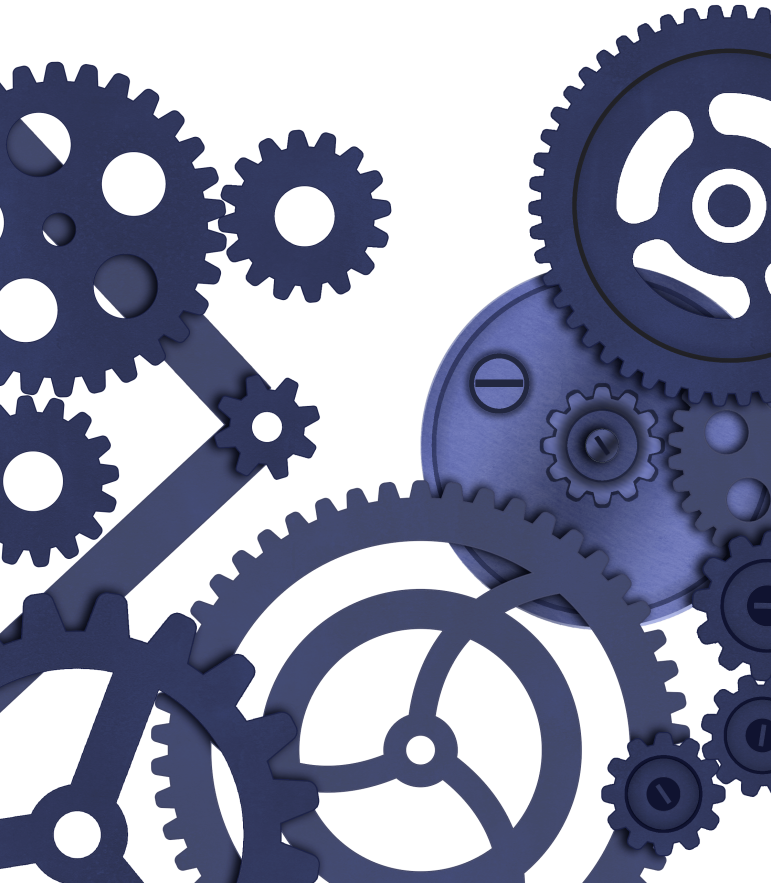
Sergey Konstantinov

The API



Sergey Konstantinov. The API.
yatwirl@gmail.com · linkedin.com/in/twirl · twirl.substack.com

API-first development is one of the hottest technical topics nowadays since many companies have started to realize that APIs serves as a multiplier to their opportunities — but it amplifies the design mistakes as well.

This book is written to share expertise and describe best practices in designing and developing APIs. It comprises six sections dedicated to the following topics:

- The API design
- API patterns
- Backward compatibility
- HTTP APIs & the REST architectural principles
- SDKs and UI libraries
- API product management.

Illustrations & inspiration by Maria Konstantinova · art.mari.ka

This book is distributed under the Creative Commons Attribution-NonCommercial 4.0 International licence.

Source code available at github.com/twirl/The-API-Book

Share: facebook · twitter · linkedin · reddit

TABLE OF CONTENTS

INTRODUCTION
- Chapter 1. On the Structure of This Book
- Chapter 2. The API Definition
- Chapter 3. An Overview of Existing API Development Solutions
- Chapter 4. API Quality Criteria
- Chapter 5. The API-First Approach
- Chapter 6. On Backward Compatibility
- Chapter 7. On Versioning
- Chapter 8. Terms and Notation Keys

SECTION I. THE API DESIGN
- Chapter 9. The API Contexts Pyramid
- Chapter 10. Defining an Application Field
- Chapter 11. Separating Abstraction Levels
- Chapter 12. Isolating Responsibility Areas
- Chapter 13. Describing Final Interfaces
- Chapter 14. Annex to Section I. Generic API Example

SECTION II. THE API PATTERNS
- Chapter 15. On Design Patterns in the API Context
- Chapter 16. Authenticating Partners and Authorizing API Calls
- Chapter 17. Synchronization Strategies
- Chapter 18. Eventual Consistency
- Chapter 19. Asynchronicity and Time Management
- Chapter 20. Lists and Accessing Them
- Chapter 21. Bidirectional Data Flows. Push and Poll Models
- Chapter 22. Multiplexing Notifications. Asynchronous Event Processing
- Chapter 23. Atomicity of Bulk Changes
- Chapter 24. Partial Updates
- Chapter 25. Degradation and Predictability

SECTION III. THE BACKWARD COMPATIBILITY
- Chapter 26. The Backward Compatibility Problem Statement
- Chapter 27. On the Waterline of the Iceberg
- Chapter 28. Extending through Abstracting
- Chapter 29. Strong Coupling and Related Problems
- Chapter 30. Weak Coupling

Chapter 31. Interfaces as a Universal Pattern
Chapter 32. The Serenity Notepad
SECTION IV. HTTP APIS & THE REST ARCHITECTURAL PRINCIPLES
Chapter 33. On the HTTP API Concept and Terminology
Chapter 34. Advantages and Disadvantages of HTTP APIs Compared to Alternative Technologies
Chapter 35. The REST Myth
Chapter 36. Components of an HTTP Request and Their Semantics
Chapter 37. Organizing HTTP APIs Based on the REST Principles
Chapter 38. Designing a Nomenclature of URLs. The CRUD Operations
Chapter 39. Working with HTTP API Errors
Chapter 40. Final Provisions and General Recommendations
[WORK IN PROGRESS] SECTION V. SDKS & UI LIBRARIES
Chapter 41. On the Content of This Section
Chapter 42. SDKs: Problems and Solutions
Chapter 43. Problems of Introducing UI Components
Chapter 44. Decomposing UI Components
Chapter 45. The MV* Frameworks
Chapter 46. The Backend-Driven UI
Chapter 47. Shared Resources and Asynchronous Locks
Chapter 48. Computed Properties
Chapter 49. Conclusion
SECTION VI. THE API PRODUCT
Chapter 50. The API as a Product
Chapter 51. API Business Models
Chapter 52. Developing a Product Vision
Chapter 53. Communicating with Developers
Chapter 54. Communicating with Business Owners
Chapter 55. An API Services Range
Chapter 56. API Key Performance Indicators
Chapter 57. Identifying Users and Preventing Fraud
Chapter 58. The Technical Means of Preventing ToS Violations
Chapter 59. Supporting customers
Chapter 60. The Documentation
Chapter 61. Testing Environments
Chapter 62. Managing Expectations

INTRODUCTION

Chapter 1. On the Structure of This Book

The book you're holding in your hands is dedicated to developing APIs as a separate engineering task. Although many concepts we're going to discuss apply to any type of software, our primary goal is to describe those problems and approaches to solving them that are most relevant in the context of the API subject area.

We expect that the reader possesses expertise in software engineering, so we do not provide detailed definitions and explanations of the terms that a developer should already be familiar with in our understanding. Without this knowledge, it will be rather uncomfortable to read the last section of the book (and even more so, other sections). We sincerely apologize for this but that's the only way of writing the book without tripling its size.

The book comprises the Introduction and six large sections. The first three (namely, “The API Design”, “The API Patterns”, and “The Backward Compatibility”) are fully abstract and not bound to any concrete technology. We hope they will help those readers who seek to build a systematic understanding of the API architecture in developing complex interface hierarchies. The proposed approach, as we see it, allows for designing APIs from start to finish, from a raw idea to concrete implementation.

The fourth and fifth sections are dedicated to specific technologies, namely developing HTTP APIs (in the “REST paradigm”) and SDKs (we will mostly talk about UI component libraries).

Finally, in the sixth section, which is the least technical of all, we will discuss APIs as products and focus on non-engineering aspects of the API lifecycle: doing market research, positioning the service, communicating to consumers, setting KPIs for the team, etc. We insist that the last section is equally important to both PMs and software engineers as products for developers thrive only if the product and technical teams work jointly on them.

Let's start.

Chapter 2. The API Definition

Before we start talking about the API design, we need to explicitly define what the API is. Encyclopedias tell us that “API” is an acronym for “Application Program Interface.” This definition is fine but useless, much like the “Man” definition by Plato: “Man stands upright on two legs without feathers.” This definition is fine again, but it gives us no understanding of what's so important about a Man. (Actually, it's not even “fine”: Diogenes of Sinope once brought a plucked chicken, saying “That's Plato's Man.” And Plato had to add “with broad nails” to his definition.)

What does the API *mean* apart from the formal definition?

You're possibly reading this book using a Web browser. To make the browser display this page correctly, a bunch of things must work correctly: parsing the URL according to the specification, the DNS service, the TLS handshake protocol, transmitting the data over the HTTP protocol, HTML document parsing, CSS document parsing, correct HTML+CSS rendering, and so on and so forth.

But those are just the tip of the iceberg. To make the HTTP protocol work you need the entire network stack (comprising 4-5 or even more different level protocols) to work correctly. HTML document parsing is performed according to hundreds of different specifications. Document rendering operations call the underlying operating system APIs, or even directly graphical processor APIs. And so on, down to modern CISC processor commands that are implemented on top of the API of microcommands.

In other words, hundreds or even thousands of different APIs must work correctly to make basic actions possible such as viewing a webpage. Modern Internet technologies simply couldn't exist without these tons of APIs working fine.

An API is an obligation. A formal obligation to connect different programmable contexts.

When I'm asked for an example of a well-designed API, I usually show a picture of a Roman aqueduct:

The Pont-du-Gard aqueduct. Built in the 1st century AD. Image Credit: **igorelick @ pixabay**

- It interconnects two areas
- Backward compatibility has not been broken even once in two thousand years.

What differs between a Roman aqueduct and a good API is that in the case of APIs, the contract is presumed to be *programmable*. To connect the two areas, *writing some code* is needed. The goal of this book is to help you design APIs that serve their purposes as solidly as a Roman aqueduct does.

An aqueduct also illustrates another problem with the API design: your customers are engineers themselves. You are not supplying water to end-users. Suppliers are plugging their pipes into your engineering structure, building their own structures upon it. On the one hand, you

may provide access to water to many more people through them, not spending your time plugging each individual house into your network. On the other hand, you can't control the quality of suppliers' solutions, and you are to blame every time there is a water problem caused by their incompetence.

That's why designing an API implies a larger area of responsibility. **An API is a multiplier to both your opportunities and your mistakes**.

Chapter 3. An Overview of Existing API Development Solutions

In the first three sections of this book, we aim to discuss API design in general, not bound to any specific technology. The concepts we describe are equally applicable to, let's say, web services and operating system (OS) APIs.

Still, two main scenarios dominate the stage when we talk about API development:

- Developing client-server applications
- Developing client SDKs.

In the first case, we almost universally talk about APIs working atop the HTTP protocol. Today, the only notable examples of non-HTTP-based client-server interaction protocols are WebSocket (though it might, and frequently does, work in conjunction with HTTP), MQTT, and highly specialized APIs like media streaming and broadcasting formats.

HTTP API

Although the technology looks homogeneous because of using the same application-level protocol, in reality, there is significant diversity regarding different approaches to realizing HTTP-based APIs.

First, implementations differ in terms of utilizing HTTP capabilities:

- Either the client-server interaction heavily relies on the features described in the HTTP standard (or rather standards, as the functionality is split across several different RFCs), or
- HTTP is used as transport, and there is an additional abstraction level built upon it (i.e., the HTTP capabilities, such as the headers and status codes nomenclatures, are deliberately reduced to a bare minimum, and all the metadata is handled by the higher-level protocol).

The APIs that belong to the first category are usually denoted as “REST” or “RESTful” APIs. The second category comprises mostly different RPC formats.

Second, different HTTP APIs rely on different data formats:

- REST APIs and some RPCs (JSON-RPC, GraphQL, etc.) use the JSON format (sometimes with some additional endpoints to transfer binary data)
- gRPC and some specialized RPC protocols like Apache Avro utilize binary formats (such as Protocol Buffers, FlatBuffers, or Apache Avro's own format)
- Finally, some RPC protocols (notably SOAP and XML-RPC) employ the XML data format (which is considered a rather outdated practice by many developers).

All the above-mentioned technologies operate in significantly dissimilar paradigms, which give rise to rather hot “holy war” debates among software engineers. However, at the moment this book is being written we observe the choice for general-purpose APIs is reduced to the “REST API (in fact, JSON-over-HTTP) vs. gRPC vs. GraphQL” triad.

SDKs

The term “SDK” (stands for “Software Development Kit”) is not, strictly speaking, related to APIs: this is a generic term for a software toolkit. As with “REST,” however, it got some popular reading as a client framework to work with some underlying API. This might be, for example, a wrapper to a client-server API or a UI to some OS API. The major difference from the APIs we discussed in the previous paragraph is that an “SDK” is implemented for a specific programming language and platform to work with some underlying low-level API.

Unlike client-server APIs, such SDKs can hardly be generalized as each of them is developed for a specific language-platform pair. Interoperable SDKs exist, notably cross-platform mobile (React Native, Flutter, Xamarin, etc.) and desktop (JavaFX, QT, etc.) frameworks and some

highly-specialized solutions (Unity). However, they are still narrowly focused on concrete technologies.

Still, SDKs feature some generality in terms of *the problems they solve*, and Section V of this book will be dedicated to solving these problems of translating contexts and making UI components.

Chapter 4. API Quality Criteria

Before we start laying out the recommendations, we ought to specify what API we consider “fine,” and what the benefits of having a “fine” API are.

Let's discuss the second question first. Obviously, API “finesse” is primarily defined through its capability to solve developers' and users' problems. (One could reasonably argue that solving problems might not be the main purpose of offering an API to developers. However, manipulating public opinion is not of interest to the author of this book. Here we assume that APIs exist primarily to help people, not for some other covertly declared purposes.)

So, how might a “fine” API design assist developers in solving their (and their users') problems? Quite simply: a well-designed API allows developers to do their jobs in the most efficient and convenient manner. The distance from formulating a task to writing working code must be as short as possible. Among other things, this means that:

- It must be totally obvious from your API's structure how to solve a task:
    - Ideally, developers should be able to understand at first glance, what entities are meant to solve their problem
- The API must be readable:
    - Ideally, developers should be able to write correct code after just looking at the methods' nomenclature, never bothering about details (especially API implementation details!)
    - It is also essential to mention that not only should the problem solution (the “happy path”) be obvious, but also possible errors and exceptions (the “unhappy path”)
- The API must be consistent:
    - While developing new functionality (i.e., while using previously unknown API entities) developers may write new code similar to the code they have already written using the known API concepts, and this new code will work.

However, the static convenience and clarity of APIs are simple parts. After all, nobody seeks to make an API deliberately irrational and unreadable. When we develop an API, we always start with clear basic concepts. Providing you have some experience in APIs, it's quite hard to make an API core that fails to meet obviousness, readability, and consistency criteria.

Problems begin when we start to expand our API. Adding new functionality sooner or later results in transforming once plain and simple API into a mess of conflicting concepts, and our efforts to maintain backward compatibility will lead to illogical, unobvious, and simply bad design solutions. It is partly related to an inability to predict the future in detail: your understanding of “fine” APIs will change over time, both in objective terms (what problems the API is to solve, and what is best practice) and in subjective terms too (what obviousness, readability, and consistency *really mean* to your API design).

The principles we are explaining below are specifically oriented towards making APIs evolve smoothly over time, without being turned into a pile of mixed inconsistent interfaces. It is crucial to understand that this approach isn't free: the necessity to bear in mind all possible extension variants and to preserve essential growth points means interface redundancy and possibly excessive abstractions being embedded in the API design. Besides, both make the developers' jobs harder. **Providing excess design complexities being reserved for future use makes sense only if this future actually exists for your API. Otherwise, it's simply overengineering.**

Chapter 5. The API-First Approach

Today, more and more IT companies are recognizing the importance of the “API-first” approach, which is the paradigm of developing software with a heavy focus on APIs.

However, we must differentiate between the product concept of the API-first approach and the technical one.

The former means that the first (and sometimes the only) step in developing a service is creating an API for it, and we will discuss it in “The API Product” section of this book.

If we talk about the API-first approach in a technical sense, we mean the following: **the contract, i.e. the obligation to connect two programmable contexts, precedes the implementation and defines it**. More specifically, two rules must be respected:

- The contract is developed and committed to in the form of a specification before the functionality is implemented.
- If it turns out that the implementation and the contract differ, the implementation is to be fixed, not the contract.

The “specification” in this context is a formal machine-readable description of the contract in one of the interface definition languages (IDL) — for example, in the form of a Swagger/OpenAPI document or a `.proto` file.

Both rules assert that partner developers' interests are given the highest priority:

- Rule #1 allows partners to write code based on the specification without coordinating the process with the API provider:
    - The possibility of auto-generating code based on the specification emerges, which might make development significantly less complex and error-prone or even automate it

    - The code might be developed without having access to the API.
- Rule #2 means partners won't need to change their implementations should some inconsistencies between the specification and the API functionality arise.

Therefore, for your API consumers, the API-first approach is a guarantee of a kind. However, it only works if the API was initially well-designed. If some irreparable flaws in the specification surface, we would have no other option but to break rule #2.

Chapter 6. On Backward Compatibility

Backward compatibility is a *temporal* characteristic of an API. The obligation to maintain backward compatibility is the crucial point where API development differs from software development in general.

Of course, backward compatibility isn't absolute. In some subject areas shipping new backward-incompatible API versions is routine. Nevertheless, every time a new backward-incompatible API version is deployed, developers need to make some non-zero effort to adapt their code to the new version. In this sense, releasing new API versions puts a sort of “tax” on customers who must spend quite real money just to ensure their product continues working.

Large companies that occupy solid market positions could afford to charge such a tax. Furthermore, they may introduce penalties for those who refuse to adapt their code to new API versions, up to disabling their applications.

From our point of view, such a practice cannot be justified. Don't impose hidden levies on your customers. **If you can avoid breaking backward compatibility, never break it**.

Of course, maintaining old API versions is a sort of tax as well. Technology changes, and you cannot foresee everything, regardless of how nicely your API is initially designed. At some point keeping old API versions results in an inability to provide new functionality and support new platforms, and you will be forced to release a new version. But at least you will be able to explain to your customers why they need to make an effort.

We will discuss API lifecycle and version policies in Section II.

Chapter 7. On Versioning

Here and throughout this book, we firmly adhere to semver principles of versioning.

1. API versions are denoted with three numbers, e.g., `1.2.3`.
2. The first number (a major version) increases when backward-incompatible changes in the API are introduced.
3. The second number (a minor version) increases when new functionality is added to the API while keeping backward compatibility intact.
4. The third number (a patch) increases when a new API version contains bug fixes only.

The sentences "a major API version" and "a new API version, containing backward-incompatible changes" are considered equivalent.

It is usually (though not necessary) agreed that the last stable API release might be referenced by either a full version (e.g., `1.2.3`) or a reduced one (`1.2` or just `1`). Some systems support more sophisticated schemes for defining the desired version (for example, `^1.2.3` reads like "get the last stable API release that is backward-compatible to the `1.2.3` version") or additional shortcuts (for example, `1.2-beta` to refer to the last beta release of the `1.2` API version family). In this book, we will mostly use designations like `v1` (`v2`, `v3`, etc.) to denote the latest stable release of the `1.x.x` version family of an API.

The practical meaning of this versioning system and the applicable policies will be discussed in more detail in the "Backward Compatibility Problem Statement" chapter.

Chapter 8. Terms and Notation Keys

Software development is characterized, among other things, by the existence of many different engineering paradigms, whose adherents are sometimes quite aggressive towards other paradigms' adherents. While writing this book, we are deliberately avoiding using terms like “method,” “object,” “function,” and so on, using the neutral term “entity” instead. “Entity” means some atomic functionality unit, like a class, method, object, monad, prototype (underline what you think is right).

As for an entity's components, we regretfully failed to find a proper term, so we will use the words “fields” and “methods.”

Most of the examples of APIs will be provided in the form of JSON-over-HTTP endpoints. This is some sort of notation that, as we see it, helps to describe concepts in the most comprehensible manner. A `GET /v1/orders` endpoint call could easily be replaced with an `orders.get()` method call, local or remote; JSON could easily be replaced with any other data format. The semantics of statements shouldn't change.

Let's take a look at the following example:

```
// Method description
POST /v1/bucket/{id}/some-resource↵
  /{resource_id}
X-Idempotency-Token: <idempotency token>
{
  …
  // This is a single-line comment
  "some_parameter": "example value",
  …
}
→ 404 Not Found
Cache-Control: no-cache
{
  /* And this is
     a multiline comment */
  "error_reason",
  "error_message":
    "Long error message↵
     that will span several↵
     lines"
}
```

It should be read like this:

- A client performs a `POST` request to a `/v1/bucket/{id}/some-resource` resource, where `{id}` is to be replaced with some `bucket`'s identifier (`{something}` notation refers to the nearest term from the left unless explicitly specified otherwise).
- A specific `X-Idempotency-Token` header is added to the request alongside standard headers (which we omit).
- Terms in angle brackets (`<idempotency token>`) describe the semantics of an entity value (field, header, parameter).
- A specific JSON, containing a `some_parameter` field and some other unspecified fields (indicated by ellipsis) is being sent as a request body payload.
- In response (marked with an arrow symbol →) the server returns a `404 Not Found` status code; the status might be omitted (treat it like a `200 OK` if no status is provided).

- The response could possibly contain additional notable headers.
- The response body is a JSON comprising two fields: `error_reason` and `error_message`. Absence of a value means that the field contains exactly what you expect it should contain — so there is some generic error reason value which we omitted.
- If some token is too long to fit on a single line, we will split it into several lines adding ↩to indicate it continues next line.

The term “client” here stands for an application being executed on a user's device, either a native or a web one. The terms “agent” and “user agent” are synonymous with “client.”

Some request and response parts might be omitted if they are irrelevant to the topic being discussed.

Simplified notation might be used to avoid redundancies, like `POST /some-resource {…, "some_parameter", …} → { "operation_id" }`; request and response bodies might also be omitted.

We will use sentences like “`POST /v1/bucket/{id}/some-resource` method” (or simply “`bucket/some-resource` method,” “`some-resource`” method — if there are no other `some-resource`s in the chapter, so there is no ambiguity) to refer to such endpoint definitions.

Apart from HTTP API notation, we will employ C-style pseudocode, or, to be more precise, JavaScript-like or Python-like one since types are omitted. We assume such imperative structures are readable enough to skip detailed grammar explanations. HTTP API-like samples intend to illustrate the *contract*, i.e., how we would design an API. Samples in pseudocode are intended to illustrate how developers might work with the API in their code, or how we would implement SDKs based on the contract.

