `keywords` property is an array of `string`, and the `supplier` one is an object with its own properties.

We may provide an optional description to capture additional information that the combination of resource name, property name, and type can't convey. For instance, the product's price is expressed in US dollars. See chapter 17 for more about API documentation.

5.2.4 Indicating required properties

In a theoretical resource model, the required flag ("req." in figure 5.9) indicates properties essential for the concept. It's mainly a subject-matter question, but we can also think about what consumers must provide as input and what the API implementation always returns as output. A product doesn't make sense without a `productReference`, `name`, or `price` but can exist without a `description` or `keywords`. We must also set this flag for deeper elements; for example, the `supplier` object must have a `supplierCode` property.

5.3 Designing inputs and outputs data models

Once we've designed the theoretical resources models, we can easily design inputs and outputs for each API operation using that base; we pick the element we need in those models according to the context. Still, we can simplify our work by identifying patterns and recipes.

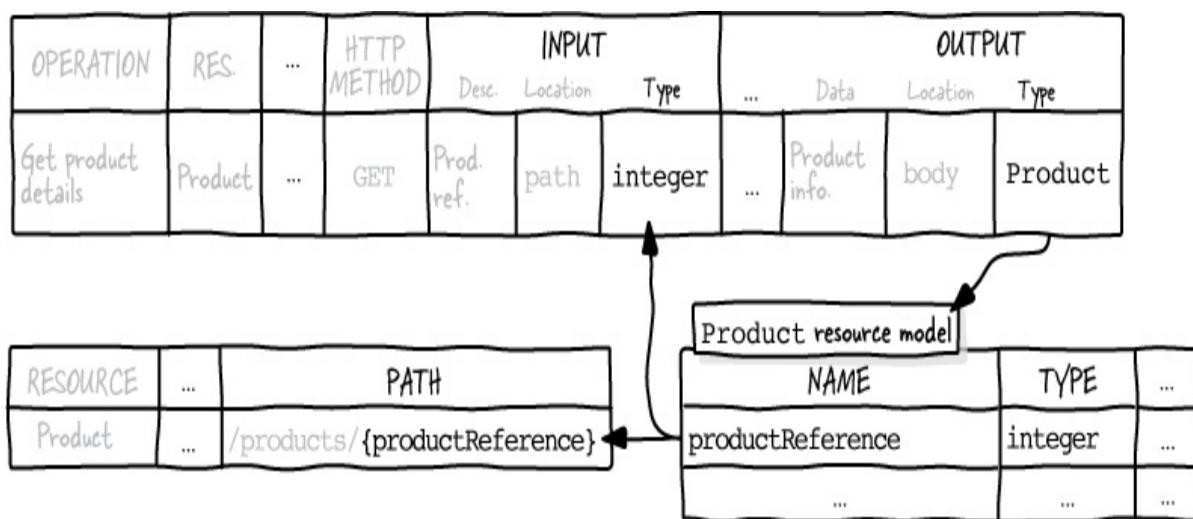
This section allows us to practice designing input and successful output from resource models for each of the "Online Shopping" typical create, read, search, update, and delete operations (listed in figure 5.3 of section 5.1). The following section 5.4 generalizes our learnings to simplify this process. It also quickly provides a temporary error model to complete our design.

5.3.1 Designing a read operation inputs and success outputs

As shown in figure 5.10, the "Get product details" (GET /products/{product reference}) operation has a single input, "Product reference," and its success output is "Product information;" we can design them taking the Product theoretical model as a base.

We "design" the output using the unmodified theoretical Product model as the operation returns a Product (see figure 5.9 of section 5.2.2). In this context, each property's required flag indicates whether it is systematically returned. For instance, the name property is required, so the operation returns it for all products. On the other hand, the operation may or may not return the non-required description property.

Figure 5.10 Modeling a path parameter based on the output



Once we've designed the output, we can work on the input, the "Product reference" that uniquely identifies the product resource. We can reuse the `productReference` property of the Product (output or theoretical) model as a

base to design it and set this input type to `integer`. We also rename the corresponding path parameter in the Product resource path (`/products/{product reference}`) to be consistent with the model (`/products/{productReference}`). Remember that `{productReference}` is a placeholder whose name never appears in an API call as an actual value replaces it.