SECTION I. THE API DESIGN

Chapter 9. The API Contexts Pyramid

The approach we use to design APIs comprises four steps:

- Defining an application field
- Separating abstraction levels
- Isolating responsibility areas
- Describing final interfaces.

This four-step algorithm actually builds an API from top to bottom, from common requirements and use case scenarios down to a refined nomenclature of entities. In fact, moving this way will eventually conclude with a ready-to-use API, and that's why we value this approach highly.

It might seem that the most useful pieces of advice are given in the last chapter, but that's not true. The cost of a mistake made at certain levels differs. Fixing the naming is simple; revising the wrong understanding of what the API stands for is practically impossible.

NB: Here and throughout we will illustrate the API design concepts using a hypothetical example of an API that allows ordering a cup of coffee in city cafes. Just in case: this example is totally synthetic. If we were to design such an API in the real world, it would probably have very little in common with our fictional example.

Chapter 10. Defining an Application Field

The key question you should ask yourself before starting to develop any software product, including an API, is: what problem do we solve? It should be asked four times, each time putting emphasis on a different word.

1. *What* problem do we solve? Could we clearly outline the situation in which our hypothetical API is needed by developers?

2. What *problem* do we solve? Are we sure that the abovementioned situation poses a problem? Does someone really want to pay (literally or figuratively) to automate a solution for this problem?

3. What problem do *we* solve? Do we actually possess the expertise to solve the problem?

4. What problem do we *solve*? Is it true that the solution we propose solves the problem indeed? Aren't we creating another problem instead?

So, let's imagine that we are going to develop an API for automated coffee ordering in city cafes, and let's apply the key question to it.

1. Why would someone need an API to make coffee? Why is ordering coffee via “human-to-human” or “human-to-machine” interfaces inconvenient? Why have a “machine-to-machine” interface?

    - Possibly, we're solving awareness and selection problems? To provide humans with full knowledge of what options they have right now and right here.
    - Possibly, we're optimizing waiting times? To save the time people waste while waiting for their beverages.
    - Possibly, we're reducing the number of errors? To help people get exactly what they wanted to order, stop losing information in imprecise conversational communication, or in dealing with unfamiliar coffee machine interfaces?

The “why” question is the most important of all questions you must ask yourself. And not only about global project goals but also locally about every single piece of functionality. **If you can't briefly and clearly answer the question “what this entity is needed for” then it's not needed**.

Here and throughout we assume, to make our example more complex and bizarre, that we are optimizing all three factors.

2. Do the problems we outlined really exist? Do we really observe unequal coffee-machine utilization in the mornings? Do people really suffer from the inability to find nearby a toffee nut latte they long for? Do they really care about the minutes they spend in lines?

3. Do we actually have resources to solve the problem? Do we have access to a sufficient number of coffee machines and users to ensure the system's efficiency?

4. Finally, will we really solve a problem? How are we going to quantify the impact our API makes?

In general, there are no simple answers to those questions. Ideally, you should start the work with all the relevant metrics measured: how much time is wasted exactly, and what numbers will we achieve providing we have such a coffee machine density. Let us also stress that in the real world obtaining these numbers is only possible if you're entering a stable market. If you try to create something new, your only option is to rely on your intuition.

Why an API?

Since our book is dedicated not to software development per se, but to developing APIs, we should look at all those questions from a different angle: why does solving those problems specifically require an API, not simply a specialized software application? In terms of our fictional example, we should ask ourselves: why provide a service to developers that allows for brewing coffee for end users instead of just making an app?

In other words, there must be a solid reason to split two software development domains: there are vendors that provide APIs, and there are vendors that develop services for end users. Their interests are somehow different to such an extent that coupling these two roles in one entity is undesirable. We will talk about the motivation to specifically provide APIs instead of apps (or as an addition to an app) in more detail in Section III.

We should also note that you should try making an API when, and only when, your answer to question (3) is "because that's our area of expertise." Developing APIs is a sort of meta-engineering: you're writing some software to allow other vendors to develop software to solve users' problems. You must possess expertise in both domains (APIs and user products) to design your API well.

As for our speculative example, let us imagine that in the nearby future, some tectonic shift happened within the coffee brewing market. Two distinct player groups took shape: some companies provide "hardware," i.e., coffee machines; other companies have access to customer audiences. Something like the modern-day flights market looks like: there are air companies that actually transport passengers, and there are trip planning services where users choose between trip options the system generates for them. We're aggregating hardware access to allow app vendors to order freshly brewed coffee.

What and How

After finishing all these theoretical exercises, we should proceed directly to designing and developing the API, having a decent understanding of two things:

- *What* we're doing exactly
- *How* we're doing it exactly.

In our coffee case, we are:

- Providing an API to services with a larger audience so that their users may order a cup of coffee in the most efficient and convenient manner
- Abstracting access to coffee machines' “hardware” and developing generalized software methods to select a beverage kind and a location to make an order.

Chapter 11. Separating Abstraction Levels

“Separate abstraction levels in your code” is possibly the most general advice for software developers. However, we don't think it would be a grave exaggeration to say that separating abstraction levels is also the most challenging task for API developers.

Before proceeding to the theory, we should clearly formulate *why* abstraction levels are so important, and what goals we're trying to achieve by separating them.

Let us remember that a software product is a medium that connects two distinct contexts, thus transforming terms and operations belonging to one subject area into concepts from another area. The more these areas differ, the more interim connecting links we have to introduce.

Returning to our coffee example, what entity abstraction levels do we see?

1. We're preparing an `order` via the API — one (or more) cups of coffee — and receiving payments for this.
2. Each cup of coffee is prepared according to some `recipe` implying the presence of various ingredients and sequences of preparation steps.
3. Each beverage is prepared on a physical `coffee machine`, occupying some position in space.

Each level presents a developer-facing “facet” in our API. While elaborating on the hierarchy of abstractions, we are primarily trying to reduce the interconnectivity of different entities. This would help us to achieve several goals:

1. Simplifying developers' work and the learning curve. At each moment, a developer is operating only those entities that are necessary for the task they're solving right now. Conversely, poorly designed isolation leads to situations where developers have to keep in mind a lot of concepts mostly unrelated to the task being solved.

2. Preserving backward compatibility. Properly separated abstraction levels allow for adding new functionality while keeping interfaces intact.

3. Maintaining interoperability. Properly isolated low-level abstractions help us to adapt the API to different platforms and technologies without changing high-level entities.

Let's assume we have the following interface:

```
// Returns the lungo recipe
GET /v1/recipes/lungo
```

```
// Posts an order to make a lungo
// using the specified coffee-machine,
// and returns an order identifier
POST /v1/orders
{
  "coffee_machine_id",
  "recipe": "lungo"
}
```

```
// Returns the order
GET /v1/orders/{id}
```

Let's consider a question: how exactly should developers determine whether the order is ready or not? Let's say we do the following:

- Add a reference beverage volume to the lungo recipe
- Add the currently prepared volume of the beverage to the order state.

```
GET /v1/recipes/lungo
→
{
  …
  "volume": "100ml"
}
```

```
GET /v1/orders/{id}
→
{
  …
  "volume": "80ml"
}
```

Then a developer just needs to compare two numbers to find out whether the order is ready.

This solution intuitively looks bad, and it really is. It violates all the aforementioned principles.

First, to solve the task “order a lungo” a developer needs to refer to the “recipe” entity and learn that every recipe has an associated volume. Then they need to embrace the concept that an order is ready at that particular moment when the prepared beverage volume becomes equal to the reference one. This concept is simply unguessable, and knowing it is mostly useless.

Second, we will have automatically got problems if we need to vary the beverage size. For example, if one day we decide to offer customers a choice of how many milliliters of lungo they desire exactly, then we have to perform one of the following tricks.

Option I: we have a list of possible volumes fixed and introduce bogus recipes like `/recipes/small-lungo` or `recipes/large-lungo`. Why “bogus”? Because it's still the same lungo recipe, same ingredients, same preparation steps, only volumes differ. We will have to start mass-

producing recipes, only different in volume, or introduce some recipe "inheritance" to be able to specify the "base" recipe and just redefine the volume.

Option II: we modify an interface, pronouncing volumes stated in recipes are just the default values. We allow requesting different cup volumes while placing an order:

```
POST /v1/orders
{
  "coffee_machine_id",
  "recipe": "lungo",
  "volume": "800ml"
}
```

For those orders with an arbitrary volume requested, a developer will need to obtain the requested volume, not from the `GET /v1/recipes` endpoint, but the `GET /v1/orders` one. Doing so we're getting a whole bunch of related problems:

- There is a significant chance that developers will make mistakes in this functionality implementation if they add arbitrary volume support in the code working with the `POST /v1/orders` handler, but forget to make corresponding changes in the order readiness check code.
- The same field (coffee volume) now means different things in different interfaces. In the context of the `GET /v1/recipes` endpoint, the `volume` field means "a volume to be prepared if no arbitrary volume is specified in the `POST /v1/orders` request"; and it cannot be renamed to "default volume" easily.

So we will get this:

```
GET /v1/orders/{id}
→
{
  …
  // this is a currently
  // prepared volume, bearing
  // the legacy name
  "volume": "80ml",
  // and this is the volume
  // requested by user
  "volume_requested": "800ml"
}
```

Third, the entire scheme becomes totally inoperable if different types of coffee machines produce different volumes of lungo. To introduce the "lungo volume depends on machine type" constraint we have to do quite a nasty thing: make recipes depend on coffee machine ids. By doing so we start actively "stir" abstraction levels: one part of our API (recipe endpoints) becomes unusable without explicit knowledge of another part (coffee machines listing). And what is even worse, developers will have to change the logic of their apps: previously it was possible to choose volume first, then a coffee machine; but now this step must be rebuilt from scratch.

Okay, we understood how to make things naughty. But how to make them *nice*?

Abstraction levels separation should go in three directions:

1. From user scenarios to their internal representation: high-level entities and their method nomenclatures must directly reflect the API usage scenarios; low-level entities reflect the decomposition of the scenarios into smaller parts.

2. From user to "raw" data subject field terms — in our case from high-level terms like "order," "recipe," and "café" to low-level terms like "beverage temperature," "coffee machine geographical coordinates," etc.

3. Finally, from data structures suitable for end users to “raw” data structures — in our case, from “lungo recipe” and “the "Chamomile" café chain” to the raw byte data stream from “Good Morning” coffee machine sensors.

The more the distance between programmable contexts our API connects, the deeper the hierarchy of the entities we are to develop.

In our example with coffee readiness detection, we clearly face the situation when we need an interim abstraction level:

- On one hand, an “order” should not store the data regarding coffee machine sensors
- On the other hand, a coffee machine should not store the data regarding order properties (and its API probably doesn't provide such functionality).

A naïve approach to this situation is to design an interim abstraction level as a “connecting link,” which reformulates tasks from one abstraction level into another. For example, introduce a `task` entity like that:

```
{
  …
  "volume_requested": "800ml",
  "volume_prepared": "200ml",
  "readiness_policy": "check_volume",
  "ready": false,
  "coffee_machine_id",
  "operation_state": {
    "status": "executing",
    "operations": [
      // description of commands
      // being executed on
      // a physical coffee machine
    ]
  }
  …
}
```

So an `order` entity will keep links to the recipe and the task, thus not dealing with other abstraction layers directly:

```
GET /v1/orders/{id}
→
{
  "recipe": "lungo",
  "task": {
    "id": <task id>
  }
}
```

We call this approach “naïve” not because it's wrong; on the contrary, that's quite a logical “default” solution if you don't know yet (or don't understand yet) how your API will look like. The problem with this approach lies in its speculativeness: it doesn't reflect the subject area's organization.

An experienced developer in this case must ask: what options do exist? how should we really determine the readiness of the beverage? If it turns out that comparing volumes *is* the only working method to tell whether the beverage is ready, then all the speculations above are wrong. You may safely include readiness-by-volume detection into your interfaces since no other methods exist. Before abstracting something we need to learn what exactly we're abstracting.

In our example let's assume that we have studied coffee machines' API specs, and learned that two device types exist:

- Coffee machines capable of executing programs coded in the firmware; the only customizable options are some beverage parameters, like the desired volume, a syrup flavor, and a kind of milk
- Coffee machines with built-in functions, like “grind specified coffee volume,” “shed the specified amount of water,” etc.; such coffee machines lack “preparation programs,” but provide access to commands and sensors.

To be more specific, let's assume those two kinds of coffee machines provide the following physical API.

- Coffee machines with pre-built programs:

```
// Returns the list of
// available programs
GET /programs
→
{
  // a program identifier
  "program": 1,
  // coffee type
  "type": "lungo"
}
```

```
// Starts an execution
// of the specified program
// and returns the execution status
POST /execute
{
  "program": 1,
  "volume": "200ml"
}
→
{
  // A unique identifier
  // of the execution
  "execution_id": "01-01",
  // An identifier of the program
  "program": 1,
  // The requested beverage volume
  "volume": "200ml"
}
```

```
// Cancels the current program
POST /cancel
```

```
// Returns the execution status.
// The response format is the same
// as in the `POST /execute` method
GET /execution/{id}/status
```

NB: This API violates a number of design principles, starting with a lack of versioning; it's described in such a manner because of two reasons: (1) to demonstrate how to design a more convenient API, (2) in the real life, you will really get something like that from vendors, and this API is actually quite a sane one.

- Coffee machines with built-in functions:

```
// Returns the list of
// available functions
GET /functions
→
{
  "functions": [
    {
      // One of the available
      // operation types:
      // * set_cup
      // * grind_coffee
      // * pour_water
      // * discard_cup
      "type": "set_cup",
      // Arguments for the
      // operation.
      // To keep it simple,
      // let's limit these to one:
      // * volume
      //     — a volume of a cup,
      //       coffee, or water
      "arguments": ["volume"]
    },
    …
  ]
}
```

```
// Takes arguments values
// and starts executing a function
POST /functions
{
  "type": "set_cup",
  "arguments": [{
    "name": "volume",
    "value": "300ml"
  }]
}
```

```
// Returns the state of the sensors
GET /sensors
→
{
  "sensors": [
    {
      // Possible values:
      // * cup_volume
      // * ground_coffee_volume
      // * cup_filled_volume
      "type": "cup_volume",
      "value": "200ml"
    },
    …
  ]
}
```

NB: The example is intentionally fictitious to model the situation described above: to determine beverage readiness you have to compare the requested volume with volume sensor readings.

Now the picture becomes more apparent: we need to abstract coffee machine API calls so that the “execution level” in our API provides general functions (like beverage readiness detection) in a unified form. We should also note that these two coffee machine API kinds belong to different abstraction levels themselves: the first one provides a higher-level API than the second one. Therefore, a “branch” of our API working with the second-kind machines will be deeper.

The next step in abstraction level separating is determining what functionality we're abstracting. To do so, we need to understand the tasks developers solve at the “order” level and learn what problems they face if our interim level is missing.

1. Obviously, the developers desire to create an order uniformly: list high-level order properties (beverage kind, volume, and special options like syrup or milk type), and don't think about how the specific coffee machine executes it.
2. Developers must be able to learn the execution state: is the order ready? If not, when can they expect it to be ready (and is there any sense to wait in case of execution errors)?

3. Developers need to address the order's location in space and time — to explain to users where and when they should pick the order up.
4. Finally, developers need to run atomic operations, like canceling orders.

Note, that the first-kind API is much closer to developers' needs than the second-kind API. An indivisible “program” is a way more convenient concept than working with raw commands and sensor data. There are only two problems we see in the first-kind API:

- Absence of explicit “programs” to “recipes” relation. A program identifier is of no use to developers since there is a “recipe” concept.
- Absence of an explicit “ready” status.

But with the second-kind API, it's much worse. The main problem we foresee is the absence of “memory” for actions being executed. The functions and sensors API is totally stateless, which means we don't even understand who called a function being currently executed, when, or to what order it relates.

So we need to introduce two abstraction levels.

1. Execution control level, which provides a uniform interface to indivisible programs. “Uniform interface” means here that, regardless of a coffee machine's kind, developers may expect:
    - Statuses and other high-level execution parameters nomenclature (for example, estimated preparation time or possible execution errors) being the same;
    - Methods nomenclature (for example, order cancellation method) and their behavior being the same.
2. Program runtime level. For the first-kind API, it will provide just a wrapper for existing programs API; for the second-kind API, the entire “runtime” concept is to be developed from scratch by us.

What does this mean in a practical sense? Developers will still be creating orders, dealing with high-level entities only:

```
POST /v1/orders
{
  "coffee_machin
  "recipe": "lungo",
  "volume": "800ml"
}
→
{ "order_id" }
```

The `POST /orders` handler checks all order parameters, puts a hold of the corresponding sum on the user's credit card, forms a request to run, and calls the execution level. First, a correct execution program needs to be fetched:

```
POST /v1/program-matcher
{ "recipe", "coffee-machine" }
→
{ "program_id" }
```

Now, after obtaining the correct `program` identifier, the handler runs the program:

```
POST /v1/programs/{id}/run
{
  "order_id",
  "coffee_machine_id",
  "parameters": [
    {
      "name": "volume",
      "value": "800ml"
    }
  ]
}
→
{ "program_run_id" }
```

Please note that knowing the coffee machine API kind isn't required at all; that's why we're making abstractions! We could possibly make the interfaces more specific by implementing different `run` and `match` endpoints for different coffee machines:

- `POST /v1/program-matcher/{api_type}`
- `POST /v1/{api_type}/programs/{id}/run`

This approach has some benefits, like the possibility to provide different sets of parameters, specific to the API kind. But we see no need for such fragmentation. The `run` method handler is capable of extracting all the program metadata and performing one of two actions:

- Call the `POST /execute` physical API method, passing the internal program identifier for the first API kind
- Initiate runtime creation to proceed with the second API kind.

Out of general considerations, the runtime level for the second-kind API will be private, so we are more or less free in implementing it. The easiest solution would be to develop a virtual state machine that creates a “runtime” (i.e., a stateful execution context) to run a program and control its state.

```
POST /v1/runtimes
{
  "coffee_machine",
  "program",
  "parameters"
}
→
{ "runtime_id", "state" }
```

The `program` here would look like that:

```
{
  "program_id",
  "api_type",
  "commands": [
    {
      "sequence_id",
      "type": "set_cup",
      "parameters"
    },
    …
  ]
}
```

And the `state` like that:

```
{
  // The `runtime` status:
  // * "pending" — awaiting execution
  // * "executing" — performing
  //      some command
  // * "ready_waiting" — the beverage
  //      is ready
  // * "finished" — all operations
  //      are done
  "status": "ready_waiting",
  // Command being currently executed.
  // Similar to line numbers
  // in computer programs
  "command_sequence_id",
  // How the execution concluded:
  // * "success"
  //     — the beverage prepared and taken
  // * "terminated"
  //     — the execution aborted
  // * "technical_error"
  //     — a preparation error
  // * "waiting_time_exceeded"
  //     — beverage prepared,
  //       but not taken;
  //       timed out then disposed
  "resolution": "success",
  // The values of all variables,
  // including the state of the sensors
  "variables"
}
```

NB: When implementing the `orders` → `match` → `run` → `runtimes` call sequence, we have two options:

- Either `POST /orders` handler requests the data regarding the recipe, the coffee machine model, and the program on its own, and forms a stateless request that contains all necessary data (API kind, command sequence, etc.)

- Or the request contains only data identifiers, and the next handler in the chain will request pieces of data it needs via some internal APIs.

Both variants are plausible and the selection between them depends on implementation details.

Abstraction Levels Isolation

A crucial quality of properly separated abstraction levels (and therefore a requirement to their design) is a level isolation restriction: **only adjacent levels may interact**. If "jumping over" is needed in the API design, then clearly mistakes were made.

Returning to our example, how would retrieving the order status work? To obtain a status the following call chain is to be performed:

- A user initiates a call to the `GET /v1/orders` method.
- The `orders` handler completes operations on its level of responsibility (e.g., checks user authorization), finds the `program_run_id` identifier and performs a call to the `runs/{program_run_id}` endpoint.
- The `runs` endpoint completes operations corresponding to its level (e.g., checks the coffee machine API kind) and, depending on the API kind, proceeds with one of two possible execution branches:
    - Either calls the `GET /execution/status` method of the physical coffee machine API, gets the coffee volume, and compares it to the reference value or
    - Invokes the `GET /v1/runtimes/{runtime_id}` method to obtain the `state.status` and converts it to the order status.
- In the case of the second-kind API, the call chain continues: the `GET /runtimes` handler invokes the `GET /sensors` method of the physical coffee machine API and performs some manipulations with the data, like comparing the cup / ground coffee / shed water volumes with the reference ones, and changing the state and the status if needed.

NB: The term “call chain” shouldn't be taken literally. Each abstraction level may be organized differently in a technical sense. For example:

- There might be explicit proxying of calls down the hierarchy
- There might be a cache at each level, which is updated upon receiving a callback call or an event. In particular, a low-level runtime execution cycle obviously must be independent of upper levels, which implies renewing its state in the background and not waiting for an explicit call.

Note what happens here: each abstraction level wields its own status (i.e., order, runtime, and sensors status respectively) formulated in subject area terms corresponding to this level. Forbidding “jumping over” results in the necessity to spawn statuses at each level independently.

Now let's examine how the order cancel operation flows through our abstraction levels. In this case, the call chain will look like this:

- A user initiates a call to the `POST /v1/orders/{id}/cancel` method.
- The method handler completes operations on its level of responsibility:
  - Checks the authorization
  - Resolves money issues (e.g., whether a refund is needed)
  - Finds the `program_run_id` identifier and calls the `runs/{program_run_id}/cancel` method.
- The `runs/cancel` handler completes operations on its level of responsibility and, depending on the coffee machine API kind, proceeds with one of two possible execution branches:
  - Calls the `POST /execution/cancel` method of a physical coffee machine API
  - Or invokes the `POST /v1/runtimes/{id}/terminate` method.
- In the second case, the call chain continues as the `terminate` handler operates its internal state:
  - Changes the `resolution` to `"terminated"`
  - Runs the `"discard_cup"` command.

Handling state-modifying operations like the `cancel` operation requires more advanced abstraction-level juggling skills compared to non-modifying calls like the `GET /status` method. There are two important moments to consider:

1. At each abstraction level the idea of “order canceling” is reformulated:
    - At the `orders` level, this action splits into several “cancels” of other levels: you need to cancel money holding and cancel order execution
    - At the second API kind, physical level the “cancel” operation itself doesn't exist; “cancel” means “executing the `discard_cup` command,” which is quite the same as any other command.

      The interim API level is needed to make this transition between different level “cancels” smooth and rational without jumping over canyons.
2. From a high-level point of view, canceling an order is a terminal action since no further operations are possible. From a low-level point of view, processing continues until the cup is discarded, and then the machine is to be unlocked (i.e., new runtimes creation allowed). It's an execution control level's task to couple those two states, outer (the order is canceled) and inner (the execution continues).

It might seem like forcing the abstraction levels isolation is redundant and makes interfaces more complicated. In fact, it is. It's essential to understand that flexibility, consistency, readability, and extensibility come with a price. One may construct an API with zero overhead, essentially just providing access to the coffee machine's microcontrollers. However using such an API would be a disaster for a developer, not to mention the inability to extend it.

Separating abstraction levels is first of all a logical procedure: how we explain to ourselves and developers what our API consists of. **The abstraction gap between entities exists objectively**, no matter what interfaces we design. Our task is just to sort this gap into levels *explicitly*. The more implicitly abstraction levels are separated (or worse — blended into each other), the more complicated your API's learning curve is, and the worse the code that uses it will be.

The Data Flow

One useful exercise that allows us to examine the entire abstraction hierarchy is to exclude all the particulars and construct a data flow chart, either on paper or in our head. This chart shows what data is flowing through your API entities, and how it's being altered at each step.

This exercise doesn't just help but also allows us design really large APIs with huge entity nomenclatures. Human memory isn't boundless; any project which grows extensively will eventually become too big to keep the entire entity hierarchy in mind. But it's usually possible to keep in mind the data flow chart, or at least keep a much larger portion of the hierarchy.

What data flow do we have in our coffee API?

1. It starts with the sensors data, e.g., volumes of coffee / water / cups. This is the lowest data level we have, and here we can't change anything.

2. A continuous sensors data stream is being transformed into discrete command execution statuses, injecting new concepts which don't exist within the subject area. A coffee machine API doesn't provide a “coffee is being poured” or a “cup is being set” notion. It's our software that treats incoming sensor data and introduces new terms: if the volume of coffee or water is less than the target one, then the process isn't over yet. If the target value is reached, then this synthetic status is to be switched, and the next command is executed. It is important to note that we don't calculate new variables out of sensor data: we need to create a new dataset first, a context, an “execution program” comprising a

sequence of steps and conditions, and fill it with initial values. If this context is missing, it's impossible to understand what's happening with the machine.

3. Having logical data about the program execution state, we can (again via creating a new high-level data context) merge two different data streams from two different kinds of APIs into a single stream, which provides in a unified form the data regarding executing a beverage preparation program with logical variables like the recipe, volume, and readiness status.

Each API abstraction level, therefore corresponds to some data flow generalization and enrichment, converting low-level (and in fact useless to end users) context terms into higher-level context terms.

We may also traverse the tree backward.

1. At the order level, we set its logical parameters: recipe, volume, execution place and possible status set.

2. At the execution level, we read the order-level data and create a lower-level execution context: the program as a sequence of steps, their parameters, transition rules, and initial state.

3. At the runtime level, we read the target parameters (which operation to execute, and what the target volume is) and translate them into coffee machine API microcommands and statuses for each command.

Also, if we take a deeper look at the “bad” decision (forcing developers to determine the actual order status on their own), being discussed at the beginning of this chapter, we could notice a data flow collision there:

- On one hand, in the order context “leaked” physical data (beverage volume prepared) is injected, stirring abstraction levels irreversibly

- On the other hand, the order context itself is deficient: it doesn't provide new meta-variables non-existent at the lower levels (the order status, in particular), doesn't initialize them, and doesn't set the game rules.

We will discuss data contexts in more detail in Section II. Here we will just state that data flows and their transformations might be and must be examined as a specific API facet, which helps us separate abstraction levels properly and check if our theoretical concepts work as intended.

Chapter 12. Isolating Responsibility Areas

In the previous chapter, we concluded that the hierarchy of abstractions in our hypothetical project would comprise:

- The user level (the entities formulated in terms understandable by users and acted upon by them: orders, coffee recipes)
- The program execution control level (the entities responsible for transforming orders into machine commands)
- The runtime level for the second API kind (the entities describing the command execution state machine).

We are now to define each entity's responsibility area: what's the reasoning for keeping this entity within our API boundaries? What operations are applicable to the entity directly (and which are delegated to other objects)? In fact, we are to apply the "why"-principle to every single API entity.

To do so, we must iterate all over the API and formulate in subject area terms what every object is. Let us remind that the abstraction levels concept implies that each level is some interim subject area per se; a step we take in the journey from describing a task in terms belonging to the first connected context ("a lungo ordered by a user") to terms belonging to the second connected context ("a command performed by a coffee machine").

As for our fictional example, it would look as follows.

1. User-level entities.
    - An `order` describes some logical unit in app-user interaction. An `order` might be:
        - Created
        - Checked for its status
        - Retrieved
        - Canceled.
    - A `recipe` describes an "ideal model" of a coffee beverage type, i.e., its customer properties. A `recipe` is an immutable entity that can only be read.

    - A `coffee-machine` is a model of a real-world device. We must be able to retrieve the coffee machine's geographical location and the options it supports from this model (which will be discussed below).
2. Program execution control-level entities.
    - A `program` describes a general execution plan for a coffee machine. Programs can only be read.
    - The `programs/matcher` entity couples a `recipe` and a `program`, which in fact means retrieving a dataset needed to prepare a specific recipe on a specific coffee machine.
    - The `programs/run` entity describes a single fact of running a program on a coffee machine. A `run` might be:
        - Initialized (created)
        - Checked for its status
        - Canceled.
3. Runtime-level entities.
    - A `runtime` describes a specific execution data context, i.e., the state of each variable. A `runtime` can be:
        - Initialized (created)
        - Checked for its status
        - Terminated.

If we look closely at the entities, we may notice that each entity turns out to be a composite. For example, a `program` operates high-level data (`recipe` and `coffee-machine`), enhancing them with its subject area terms (`program_run_id` for instance). This is totally fine as connecting contexts is what APIs do.

Use Case Scenarios

At this point, when our API is in general clearly outlined and drafted, we must put ourselves in the developer's shoes and try writing code. Our task is to look at the entity nomenclature and make some guesses regarding their future usage.

So, let us imagine we've got a task to write an app for ordering coffee based on our API. What code would we write?

Obviously, the first step is to offer a choice to the user, to make them point out what they want. And this very first step reveals that our API is quite inconvenient. There are no methods allowing for choosing something. Developers have to implement these steps:

- Retrieve all possible recipes from the `GET /v1/recipes` endpoint
- Retrieve a list of all available coffee machines from the `GET /v1/coffee-machines` endpoint
- Write code that traverses all this data.

If we try writing pseudocode, we will get something like this:

```
// Retrieve all possible recipes
let recipes =
  api.getRecipes();
// Retrieve a list of
// all available coffee machines
let coffeeMachines =
  api.getCoffeeMachines();
// Build a spatial index
let coffeeMachineRecipesIndex =
  buildGeoIndex(
    recipes,
    coffeeMachines
  );
// Select coffee machines
// matching user's needs
let matchingCoffeeMachines =
  coffeeMachineRecipesIndex.query(
    parameters,
    { "sort_by": "distance" }
  );
// Finally, show offers to the user
app.display(matchingCoffeeMachines);
```

As you see, developers are to write a lot of redundant code (to say nothing about the complexity of implementing spatial indexes). Besides, if we take into consideration our Napoleonic plans to cover all coffee machines in the world with our API, then we need to admit that this

algorithm is just a waste of computational resources on retrieving lists and indexing them.

The necessity of adding a new endpoint for searching becomes obvious. To design such an interface we must imagine ourselves being UX designers, and think about how an app could try to arouse users' interest. Two scenarios are evident:

- Display all cafes in the vicinity and the types of coffee they offer (a “service discovery” scenario) — for new users or just users with no specific preferences
- Display nearby cafes where a user could order a particular type of coffee — for users seeking a certain beverage type.

Then our new interface would look like this:

```
POST /v1/offers/search
{
  // optional
  "recipes": ["lungo", "americano"],
  "position": <geographical coordinates>,
  "sort_by": [
    { "field": "distance" }
  ],
  "limit": 10
}
→
{
  "results": [
    {
      "coffee_machine",
      "place",
      "distance",
      "offer"
    }
  ],
  "cursor"
}
```

Here:

- An `offer` is a marketing bid: on what conditions a user could have the requested coffee beverage (if specified in the request), or some kind of marketing offer — prices for the most popular or interesting products (if no specific preference was set).
- A `place` is a spot (café, restaurant, street vending machine) where the coffee machine is located. We never introduced this entity before, but it's quite obvious that users need more convenient guidance to find a proper coffee machine than just geographical coordinates.

NB: We could have enriched the existing `/coffee-machines` endpoint instead of adding a new one. Although this decision looks less semantically viable, coupling different modes of listing entities in one interface, by relevance and by order, is usually a bad idea because these two types of rankings imply different features and usage scenarios. Furthermore, enriching the search with "offers" pulls this functionality out of the `coffee-machines` namespace: the fact of getting offers to prepare specific beverages in specific conditions is a key feature for users, with specifying the coffee machine being just a part of an offer. In reality, users rarely care about coffee machine models.

NB: Having the `coffee_machine_id` in the interface is to some extent violating the abstraction separation principle. It should be organized in a more complex way: coffee shops should somehow map incoming orders against available coffee machines, and only the type of the coffee machine (if a coffee shop really operates several of them) is something meaningful in the context of order creation. However, we deliberately simplified our study by making a coffee machine selectable in the API to keep our API example readable.

Coming back to the code developers write, it would now look like that:

```
// Searching for offers
// matching a user's intent
let offers = api.search(parameters);
// Display them to a user
app.display(offers);
```

Helpers

Methods similar to the newly invented `offers/search` one are called helpers. The purpose of their existence is to generalize known API usage scenarios and facilitate their implementation. By “facilitating,” we mean not only reducing wordiness (getting rid of “boilerplates”) but also helping developers avoid common problems and mistakes.

For instance, let's consider the problem of the monetary value of an order. Our search function returns some “offers” with prices. However, the price is volatile; coffee could cost less during “happy hours,” for example. Developers could make a mistake three times while implementing this functionality:

- Cache search results on a client device for too long (as a result, the price will always be outdated).
- Contrary to the previous point, call the search endpoint excessively just to actualize prices, thus overloading the network and the API servers.
- Create an order with an invalid price (thereby deceiving a user, displaying one sum, and debiting another).

To solve the third problem we could demand that the displayed price be included in the order creation request and return an error if it differs from the actual one. (In fact, any API working with money *must* do so.) However, this solution does not help with the first two problems, and also deteriorates the user experience. Displaying the actual price is always a much more convenient behavior than displaying errors upon pressing the “place an order” button.

One solution is to provide a special identifier to an offer. This identifier must be specified in an order creation request:

```
{
  "results": [
    {
      "coffee_machine",
      "place",
      "distance",
      "offer": {
        "id",
        "price",
        "currency_code",
        // Date and time
        // when the offer expires
        "valid_until"
      }
    }
  ],
  "cursor"
}
```

By doing so we're not only helping developers grasp the concept of getting the relevant price but also solving a UX task of informing users about “happy hours.”

As an alternative, we could split the endpoints: one for searching, and one for obtaining offers. The second endpoint would only be needed to actualize prices if necessary.

Error Handling

And one more step towards making developers' lives easier: what would an “invalid price” error look like?

```
POST /v1/orders
{ "offer_id", … }
→ 409 Conflict
{ "message": "Invalid price" }
```

Formally speaking, this error response is sufficient: users get the “Invalid price” message, and they have to repeat the order. But from a UX point of view, this would be a terrible decision: the user hasn't made any mistakes, and this message isn't helpful at all.

The main rule of error interfaces in APIs is that an error response must help a client understand *what to do with the error*. An error response's content must address the following questions:

1. Which party is the source of the problem: the client or the server?
   For example, HTTP APIs traditionally employ the `4xx` status codes to indicate client problems and `5xx` to indicate server problems (with the exception of the `404` code, which is an uncertainty status).

2. If the error is caused by the server, is there any sense in repeating the request? If yes, then when?

3. If the error is caused by the client, is it resolvable or not?
   For example, the invalid price error is resolvable: a client could obtain a new price offer and create a new order with it. But if the error occurred because of a mistake in the client code, then eliminating the cause is impossible, and there is no need to make the user press the “place an order” button again: this request will never succeed.

   NB: Here and throughout we indicate resolvable problems with the `409 Conflict` code and unresolvable ones with the `400 Bad Request` code.

4. If the error is resolvable then what kind of problem is it? Obviously, application engineers couldn't resolve a problem they are unaware of. For every resolvable problem, developers must *write some code* (re-obtaining the offer in our case), so there must be a list of possible error reasons and the corresponding fields in the error response to tell one problem from another.

5. If passing invalid values in different parameters arises the same kind of error, then how to learn which parameter value is wrong exactly?

6. Finally, if some parameter value is unacceptable, then what values are acceptable?

In our case, the price mismatch error should look like this:

```
409 Conflict
{
  // Error kind
  "reason": "offer_invalid",
  "localized_message":
    "Something went wrong.↵
     Try restarting the app."
  "details": {
    // What's wrong exactly?
    // Which validity checks failed?
    "checks_failed": [
      "offer_lifetime"
    ]
  }
}
```

After receiving this error, a client should check the error's kind (“some problem with the offer”) and the specific error reason (“order lifetime expired”), and send the offer retrieval request again. If the `checks_failed` field indicated a different error reason (for example, the offer isn't bound to the specified user), client actions would be different (re-authorize the user, then get a new offer). If there was no error

handler for this specific reason, a client should show the `localized_message` to the user and invoke the standard error recovery procedure.

It is also worth mentioning that unresolvable errors are useless to a user at the time of the error occurrence (since the client couldn't react meaningfully to unknown errors). Still, providing extended error data is not excessive as a developer will read it while fixing the issue in their code.

Decomposing Interfaces. The “7±2” Rule

From our own API development experience, we can tell without a doubt that the greatest final interface design mistake (and the greatest developer's pain accordingly) is the excessive overloading of entities' interfaces with fields, methods, events, parameters, and other attributes.

Meanwhile, there is the “Golden Rule” of interface design (applicable not only to APIs but almost to anything): humans can comfortably keep 7±2 entities in short-term memory. Manipulating a larger number of chunks complicates things for most humans. The rule is also known as Miller's Law.

The only possible method of overcoming this law is decomposition. Entities should be grouped under a single designation at every concept level of the API so that developers never have to operate on more than a reasonable amount of entities (let's say, ten) at a time.

Let's take a look at the coffee machine search function response in our API. To ensure an adequate UX of the app, quite bulky datasets are required:

```
{
  "results": [{
    // Coffee machine data
    "coffee_machine_id",
    "coffee_machine_type",
    "coffee_machine_brand",
    // Place data
    "place_name": "The Chamomile",
    "place_location_latitude",
    "place_location_longitude",
    "place_open_now",
    "working_hours",
    // Walking route parameters
    "walking_distance",
    "walking_time",
    // How to find the place
    "location_tip",
    // Offers
    "offers": [{
      // Recipe data
      "recipe",
      "recipe_name",
      "recipe_description",
      // Order parameters
      "volume",
      // Offer data
      "offer_id",
      "offer_valid_until",
      "price": "19.00",
      "localized_price":
        "Just $19 for a large coffee cup",
      "currency_code",
      "estimated_waiting_time"
    }, …]
  }, …]
}
```

This approach is regretfully quite common and could be found in almost every API. Fields are mixed into one single list and often prefixed to indicate the related ones.

In this situation, we need to split this structure into data domains by grouping fields that are logically related to a single subject area. In our case, we may identify at least 7 data clusters:

- Data regarding the place where the coffee machine is located
- Properties of the coffee machine itself
- Route data
- Recipe data
- Order options
- Offer data
- Pricing data.

Let's group them together:

```
{
  "results": [{
    // Place data
    "place": { "name", "location" },
    // Coffee machine properties
    "coffee-machine": {
      "id", "brand", "type"
    },
    // Route data
    "route": {
      "distance",
      "duration",
      "location_tip"
    },
    "offers": [{
      // Recipe data
      "recipe": {
        "id",
        "name",
        "description"
      },
      // Order options
      "options":
        { "volume" },
      // Offer metadata
      "offer":
        { "id", "valid_until" },
      // Pricing
      "pricing": {
        "currency_code",
        "price",
        "localized_price"
      },
      "estimated_waiting_time"
    }, …]
  }, …]
}
```

Such a decomposed API is much easier to read than a long list of different attributes. Furthermore, it's probably better to group even more entities in advance. For example, a `place` and a `route` could be nested fields under a synthetic `location` property, or `offer` and `pricing` fields might be combined into some generalized object.

It is important to say that readability is achieved not only by merely grouping the entities. Decomposing must be performed in such a manner that a developer, while reading the interface, instantly understands, “Here is the place description of no interest to me right now, no need to traverse deeper.” If the data fields needed to complete some action are scattered all over different composites, the readability doesn't improve and even degrades.

Proper decomposition also helps with extending and evolving an API. We'll discuss the subject in Section III.

Chapter 13. Describing Final Interfaces

When all entities, their responsibilities, and their relations to each other are defined, we proceed to the development of the API itself. We need to describe the objects, fields, methods, and functions nomenclature in detail. In this chapter, we provide practical advice on making APIs usable and understandable.

One of the most important tasks for an API developer is to ensure that code written by other developers using the API is easily readable and maintainable. Remember that the law of large numbers always works against you: if a concept or call signature can be misunderstood, it will be misunderstood by an increasing number of partners as the API's popularity grows.

NB: The examples in this chapter are meant to illustrate the consistency and readability problems that arise during API development. We do not provide specific advice on designing REST APIs (such advice will be given in the corresponding section of this book) or programming languages' standard libraries. The focus is o the idea, not specific syntax.

An important assertion number one:

1. Rules Must Not Be Applied Unthinkingly

Rules are simply formulated generalizations based on one's experience. They are not to be applied unconditionally, and they do not make thinking redundant. Every rule has a rational reason to exist. If your situation does not justify following a rule, then you should not do it.

This idea applies to every concept listed below. If you end up with an unusable, bulky, or non-obvious API because you followed the rules, it's a motivation to revise the rules (or the API).

It is important to understand that you can always introduce your own concepts. For example, some frameworks intentionally reject paired `set_entity` / `get_entity` methods in favor of a single `entity()` method with an optional argument. The crucial part is being systematic in

applying the concept. If it is implemented, you must apply it to every single API method or at the very least develop a naming rule to distinguish such polymorphic methods from regular ones.

2. Explicit Is Always Better Than Implicit

The entity name should explicitly indicate what the entity does and what side effects to expect when using it.

Bad:

```
// Cancels an order
order.canceled = true;
```

It is not obvious that a state field might be modified, and that this operation will cancel the order.

Better:

```
// Cancels an order
order.cancel();
```

Bad:

```
// Returns aggregated statistics
// since the beginning of time
orders.getStats()
```

Even if the operation is non-modifying but computationally expensive, you should explicitly indicate that, especially if clients are charged for computational resource usage. Furthermore, default values should not be set in a way that leads to maximum resource consumption.

Better:

```
// Calculates and returns
// aggregated statistics
// for a specified period of time
orders.calculateAggregatedStats({
  begin_date,
  end_date
});
```

Try to design function signatures that are transparent about what the function does, what arguments it takes, and what the outcome is. When reading code that works with your API, it should be easy to understand what it does without referring to the documentation.

Two important implications:

1.1. If the operation is modifying, it must be obvious from the signature. In particular, there should not be modifying operations named `getSomething` or using the `GET` HTTP verb.

1.2. If your API's nomenclature contains both synchronous and asynchronous operations, then (a)synchronicity must be apparent from signatures, **or** a naming convention must exist.

3. Specify Which Standards Are Used

Regrettably, humanity is unable to agree on even the most trivial things, like which day starts the week, let alone more sophisticated standards.

Therefore, *always* specify exactly which standard is being used. Exceptions are possible if you are 100% sure that only one standard for this entity exists in the world and every person on Earth is totally aware of it.

Bad: `"date": "11/12/2020"` — there are numerous date formatting standards. It is unclear which number represents the day and which number represents the month.

Better: `"iso_date": "2020-11-12"`.

Bad: `"duration": 5000` — five thousand of what?

Better:
`"duration_ms": 5000`
or
`"duration": "5000ms"`
or
`"iso_duration": "PT5S"`
or
`"duration": {"unit": "ms", "value": 5000}`.

One particular implication of this rule is that money sums must *always* be accompanied by a currency code.

It is also worth mentioning that in some areas the situation with standards is so spoiled that no matter what you do, someone will be upset. A "classical" example is the order of geographical coordinates (latitude-longitude vs longitude-latitude). Unfortunately, the only effective method to address the frustration in such cases is the Serenity Notepad which will be discussed in the corresponding chapter.

4. Entities Must Have Concrete Names

Avoid using single amoeba-like words, such as "get," "apply," "make," etc.

Bad: `user.get()` — it is difficult to guess what is actually returned.

Better: `user.get_id()`.

5. Don't Spare the Letters

In the 21st century, there's no need to shorten entities' names.

Bad: `order.getTime()` — it is unclear what time is actually returned: order creation time, order preparation time, order waiting time, or something else.

Better: `order.getEstimatedDeliveryTime()`.

Bad:

```
// Returns a pointer to the first occurrence
// in str1 of any of the characters
// that are part of str2
strpbrk (str1, str2)
```

Possibly, the author of this API thought that the abbreviation `pbrk` would mean something to readers, but that is clearly mistaken. It is also hard to understand from the signature which string (`str1` or `str2`) represents a character set.

Better:

```
str_search_for_characters(
  str,
  lookup_character_set
)
```

— though it is highly debatable whether this function should exist at all; a feature-rich search function would be much more convenient. Also, shortening a `string` to `str` bears no practical sense, unfortunately being a common practice in many subject areas.

NB: Sometimes field names are shortened or even omitted (e.g., a heterogeneous array is passed instead of a set of named fields) to reduce the amount of traffic. In most cases, this is absolutely meaningless as the data is usually compressed at the protocol level.

6. Naming Implies Typing

A field named `recipe` must be of type `Recipe`. A field named `recipe_id` must contain a recipe identifier that can be found within the `Recipe` entity.

The same applies to basic types. Arrays must be named in the plural form or as collective nouns, e.g., `objects`, `children`. If it is not possible, it is better to add a prefix or a postfix to avoid ambiguity.

Bad: `GET /news` — it is unclear whether a specific news item is returned, or a list of them.

Better: `GET /news-list`.

Similarly, if a Boolean value is expected, entity naming must describe a qualitative state, e.g., `is_ready`, `open_now`.

Bad: `"task.status": true`
— statuses are not explicitly binary. Additionally, such an API is not extendable.

Better: `"task.is_finished": true`.