Tip

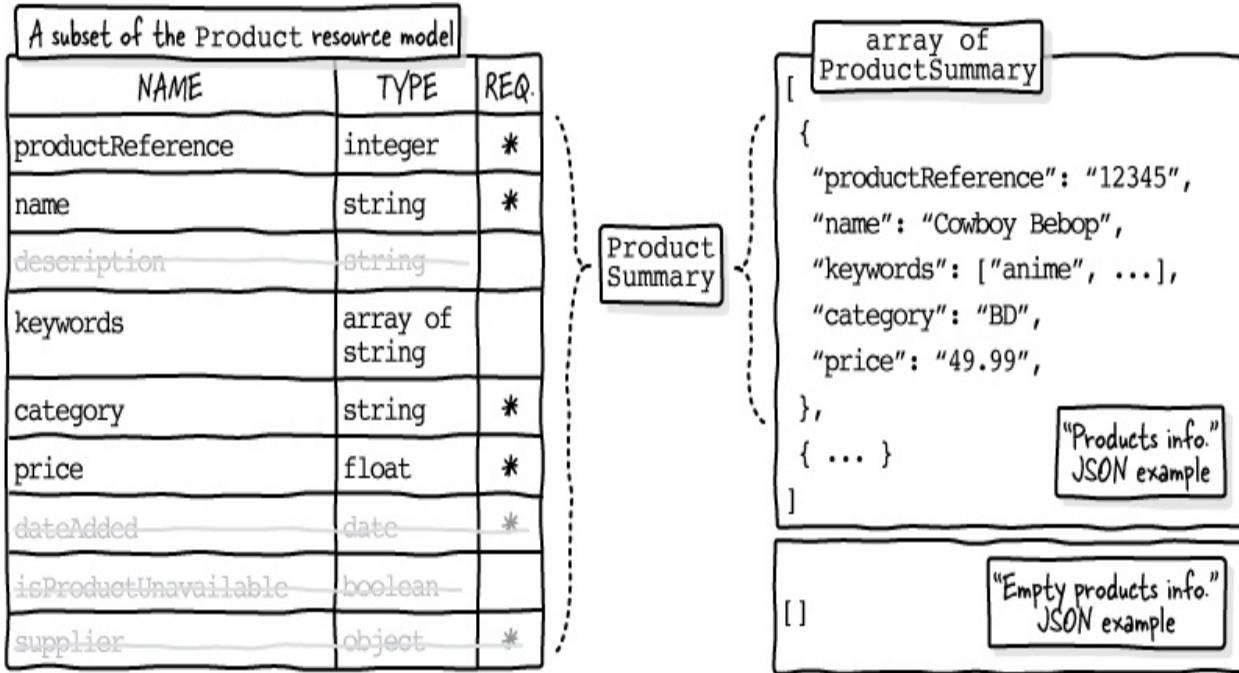
When modeling operation data, starting with the output design is convenient since each piece of input data is identical to a part of the output.

5.3.2 Designing a search operation inputs and success outputs

The "Search for products" operation (`GET /products`) has a "Filters" input and "Products information" and "Empty products information" successful outputs.

Figure 5.11 shows the modelization of the outputs. Both outputs represent the result of a search across the "Catalog" resource. Hence, they are an array of `Product`; one contains some `Product` while the other is empty.