Specific platforms imply specific additions to this rule depending on the first-class citizen types they provide. For example, JSON doesn't have a `Date` object type, so dates are typically passed as numbers or strings. In this case, it's convenient to mark dates somehow, for example, by adding `_at` or `_date` postfixes, i.e. `created_at`, `occurred_at`.

If an entity name is a polysemantic term itself, which could confuse developers, it is better to add an extra prefix or postfix to avoid misunderstanding.

Bad:

```
// Returns a list of
// coffee machine builtin functions
GET /coffee-machines/{id}/functions
```

The word “function” is ambiguous. It might refer to built-in functions, but it could also mean “a piece of code,” or a state (machine is functioning).

Better:

```
GET /v1/coffee-machines/{id}↵
  /builtin-functions-list
```

7. Matching Entities Must Have Matching Names and Behave Alike

Bad: `begin_transition` / `stop_transition`
— The terms `begin` and `stop` don't match; developers will have to refer to the documentation to find a paired method.

Better: either `begin_transition` / `end_transition` or `start_transition` / `stop_transition`.

Bad:

```
// Find the position of the first occurrence
// of a substring in a string
strpos(haystack, needle)
// Replace all occurrences
// of the search string
// with the replacement string
str_replace(needle, replace, haystack)
```

Several rules are violated:

- The usage of an underscore is not consistent
- Functionally close methods have different `needle/haystack` argument ordering
- The first function finds the first occurrence while the second one finds all occurrences, and there is no way to deduce that fact from the function signatures.

Improving these function signatures is left as an exercise for the reader.

8. Avoid Double Negations

Bad: `"dont_call_me": false`
— humans are bad at perceiving double negation and can make mistakes.

Better: `"prohibit_calling": true` or `"avoid_calling": true`
— this is easier to read. However, you should not deceive yourself: it is still a double negation, even if you've found a "negative" word without a "negative" prefix.

It is also worth mentioning that mistakes in using De Morgan's laws are even more common. For example, if you have two flags:

```
GET /coffee-machines/{id}/stocks
→
{
  "has_beans": true,
  "has_cup": true
}
```

The condition "coffee might be prepared" would look like `has_beans && has_cup` — both flags must be true. However, if you provide the negations of both flags:

```
{
  "beans_absence": false,
  "cup_absence": false
}
```

— then developers will have to evaluate the `!beans_absence && !cup_absence` flag which is equivalent to the `!(beans_absence || cup_absence)` condition. In this transition, people tend to make mistakes. Avoiding double negations helps to some extent, but the best advice is to avoid situations where developers have to evaluate such flags.

9. Avoid Implicit Type Casting

This advice contradicts the previous one, ironically. When developing APIs you frequently need to add a new optional field with a non-empty default value. For example:

```
const orderParams = {
  contactless_delivery: false
};
const order = api.createOrder(
  orderParams
);
```

This new `contactless_delivery` option isn't required, but its default value is `true`. A question arises: how should developers discern the explicit intention to disable the option (`false`) from not knowing if it exists (the field isn't set)? They would have to write something like:

```
if (
  Type(
    orderParams.contactless_delivery
  ) == 'Boolean' &&
  orderParams
    .contactless_delivery == false) {
  …
}
```

This practice makes the code more complicated, and it's quite easy to make mistakes resulting in effectively treating the field as the opposite. The same can happen if special values (e.g., `null` or `-1`) are used to denote value absence.

If the protocol does not support resetting to default values as a first-class citizen, the universal rule is to make all new Boolean flags false by default.

Better

```
const orderParams = {
  force_contact_delivery: true
};
const order = api.createOrder(
  orderParams
);
```

If a non-Boolean field with a specially treated absence of value is to be introduced, then introduce two fields.

Bad:

```
// Creates a user
POST /v1/users
{ … }
→
// Users are created with a monthly
// spending limit set by default
{
  "spending_monthly_limit_usd": "100",
  …
}
// To cancel the limit null value is used
PUT /v1/users/{id}
{
  "spending_monthly_limit_usd": null,
  …
}
```

Better

```
POST /v1/users
{
  // true — user explicitly cancels
  //   monthly spending limit
  // false — limit isn't canceled
  //   (default value)
  "abolish_spending_limit": false,
  // Non-required field
  // Only present if the previous flag
  // is set to false
  "spending_monthly_limit_usd": "100",
  …
}
```

NB: The contradiction with the previous rule lies in the necessity of introducing “negative” flags (the “no limit” flag), which we had to rename to `abolish_spending_limit`. Though it's a decent name for a negative flag, its semantics is still not obvious, and developers will have

to read the documentation. This is the way.

10. Declare Technical Restrictions Explicitly

Every field in your API comes with restrictions: the maximum allowed text length, the size of attached documents, the allowed ranges for numeric values, etc. Often, describing those limits is neglected by API developers — either because they consider it obvious, or because they simply don't know the boundaries themselves. This is of course an antipattern: not knowing the limits automatically implies that partners' code might stop working at any moment due to reasons they don't control.

Therefore, first, declare the boundaries for every field in the API without any exceptions, and, second, generate proper machine-readable errors describing the exact boundary that was violated should such a violation occur.

The same reasoning applies to quotas as well: partners must have access to the statistics on which part of the quota they have already used, and the errors in the case of exceeding quotas must be informative.

11. All Requests Must Be Limited

The restrictions should apply not only to field sizes but also to list sizes or aggregation intervals.

Bad: `getOrders()` — what if a user made a million orders?

Better: `getOrders({ limit, parameters })` — there must be a cap on the amount of processed and returned data. This also implies providing the possibility to refine the query if a partner needs more data than what is allowed to be returned in one request.

12. Describe the Retry Policy

One of the most significant performance-related challenges that nearly any API developer encounters, regardless of whether the API is internal or public, is service denial due to a flood of re-requests. Temporary backend API issues, such as increased response times, can lead to complete server failure if clients rapidly repeat requests after receiving an error or a timeout, resulting in generating a significantly larger workload than usual in a short period of time.

The best practice in such a situation is to require clients to retry API endpoints with increasing intervals (for example, the first retry occurs after one second, the second after two seconds, the third after four seconds, and so on, up to a maximum of, let's say, one minute). Of course, in the case of a public API, no one is obliged to comply with such a requirement, but its presence certainly won't make things worse for you. At the very least, some partners will read the documentation and follow your recommendations.

Moreover, you can develop a reference implementation of the retry policy in your public SDKs and ensure it is correctly implemented in open-source modules for your API.

13. Count the Amount of Traffic

Nowadays the amount of traffic is rarely taken into account as the Internet connection is considered unlimited almost universally. However, it is not entirely unlimited: with some degree of carelessness, it's always possible to design a system that generates an uncomfortable amount of traffic even for modern networks.

There are three obvious reasons for inflating network traffic:

- Clients query the data too frequently or cache it too little
- No data pagination is provided
- No limits are set on the data fields, or too large binary data (graphics, audio, video, etc.) is transmitted.

All these problems must be addressed by setting limitations on field sizes and properly decomposing endpoints. If an entity comprises both "lightweight" data (such as the name and description of a recipe) and "heavy" data (such as the promotional picture of a beverage which might easily be a hundred times larger than the text fields), it's better to split endpoints and pass only a reference to the "heavy" data (e.g., a link to the image). This will also allow for setting different cache policies for different kinds of data.

As a useful exercise, try modeling the typical lifecycle of a partner's app's main functionality (e.g., making a single order) to count the number of requests and the amount of traffic it requires. It might turn out that the high number of requests or increased network traffic consumption is due to a mistake in the design of state change notification endpoints. We will discuss this issue in detail in the "Bidirectional Data Flow" chapter of "The API Patterns" section of this book.

14. No Results Is a Result

If a server processes a request correctly and no exceptional situation occurs, there should be no error. Unfortunately, the antipattern of throwing errors when no results are found is widespread.

Bad

```
POST /v1/coffee-machines/search
{
  "query": "lungo",
  "location": <customer's location>
}
→ 404 Not Found
{
  "localized_message":
    "No one makes lungo nearby"
}
```

The response implies that a client made a mistake. However, in this case, neither the customer nor the developer made any mistakes. The client cannot know beforehand whether lungo is served in this location.

Better:

```
POST /v1/coffee-machines/search
{
  "query": "lungo",
  "location": <customer's location>
}
→ 200 OK
{
  "results": []
}
```

This rule can be summarized as follows: if an array is the result of the operation, then the emptiness of that array is not a mistake, but a correct response. (Of course, this applies if an empty array is semantically acceptable; an empty array of coordinates, for example, would be a mistake.)

NB: This pattern should also be applied in the opposite case. If an array of entities is an optional parameter in the request, the empty array and the absence of the field must be treated differently. Let's consider the example:

```
// Finds all coffee recipes
// that contain no milk
POST /v1/recipes/search
{
  "filter": {
    "no_milk": true
  }
}
→ 200 OK
{
  "results": [{
    "recipe": "espresso"
    …
  }, {
    "recipe": "lungo",
    …
  }]
}
// Finds offers for
// the given recipes
POST /v1/offers/search
{
  "location",
  "recipes": [
    "espresso",
    "lungo"
  ]
}
```

Now let's imagine that the first request returned an empty array of results meaning there are no known recipes that satisfy the condition. Ideally, the developer would have expected this situation and installed a guard to prevent the call to the offer search function in this case. However, we can't be 100% sure they did. If this logic is missing, the application will make the following call:

```
POST /v1/offers/search
{
  "location",
  "recipes": []
}
```

Often, the endpoint implementation ignores the empty recipe array and returns a list of offers as if no recipe filter was supplied. In our case, it means that the application seemingly ignores the user's request to show only milk-free beverages, which we consider unacceptable behavior. Therefore, the response to such a request with an empty array parameter should either be an error or an empty result.

15. Validate Inputs

The decision of whether to use an exception or an empty response in the previous example depends directly on what is stated in the contract. If the specification specifies that the `recipes` parameter must not be empty, an error should be generated (otherwise, you would violate your own spec).

This rule applies not only to empty arrays but to every restriction specified in the contract. “Silently” fixing invalid values rarely makes practical sense.

Bad:

```
POST /v1/offers/search
{
  "location": {
    "longitude": 20,
    "latitude": 100
  }
}
→ 200 OK
{
  // Offers for the
  // [0, 90] point
  "offers"
}
```

As we can see, the developer somehow passed the wrong latitude value (100 degrees). Yes, we can "fix" it by reducing it to the closest valid value, which is 90 degrees, but who benefits from this? The developer will never learn about this mistake, and we doubt that coffee offers in the Northern Pole vicinity are relevant to users.

Better:

```
POST /v1/coffee-machines/search
{
  "location": {
    "longitude": 20,
    "latitude": 100
  }
}
→ 400 Bad Request
{
  // Error description
}
```

It is also useful to proactively notify partners about behavior that appears to be a mistake:

```
POST /v1/coffee-machines/search
{
  "location": {
    "latitude": 0,
    "longitude": 0
  }
}
→
{
  "results": [],
  "warnings": [{
    "type": "suspicious_coordinates",
    "message": "Location [0, 0]↵
      is probably a mistake"
  }, {
    "type": "unknown_field",
    "message": "unknown field:↵
      `force_convact_delivery`. Did you↵
      mean `force_contact_delivery`?"
  }]
}
```

If it is not possible to add such notices, we can introduce a debug mode or strict mode in which notices are escalated:

```
POST /v1/coffee-machines/search↵
  strict_mode=true
{
  "location": {
    "latitude": 0,
    "longitude": 0
  }
}
→ 404 Bad Request
{
  "errors": [{
    "type": "suspicious_coordinates",
    "message": "Location [0, 0]↵
      is probably a mistake"
  }],
  …
}
```

If the [0, 0] coordinates are not an error, it makes sense to allow for manual bypassing of specific errors:

```
POST /v1/coffee-machines/search↵
  strict_mode=true↵
  disable_errors=suspicious_coordinates
```

16. Default Values Must Make Sense

Setting default values is one of the most powerful tools that help avoid verbosity when working with APIs. However, these values should help developers rather than hide their mistakes.

Bad:

```
POST /v1/coffee-machines/search
{
  "recipes": ["lungo"]
  // User location is not set
}
→
{
  "results": [
    // Results for some default
    // location
  ]
}
```

Formally speaking, having such behavior is feasible: why not have a “default geographical coordinates” concept? However, in reality, such policies of “silently” fixing mistakes lead to absurd situations like “the null island” — the most visited place in the world. The more popular an API becomes, the higher the chances that partners will overlook these edge cases.

Better:

```
POST /v1/coffee-machines/search
{
  "recipes": ["lungo"]
  // User location is not set
}
→ 400 Bad Request
{
  // Error description
}
```

17. Errors Must Be Informative

It is not enough to simply validate inputs; providing proper descriptions of errors is also essential. When developers write code, they encounter problems, sometimes quite trivial, such as invalid parameter types or boundary violations. The more convenient the error responses returned by your API, the less time developers will waste struggling with them, and the more comfortable working with the API will be for them.

Bad:

```
POST /v1/coffee-machines/search
{
  "recipes": ["lngo"],
  "position": {
    "latitude": 110,
    "longitude": 55
  }
}
→ 400 Bad Request
{}
```

— of course, the mistakes (typo in `"lngo"`, wrong coordinates) are obvious. But the handler checks them anyway, so why not return readable descriptions?

Better:

```
{
  "reason": "wrong_parameter_value",
  "localized_message":
    "Something is wrong.↵
     Contact the developer of the app.",
  "details": {
    "checks_failed": [
      {
        "field": "recipe",
        "error_type": "wrong_value",
        "message":
          "Unknown value: 'lngo'.↵
           Did you mean 'lungo'?"
      },
      {
        "field": "position.latitude",
        "error_type":
          "constraint_violation",
        "constraints": {
          "min": -90,
          "max": 90
        },
        "message":
          "'position.latitude' value↵
            must fall within↵
            the [-90, 90] interval"
      }
    ]
  }
}
```

It is also a good practice to return all detectable errors at once to save developers time.

18. Return Unresolvable Errors First

```
POST /v1/orders
{
  "recipe": "lngo",
  "offer"
}
→ 409 Conflict
{
  "reason": "offer_expired"
}
// Request repeats
// with the renewed offer
POST /v1/orders
{
  "recipe": "lngo",
  "offer"
}
→ 400 Bad Request
{
  "reason": "recipe_unknown"
}
```

— what was the point of renewing the offer if the order cannot be created anyway? For the user, it will look like meaningless efforts (or meaningless waiting) that will ultimately result in an error regardless of what they do. Yes, maintaining error priorities won't change the result — the order still cannot be created. However, first, users will spend less time (also make fewer mistakes and contribute less to the error metrics) and second, diagnostic logs for the problem will be much easier to read.

19. Prioritize Significant Errors

If the errors under consideration are resolvable (i.e., the user can take some actions and still get what they need), you should first notify them of those errors that will require more significant state updates.

Bad:

```
POST /v1/orders
{
  "items": [{
    "item_id": "123",
    "price": "0.10"
  }]
}
→
409 Conflict
{
  // Error: while the user
  // was making an order,
  // the product price has changed
  "reason": "price_changed",
  "details": [{
    "item_id": "123",
    "actual_price": "0.20"
  }]
}
// Repeat the request
// to get the actual price
POST /v1/orders
{
  "items": [{
    "item_id": "123",
    "price": "0.20"
  }]
}
→
409 Conflict
{
  // Error: the user already has
  // too many parallel orders,
  // creating a new one
  // is prohibited
  "reason": "order_limit_exceeded",
  "localized_message":
    "Order limit exceeded"
}
```

— what was the point of showing the price changed dialog, if the user still can't make an order, even if the price is right? When one of the concurrent orders has finished, and the user is able to commit another one, prices, item availability, and other order parameters will likely need another correction.

20. Analyze Potential Error Deadlocks

In complex systems, it might happen that resolving one error leads to another one, and vice versa.

```
// Create an order
// with paid delivery
POST /v1/orders
{
  "items": 3,
  "item_price": "3000.00"
  "currency_code": "MNT",
  "delivery_fee": "1000.00",
  "total": "10000.00"
}
→ 409 Conflict
// Error: if the order sum
// is more than 9000 tögrögs,
// delivery must be free
{
  "reason": "delivery_is_free"
}
// Create an order
// with free delivery
POST /v1/orders
{
  "items": 3,
  "item_price": "3000.00"
  "currency_code": "MNT",
  "delivery_fee": "0.00",
  "total": "9000.00"
}
→ 409 Conflict
// Error: the minimal order sum
// is 10000 tögrögs
{
  "reason": "below_minimal_sum",
  "currency_code": "MNT",
  "minimal_sum": "10000.00"
}
```

You may note that in this setup the error can't be resolved in one step: this situation must be elaborated on, and either order calculation parameters must be changed (discounts should not be counted against the minimal order sum), or a special type of error must be introduced.

21. Specify Caching Policies and Lifespans of Resources

In modern systems, clients usually have their own state and almost universally cache results of requests. Every entity has some period of autonomous existence, whether session-wise or long-term. So it's highly desirable to provide clarifications: it should be understandable how the data is supposed to be cached, if not from operation signatures, but at least from the documentation.

Let's emphasize that we understand “cache” in the extended sense: which variations of operation parameters (not just the request time, but other variables as well) should be considered close enough to some previous request to use the cached result?

Bad:

```
// Returns lungo prices including
// delivery to the specified location
GET /price?recipe=lungo↵
  &longitude={longitude}↵
  &latitude={latitude}
→
{ "currency_code", "price" }
```

Two questions arise:

- Until when is the price valid?
- In what vicinity of the location is the price valid?

Better: you may use standard protocol capabilities to denote cache options, such as the `Cache-Control` header. If you need caching in both temporal and spatial dimensions, you should do something like this:

```
GET /price?recipe=lungo↵
  &longitude={longitude}↵
  &latitude={latitude}
→
{
  "offer": {
    "id",
    "currency_code",
    "price",
    "conditions": {
      // Until when the price is valid
      "valid_until",
      // In what vicinity
      // the price is valid
      // * city
      // * geographical object
      // * …
      "valid_within"
    }
  }
}
```

NB: Sometimes, developers set very long caching times for immutable resources, spanning a year or even more. It makes little practical sense as the server load will not be significantly reduced compared to caching for, let's say, one month. However, the cost of a mistake increases dramatically: if wrong data is cached for some reason (for example, a `404` error), this problem will haunt you for the next year or even more. We would recommend selecting reasonable cache parameters based on how disastrous invalid caching would be for the business.

22. Keep the Precision of Fractional Numbers Intact

If the protocol allows, fractional numbers with fixed precision (such as money sums) must be represented as a specially designed type like `Decimal` or its equivalent.

If there is no `Decimal` type in the protocol (for instance, JSON doesn't have one), you should either use integers (e.g., apply a fixed multiplier) or strings.

If converting to a float number will certainly lead to a loss of precision (for example, if we translate "20 minutes" into hours as a decimal fraction), it's better to either stick to a fully precise format (e.g., use `00:20` instead of `0.33333…`), or provide an SDK to work with this data. As a last resort, describe the rounding principles in the documentation.

23. All API Operations Must Be Idempotent

Let us remind the reader that idempotency is the following property: repeated calls to the same function with the same parameters won't change the resource state. Since we are primarily discussing client-server interaction, repeating requests in case of network failure is not something exceptional but a common occurrence.

If an endpoint's idempotency can not be naturally assured, explicit idempotency parameters must be added in the form of a token or a resource version.

Bad:

```
// Creates an order
POST /orders
```

A second order will be produced if the request is repeated!

Better:

```
// Creates an order
POST /v1/orders
X-Idempotency-Token: <random string>
```

The client must retain the `X-Idempotency-Token` in case of automated endpoint retrying. The server must check whether an order created with this token already exists.

Alternatively:

```
// Creates order draft
POST /v1/orders/drafts
→
{ "draft_id" }
```

```
// Confirms the draft
PUT /v1/orders/drafts↵
  /{draft_id}/confirmation
{ "confirmed": true }
```

Creating order drafts is a non-binding operation as it doesn't entail any consequences, so it's fine to create drafts without the idempotency token. Confirming drafts is a naturally idempotent operation, with the `draft_id` serving as its idempotency key.

It is also worth mentioning that adding idempotency tokens to naturally idempotent handlers is not meaningless. It allows distinguishing between two situations:

- The client did not receive the response due to network issues and is now repeating the request.
- The client made a mistake by posting conflicting requests.

Consider the following example: imagine there is a shared resource, characterized by a revision number, and the client tries to update it.

```
POST /resource/updates
{
  "resource_revision": 123
  "updates"
}
```

The server retrieves the actual resource revision and finds it to be 124. How should it respond correctly? Returning the `409 Conflict` code will force the client to try to understand the nature of the conflict and somehow resolve it, potentially confusing the user. It is also unwise to fragment the conflict-resolving algorithm and allow each client to implement it independently.

The server can compare request bodies, assuming that identical requests mean retrying. However, this assumption might be dangerously wrong (for example if the resource is a counter of some kind, repeating identical requests is routine).

Adding the idempotency token (either directly as a random string or indirectly in the form of drafts) solves this problem.

```
POST /resource/updates
X-Idempotency-Token: <token>
{
  "resource_revision": 123
  "updates"
}
→ 201 Created
```

— the server determined that the same token was used in creating revision 124 indicating the client is retrying the request.

Or:

```
POST /resource/updates
X-Idempotency-Token: <token>
{
  "resource_revision": 123
  "updates"
}
→ 409 Conflict
```

— the server determined that a different token was used in creating revision 124 indicating an access conflict.

Furthermore, adding idempotency tokens not only fixes the issue but also enables advanced optimizations. If the server detects an access conflict, it could attempt to resolve it by "rebasing" the update like modern version control systems do, and return a `200 OK` instead of a `409 Conflict`. This logic dramatically improves the user experience, being fully backward-compatible, and helps avoid code fragmentation for conflict resolution algorithms.

However, be warned: clients are bad at implementing idempotency tokens. Two common problems arise:

- You can't really expect clients to generate truly random tokens. They might share the same seed or simply use weak algorithms or entropy sources. Therefore constraints must be placed on token checking, ensuring that tokens are unique to the specific user and resource rather than globally.
- Client developers might misunderstand the concept and either generate new tokens for each repeated request (which degrades the UX but is otherwise harmless) or conversely use a single token in several requests (which is not harmless at all and could lead to catastrophic disasters; this is another reason to implement the suggestion in the previous clause). Writing an SDK and/or detailed documentation is highly recommended.

24. Don't Invent Security Practices

If the author of this book were given a dollar each time he had to implement an additional security protocol invented by someone, he would be retired by now. API developers' inclination to create new signing procedures for requests or complex schemes of exchanging passwords for tokens is both obvious and meaningless.