Figure 5.11 The product summary model is a subset of the product model



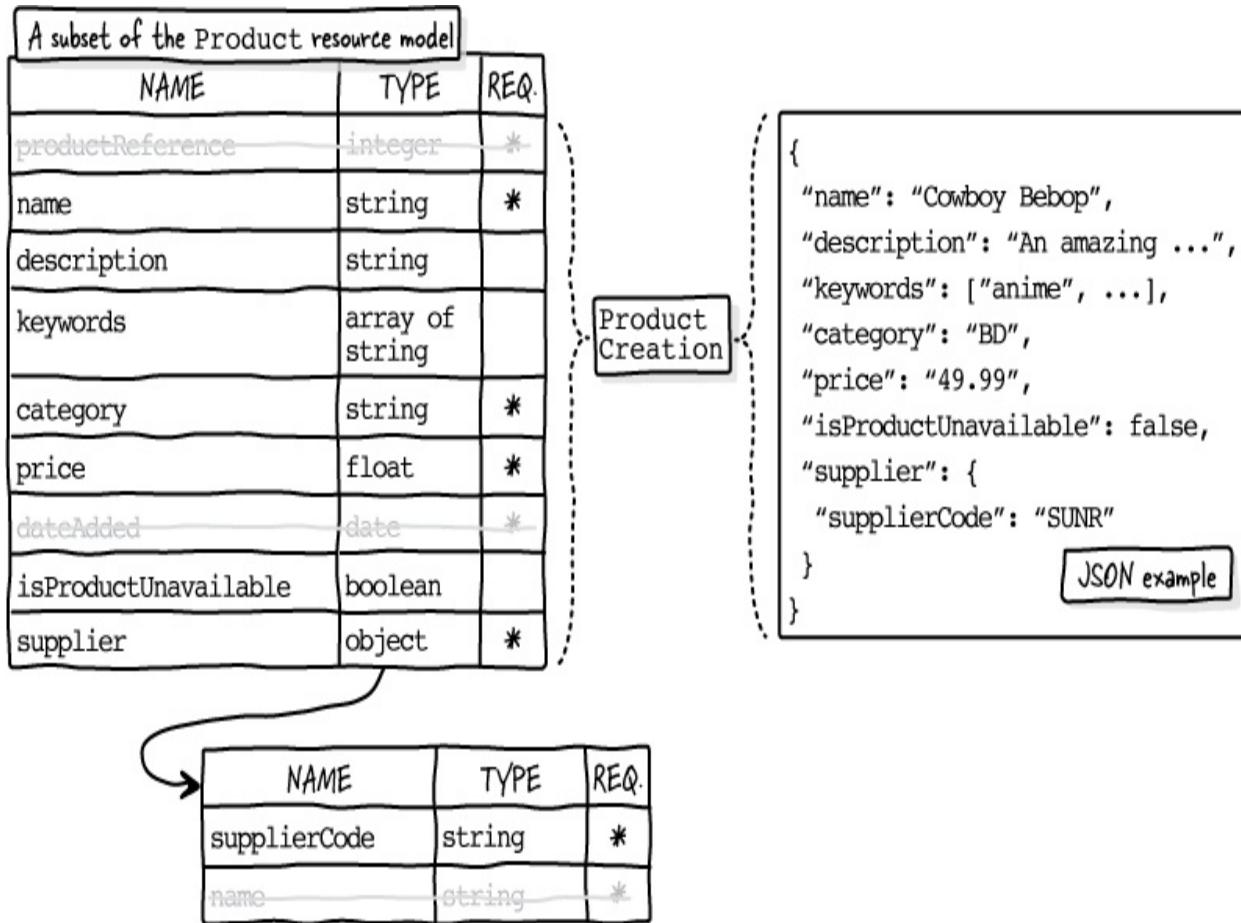
We have two options for each array element: returning all data of the Product theoretical model or a subset of it; this choice is discussed in parts 2 and 3). At this stage of our learning, we choose the most common option: returning a subset or summary of available information, an array of ProductSummary. The ProductSummary model has properties like productReference, name, category, keywords, and price. We only need select properties giving a good overview of a product, not necessarily all required ones; for instance, we excluded dateAdded. To get all the data of a specific product, consumers can use the "Get product details" operation by sending a GET /products/{productReference} request using the productReference property value in the {productReference} path parameter.

The "Filters" query input selects a subset of products. Parts 2 and 3, thoroughly discuss the design of query and search parameters. At this stage of our learning, we choose a common option: mapping search filters to the output model properties. For example, we can have two query parameters to allow filtering on keywords or category, their name and type identical to the original properties of the ProductSummary model. An example of a filtered request could be GET /products?keywords=anime,fantasy&category=BD. Note how comma-separated values represent the keywords (array of string); later chapters discuss other options.