First, there is no need to reinvent the wheel when it comes to security-enhancing procedures for various operations. All the algorithms you need are already invented, just adopt and implement them. No self-invented algorithm for request signature checking can provide the same level of protection against a Man-in-the-Middle attack as a TLS connection with mutual certificate pinning.

Second, assuming oneself to be an expert in security is presumptuous and dangerous. New attack vectors emerge daily, and staying fully aware of all actual threats is a full-time job. If you do something different during workdays, the security system you design will contain vulnerabilities that you have never heard about — for example, your password-checking algorithm might be susceptible to a timing attack or your webserver might be vulnerable to a request splitting attack.

The OWASP Foundation compiles a list of the most common vulnerabilities in APIs every year, which we strongly recommend studying.

And just in case: all APIs must be provided over TLS 1.2 or higher (preferably 1.3).

25. Help Partners With Security

It is equally important to provide interfaces to partners that minimize potential security problems for them.

Bad:

```
// Allows partners to set
// descriptions for their beverages
PUT /v1/partner-api/{partner-id}↵
  /recipes/lungo/info
"<script>alert(document.cookie)</script>"
```

```
// Returns the desciption
GET /v1/partner-api/{partner-id}↵
  /recipes/lungo/info
→
"<script>alert(document.cookie)</script>"
```

Such an interface directly creates a stored XSS vulnerability that potential attackers might exploit. While it is the partners' responsibility to sanitize inputs and display them safely, the large numbers work against you: there will always be inexperienced developers who are unaware of this vulnerability or haven't considered it. In the worst case, this stored XSS might affect all API consumers, not just a specific partner.

In these situations, we recommend, first, sanitizing the data if it appears potentially exploitable (e.g. if it is meant to be displayed in the UI and/or is accessible through a direct link). Second, limiting the blast radius so that stored exploits in one partner's data space can't affect other partners. If the functionality of unsafe data input is still required, the risks must be explicitly addressed:

Better (though not perfect):

```
// Allows for setting a potentially
// unsafe description for a beverage
PUT /v1/partner-api/{partner-id}↵
  /recipes/lungo/info
X-Dangerously-Disable-Sanitizing: true
"<script>alert(document.cookie)</script>"
```

```
// Returns the potentially
// unsafe description
GET /v1/partner-api/{partner-id}↵
  /recipes/lungo/info
X-Dangerously-Allow-Raw-Value: true
→
"<script>alert(document.cookie)</script>"
```

One important finding is that if you allow executing scripts via the API, always prefer typed input over unsafe input:

Bad:

```
POST /v1/run/sql
{
  // Passes the full script
  "query": "INSERT INTO data (name)↵
    VALUES ('Robert');↵
    DROP TABLE students;--')"
}
```

Better:

```
POST /v1/run/sql
{
  // Passes the script template
  "query": "INSERT INTO data (name)↵
    VALUES (?)",
  // and the parameters to set
  values: [
    "Robert');↵
     DROP TABLE students;--"
  ]
}
```

In the second case, you will be able to sanitize parameters and avoid SQL injections in a centralized manner. Let us remind the reader that sanitizing must be performed with state-of-the-art tools, not self-written regular expressions.

26. Use Globally Unique Identifiers

It's considered good practice to use globally unique strings as entity identifiers, either semantic (e.g., "lungo" for beverage types) or random ones (e.g., UUID-4). It might turn out to be extremely useful if you need to merge data from several sources under a single identifier.

In general, we tend to advise using URN-like identifiers, e.g. `urn:order:<uuid>` (or just `order:<uuid>`). That helps a lot in dealing with legacy systems with different identifiers attached to the same entity. Namespaces in URNs help to quickly understand which identifier is used and if there is a usage mistake.

One important implication: **never use increasing numbers as external identifiers**. Apart from the abovementioned reasons, it allows counting how many entities of each type there are in the system. Your competitors will be able to calculate the precise number of orders you have each day, for example.

27. Stipulate Future Restrictions

With the growth of API popularity, it will inevitably become necessary to introduce technical means of preventing illicit API usage, such as displaying captchas, setting honeypots, raising "too many requests" exceptions, installing anti-DDoS proxies, etc. All these things cannot be done if the corresponding errors and messages were not described in the docs from the very beginning.

You are not obliged to actually generate those exceptions, but you might stipulate this possibility in the docs. For example, you might describe the `429 Too Many Requests` error or captcha redirect but implement the functionality when it's actually needed.

It is extremely important to leave room for multi-factor authentication (such as TOTP, SMS, or 3D-secure-like technologies) if it's possible to make payments through the API. In this case, it's a must-have from the very beginning.

NB: This rule has an important implication: **always separate endpoints for different API families**. (This may seem obvious, but many API developers fail to follow it.) If you provide a server-to-server API, a service for end users, and a widget to be embedded in third-party apps — all these APIs must be served from different endpoints to allow for different security measures (e.g., mandatory API keys, forced login, and solving captcha respectively).

28. No Bulk Access to Sensitive Data

If it's possible to access the API users' personal data, bank card numbers, private messages, or any other kind of information that, if exposed, might seriously harm users, partners, and/or the API vendor, there must be *no* methods for bulk retrieval of the data, or at least there must be rate limiters, page size restrictions, and ideally, multi-factor authentication in front of them.

Often, making such offloads on an ad-hoc basis, i.e., bypassing the API, is a reasonable practice.

29. Localization and Internationalization

All endpoints must accept language parameters (e.g., in the form of the `Accept-Language` header), even if they are not currently being used.

It is important to understand that the user's language and the user's jurisdiction are different things. Your API working cycle must always store the user's location. It might be stated either explicitly (requests contain geographical coordinates) or implicitly (initial location-bound request initiates session creation which stores the location) — but no correct localization is possible in the absence of location data. In most cases reducing the location to just a country code is enough.

The thing is that lots of parameters that potentially affect data formats depend not on language but on the user's location. To name a few: number formatting (integer and fractional part delimiter, digit groups delimiter), date formatting, the first day of the week, keyboard layout, measurement units system (which might be non-decimal!), etc. In some situations, you need to store two locations: the user's residence location and the user's "viewport." For example, if a US citizen is planning a European trip, it's convenient to show prices in the local currency but measure distances in miles and feet.

Sometimes explicit location passing is not enough since there are lots of territorial conflicts in the world. How the API should behave when user coordinates lie within disputed regions is a legal matter, regretfully. The author of this book once had to implement a "state A territory according to state B official position" concept.

Important: mark a difference between localization for end users and localization for developers. In the examples above, the `localized_message` field is meant for the user; the app should show it if no specific handler for this error exists in the client code. This message must be written in the user's language and formatted according to the user's location. But the `details.checks_failed[].message` is meant to be read by developers examining the problem. So it must be written and formatted in a manner that suits developers best — which usually means "in English," as English is a *de facto* standard in software development.

It is worth mentioning that the `localized_` prefix in the examples is used to differentiate messages to users from messages to developers. A concept like that must be, of course, explicitly stated in your API docs.

And one more thing: all strings must be UTF-8, no exclusions.

Chapter 14. Annex to Section I. Generic API Example

Let's summarize the current state of our API study.

1. Offer Search

```
POST /v1/offers/search
{
  // optional
  "recipes": ["lungo", "americano"],
  "position": <geographical coordinates>,
  "sort_by": [
    { "field": "distance" }
  ],
  "limit": 10
}
→
{
  "results": [{
    // Place data
    "place":
      { "name", "location" },
    // Coffee machine properties
    "coffee-machine":
      { "id", "brand", "type" },
    // Route data
    "route": {
      "distance",
      "duration",
      "location_tip"
    },
    "offers": [{
      // Recipe data
      "recipe":
        { "id", "name", "description" },
      // Recipe specific options
      "options":
        { "volume" },
      // Offer metadata
      "offer":
        { "id", "valid_until" },
      // Pricing
      "pricing": {
        "currency_code",
        "price",
        "localized_price"
      },
```

```
      "estimated_waiting_time"
    }, …]
  }, …],
  "cursor"
}
```

2. Working with Recipes

```
// Returns a list of recipes
// Cursor parameter is optional
GET /v1/recipes?cursor=<cursor>
→
{ "recipes", "cursor" }
```

```
// Returns the recipe by its id
GET /v1/recipes/{id}
→
{
  "recipe_id",
  "name",
  "description"
}
```

3. Working with Orders

```
// Creates an order
POST /v1/orders
{
  "coffee_machine_id",
  "currency_code",
  "price",
  "recipe": "lungo",
  // Optional
  "offer_id",
  // Optional
  "volume": "800ml"
}
→
{ "order_id" }
```

```
// Returns the order by its id
GET /v1/orders/{id}
→
{ "order_id", "status" }
```

```
// Cancels the order
POST /v1/orders/{id}/cancel
```

4. Working with Programs

```
// Returns an identifier of the program
// corresponding to specific recipe
// on specific coffee-machine
POST /v1/program-matcher
{ "recipe", "coffee-machine" }
→
{ "program_id" }
```

```
// Return program description
// by its id
GET /v1/programs/{id}
→
{
  "program_id",
  "api_type",
  "commands": [
    {
      "sequence_id",
      "type": "set_cup",
      "parameters"
    },
    …
  ]
}
```

5. Running Programs

```
// Runs the specified program
// on the specified coffee-machine
// with specific parameters
POST /v1/programs/{id}/run
{
  "order_id",
  "coffee_machine_id",
  "parameters": [
    {
      "name": "volume",
      "value": "800ml"
    }
  ]
}
→
{ "program_run_id" }
```

```
// Stops program running
POST /v1/runs/{id}/cancel
```

6. Managing Runtimes

```
// Creates a new runtime
POST /v1/runtimes
{
  "coffee_machine_id",
  "program_id",
  "parameters"
}
→
{ "runtime_id", "state" }
```

```
// Returns the state
// of the specified runtime
GET /v1/runtimes/{runtime_id}/state
{
  "status": "ready_waiting",
  // Command being currently executed
  // (optional)
  "command_sequence_id",
  "resolution": "success",
  "variables"
}
```

```
// Terminates the runtime
POST /v1/runtimes/{id}/terminate
```

SECTION II. THE API PATTERNS

Chapter 15. On Design Patterns in the API Context

The concept of "Patterns" in the field of software engineering was introduced by Kent Beck and Ward Cunningham in 1987 and popularized by "The Gang of Four" (Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides) in their book "Design Patterns: Elements of Reusable Object-Oriented Software," which was published in 1994. According to the most widespread definition, a software design pattern is a "general, reusable solution to a commonly occurring problem within a given context."

If we talk about APIs, especially those to which developers are end users (e.g., frameworks or operating system interfaces), the classical software design patterns are well applicable to them. Indeed, many examples in the previous Section of this book are just about applying some design patterns.

However, if we try to extend this approach to include API development in general, we will soon find that many typical API design issues are high-level and can't be reduced to basic software patterns. Let's say, caching resources (and invalidating the cache) or organizing paginated access are not covered in classical writings.

In this Section, we will specify those API design problems that we see as the most important ones. We are not aiming to encompass *every* problem, let alone every solution, and rather focus on describing approaches to solving typical problems with their pros and cons. We do understand that readers familiar with the works of "The Gang of Four," Grady Booch, and Martin Fowler might expect a more systematic approach and greater depth of outreach from a section called "The API Patterns," and we apologize to them in advance.

NB: The first such pattern we need to mention is the API-first approach to software engineering, which we described in the corresponding chapter.

The Fundamentals of Solving Typical API Design Problems

Before we proceed to the patterns, we need to understand first, how developing APIs differs from developing other kinds of software. Below, we will formulate three important concepts, which we will be referring to in the subsequent chapters.

1. The more distributed and multi-faceted systems are built and the more general-purpose channels of communication are used, the more errors occur in the process of interaction. In the most interesting case of distributed many-layered client-server systems, raising an exception on the side of a client (like losing context as a result of app crash and restart), server (the pipeline of executing a query threw at some stage), communication channel (connection fully or partially lost), or any other interim agent (intermediate web-server hasn't got a response from backend and returned a gateway error) is a norm of life, and all systems must be designed in a manner that in a case of an exception of any kind, API clients must be able to restore their state and continue operating normally.

2. The more partners use the API, the more chance is that some of the mechanisms of the expected workflow are implemented wrongly. In other words, not only genuine errors related to network or server overload should be expected, but also logical ones caused by improper API usage (and, in particular, there should be safeguards to avoid errors in one partner's code leading to a denial of service for other partners).

3. Any part of the system might introduce unpredictable latencies when serving requests, and these latencies could be quite high, up to seconds and tens of seconds. Even if you have full control over the execution environment and network, client apps may hinder themselves due to suboptimal code or execution on low-performing or overloaded devices. As a result, it is important to ensure that proper API design does not rely on critical operations being executed quickly. This includes:

- If carrying out a task through the API requires making a sequence of calls, there should be a mechanism in place to resume the operation from the current step if needed, instead of restarting it from the beginning.
- Operations that affect shared resources should have locking mechanisms in place for the duration of the operation.

Chapter 16. Authenticating Partners and Authorizing API Calls

Before we proceed further to discussing technical matters, we feel obliged to provide an overview of the problems related to authorizing API calls and authenticating clients. Based on the main principle that "an API serves as a multiplier to both your opportunities and mistakes," organizing authorization and authentication (AA) is one of the most important challenges that any API vendor faces, especially when it comes to public APIs. It is rather surprising that there is no standard approach to this issue, as every big vendor develops its own interface to solve AA problems, and these interfaces are often quite archaic.

If we set aside implementation details (for which we strongly recommend not reinventing the wheel and using standard techniques and security protocols), there are basically two approaches to authorizing an API call:

- Introducing a special "robot" type of account into the system, and carrying out the operations on behalf of the robot account.
- Authorizing the caller system (backend or client application) as a single entity, using API keys, signatures, or certificates for the purpose of authenticating such calls.

The difference between the two approaches lies in the access granularity:

- If an API client is making requests as a regular user of the system, then it can only perform operations allowed for a specific user, which often means it might have access only to a partial dataset within the API endpoint.
- If the caller system is authorized, it implies that it has full access to the endpoint and can supply any parameters, i.e., might operate the full dataset exposed through the endpoint.

Therefore, the first approach is more granular (the robot might be a "virtual employee" with access only to a limited dataset) and is a natural choice for APIs that are supplemental to an existing service for end users (and thus can reuse the existing AA solutions). However, this approach

has some disadvantages:

- The need to develop a process for securely fetching authorization tokens for the robot user (e.g., via having a real user generate tokens in the web UI), as regular login-password authentication (especially multi-factored) is not well-suited for API clients.
- The need to make exceptions for robot users in almost every security protocol:
    - Robots might make many more requests per second than real users and might perform several queries in parallel (possibly from different IP addresses located in different availability zones).
    - Robots do not accept cookies and cannot solve captchas.
    - Robots should not be logged out or have their token invalidated (as it would impact the partner's business processes), so it is usually necessary to invent specific long-lived tokens for robots and/or token renewal procedures.
- Finally, you may encounter significant challenges if you need to allow robots to perform operations on behalf of other users (as you will have to either expose this functionality to all users or, vice versa, hide its existence from them).

If the API is not about providing additional access to a service for end users, it is usually much easier to opt for the second approach and authorize clients with API keys. In this case, per-endpoint granularity can be achieved (i.e., allowing partners to regulate the set of permitted endpoints for a key), while developing more granular access can be much more complex and because of that rarely see implementations.

Both approaches can be morphed into each other (e.g., allowing robot users to perform operations on behalf of any other users effectively becomes API key-based authorization; allowing binding of a limited dataset to an API key effectively becomes a user account), and there are some hybrid systems in the wild (where the request must be signed with both an API key and a user token).

Chapter 17. Synchronization Strategies

Let's proceed to the technical problems that API developers face. We begin with the last one described in the introductory chapter: the necessity to synchronize states. Let us imagine that a user creates a request to order coffee through our API. While this request travels from the client to the coffee house and back, many things might happen. Consider the following chain of events:

1. The client sends the order creation request
2. Because of network issues, the request propagates to the server very slowly, and the client gets a timeout
    - Therefore, the client does not know whether the query was served or not.
3. The client requests the current state of the system and gets an empty response as the initial request still hasn't reached the server:

```
const pendingOrders = await
  api.getOngoingOrders(); // → []
```

4. The server finally gets the initial request for creating an order and serves it.
5. The client, being unaware of this, tries to create an order anew.

As the operations of reading the list of ongoing orders and of creating a new order happen at different moments of time, we can't guarantee that the system state hasn't changed in between. If we do want to have this guarantee, we must implement some synchronization strategy. In the case of, let's say, operating system APIs or client frameworks we might rely on the primitives provided by the platform. But in the case of distributed client-server APIs, we would need to implement such a primitive of our own.

There are two main approaches to solving this problem: the pessimistic one (implementing locks in the API) and the optimistic one (resource versioning).

NB: Generally speaking, the best approach to tackling an issue is not having the issue at all. Let's say, if your API is idempotent, the duplicating calls are not a problem. However, in the real world, not every operation is idempotent; for example, creating new orders is not. We might add mechanisms to prevent *automatic* retries (such as client-generated idempotency tokens) but we can't forbid users from just creating a second identical order.

API Locks

The first approach is to literally implement standard synchronization primitives at the API level. Like this, for example:

```
let lock;
try {
  // Capture the exclusive
  // right to create new orders
  lock = await api.
    acquireLock(ORDER_CREATION);
  // Get the list of current orders
  // known to the system
  const pendingOrders = await
    api.getPendingOrders();
  // If our order is absent,
  // create it
  if (pendingOrders.length == 0) {
    const order = await api
      .createOrder(…)
  }
} catch (e) {
  // Deal with errors
} finally {
  // Unblock the resource
  await lock.release();
}
```

Rather unsurprisingly, this approach sees very rare use in distributed client-server APIs because of the plethora of related problems:

1. Waiting for acquiring a lock introduces new latencies to the interaction that are hardly predictable and might potentially be quite significant.
2. The lock itself is one more entity that constitutes a subsystem of its own, and quite a demanding one as strong consistency is required for implementing locks: the `getPendingOrders` function must return the up-to-date state of the system otherwise the duplicate order will be anyway created.
3. As it's partners who develop client code, we can't guarantee it works with locks always correctly. Inevitably, “lost” locks will occur in the system, and that means we need to provide some tools to partners so they can find the problem and debug it.
4. A certain granularity of locks is to be developed so that partners can't affect each other. We are lucky if there are natural boundaries for a lock — for example, if it's limited to a specific user in the specific partner's system. If we are not so lucky (let's say all partners share the same user profile), we will have to develop even more complex systems to deal with potential errors in the partners' code — for example, introduce locking quotas.

Optimistic Concurrency Control

A less implementation-heavy approach is to develop an optimistic concurrency control system, i.e., to require clients to pass a flag proving they know the actual state of a shared resource.

```
// Retrieve the state
const orderState =
  await api.getOrderState();
// The version is a part
// of the state of the resource
const version =
  orderState.latestVersion;
// An order might only be created
// if the resource version hasn't
// changed since the last read
try {
  const task = await api
    .createOrder(version, …);
} catch (e) {
  // If the version is wrong, i.e.,
  // another client changed the
  // resource state, an error occurs
  if (Type(e) == INCORRECT_VERSION) {
    // Which should be handled…
  }
}
```

NB: An attentive reader might note that the necessity to implement some synchronization strategy and strongly consistent reading has not disappeared: there must be a component in the system that performs a locking read of the resource version and its subsequent change. It's not entirely true as synchronization strategies and strongly consistent reading have disappeared *from the public API*. The distance between the client that sets the lock and the server that processes it became much smaller, and the entire interaction now happens in a controllable environment. It might be a single subsystem in a form of an ACID-compatible database or even an in-memory solution.

Instead of a version, the date of the last modification of the resource might be used (which is much less reliable as clocks are not ideally synchronized across different system nodes; at least save it with the maximum possible precision!) or entity identifiers (ETags).

The advantage of optimistic concurrency control is therefore the possibility to hide under the hood the complexity of implementing locking mechanisms. The disadvantage is that the versioning errors are no longer exceptional situations — it's now a regular behavior of the system. Furthermore, client developers *must* implement working with them otherwise the application might render inoperable as users will be infinitely creating an order with the wrong version.

NB: Which resource to select for making versioning is extremely important. If in our example we create a global system version that is incremented after any order comes, users' chances to successfully create an order will be close to zero.

The approach described in the previous chapter is in fact a trade-off: the API performance issues are traded for "normal" (i.e., expected) background errors that happen while working with the API. This is achieved by isolating the component responsible for controlling concurrency and only exposing read-only tokens in the public API. Still, the achievable throughput of the API is limited, and the only way of scaling it up is removing the strict consistency from the external API and thus allowing reading system state from read-only replicas:

```
// Reading the state,
// possibly from a replica
const orderState =
  await api.getOrderState();
const version =
  orderState.latestVersion;
try {
  // The request handler will
  // read the actual version
  // from the master data
  const task = await api
    .createOrder(version, …);
} catch (e) {
  …
}
```

As orders are created much more rarely than read, we might significantly increase the system performance if we drop the requirement of returning the most recent state of the resource from the state retrieval endpoints. The versioning will help us avoid possible problems: creating an order will still be impossible unless the client has the actual version. In fact, we transited to the eventual consistency model: the client will be able to fulfill its request *sometime* when it finally gets the actual data. In modern microservice architectures, eventual consistency is rather an industrial standard, and it might be close to impossible to achieve the opposite, i.e., strict consistency.

NB: Let us stress that you might choose the approach only in the case of exposing new APIs. If you're already providing an endpoint implementing some consistency model, you can't just lower the consistency level (for instance, introduce eventual consistency instead of the strict one) even if you never documented the behavior. This will be discussed in detail in the "On the Waterline of the Iceberg" chapter of "The Backward Compatibility" section of this book.

Choosing weak consistency instead of a strict one, however, brings some disadvantages. For instance, we might require partners to wait until they get the actual resource state to make changes — but it is quite unobvious for partners (and actually inconvenient) they must be prepared to wait for changes they made themselves to propagate.

```
// Creates an order
const api = await api
  .createOrder(…)
// Returns a list of orders
const pendingOrders = await api.
  getOngoingOrders(); // → []
  // The list is empty
```

If strict consistency is not guaranteed, the second call might easily return an empty result as it reads data from a replica, and the newest order might not have hit it yet.

An important pattern that helps in this situation is implementing the "read-your-writes" model, i.e., guaranteeing that clients observe the changes they have just made. The consistency might be lifted to the read-your-writes level by making clients pass some token that describes the last changes known to the client.

```
const order = await api
  .createOrder(…);
const pendingOrders = await api.
  getOngoingOrders({
    …,
    // Pass the identifier of the
    // last operation made by
    // the client
    last_known_order_id: order.id
  })
```

Such a token might be:

- An identifier (or identifiers) of the last modifying operations carried out by the client
- The last known resource version (modification date, ETag) known to the client.