5.3.3 Designing a create operation inputs and success outputs

The "Add a product to the catalog" operation (`POST /products`) has a single input, "Product information," and returns "Product information" and "Product URL" when a product is successfully created.

Figure 5.12 The data participating in a product creation



The "Product information" output is the data of the created product, hence the Product theoretical model, hence the same data "Get product details" returns. However, note that for performance reasons, we may return a `ProductMinimal` model containing only the `productReference` property; check out chapter 13 for more. As seen in section 4.6.2, the "Product URL" is the created product URL. We strictly follow HTTP documentation and put it into the standard `Location` header as a string. Its value is, for example, `/products/12345` (`/products/{productReference}`); 12345 is the

`productReference` of the created product.

Though they share the same name in our API spreadsheet, the "Product information" input differs from the output. It is the data needed to create a product, a subset of the `Product` output model stripped of implementation-managed properties. After discussing with the SMEs and implementation team, we decided the `ProductCreation` model contains all of the `Product` properties minus `productReference`, `dateAdded`, and the name of the supplier. The implementation generates the first two and sets the third based on the `supplierCode`. In this context, each property's required flag indicates whether consumers must provide it. For instance, consumers can't create a product without a name but don't need to provide a description. Chapter 9 discusses how the required flag may impact user experience.

5.3.4 Designing an update operation inputs and success outputs

The "Modify a product" operation (`PUT` or `PATCH` `/products/{productReference}`) has two inputs, "Product reference" and "Modified product information," and returns "Product information" when the product is successfully updated.

We have already designed the output data. The returned "Product information" is the new state of the `Product` resource; thus, its data is the `Product` theoretical model, the same "Get product details" or "Add a product to the catalog" return. The "Product reference" path parameter is the same as for "Get product details" as both operations share the same resource (see section 5.3.1).

The input design may vary depending on the HTTP method (`PUT` or `PATCH`). If we use `PUT /products/{productReference}`, the "Modified product information" input should contain all properties necessary to replace entirely, hence re-create the product identified by the path. So we need the same information as in the `ProductCreation` input of "Add a product to the catalog" (`POST /product`). We may rename the model `ProductCreationOrReplacement`. If we use `PATCH /products/{productReference}`, which allows for a partial update, the common option would be to have a `ProductModification` model that is

similar to `ProductCreation`, but all properties are optional (see Chapter 9 for more about `PATCH` options).

Some data may be restricted from updates due to subject matter-related reasons. For example, a product's category may be defined upon creation and cannot be modified afterward. To limit what can be updated, the input data model may be a subset of the one used for creation. Check out chapters 9 and 17, which discuss this topic further.

5.3.5 Designing a delete operation inputs and success outputs

The "Remove a product from the catalog" operation (`DELETE /products/{product reference}`) has a single input, "Product reference," and no outputs. We're in the same situation as with `GET`, `PUT`, or `PATCH /products/{product reference}`. This operation manipulates a Product resource, so we end with a `DELETE /products/{productReference}` where the `{productReference}` path parameter maps the `productReference` property of the theoretical product resource.

5.3.6 Designing a temporary error data model

This chapter doesn't cover error output modeling, but we can design a temporary simple data model for all error outputs (4xx and 5xx) of all operations (figure 5.13). The `Error` model is an object with a required `message` property, a `string`, conveying explicit human-readable information about the problem. See section 9.6 for a complete design.

Figure 5.13 A simplistic and generic error model

NAME	TYPE	REQ.
message	string	*

JSON example

```
{  
  "message": "The product type is missing"  
}
```

5.4 Streamlining data modelization

Now that we have designed all resources and operations' inputs and success

outputs, we can identify seven typical models that allow us to streamline the outputs and inputs design for CRUD and "Do" operations: complete (or theoretical), summarized, minimal, identifier, creation, replacement, and modification. Figure 5.14 shows where to use these models. This section explains how to design and use these typical CRUD and "Do" operations models. It also explains how to simplify listing and modeling properties and discusses some risks related to non-differentiating similarly named elements.

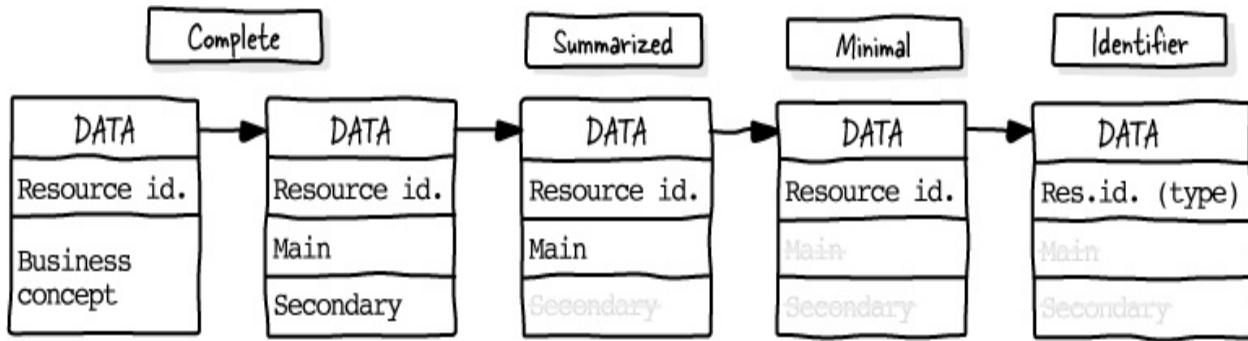
Figure 5.14 Usages of the typical models in operations

OPERATION		PATH	QUERY	REQUEST BODY	RESPONSE BODY
Create resource	POST /resources			Creation	Complete, Minimal
Search resources	GET /resources		Break down of response		List of Complete or Summarized
Read resource	GET /resources/{resourceId}				
Replace resource, Create resource	PUT /resources/{resourceId}	Identifier		Replacement, Creation	Complete
Partial res. update	PATCH /resources/{resourceId}			Modification	
Delete resource	DELETE /resources/{resourceId}				
Action resource	POST /do			Creation	Complete

5.4.1 Designing and using the complete, summarized, minimal, and identifier models

Figure 5.15 shows we can derive the complete model to design the summarized, minimal, and identifier ones.

Figure 5.15 From complete to summarized, minimal, and identifier models



A *complete* (or *theoretical*) resource model, such as `Product`, should be designed first, as it is the source for the other models. It contains all possible business concept properties, including a resource identifier. We can use it as a successful output body for create, read, search (in a list), and update operations.

A *summarized* model, such as `ProductSummary`, contains a subset of the data from the *complete* resource model, including resource identifiers and "main" properties representing a meaningful summary. We can use it as an output of search operations (in a list).

On search operations, we can leverage the properties of the *complete* or *summarized* model we use in the list as a base for query parameters, as in `GET /products?keywords=anime,fantasy&category=BD`.

A *minimal* model, such as `ProductMinimal`, is a subset of the *complete* or *summarized* models, containing only the resource identifier. We can use it as an output for a create operation (discussed in chapter 13).