Upon getting the token, the server must check that the response (e.g., the list of ongoing operations it returns) matches the token, i.e., the eventual consistency converged. If it did not (the client passed the modification date / version / last order id newer than the one known to the server), one of the following policies or their combinations might be applied:

- The server might repeat the request to the underlying DB or to the other kind of data storage in order to get the newest version (eventually)
- The server might return an error that requires the client to try again later
- The server queries the main node of the DB, if such a thing exists, or otherwise initiates retrieving the master data.

The advantage of this approach is client development convenience (compared to the absence of any guarantees): by preserving the version token, client developers get rid of the possible inconsistency of the data got from API endpoints. There are two disadvantages, however:

- It is still a trade-off between system scalability and a constant inflow of background errors:
    - If you're querying master data or repeating the request upon the version mismatch, the load on the master storage is increased in poorly a predictable manner
    - If you return a client error instead, the number of such errors might be considerable, and partners will need to write some additional code to deal with the errors.
- This approach is still probabilistic, and will only help in a limited number of use cases (to be discussed below).

There is also an important question regarding the default behavior of the server if no version token was passed. Theoretically, in this case, master data should be returned, as the absence of the token might be the result of an app crash and subsequent restart or corrupted data storage. However, this implies an additional load on the master node.

Evaluating the Risks of Switching to Eventual Consistency

Let us state an important assertion: the methods of solving architectural problems we're discussing in this section are probabilistic. Abolishing strict consistency means that even if all components of the system work perfectly, client errors will still occur. It might appear that they could be simply ignored, but in reality, doing so means introducing risks.

Imagine that because of eventual consistency, users of our API sometimes cannot create orders with their first attempt. For example, a customer adds a new payment method in the application, but their subsequent order creation request is routed to a replica that hasn't yet received the information regarding the newest payment method. As these two actions (adding a bank card and making an order) often go in conjunction, there will be a noticeable percentage of errors — let's say, 1%. At this stage, we could disregard the situation as it appears harmless: in the worst-case scenario, the client will repeat the request.

But let's go a bit further and imagine there is an error in a new version of the application, and 0.1% of end users cannot make an order at all because the client sends a wrong payment method identifier. In the absence of this 1% background noise of consistency-bound errors, we would find the issue very quickly. However, amidst this constant inflow of errors, identifying problems like this one could be very challenging as it requires configuring monitoring systems to reliably exclude the data consistency errors, and this could be very complicated or even impossible. The author of this book, in his job, has seen several situations when critical mistakes that affect a small percentage of users were not noticed for months.

Therefore, the task of proactively lowering the number of these background errors is crucially important. We may try to reduce their occurrence for typical usage profiles.

NB: The “typical usage profile” stipulation is important: an API implies the variability of client scenarios, and API usage cases might fall into several groups, each featuring quite different error profiles. The classical example is client APIs (where it's an end user who makes actions and waits for results) versus server APIs (where the execution time is per se not so important — but let's say mass parallel execution might be). If this happens, it's a strong signal to make a family of API products covering different usage scenarios, as we will discuss in “The API Services Range” chapter of “The API Product” section of this book.

Let's return to the coffee example, and imagine we implemented the following scheme:

- Optimistic concurrency control (through, let's say, the id of the last user's order)
- The “read-your-writes” policy of reading the order list (again with passing the last known order id as a token)
- Retrieving master data in the case the token is absent.

In this case, the order creation error might only happen in one of the two cases:

- The client works with the data incorrectly (does not preserve the identifier of the last order or the idempotency key while repeating the request)
- The client tries to create an order from two different instances of the app that do not share the common state.

The first case means there is a bug in the partner's code; the second case means that the user is deliberately testing the system's stability — which is hardly a frequent case (or, let's say, the user's phone went off and they quickly switched to a tablet — rather rare case as well, we must admit).

Let's now imagine that we dropped the third requirement — i.e., returning the master data if the token was not provided by the client. We would get the third case when the client gets an error:

- The client application lost some data (restarted or corrupted), and the user tries to replicate the last request.

NB: The repeated request might happen without any automation involved if, let's say, the user got bored of waiting, killed the app and manually re-orders the coffee again.

Mathematically, the probability of getting the error is expressed quite simply. It's the ratio between two durations: the time period needed to get the actual state to the time period needed to restart the app and repeat the request. (Keep in mind that the last failed request might be automatically repeated on startup by the client.) The former depends on the technical properties of the system (for instance, on the replication latency, i.e., the lag between the master and its read-only copies) while the latter depends on what client is repeating the call.

If we talk about applications for end users, the typical restart time there is measured in seconds, which normally should be much less than the overall replication latency. Therefore, client errors will only occur in case of data replication problems / network issues / server overload.

If, however, we talk about server-to-server applications, the situation is totally different: if a server repeats the request after a restart (let's say because the process was killed by a supervisor), it's typically a millisecond-scale delay. And that means that the number of order creation errors will be significant.

As a conclusion, returning eventually consistent data by default is only viable if an API vendor is either ready to live with background errors or capable of making the lag of getting the actual state much less than the typical app restart time.

Let's continue working with the previous example: the application retrieves some system state upon start-up, perhaps not the most recent one. What else does the probability of collision depend on, and how can we lower it?

We remember that this probability is equal to the ratio of time periods: getting an actual state versus starting an app and making an order. The latter is almost out of our control (unless we deliberately introduce additional waiting periods in the API initialization function, which we consider an extreme measure). Let's then talk about the former.

Our usage scenario looks like this:

```
const pendingOrders = await api.
  getOngoingOrders();
if (pendingOrders.length == 0) {
  const order = await api
    .createOrder(…);
}
// App restart happens here,
// and all the same requests
// are repeated
const pendingOrders = await api.
  getOngoingOrders(); // → []
if (pendingOrders.length == 0) {
  const order = await api
    .createOrder(…);
}
```

Therefore, we're trying to minimize the following interval: network latency to deliver the `createOrder` call plus the time of executing the `createOrder` plus the time needed to propagate the newly created order to the replicas. We don't control the first summand (but we might expect the network latencies to be more or less constant during the session duration, so the next `getOngoingOrders` call will be delayed for roughly

the same time period). The third summand depends on the infrastructure of the backend. Let's talk about the second one.

As we can see if the order creation itself takes a lot of time (meaning that it is comparable to the app restart time) then all our previous efforts were useless. The end user must wait until they get the server response back and might just restart the app to make a second `createOrder` call. It is in our best interest to ensure this never happens.

However, what we could do to improve this timing remains unclear. Creating an order might *indeed* take a lot of time as we need to carry out necessary checks and wait for the payment gateway response and confirmation from the coffee shop.

What could help us here is the asynchronous operations pattern. If our goal is to reduce the collision rate, there is no need to wait until the order is *actually* created as we need to quickly propagate the knowledge that the order is *accepted for creation*. We might employ the following technique: create *a task for order creation* and return its identifier, not the order itself.

```
const pendingOrders = await api.
  getOngoingOrders();
if (pendingOrders.length == 0) {
  // Instead of creating an order,
  // put the task for the creation
  const task = await api
    .putOrderCreationTask(…);
}
// App restart happens here,
// and all the same requests
// are repeated
const pendingOrders = await api.
  getOngoingOrders();
  // → { tasks: [task] }
```

Here we assume that task creation requires minimal checks and doesn't wait for any lingering operations, and therefore, it is carried out much faster. Furthermore, this operation (of creating an asynchronous task) might be isolated as a separate backend service for performing abstract asynchronous tasks. By having the functionality of creating tasks and retrieving the list of ongoing tasks we can significantly narrow the “gray zones” when clients can't learn the actual system state precisely.

Thus we naturally came to the pattern of organizing asynchronous APIs through task queues. Here we use the term “asynchronous” logically meaning the absence of mutual *logical* locks: the party that makes a request gets a response immediately and does not wait until the requested procedure is fully carried out being able to continue to interact with the API. *Technically* in modern application environments, locking (of both the client and server) almost universally doesn't happen during long-responding calls. However, *logically* allowing users to work with the API while waiting for a response from a modifying endpoint is error-prone and leads to collisions like the one we described above.

The asynchronous call pattern is useful for solving other practical tasks as well:

- Caching operation results and providing links to them (implying that if the client needs to reread the operation result or share it with another client, it might use the task identifier to do so)
- Ensuring operation idempotency (through introducing the task confirmation step we will actually get the draft-commit system as discussed in the “Describing Final Interfaces” chapter)
- Naturally improving resilience to peak loads on the service as the new tasks will be queuing up (possibly prioritized) in fact implementing the “token bucket” technique
- Organizing interaction in the cases of very long-lasting operations that require more time than typical timeouts (which are tens of seconds in the case of network calls) or can take unpredictable time.

Also, asynchronous communication is more robust from a future API development point of view: request handling procedures might evolve towards prolonging and extending the asynchronous execution pipelines whereas synchronous handlers must retain reasonable execution times which puts certain restrictions on possible internal architecture.

NB: In some APIs, an ambivalent decision is implemented where endpoints feature a double interface that might either return a result or a link to a task. Although from the API developer's point of view, this might look logical (if the request was processed "quickly", e.g., served from cache, the result is to be returned immediately; otherwise, the asynchronous task is created), for API consumers, this solution is quite inconvenient as it forces them to maintain two execution branches in their code. Sometimes, a concept of providing a double set of endpoints (synchronous and asynchronous ones) is implemented, but this simply shifts the burden of making decisions onto partners.

The popularity of the asynchronicity pattern is also driven by the fact that modern microservice architectures "under the hood" operate in asynchronous mode through event queues or pub/sub middleware. Implementing an analogous approach in external APIs is the simplest solution to the problems caused by asynchronous internal architectures (the unpredictable and sometimes very long latencies of propagating changes). Ultimately, some API vendors make all API methods asynchronous (including the read-only ones) even if there are no real reasons to do so.

However, we must stress that excessive asynchronicity, though appealing to API developers, implies several quite objectionable disadvantages:

1. If a single queue service is shared by all endpoints, it becomes a single point of failure for the system. If unpublished events are piling up and/or the event processing pipeline is overloaded, all the API endpoints start to suffer. Otherwise, if there is a separate queue service instance for every functional domain, the internal architecture becomes much more complex, making monitoring and troubleshooting increasingly costly.

2. For partners, writing code becomes more complicated. It is not only about the physical volume of code (creating a shared component to communicate with queues is not that complex of an engineering task) but also about anticipating every endpoint to possibly respond slowly. With synchronous endpoints, we assume by default that they respond within a reasonable time, less than a typical response timeout (which, for client applications, means that just a spinner might be shown to a user). With asynchronous endpoints, we don't have such a guarantee as it's simply impossible to provide one.

3. Employing task queues might lead to some problems specific to the queue technology itself, i.e., not related to the business logic of the request handler:

    - Tasks might be “lost” and never processed
    - Events might be received in the wrong order or processed twice, which might affect public interfaces
    - Under the task identifier, wrong data might be published (corresponding to some other task) or the data might be corrupted.

    These issues will be totally unexpected by developers and will lead to bugs in applications that are very hard to reproduce.

4. As a result of the above, the question of the viability of such an SLA level arises. With asynchronous tasks, it's rather easy to formally make the API uptime 100.00% — just some requests will be served in a couple of weeks when the maintenance team finds the root cause of the delay. Of course, that's not what API consumers want: their users need their problems solved *now* or at least *in a reasonable time*, not in two weeks.

Therefore, despite all the advantages of the approach, we tend to recommend applying this pattern only to those cases when they are really needed (as in the example we started with when we needed to lower the probability of collisions) and having separate queues for each case. The perfect task queue solution is the one that doesn't look like a task queue. For example, we might simply make the “order creation task

is accepted and awaits execution” state a separate order status and make its identifier the future identifier of the order itself:

```
const pendingOrders = await api.
  getOngoingOrders();
if (pendingOrders.length == 0) {
  // Don't call it a “task”,
  // just create an order
  const order = await api
    .createOrder(…);
}
// App restart happens here,
// and all the same requests
// are repeated
const pendingOrders = await api.
  getOngoingOrders();
  /* → { orders: [{
    order_id: <task identifier>,
    status: "new"
  }]} */
```

NB: Let us also mention that in the asynchronous format, it's possible to provide not only binary status (task done or not) but also execution progress as a percentage if needed.

Chapter 20. Lists and Accessing Them

In the previous chapter, we concluded with the following interface that allows minimizing collisions while creating orders:

```
const pendingOrders = await api
  .getOngoingOrders();
→
{ orders: [{
    order_id: <task identifier>,
    status: "new"
}, …]}
```

However, an attentive reader might notice that this interface violates the recommendation we previously gave in the "Describing Final Interfaces" chapter: the returned data volume must be limited, but there are no restrictions in our design. This problem was already present in the previous versions of the endpoint, but abolishing asynchronous order creation makes it much worse. The task creation operation must work as quickly as possible, and therefore, almost all limit checks are to be executed asynchronously. As a result, a client might easily create a large number of ongoing tasks which would potentially inflate the size of the `getOngoingOrders` response.

NB: Having *no limit at all* on order task creation is unwise, and there must be some (involving as lightweight checks as possible). Let us, however, focus on the response size issue in this chapter.

Fixing this problem is rather simple: we might introduce a limit for the items returned in the response, and allow passing filtering and sorting parameters, like this:

```
api.getOngoingOrders({
  // The `limit` parameter
  // is optional, but there is
  // a reasonable default value
  "limit": 100,
  "parameters": {
    "order_by": [{
      "field": "created_iso_time",
      "direction": "desc"
    }]
  }
})
```

However, introducing limits leads to another issue: if the number of items to return is higher than the limit, how would clients access them?

The standard approach is to add an `offset` parameter or a page number:

```
api.getOngoingOrders({
  // The `limit` parameter
  // is optional, but there is
  // a reasonable default value
  "limit": 100,
  // The default value is 0
  "offset": 100,
  "parameters"
})
```

With this approach, however, other problems arise. Let us imagine three orders are being processed on behalf of the user:

```
[{
  "id": 3,
  "created_iso_time": "2022-12-22T15:35",
  "status": "new"
}, {
  "id": 2,
  "created_iso_time": "2022-12-22T15:34",
  "status": "new"
}, {
  "id": 1,
  "created_iso_time": "2022-12-22T15:33",
  "status": "new"
}]
```

A partner application requested the first page of the list:

```
api.getOrders({
  "limit": 2,
  "parameters": {
    "order_by": [{
      "field": "created_iso_time",
      "direction": "desc"
    }]
  }
})
→
{
  "orders": [{
    "id": 3, …
  }, {
    "id": 2, …
  }]
}
```

Then the application requests the second page (`"limit": 2, "offset": 2`) and expects to retrieve the order with `"id": 1`. However, during the interval between the requests, another order, with `"id": 4`, happened.

```
[{
  "id": 4,
  "created_iso_time": "2022-12-22T15:36",
  "status": "new"
}, {
  "id": 3,
  "created_iso_time": "2022-12-22T15:35",
  "status": "new"
}, {
  "id": 2,
  "created_iso_time": "2022-12-22T15:34",
  "status": "ready"
}, {
  "id": 1,
  "created_iso_time": "2022-12-22T15:33",
  "status": "new"
}]
```

Then upon requesting the second page of the order list, instead of getting exactly one order with `"id": 1`, the application will get the `"id": 2` order once again:

```
api.getOrders({
  "limit": 2,
  "offset": 2
  "parameters"
})
→
{
  "orders": [{
    "id": 2, …
  }, {
    "id": 1, …
  }]
}
```

These permutations are rather inconvenient in user interfaces (if let's say, the partner's accountant is requesting orders to calculate fees, they might easily overlook the duplicate identifiers and process one order twice). But in the case of *programmable* integrations, the situation becomes even more complicated: the application developer needs to write rather unobvious code (which preserves the information regarding which pages were already processed) to carry out this enumeration correctly.

The problem might easily become even more sophisticated. For example, if we add sorting by two fields, creation date and order status:

```
api.getOrders({
  "limit": 2,
  "parameters": {
    "order_by": [{
      "field": "status",
      "direction": "desc"
    }, {
      "field": "created_iso_time",
      "direction": "desc"
    }]
  }
})
→
{
  "orders": [{
    "id": 3,
    "status": "new"
  }, {
    "id": 2,
    "status": "new"
  }]
}
```

Imagine, that in between requesting the first and the second pages, the `"id": 1` order changed its status and moved to the top of the list. Upon requesting the second page, the partner application will only receive the `"id": 2` order (for the second time) and miss the `"id": 1` completely —

and there is no method to learn this fact!

Let us reiterate: this approach works poorly with visual interfaces, but with program ones, it inevitably leads to mistakes. **An API must provide methods of traversing large lists that guarantee clients can retrieve the full and consistent dataset**.

If we don't go into implementation details, we can identify three main patterns of realizing such traversing, depending on how the data itself is organized.

Immutable Lists

The easiest case is with immutable lists, i.e., when the set of items never changes. The `limit/offset` scheme then works perfectly and no additional tricks are needed. Unfortunately, this rarely happens in real subject areas.

Additive Lists, Immutable Data

The case of a list with immutable items and the operation of adding new ones is more typical. Most notably, we talk about event queues containing, for example, new messages or notifications. Let's imagine there is an endpoint in our coffee API that allows partners to retrieve the history of offers:

```
GET /v1/partners/{id}/offers/history↵
  limit=<limit>
→
{
  "offer_history": [{
    // A list item identifier
    "id",
    // An identifier of the user
    // that got the offer
    "user_id",
    // Date and time of the search
    "occurred_at",
    // The search parameter values
    // set by the user
    "search_parameters",
    // The offers that the user got
    "offers"
  }]
}
```

The data returned from this endpoint is naturally immutable because it reflects a completed action: a user searched for offers and received a response. However, new items are continuously added to the list, potentially in large chunks, as users might make multiple searches in succession.

Partners can utilize this data to implement various features, such as:

1. Real-time user behavior analysis (e.g., sending push notifications with discount codes to encourage users to convert offers to orders)
2. Statistical analysis (e.g., calculating conversion rates per hour).

To enable these scenarios, we need to expose through the API two operations with the offer history:

1. For the first task, the real-time fetching of new offers that were made since the last request.

2. For the second task, traversing the list, i.e., retrieving all queries until some condition is reached (possibly, the end of the list).

Both scenarios are covered with the `limit/offset` approach but require significant effort to write code properly as partners need to somehow align their requests with the rate of incoming queries. Additionally, note that using the `limit/offset` scheme makes caching impossible as repeating requests with the same `limit/offset` values will emit different results.

To solve this issue, we need to rely not on an attribute that constantly changes (such as the item position in the list) but on other anchors. The important rule is that this attribute must provide the possibility to unambiguously tell which list elements are "newer" compared to the given one (i.e., precede it in the list) and which are "older".

If the data storage we use for keeping list items offers the possibility of using monotonically increased identifiers (which practically means two things: (1) the DB supports auto-incremental columns and (2) there are insert locks that guarantee inserts are performed sequentially), then using the monotonous identifier is the most convenient way of organizing list traversal:

```
// Retrieve the records that precede
// the one with the given id
GET /v1/partners/{id}/offers/history↵
  newer_than=<item_id>&limit=<limit>
// Retrieve the records that follow
// the one with the given id
GET /v1/partners/{id}/offers/history↵
  older_than=<item_id>&limit=<limit>
```

The first request format allows for implementing the first scenario, i.e., retrieving the fresh portion of the data. Conversely, the second format makes it possible to consistently iterate over the data to fulfill the second scenario. Importantly, the second request is cacheable as the tail of the list never changes.

NB: In the "Describing Final Interfaces" chapter we recommended avoiding exposing incremental identifiers in publicly accessible APIs. Note that the scheme described above might be augmented to comply with this rule by exposing some arbitrary secondary identifiers. The requirement is that these identifiers might be unequivocally converted into monotonous ones.

Another possible anchor to rely on is the record creation date. However, this approach is harder to implement for the following reasons:

- Creation dates for two records might be identical, especially if the records are mass-generated programmatically. In the worst-case scenario, it might happen that at some specific moment, more records were created than one request page contains making it impossible to traverse them.
- If the storage supports parallel writing to several nodes, the most recently created record might have a slightly earlier creation date than the second-recent one because clocks on different nodes might tick slightly differently, and it is challenging to achieve even microsecond-precision coherence[1]. This breaks the monotonicity invariant, which makes it poorly fit for use in public APIs. If there is no other choice but relying on such storage, one of two evils is to be chosen:
    - Introducing artificial delays, i.e., returning only items created earlier than N seconds ago, selecting this N to be certainly less than the clock irregularity. This technique also works in the case of asynchronously populated lists. Keep in mind, however, that this solution is probabilistic, and wrong data will be served to clients in case of backend synchronization problems.
    - Describe the instability of ordering list items in the docs (and thus make partners responsible for solving arising issues).

Often, the interfaces of traversing data through stating boundaries are generalized by introducing the concept of a "cursor":

```
// Initiate list traversal
POST /v1/partners/{id}/offers/history↵
  search
{
  "order_by": [{
    "field": "created",
    "direction": "desc"
  }]
}
→
{
  "cursor": "TmluZSBQcmluY2VzIGluIEFtYmVy"
}
```

```
// Get the next data chunk
GET /v1/partners/{id}/offers/history↵
  ?cursor=TmluZSBQcmluY2VzIGluIEFtYmVy↵
  &limit=100
→
{
  "items": […],
  // Pointer to the next data chunk
  "cursor": "R3VucyBvZiBBdmFsb24"
}
```

A *cursor* might be just an encoded identifier of the last record or it might comprise all the searching parameters. One advantage of using cursors instead of exposing raw monotonous fields is the possibility to change the underlying technology. For example, you might switch from using an auto-incremental key to using the date of the last known record's creation without breaking backward compatibility. (That's why cursors are usually opaque strings: providing readable cursors would mean that you now have to maintain the cursor format even if you never documented it. It's better to return cursors encrypted or at least coded in a form that will not arise the desire to decode it and experiment with parameters.)

The cursor-based approach also allows adding new filters and sorting directions in a backward-compatible manner — provided you organize the data in a way that cursor-based traversal will continue working.

```
// Initialize list traversal
POST /v1/partners/{id}/offers/history↵
  search
{
  // Add a filter by the recipe
  "filter": {
    "recipe": "americano"
  },
  // Add a new sorting mode
  // by the distance from some
  // location
  "order_by": [{
    "mode": "distance",
    "location": [-86.2, 39.8]
  }]
}
→
{
  "items": […],
  "cursor":
    "Q29mZmVlIGFuZCBDb250ZW1wbGF0aW9u"
}
```

A small footnote: sometimes, the absence of the next-page cursor in the response is used as a flag to signal that iterating is over and there are no more elements in the list. However, we would rather recommend not using this practice and always returning a cursor even if it points to an empty page. This approach allows for adding the functionality of dynamically inserting new items at the end of the list.