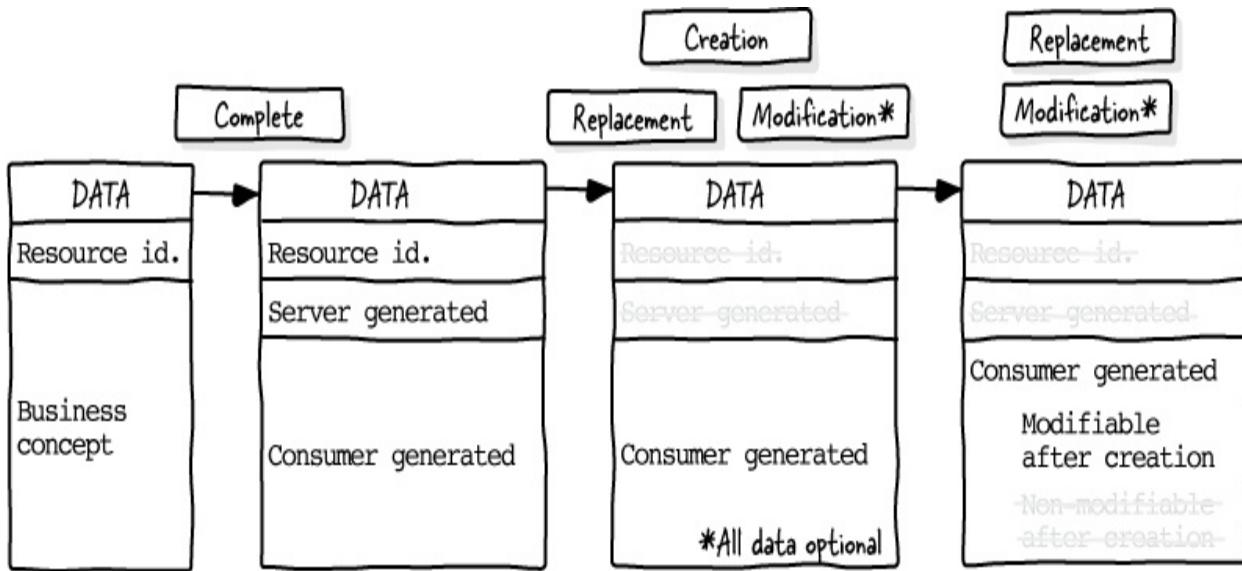
The *summarized* and *minimal* models can also be embedded in other resources. For example, an `Order` may contain a list of products whose elements are one of these models.

An *identifier* model is the type of the resource identifier of the *complete* data model. We can use it as a path parameter (`/products/{productReference}`).

5.4.2 Designing and using the creation, replacement, and modification models

Figure 5.16 shows how we can derive the complete model to design the creation, replacement, and modification ones.

Figure 5.16 From complete to creation, replacement, and modification input models



A *creation* model, such as `ProductCreation`, contains all the properties needed to create a resource and goes into the body input of a "Create resource" operation. It is a subset of the *complete* model that excludes data managed by the implementation, such as the resource unique identifier (`productReference`) or creation date (`dateAdded`).

A *replacement* model, such as `ProductReplacement`, is the body input for an "Update resource" operation using `PUT`. It's usually the same model as the *creation* one (`ProductCreationOrReplacement`).

A *modification* model, such as `ProductModification`, is the body input for an "Update resource" operation using `PATCH`. It's usually a copy of the *creation* model, where all properties are non-required.

Both *replacement* and *modification* models can also be a subset of the *creation* model if some properties are not modifiable after creation.

Note

Chapters 9 and 13 discuss the pros and cons, modeling alternatives, and concerns for "Update resource" operations with PUT and PATCH.

5.4.3 Modeling data for "Do" operations

In section 4.7, we learned about representing "Do" or non-CRUD operations using the REST model; we fundamentally defined two options: creating a business concept resource or an action resource. This section discusses how to model their data.

If we choose the business concept option and create an "Executions" or "Results" resource, "Do" is represented by a create operation (`POST /executions` or `POST /results`). We can model these resources like any other resource, as explained in sections 5.3 and 5.4. Design "Results" resource like any business concept. The work is similar for "Executions" resources; the complete model, used as output, contains all "Do" input and output data (the numbers to sum and their sum, for example) plus a resource identifier. The input is a creation model only containing what is needed to perform the action (the numbers to sum).

With an action resource, we can have a `POST /do` behaving like a function whose body input, as in the previous option, contains all needed to perform the action (numbers to sum) and whose body output has the result (the sum). Still, I recommend designing the output models similarly to "Executions" resources minus the resource identifier; returning all input and output data (complete model) allows consumers to understand the source of the result.

5.4.4 Listing and modeling properties

To streamline the design process, list properties without worrying about details (final names or types), group those belonging to a sub-concept to create sub-objects, evaluate each element, and remove any that don't make sense. Reiterate with deeper objects, such as `supplier`, if necessary. Finally, choose the most appropriate name and type for each element and determine if it's required. For more information, refer to 2 and 3.