NB: In some articles, organizing list traversals through monotonous identifiers / creation dates / cursors is not recommended because it is impossible to show a page selection to the end user and allow them to choose the desired result page. However, we should consider the following:

- This case, of showing a pager and selecting a page, makes sense for end-user interfaces only. It's unlikely that an API would require access to random data pages.
- If we talk about the internal API for an application that provides the UI control element with a pager, the proper approach is to prepare the data for this control element on the server side, including generating links to pages.
- The boundary-based approach doesn't mean that using `limit`/`offset` parameters is prohibited. It is quite possible to have a double interface that would respond to both `GET /items?cursor=…` and `GET /items?offset=…&limit=…` queries.
- Finally, if the need to have access to an arbitrary data page in the UI exists, we need to ask ourselves a question: what is the user's problem that we're solving with this UI? Most likely, users *are searching* for something, such as a specific list item or where they were the last time they worked with the list. Specific UI control elements to help them will be likely more convenient than a pager.

The General Case

Unfortunately, it is not universally possible to organize the data in a way that would not require mutable lists. For example, we cannot paginate the list of ongoing orders consistently because orders change their status and randomly enter and leave this list. In these general scenarios, we need to focus on the *use cases* for accessing the data.

Sometimes, the task can be *reduced* to an immutable list if we create a snapshot of the data. In many cases, it is actually more convenient for partners to work with a snapshot that is current for a specific date as it eliminates the necessity of taking ongoing changes into account. This approach works well with accessing “cold” data storage by downloading chunks of data and putting them into “hot” storage upon request.

```
POST /v1/orders/archive/retrieve
{
  "created_iso_date": {
    "from": "1980-01-01",
    "to": "1990-01-01"
  }
}
→
{
  "task_id": <an identifier of
    a task to retrieve the data>
}
```

The disadvantage of this approach is also clear: it requires additional (sometimes quite considerable) computational resources to create and store a snapshot (and therefore requires a separate tariff). And we actually haven't solved the problem: though we don't expose the real-time traversal functionality in public APIs, we still need to implement it internally to be able to make a snapshot.

The inverse approach to the problem is to never provide more than one page of data, meaning that partners can only access the “newest” data chunk. This technique is viable in one of three cases:

- If the endpoint features a search algorithm that fetches the most relevant data. As we are well aware, nobody needs a second search result page.
- If the endpoint is needed to *modify* data. For example, the partner's service retrieves all “new” orders to transit them into the “accepted” status; then pagination is not needed at all as with each request the partner is *removing* items from the top of the list.
    - The important case for such modifications is marking the received data as “read”.
- Finally, if the endpoint is needed to access only real-time “raw” data while the processed and classified data are available through other interfaces.

If none of the approaches above works, our only solution is changing the subject area itself. If we can't consistently enumerate list elements, we need to find a facet of the same data that we *can* enumerate. In our example with the ongoing orders we might make an ordered list of the *events* of creating new orders:

```
// Retrieve all the events older
// than the one with the given id
GET /v1/orders/created-history↵
  ?older_than=<item_id>&limit=<limit>
→
{
  "orders_created_events": [{
    "id": <event id>,
    "occured_at",
    "order_id"
  }, …]
}
```

Events themselves and the order of their occurrence are immutable. Therefore, it's possible to organize traversing the list. It is important to note that the order creation event is not the order itself: when a partner reads an event, the order might have already changed its status. However, accessing *all* new orders is ultimately doable, although not in the most efficient manner.

NB: In the code samples above, we omitted passing metadata for responses, such as the number of items in the list, the `has_more_items` flag, etc. Although this metadata is not mandatory (i.e., clients will learn the list size when they retrieve it fully), having it makes working with the API more convenient for developers. Therefore we recommend adding it to responses.

Chapter 21. Bidirectional Data Flows. Push and Poll Models

In the previous chapter, we discussed the following scenario: a partner receives information about new events occuring in the system by periodically requesting an endpoint that supports retrieving ordered lists.

```
GET /v1/orders/created-history↵
  older_than=<item_id>&limit=<limit>
→
{
  "orders_created_events": [{
    "id",
    "occured_at",
    "order_id"
  }, …]
}
```

This pattern (known as *polling*) is the most common approach to organizing two-way communication in an API when a partner needs not only to send data to the server but also to receive notifications from the server about changes in some state.

Although this approach is quite easy to implement, polling always requires a compromise between responsiveness, performance, and system throughput:

- The longer the interval between consecutive requests, the greater the delay between the change of state on the server and receiving the information about it on the client, and the potentially larger the traffic volume that needs to be transmitted in one iteration.
- On the other hand, the shorter this interval, the more requests will be made in vain, as no changes in the system have occurred during the elapsed time.

In other words, polling always generates some background traffic in the system but never guarantees maximum responsiveness. Sometimes, this problem is solved by using the so-called "long polling," which intentionally delays the server's response for a prolonged period (seconds, tens of seconds) until some state change occurs. However, we do not recommend using this approach in modern systems due to associated technical problems, particularly in unreliable network conditions where the client has no way of knowing that the connection is lost, and a new request needs to be sent.

If regular polling is insufficient to solve the user's problem, you can switch to a reverse model (push) in which the server itself informs the client that changes have occurred in the system.

Although the problem and the ways to solve it may appear similar, completely different technologies are currently used to deliver messages from the backend to the backend and from the backend to the client device.

Delivering Notifications to Client Devices

As various mobile platforms currently constitute a major share of all client devices, this implies significant limitations in terms of battery and partly traffic savings on the technologies for data exchange between the server and the end user. Many platform and device manufacturers monitor the resources consumed by the application and can send it to the background or close open connections. In such a situation, frequent polling should only be used in active phases of the application work cycle (i.e., when the user is directly interacting with the UI) or in controlled environments (for example, if employees of a partner company use the application in their work and can add it to system exceptions).

Three alternatives to polling might be proposed:

1. Duplex Connections

The most obvious option is to use technologies that can transmit messages in both directions over a single connection. The best-known example of such technology is WebSocket. Sometimes, the Server Push functionality of the HTTP/2 protocol is used for this purpose; however, we must note that the specification formally does not allow such usage. There is also the WebRTC protocol; its main purpose is a peer-to-peer exchange of media data, and it's rarely used in client-server interaction.

Although the idea looks simple and attractive, its applicability to real-world use cases is limited. Popular server software and frameworks do not support server-initiated message sending (gRPC does support it, but the client should initiate the exchange; using gRPC server streams to send server-initiated events is essentially employing HTTP/2 server pushes for this purpose, and it's the same technique as in the long polling approach, just a bit more modern), and the existing specification definition standards do not support it — as WebSocket is a low-level protocol, and you will need to design the interaction format on your own.

Duplex connections still suffer from the unreliability of the network and require implementing additional tricks to tell the difference between a network problem and the absence of new messages. All these issues result in limited applicability of the technology; it's mostly used in web applications.

2. Separate Callback Channels

Instead of a duplex connection, two separate connections might be used: one for sending requests to the server and one to receive notifications from the server. The most popular technology of this kind is MQTT. Although it is considered very effective because of utilizing low-level protocols, its disadvantages follow from its advantages:

- The technology is meant to implement the pub/sub pattern, and its main value is that the server software (MQTT Broker) is provided alongside the protocol itself. Applying it to other tasks, especially bidirectional communication, might be challenging.

- The low-level protocols force you to develop your own data formats.

There is also a Web standard for sending server notifications called Server-Sent Events (SSE). However, it's less functional than WebSocket (only text data and unidirectional flow are allowed) and rarely used.

3. Third-Party Push Notifications

One of the notorious problems with the long polling / WebSocket / SSE / MQTT technologies is the necessity to maintain an open network connection between the client and the server, which might be a problem for mobile applications and IoT devices from in terms of performance and battery life. One option that allows for mitigating the issue is delegating sending push notifications to a third-party service (the most popular choice today is Google's Firebase Cloud Messaging) that delivers notifications through the built-in mechanisms of the platform. Using such integrated services takes most of the load of maintaining open connections and checking their status off the developer's shoulders. The disadvantages of using third-party services are the necessity to pay for them and strict limits on message sizes.

Also, sending push notifications to end-user devices suffers from one important issue: the percentage of successfully delivered messages never reaches 100%; the message drop rate might be tens of percent. Taking into account the message size limitations, it's actually better to implement a mixed model than a pure push model: the client continues polling the server, just less frequently, and push notifications just trigger ahead-of-time polling. (This problem is actually applicable to any notification delivery technology. Low-level protocols offer more options to set delivery guarantees; however, given the situation with forceful closing of open connections by OSes, having low-frequency polling as a precaution in an application is almost never a bad thing.)

Using Push Technologies in Public APIs

As a consequence of the fragmentation of client technologies described above, it's virtually impossible to use any of them but polling in public APIs. Requiring partners to implement receiving notifications through WebSocket, MQTT, or SSE channels raises the bar for adopting the API as working with low-level protocols, which are poorly covered by existing IDLs and code-generation tools, requires a significant amount of effort and is prone to implementation errors. If you decide to provide ready-to-use SDKs to ease working with the API, you will need to develop them for every applicable platform (which is, let us reiterate, quite labor-consuming). Given that HTTP polling is *much* easier to implement and its disadvantages play their role only in situations when one *really* needs to think about saving traffic and computational resources, we would rather recommend exposing additional channels for receiving server-sent notifications *as an addition* to polling, not instead of it.

Using platform pushes might be a fine solution for public APIs, but there another problem arises: application developers are not eager to allow other third-party services send push notifications, and that's for a list of reasons, starting with the costs of sending pushes and ending with security considerations.

In fact, the most convenient way of organizing message delivery from the public API backend to a partner service's user is by delivering messages backend-to-backend. This way, the partner service can relay it further using push notifications or any other technology that the partner selected for developing their applications.

Delivering Backend-to-Backend Notifications

Unlike client applications, server-side integrations universally utilize a single approach to implementing a bidirectional data flow, apart from polling (which is as applicable to server-to-server integrations as to client-server ones, and bears the same pros and cons). The approach is using a separate communication channel for callbacks. In the case of public APIs, the dominating practice is using callback URLs, also known as “webhooks.”

Although long polling, WebSocket, HTTP/2 Push, and other technologies discussed above are also applicable to realizing backend-to-backend communication, we find it difficult to name a popular API that utilizes any of them. We assume that the reasons for this are:

- Server-to-server integrations are less susceptible to performance issues (servers rarely hit any limits on network bandwidth, and keeping an open connection is not a problem as well)
- There are higher expectations regarding message delivery guarantees
- A broad choice of ready-to-use components to develop a *webhook* service (as it's basically a regular webserver) is available
- It is possible to have a specification covering the communication format and use the advantages of code-generation.

To integrate via a *webhook*, a partner specifies a URL of their own message processing server, and the API provider calls this endpoint to notify about status changes.

Let us imagine that in our coffee example the partner has a backend capable of processing newly created orders to be processed by partner's coffee shops, and we need to organize such communication. Realizing this task comprise several steps:

1. Negotiate a Contract

Depending on how important the partner is for our business, different options are possible:

- The API vendor might develop the functionality of calling the partner's *webhook* utilizing a protocol proposed by the partner
- Contrary to the previous, it's partner's job to develop an endpoint to support a format proposed by the API developers
- Any combination of the above

What is important is that the *must* be a formal contract (preferably in a form of a specification) for *webhook*'s request and response formats and all the errors that might happen.

2. Agree on Authorization and Authentication Methods

As a *webhook* is a callback channel, you will need to develop a separate authorization system to deal with it as it's *partners* duty to check that the request is genuinely coming from the API backend, not vice versa. We reiterate here our strictest recommendation to stick to existing standard techniques, for example, mTLS; though in the real world, you will likely have to use archaic methods like fixing the caller server's IP address.

3. Develop an Interface for Setting the URL of a *Webhook*

As the callback endpoint is developed by partners, we do not know its URL beforehand. It implies some interface must exist for setting this URL and authorized public keys (probably in a form of a control panel for partners).

Importantly, the operation of setting a *webhook* URL is to be treated as a potentially hazardous one. It is highly desirable to request a second authentication factor to authorize the operations as a potential attacker wreak a lot of havoc if there is a vulnerability in the procedure:

- By setting an arbitrary URL, the perpetrator might get access to all partner's orders (and the partner might lose access)
- This vulnerability might be used for organizing DoS attacks on third parties
- If an internal URL might be set as a *webhook*, a SSRF attack might be directed toward the API vendor's own infrastructure.

Typical Problems of *Webhook*-Powered Integrations

Bidirectional data flows (both client-server and server-server ones, though the latter to a greater extent) bear quite undesirable risks for an API provider. In general, the quality of integration primarily depends on the API developers. In the callback-based integration, it's vice versa: the integration quality depends on how partners implemented the *webhook*. We might face numerous problems with the partners' code:

- *Webhook* might return false-positive responses meaning the notification was not actually processed but the success status was returned by the partner's server
- On other hand, false-negative responses are also possible if the operation was actually accepted but erroneously returned an error (or just responded in invalid format)
- *Webhook* might be processing incoming requests very slowly — up to a point when the requesting server will be just unable to deliver subsequent messages on time
- Partner's developers might make a mistake in implementing the idempotency policies, and repeated requests to the *webhook* will lead to errors or data inconsistency on the partner's side
- The size of the message body might exceed the limit set in the partner's webserver configuration
- On the partner's side, authentication token checking might be missing or flawed so some malefactor might be able to issue requests pretending they come from the genuine API server
- Finally, the endpoint might simply be unavailable because of many reasons, starting from technical issues in the data center where partner's servers are located and ending with a human error in setting *webhook*'s URL.

Obviously, we can't guarantee partners don't make any of these mistakes. The only thing we *can* do is to minimize the impact radius:

1. The system state must be restorable. If the partner erroneously responded that messages are processed while they are not, there must be a possibility for them to redeem themselves and get the list of missed events and/or the full system state and fix all the issues
2. Help partners to write proper code by describing in the documentation all unobvious subtleties that inexperienced developers might be unaware of:
    - Idempotency keys for every operation
    - Delivery guarantees (“at least once,” “exactly ones,” etc.; see the reference description on the example of Apache Kafka API)

    - Possibility of the server generating parallel requests and the maximum number of such requests at a time
    - Guarantees of message ordering (i.e., the notifications are always delivered ordered from the oldest one to the newest one) or the absence of such guarantees
    - The sizes of all messages and message fields in bytes
    - The retry policy in case an error is returned by the partner's server
3. Implement a monitoring system to check the health of partners' endpoints:
    - If a large number of errors or timeouts occurs, it must be escalated (including notifying the partner about the problem), probably with several escalation tiers,
    - If too many un-processed notifications are stuck, there must be a mechanism of controllable degradation (limiting the number of requests toward the partner, e.g. cutting the demand by disallowing some users to make an order) up to fully disconnecting the partner from the platform.

Message Queues

As for internal APIs, the *webhook* technology (i.e., the possibility to programmatically define a callback URL) is either not needed at all or is replaced with the Service Discovery protocol as services comprising a single backend are symmetrically able to call each other. However, the problems of callback-based integration discussed above are equally actual for internal calls. Requesting an internal API might result in a false-negative mistake, internal clients might be unaware that ordering is not guaranteed, etc.

To solve these problems, and also to ensure better horizontal scalability, message queues were developed, most notably numerous pub/sub pattern implementations. At present moment, pub/sub-based architectures are very popular in enterprise software development, up to switching any inter-service communication to message queues.

NB: Let us note that everything comes with a price, and these delivery guarantees and horizontal scalability are not an exclusion:

- All communication becomes eventually consistent with all the implications
- Decent horizontal scalability and cheap message queue usage are only achievable with at least once/at most once policies and no ordering guarantee
- Queues might accumulate unprocessed events, introducing increasing delays, and solving this issue on the subscriber's side might be quite non-trivial.

Also, in public APIs both technologies are frequently used in conjunction: the API backend sends a task to call the *webhook* in the form of publishing an event which the specially designed internal service will try to process by making the call.

Theoretically, we can imagine an integration that exposes directly accessible message queues in one of the standard formats for partners to subscribe. However, we are unaware of any examples of such APIs.

Chapter 22. Multiplexing Notifications. Asynchronous Event Processing

One of the vexing restrictions of almost every technology mentioned in the previous chapter is the limited size of messages. With client push notifications the situation is the most problematic: Google Firebase Messaging at the moment this chapter is being written allowed no more than 4000 bytes of payload. In backend development, the restrictions are also notable; let's say, Amazon SQS limits the size of messages to 256 KiB. While developing *webhook*-based integrations, you risk hitting the maximum body size allowed by the partner's webserver (for example, in nginx the default value is 1MB). This leads us to the necessity of making two technical decisions regarding the notification formats:

- Whether a message contains all data needed to process it or just notifies some state change has happened
- If we choose the latter, whether a single notification contains data on a single change, or it might bear several such events.

On the example of our coffee API:

```
// Option #1: the message
// contains all the order data
POST /partner/webhook
Host: partners.host
{
  "event_id",
  "occurred_at",
  "order": {
    "id",
    "status",
    "recipe_id",
    "volume",
    // Other data fields
    …
  }
}
```

```
// Option #2: the message body
// contains only the notification
// of the status change
POST /partner/webhook
Host: partners.host
{
  "event_id",
  // Message type: a notification
  // about a new order
  "event_type": "new_order",
  "occurred_at",
  // Data sufficient to
  // retrieve the full state,
  // in our case, the order identifier
  "order_id"
}
// To process the event, the partner
// must request some endpoint
// on the API vendor's side,
// possibly asynchronously
GET /v1/orders/{id}
→
{ /* full data regarding
     the order */ }
```

```
// Option #3: the API vendor
// notifies partners that
// several orders await their
// reaction
POST /partner/webhook
Host: partners.host
{
  // The system state revision
  // and/or a cursor to retrieve
  // the orders might be provided
  "occurred_at",
  "pending_order_count":
    <the number of pending orders>
}
// In response to such a call,
// partners should retrieve the list
// of ongoing orders
GET /v1/orders/pending
→
{
  "orders",
  "cursor"
}
```

Which option to select depends on the subject area (and on the allowed message sizes in particular) and on the procedure of handling messages by partners. In our case, every order must be processed independently and the number of messages during the order life cycle is low, so our natural choice would be either option #1 (if order data cannot contain unpredictably large fields) or #2. Option #3 is viable if:

- The API generates a lot of notifications for a single logical entity
- Partners are interested in fresh state changes only
- Or events must be processed sequentially, and no parallelism is allowed.

NB: The approach #3 (and partly #2) naturally leads us to the scheme that is typical for client-server integration: the push message itself contains almost no data and is only a trigger for ahead-of-time polling.

The technique of sending only essential data in the notification has one important disadvantage, apart from more complicated data flows and increased request rate. With option #1 implemented (i.e., the message contains all the data), we might assume that returning a success response by the subscriber is equivalent to successfully processing the state change by the partner (although it's not guaranteed if the partner uses asynchronous techniques). With options #2 and #3, this is certainly not the case: the partner must carry out additional actions (starting from retrieving the actual order state) to fully process the message. This implies that two separate statuses might be needed: “message received” and “message processed.” Ideally, the latter should follow the logic of the API work cycle, i.e., the partner should carry out some follow-up action upon processing the event, and this action might be treated as the “message processed” signal. In our coffee example, we can expect that the partner will either accept or reject an order after receiving the “new order” message. Then the full message processing flow will look like this:

```
// The API vendor
// notifies the partner that
// several orders await their
// reaction
POST /partner/webhook
Host: partners.host
{
  "occurred_at",
  "pending_order_count":
    <the number of pending orders>
}
```

```
// In response, the partner
// retrieves the list of
// pending orders
GET /v1/orders/pending
→
{
  "orders",
  "cursor"
}
```

```
// After the orders are processed,
// the partners notify about this
// by calling the specific API
// endpoint
POST /v1/orders/bulk-status-change
{
  "status_changes": [{
    "order_id",
    "new_status": "accepted",
    // Other relevant information
    // e.g. the preparation time
    // estimates
    …
  }, {
    "order_id",
    "new_status": "rejected",
    "reason"
  }, …]
}
```

If there is no genuine follow-up call expected during our API work cycle, we can introduce an endpoint to explicitly mark notifications as processed. This step is not mandatory as we can always stipulate that it is the partner's responsibility to process notifications and we do not expect any confirmations. However, we will lose an important monitoring tool if we do so, as we can no longer track what's happening on the partner's side, i.e., whether the partner is able to process

notifications on time. This, in turn, will make it harder to develop the degradation and emergency shutdown mechanisms we talked about in the previous chapter.

Chapter 23. Atomicity of Bulk Changes

Let's transition from *webhooks* back to developing direct-call APIs. The design of the `orders/bulk-status-change` endpoint, as described in the previous chapter, raises an interesting question: what should we do if some changes were successfully processed by our backend while others were not?

Let's consider a scenario where the partner notifies us about status changes that have occurred for two orders:

```
POST /v1/orders/bulk-status-change
{
  "status_changes": [{
    "order_id": "1",
    "new_status": "accepted",
    // Other relevant data,
    // such as estimated
    // preparation time
    …
  }, {
    "order_id": "2",
    "new_status": "rejected",
    "reason"
  }]
}
→
500 Internal Server Error
```

In this case, if changing the status of one order results in an error, how should we organize this "umbrella" endpoint (which acts as a proxy to process a list of nested sub-requests)? We can propose at least four different options:

- A. Guarantee atomicity and idempotency. If any of the sub-requests fail, none of the changes are applied.

- B. Guarantee idempotency but not atomicity. If some sub-requests fail, repeating the call with the same idempotency key results in no action and leaves the system exactly in the same state (i.e., unsuccessful calls will never be executed, even if the obstacles are resolved, until a new call with a new idempotency key is made).
- C. Guarantee neither idempotency nor atomicity and process the sub-requests independently.
- D. Do not guarantee atomicity and completely prohibit retries by requiring the inclusion of the actual resource revision in the request (see the "Synchronization Strategies" chapter).

From a general standpoint, it appears that the first option is most suitable for public APIs: if you can guarantee atomicity (despite it potentially poses scalability challenges), it is advisable to do so. In the first revision of this book, we unconditionally recommended adhering to this solution.