5.4.5 Differentiating similarly named elements

Similarly named resources, inputs, or outputs may be the same but not share the same model or not be the same concept depending on the context. It's essential to differentiate them and identify what they are to design the appropriate models contributing to creating an API that fulfills the needs.

In section 5.3.3, we realized "Product information" was both an input and output for the "Add a product to the catalog" operation, but with different modelizations. Thanks to the typical models of section 5.4, we can seamlessly differentiate elements between inputs and outputs and across operations that manipulate the same resource.

Once we identify and model resources, we must be careful not to use them whenever they're mentioned. Business concepts with similar names may differ depending on the context and require different resources. For instance, the "Product" resource identified for catalog-related operations likely differs from the "Product" in the cart context. Adding a "Product" to a cart typically requires a product reference, whose value is the same as the `productReference` of the `Product` model and the quantity. We should differentiate these concepts by naming them "Catalog Product" and "Cart Item." Additionally, when listing the items (or products) added to the cart, we can embed some of the "Catalog Product" information in the "Cart Item Summary" to provide a complete picture.

What may seem like a unique business concept is often multiple concepts adapted to a context or use case. Always check with SMEs if seemingly identical concepts are the same; rename them using a suffix, prefix, or more precise name. At last resort, fine-grained modelization can uncover non-differentiation issues by demonstrating the two concepts are different. When concepts are related, leverage typical models of section 5.4 or their elements to compose new models.

5.5 Leveraging data to ensure completeness and proper focus

We now have a solid draft of our programming interface design. Still, before moving forward, we must review it for completeness and accuracy as we have investigated the needs deeper with data modelization. Similar to what

we did during the initial needs analysis (see chapter 2), this review involves checking input sources and output usages and avoiding exposing provider or too specific consumer perspectives. We also add a thorough error handling check. This section overviews these concerns using the "Online Shopping" programming interface data as an example.

5.5.1 Spotting missing elements with sources and usages

As in section 2.3.3, we must check if consumers can provide requested inputs and what they do with outputs to spot missing use cases, steps, and operations.

The fine-grained inputs source check can reveal new API parts that were previously unknown during the needs analysis. For instance, to "Add a product to the catalog," consumers must provide information about the supplier, especially a code identifying them. After discussing with SMEs, we realized a "Select a supplier" step needs to be added to the "Fill the catalog" use case (see section 2.3.3).

We likely spotted all the use cases and steps we could discover during needs analysis by investigating output usage. However, we can do a quick second pass. For example, we can consider what end-users may do with the supplier information returned with products. We may add this as a filter to "Search products," but if we decide the supplier property is irrelevant for end-users, we can remove this information. But catalog administrators need it; that leads to security questions discussed in chapter 11, so we keep it for now.

5.5.2 Ensuring complete business error handling

Part 2 and 3, especially chapters 9 and 17, thoroughly discuss errors and their documentation. At this stage of our learning, we focus on identifying them. Now that we have a fine view of all inputs, we can double-check with SMEs what possible business errors can happen, especially on creation and modification operations.

In most cases, the newly identified errors should be a refinement of the ones

already identified during needs analysis. For instance, "You can't add a product with a price which is negative or above 100,000" is a refinement of "wrong product information." We can add the newly detected specific error cases to the description of the 400 error cases. We must also check if errors impact the use cases identified during need analysis (adding new branches, operations, etc.), though that should be rare.

5.5.3 Focusing on the proper elements

We must analyze each fine-grained input and output data model element from the angles uncovered in section 2.6, data must be aligned with the needs and free of unwanted provider or too specific consumer influence. Note that later chapters will show us more possible issues and solutions.

We must ensure consumers can send the data they want and get the data they need to achieve their goals in the context of the use cases identified during needs analysis (see 2). For example, the "Product" resource data model returned by "Get product details" may miss data about the size of the product, which is crucial for consumers, though not used elsewhere in the API.

We must ensure that the names, types, or data organization are not exposing the provider's perspective, hence exposing data organization or business logic (there's less risk of software architecture issues here). For instance, if the price description says, "Add 10% on Fridays between 4 pm and 7 pm", it would be better to find a way to avoid consumers having to deal with that. We'll get back to that topic in part 2.

We must ensure a specific consumer's perspective doesn't taint data by checking if it doesn't mimic existing UI or isn't specialized for one consumer. A typical UI influence would be to have description and keyword grouped under a summary property in the product model because that's how information is presented on the existing website. But from a pure data perspective, agnostic of the context, this organization doesn't make any sense.

5.6 Summary

- Use JSON portable data types (strings, numbers, booleans, arrays, and objects) to model all data for better compatibility between providers and consumers.
- Design data models for resource/business concepts and derive them into inputs and outputs for each API operation.
- Typical data models used as input or output for CRUD and "Do" operations include complete, summarized, minimal, identifier, creation, replacement, and modification.
- The complete (or theoretical) model contains all business concept properties, including a resource identifier. Design it first; it is the source for the other models. Use it for create, read, search (list), or update operations.
- The summarized model is a subset of the complete model, including the resource identifier and properties representing a meaningful summary. Use it as output for search operations (list).
- The minimal model is a subset of the complete model, containing only the resource identifier. Use it as output for create operations.
- The identifier model is the type of the resource identifier of the complete data model. Use it for path parameters.
- The creation model is a subset of the complete model that excludes data managed by the implementation. Use it as input for create operations.
- The replacement model is usually the same as the creation one. Use it as input for update operations using PUT.
- The modification model is usually a copy of the creation model where all properties are non-required. Use it as input for update operation using PATCH.
- To design models, list properties without worrying about details (final names or types), reorganize and filter them, and finally, choose name, type, and required status.
- Investigate fine-grained data sources and usages to spot missing use cases or steps.
- Identify all business errors by leveraging input data.
- Ensure fine-grained data is aligned with the needs and free of unwanted provider or too specific consumer influence.

welcome

Thank you for purchasing *The Design of Web APIs, Second Edition*.

Web APIs are everywhere; we use them all the time, often without even realizing it. Whether sharing a photo on social media or hailing a ride through an app, web APIs are crucial in making it happen. For developers, APIs are essential as most modern systems rely on multiple software components communicating with each other. We need them to build simple web applications to complex distributed systems. APIs are also products in their own right, as exemplified by Stripe or Twilio. Even government agencies rely on APIs to power their digital services.

The design quality is crucial for web APIs, whether seen as technical interfaces or products, used by a single application or multiple, or created for internal use or third-party. Poorly designed public or private APIs can harm developers' productivity, system performance and integrity, end-users experience, and organization's revenue.

This book aims to help you develop an API designer's mindset and design exceptional web APIs, specifically REST APIs. In these chapters, we will explore the true nature of API design as both a result and a process. We will learn how to analyze and evaluate requirements to identify the API capabilities, discover HTTP and REST, and understand how to use them to represent these capabilities. We will discuss how to create interoperable and user-friendly APIs, ensuring that anyone can instantly use the API's data and operations. We will also learn how to integrate various constraints, especially security, into our design. Additionally, we will focus on handling modifications and preventing breaking an API design unintentionally or breaking it intentionally when it makes sense. Furthermore, we will learn how to become efficient API designers by learning various principles and recipes for making design decisions when faced with new problems. We will learn to convince others (and ourselves) that our design decisions are correct.

I am writing the second edition of this book to address what was not working

well in the first edition and expand it. I also reorganize the content to make it easier to follow and integrate new ideas and feedback received from readers. I am keeping the spirit of the first edition but rewriting everything. In a way, this is almost a new book. Your feedback is essential to make this new edition the best companion on your journey of API design; I hope you'll add your comments to the [Livebook discussion forum](#).

— Arnaud Lauret

In this book

[welcome](#) [1 What is API design?](#) [2 Analyzing needs](#) [3 Observing operations from the REST angle](#) [4 Representing operations with HTTP](#) [5 Modeling data](#)