However, if we consider the situation from the partner's perspective, we realize that the decision is not as straightforward as one might initially think. Let's imagine that the partner has implemented the following functionality:

1. The partner's backend processes notifications about incoming orders through a *webhook*.
2. The backend makes inquiries to coffee shops regarding whether they can fulfill the orders.
3. Periodically, let's say once every 10 seconds, the partner collects all the status changes (i.e., responses from the coffee shops) and calls the `bulk-status-change` endpoint with the list of changes.

Now, let's consider a scenario where the partner receives an error from the API endpoint during the third step. What would developers do in such a situation? Most probably, one of the following solutions might be implemented in the partner's code:

1. Unconditional retry of the request:

```
// Retrieve the ongoing orders
const pendingOrders = await api
  .getPendingOrders();
// The partner checks the status
// of every order in its system
// and prepares the list of
// changes to perform
const changes =
  await prepareStatusChanges(
    pendingOrders
  );

let result;
let tryNo = 0;
let timeout =
  DEFAULT_RETRY_TIMEOUT;
while (
  result &&
  tryNo++ < MAX_RETRIES
) {
  try {
    // Send the list
    // of changes
    result = await api
      .bulkStatusChange(
        changes,
        // Provide the newest
        // known revision
        pendingOrders.revision
      );
  } catch (e) {
    // If there is an error,
    // repeat the request
    // with some delay
    logger.error(e);
    await wait(timeout);
    timeout = min(
      timeout * 2,
      MAX_TIMEOUT
    );
  }
}
```

NB: In the code sample above, we provide the “right” retry policy with exponentially increasing delays and a total limit on the number of retries, as we recommended earlier in the “Describing Final Interfaces” chapter. However, be warned that real partners' code may frequently lack such precautions. For the sake of readability, we will skip this bulky construct in the following code samples.

2. Retrying only failed sub-requests:

```
const pendingOrders = await api
  .getPendingOrders();
let changes =
  await prepareStatusChanges(
    pendingOrders
  );

let result;
while (changes.length) {
  let failedChanges = [];
  try {
    result = await api
      .bulkStatusChange(
        changes,
        pendingOrders.revision
      );
  } catch (e) {
    let i = 0;
    // Assuming that the `e.changes`
    // field contains the errors
    // breakdown
    for (
      i < e.changes.length; i++
    ) {
      if (e.changes[i].status ==
        'failed') {
        failedChanges.push(
          changes[i]
        );
      }
    }
  }
  // Prepare a new request
  // comprising only the failed
  // sub-requests
  changes = failedChanges;
}
```

3. Restarting the entire pipeline. In this case, the partner retrieves the list of pending orders anew and forms a new bulk change request:

```
do {
  const pendingOrders = await api
    .getPendingOrders();
  const changes =
    await prepareStatusChanges(
      pendingOrders
    );
  // Request changes,
  // if there are any
  if (changes.length) {
    await api.bulkStatusChange(
      changes,
      pendingOrders.revision
    );
  }
} while (pendingOrders.length);
```

If we examine the possible combinations of client and server implementation options, we will discover that approaches (B) and (D) are incompatible with solution (1). Retrying the same request after a partial failure will never succeed, and the server will repeatedly attempt the failing request until it exhausts the remaining retry attempts.

Now, let's introduce another crucial condition to the problem statement: imagine that certain issues with a sub-request can not be resolved by retrying it. For example, if the partner attempts to confirm an order that has already been canceled by the customer. If a bulk status change request contains such a sub-request, the atomic server that implements paradigm (A) will immediately “penalize” the partner. Regardless of how many times and in what order the set of sub-requests is repeated, *valid sub-requests will never be executed if there is even a single invalid one*. On the other hand, a non-atomic server will at least continue processing the valid parts of bulk requests.

This leads us to a seemingly paradoxical conclusion: in order to ensure the partners' code continues to function *somehow* and to allow them time to address their invalid sub-requests we should adopt the least strict non-idempotent non-atomic approach to the design of the bulk state change endpoint. However, we consider this conclusion to be incorrect: the “zoo” of possible client and server implementations and the associated problems demonstrate that *bulk state change endpoints are inherently undesirable*. Such endpoints require maintaining an

additional layer of logic in both server and client code, and the logic itself is quite non-obvious. The non-atomic non-idempotent bulk state changes will very soon result in nasty issues:

```
// A partner issues a refund
// and cancels the order
POST /v1/bulk-status-change
{
  "changes": [{
    "operation": "refund",
    "order_id"
  }, {
    "operation": "cancel",
    "order_id"
  }]
}
→
// During bulk change execution,
// the user was able to walk in
// and fetch the order
{
  "changes": [{
    // The refund is successful…
    "status": "success"
  }, {
    // …while canceling the order
    // is not
    "status": "fail",
    "reason": "already_served"
  }]
}
```

If sub-operations in the list depend on each other (as in the example above: the partner needs *both* refunding and canceling the order to succeed as there is no sense to fulfill only one of them) or the execution order is important, non-atomic endpoints will constantly lead to new problems. And if you think that in your subject area, there are no such problems, it might turn out at any moment that you have overlooked something.

So, our recommendations for bulk modifying endpoints are:

1. If you can avoid creating such endpoints — do it. In server-to-server integrations, the profit is marginal. In modern networks that support QUIC and request multiplexing, it's also dubious.
2. If you can not, make the endpoint atomic and provide SDKs to help partners avoid typical mistakes.
3. If implementing an atomic endpoint is not possible, elaborate on the API design thoroughly, keeping in mind the caveats we discussed.
4. Whichever option you choose, it is crucially important to include a breakdown of the sub-requests in the response. For atomic endpoints, this entails ensuring that the error message contains a list of errors that prevented the request execution, ideally encompassing the potential errors as well (i.e., the results of validity checks for all the sub-requests). For non-atomic endpoints, it means returning a list of statuses corresponding to each sub-request along with errors that occurred during the execution.

One of the approaches that helps minimize potential issues is developing a “mixed” endpoint, in which the operations that can affect each other are grouped:

```
POST /v1/bulk-status-change
{
  "changes": [{
    "order_id": <first id>
    // Operations related
    // to a specific endpoint
    // are grouped in a single
    // structure and executed
    // atomically
    "operations": [
      "refund",
      "cancel"
    ]
  }, {
    // Operation sets for
    // different orders might
    // be executed in parallel
    // and non-atomically
    "order_id": <second id>
    …
  }]
}
```

Let us also stress that nested operations (or sets of operations) must be idempotent per se. If they are not, you need to somehow deterministically generate internal idempotency tokens for each operation. The simplest approach is to consider the internal token equal to the external one if it is possible within the subject area. Otherwise, you will need to employ some constructed tokens — in our case, let's say, in the `<order_id>:<external_token>` form.

Chapter 24. Partial Updates

The case of partial application of the list of changes described in the previous chapter naturally leads us to the next typical API design problem. What if the operation involves a low-level overwriting of several data fields rather than an atomic idempotent procedure (as in the case of changing the order status)? Let's take a look at the following example:

```
// Creates an order
// consisting of two beverages
POST /v1/orders/
X-Idempotency-Token: <token>
{
  "delivery_address",
  "items": [{
    "recipe": "lungo"
  }, {
    "recipe": "latte",
    "milk_type": "oat"
  }]
}
→
{ "order_id" }
```

```
// Partially updates the order
// by changing the volume
// of the second beverage
PATCH /v1/orders/{id}
{
  "items": [
    // `null` indicates
    // no changes for the
    // first beverage
    null,
    // list of properties
    // to change for
    // the second beverage
    {"volume": "800ml"}
  ]
}
→
{ /* Changes accepted */ }
```

This signature is inherently flawed as its readability is dubious. What does the empty first element in the array mean, deletion of an element or absence of changes? What will happen with fields that are not passed (`delivery_address`, `milk_type`)? Will they reset to default values or remain unchanged?

The most notorious thing here is that no matter which option you choose, your problems have just begun. Let's say we agree that the `"items":[null, {…}]` construct means the first array element remains untouched. So how do we delete it if needed? Do we invent another "nullish" value specifically to denote removal? The same issue applies to field values: if skipping a field in a request means it should remain unchanged, then how do we reset it to the default value?

Partially updating a resource is one of the most frequent tasks that API developers have to solve, and unfortunately, it is also one of the most complicated. Attempts to take shortcuts and simplify the implementation often lead to numerous problems in the future.

A **trivial solution** is to always overwrite the requested entity completely, which means requiring the passing of the entire object to fully replace the current state and return the new one. However, this simple solution is frequently dismissed due to several reasons:

- Increased request sizes and, consequently, higher traffic consumption
- The necessity to detect which fields were actually changed in order to generate proper signals (events) for change listeners
- The inability to facilitate collaborative editing of the object, meaning allowing two clients to edit different properties of the object in parallel as clients send the full object state as they know it and overwrite each other's changes as they are unaware of them.

To avoid these issues, developers sometimes implement a **naïve solution**:

- Clients only pass the fields that have changed
- To reset the values of certain fields and to delete or skip array elements some “special” values are used.

A full example of an API implementing the naïve approach would look like this:

```
// Partially rewrites the order:
//   * Resets the delivery address
//     to the default values
//   * Leaves the first beverage
//     intact
//   * Removes the second beverage.
PATCH /v1/orders/{id}
{
  // “Special” value #1:
  // reset the field
  "delivery_address": null
  "items": [
    // “Special” value #2:
    // do nothing to the entity
    {},
    // “Special” value #3:
    // delete the entity
    false
  ]
}
```

This solution allegedly solves the aforementioned problems:

- Traffic consumption is reduced as only the changed fields are transmitted, and unchanged entities are fully omitted (in our case, replaced with the special value `{}`).
- Notifications regarding state changes will only be generated for the fields and entities passed in the request.
- If two clients edit different fields, no access conflict is generated and both sets of changes are applied.

However, upon closer examination all these conclusions seem less viable:

- We have already described the reasons for increased traffic consumption (excessive polling, lack of pagination and/or field size restrictions) in the “Describing Final Interfaces” chapter, and these issues have nothing to do with passing extra fields (and if

they do, it implies that a separate endpoint for “heavy” data is needed).

- The concept of passing only the fields that have actually changed shifts the burden of detecting which fields have changed onto the client developers' shoulders:
    - Not only does the complexity of implementing the comparison algorithm remain unchanged but we also run the risk of having several independent realizations.
    - The capability of the client to calculate these diffs doesn't relieve the server developers of the duty to do the same as client developers might make mistakes or overlook certain aspects.
- Finally, the naïve approach of organizing collaborative editing by allowing conflicting operations to be carried out if they don't touch the same fields works only if the changes are transitive. In our case, they are not: the result of simultaneously removing the first element in the list and editing the second one depends on the execution order.
    - Often, developers try to reduce the outgoing traffic volume as well by returning an empty server response for modifying operations. Therefore, two clients editing the same entity do not see the changes made by each other until they explicitly refresh the state, which further increases the chance of yielding highly unexpected results.

The solution could be enhanced by introducing explicit control sequences instead of relying on “magical” values and adding meta settings for the operation (such as a field name filter as it's implemented in gRPC). Here's an example:

```
// Partially rewrites the order:
//   * Resets the delivery address
//     to the default values
//   * Leaves the first beverage
//     intact
//   * Removes the second beverage.
PATCH /v1/orders/{id}?↵
  // A meta filter: which fields
  // are allowed to be modified
  field_mask=delivery_address,items
{
  // “Special” value #1:
  // reset the field
  "delivery_address": {
    // The `__` prefix is needed
    // to avoid collisions
    // with real field names
    "__operation": "reset"
  },
  "items": [
    // “Special” value #2:
    // do nothing to the entity
    { "__operation": "skip" },
    // “Special” value #3:
    // delete the entity
    { "__operation": "delete" }
  ]
}
```

While this approach may appear more robust, it doesn't fundamentally address the problems:

- “Magical” values are replaced with “magical” prefixes
- The fragmentation of algorithms and the non-transitivity of operations persist.

Given that the format becomes more complex and less intuitively understandable, we consider this enhancement dubious.

A **more consistent solution** is to split an endpoint into several idempotent sub-endpoints, each having its own independent identifier and/or address (which is usually enough to ensure the transitivity of independent operations). This approach aligns well with the decomposition principle we discussed in the "Isolating Responsibility Areas" chapter.

```
// Creates an order
// comprising two beverages
POST /v1/orders/
{
  "parameters": {
    "delivery_address"
  },
  "items": [{
    "recipe": "lungo"
  }, {
    "recipe": "latte",
    "milk_type": "oats"
  }]
}
→
{
  "order_id",
  "created_at",
  "parameters": {
    "delivery_address"
  },
  "items": [
    { "item_id", "status"},
    { "item_id", "status"}
  ]
}
```

```
// Changes the parameters
// of the second order
PUT /v1/orders/{id}/parameters
{ "delivery_address" }
→
{ "delivery_address" }
```

```
// Partially changes the order
// by rewriting the parameters
// of the second beverage
PUT /v1/orders/{id}/items/{item_id}
{
  // All the fields are passed,
  // even if only one has changed
  "recipe", "volume", "milk_type"
}
→
{ "recipe", "volume", "milk_type" }
```

```
// Deletes one of the beverages
DELETE /v1/orders/{id}/items/{item_id}
```

Now to reset the `volume` field it is enough *not* to pass it in the `PUT items/{item_id}`. Also note that the operations of removing one beverage and editing another one became transitive.

This approach also allows for separating read-only and calculated fields (such as `created_at` and `status`) from the editable ones without creating ambivalent situations (such as what should happen if the client tries to modify the `created_at` field).

Applying this pattern is typically sufficient for most APIs that manipulate composite entities. However, it comes with a price as it sets high standards for designing the decomposed interfaces (otherwise a once neat API will crumble with further API expansion) and the necessity to make many requests to replace a significant subset of the entity's fields (which implies exposing the functionality of applying bulk changes, the undesirability of which we discussed in the previous chapter).

NB: While decomposing endpoints, it's tempting to split editable and read-only data. Then the latter might be cached for a long time and there will be no need for sophisticated list iteration techniques. The plan looks great on paper; however, with API expansion, immutable data often ceases to be immutable which is only solvable by creating new versions of the interfaces. We recommend explicitly pronouncing some data non-modifiable in one of the following two cases: either (1) it really cannot become editable without breaking backward compatibility or (2) the reference to the resource (such as, let's say, a link to an image) is fetched via the API itself and you can make these links persistent (i.e., if the image is updated, a new link is generated instead of overwriting the content the old one points to).

Resolving Conflicts of Collaborative Editing

The idea of applying changes to a resource state through independent atomic idempotent operations looks attractive as a conflict resolution technique as well. As subcomponents of the resource are fully overwritten, it is guaranteed that the result of applying the changes will be exactly what the user saw on the screen of their device, even if they had observed an outdated version of the resource. However, this approach helps very little if we need a high granularity of data editing as it's implemented in modern services for collaborative document editing and version control systems (as we will need to implement endpoints with the same level of granularity, literally one for each symbol in the document).

To make true collaborative editing possible, a specifically designed format for describing changes needs to be implemented. It must allow for:

- Ensuring the maximum granularity (each operation corresponds to one distinct user's action)
- Implementing conflict resolution policies.

In our case, we might take this direction:

```
POST /v1/order/changes
X-Idempotency-Token: <token>
{
  // The revision the client
  // observed when making
  // the changes
  "known_revision",
  "changes": [{
    "type": "set",
    "field": "delivery_address",
    "value": <new value>
  }, {
    "type": "unset_item_field",
    "item_id",
    "field": "volume"
  }],
  …
}
```

This approach is much more complex to implement, but it is the only viable technique for realizing collaborative editing as it explicitly reflects the exact actions the client applied to an entity. Having the changes in this format also allows for organizing offline editing with accumulating changes on the client side for the server to resolve the conflict later based on the revision history.

NB: One approach to this task is developing a set of operations in which all actions are transitive (i.e., the final state of the entity does not change regardless of the order in which the changes were applied). One example of such a nomenclature is CRDT. However, we consider this approach viable only in some subject areas, as in real life, non-transitive changes are always possible. If one user entered new text in the document and another user removed the document completely, there is no way to automatically resolve this conflict that would satisfy both users. The only correct way of resolving this conflict is explicitly asking users which option for mitigating the issue they prefer.

Chapter 25. Degradation and Predictability

In the previous chapters, we repeatedly discussed that the background level of errors is not just unavoidable, but in many cases, APIs are deliberately designed to tolerate errors to make the system more scalable and predictable.

But let's ask ourselves a question: what does a “more predictable system” mean? For an API vendor, the answer is simple: the distribution and number of errors are both indicators of technical problems (if the numbers are growing unexpectedly) and KPIs for technical refactoring (if the numbers are decreasing after the release).

However, for partner developers, the concept of “API predictability” means something completely different: how solidly they can cover the API use cases (both happy and unhappy paths) in their code. In other words, how well one can understand based on the documentation and the nomenclature of API methods what errors might arise during the API work cycle and how to handle them.

Why is optimistic concurrency control better than acquiring locks from the partner's point of view? Because if the revision conflict error is received, it's obvious to a developer what to do about it: update the state and try again (the easiest approach is to show the new state to the end user and ask them what to do next). But if the developer can't acquire a lock in a reasonable time then... what useful action can they take? Retrying most certainly won't change anything. Show something to the user... but what exactly? An endless spinner? Ask the user to make a decision — give up or wait a bit longer?

While designing the API behavior, it's extremely important to imagine yourself in the partner developer's shoes and consider the code they must write to solve the arising issues (including timeouts and backend unavailability). This book comprises many specific tips on typical problems; however, you need to think about atypical ones on your own.

Here are some general pieces of advice that might come in handy:

- If you can include recommendations on resolving the error in the error response itself, do it unconditionally (but keep in mind there should be two sets of recommendations, one for the user who will see the message in the application and one for the developer who will find it in the logs)
- If errors emitted by some endpoint are not critical for the main functionality of the integration, explicitly describe this fact in the documentation. Developers may not guess to wrap the corresponding code in a `try-catch` block. Providing code samples and guidance on what default value or behavior to use in case of an error is even better.
- Remember that no matter how exquisite and comprehensive your error nomenclature is, a developer can always encounter a transport-level error or a network timeout, which means they need to restore the application state when the tips from the backend are not available. There should be an obvious default sequence of steps to handle unknown problems.
- Finally, when introducing new types of errors, don't forget about old clients that are unaware of these new errors. The aforementioned "default reaction" to obscure issues should cover these new scenarios.

In an ideal world, to help partners "degrade properly," a meta-API should exist, allowing for determining the status of the endpoints of the main API. This way, partners would be able to automatically enable fallbacks if some functionality is unavailable. In the real world, alas, if a widespread outage occurs, APIs for checking the status of APIs are commonly unavailable as well.

SECTION III. THE BACKWARD COMPATIBILITY

Chapter 26. The Backward Compatibility Problem Statement

As usual, let's conceptually define “backward compatibility” before we start.

Backward compatibility is a feature of the entire API system to be stable in time. It means the following: **the code that developers have written using your API continues to work functionally correctly for a long period of time**. There are two important questions regarding this definition and two explanations:

1. What does “functionally correctly” mean?

   It means that the code continues to serve its intended function which is to solve users' problems. It does not necessarily mean that it continues to work indistinguishably from the previous version. For example, if you're maintaining a UI library, making functionally insignificant design changes like adjusting shadow depth or border stroke type would still be considered backward-compatible unlike changing the sizes of the visual components.

2. What does “a long period of time” mean?

   From our point of view, the backward compatibility maintenance period should be aligned with the typical lifetime of applications in the subject area. Platform LTS (Long-Term Support) periods can serve as helpful guidelines in most cases. Since applications will be rewritten when the platform maintenance period ends, it is reasonable to expect developers to transition to the new API version as well. In mainstream subject areas such as desktop and mobile operating systems, this period typically spans several years.

The definition makes it evident why maintaining backward compatibility is crucial, including taking necessary measures at the API design stage. An outage, whether full or partial, caused by an API vendor, is an extremely inconvenient situation for every developer, if not a disaster —

especially if they are paying for the API usage.

But let's take a look at the problem from another angle: why does the problem of maintaining backward compatibility exist in the first place? Why would anyone *want* to break it? This question, though it may seem trivial, is much more complicated than the previous one.

We could say that *we break backward compatibility to introduce new features to the API*. However, this would be deceiving since new features are called *"new"* for a reason — they cannot affect existing implementations that do not use them. We must admit that there are several associated problems that lead to the aspiration to rewrite *our* code, the code of the API itself, and ship a new major version:

- The codebase eventually becomes outdated making it impractical to introduce changes or even new functionality
- The old interfaces are not suited to accommodate new features. We would love to extend existing functionality with new properties, but we simply cannot
- Finally, with years passing since the initial release, we have gained a better understanding of the subject area and API best practices. We would implement many things differently now.

These arguments can be summarized frankly as "API vendors do not want to support old code." However, this explanation is still incomplete. If you're not planning to rewrite the API code to add new functionality or even if you're not planning to add it at all, you still need to release new API versions, both minor and major.

NB: In this chapter, we don't make any difference between minor versions and patches. "Minor version" means any backward-compatible API release.

Let us remind the reader that an API is a bridge, a means of connecting different programmable contexts. No matter how strong our desire is to keep the bridge intact, our capabilities are limited: we can lock the bridge, but we cannot command the rifts and the canyon itself. That's the source of the problem: we can't guarantee that *our own* code won't

change. So at some point, we will have to ask the clients to rewrite *their* code.

Apart from our aspirations to change the API architecture, three other tectonic processes are happening at the same time: user agents, subject areas, and the erosion of underlying platforms.

1. The Fragmentation of Consumer Applications

When you shipped the very first API version, and the initial clients started using it, the situation was perfect. However, this perfection doesn't last, and two scenarios are possible.

1. If the platform allows for fetching code on-demand, like the good old Web does, and you weren't too lazy to implement that code-on-demand feature (in the form of a platform SDK, such as JS API), then the evolution of your API is more or less under your control. Maintaining backward compatibility effectively means keeping *the client library* backward-compatible. As for client-server interaction, you have freedom.

   This doesn't mean that you can't break backward compatibility. You can still make a mistake with cache-control headers or simply overlook a bug in the code. Additionally, even code-on-demand systems don't get updated instantly. The author of this book faced a situation where users deliberately kept a browser tab open *for weeks* to avoid updates. However, in general, you usually don't have to support more than two API versions — the latest one and the penultimate one. Furthermore, you may consider rewriting the previous major version of the library, implementing it on top of the actual API version.

2. If the code-on-demand feature isn't supported or is prohibited by the platform, as is the case with modern mobile operating systems, the situation becomes more severe. Each client effectively borrows a snapshot of the code that works with your API, frozen at the moment of compilation. Client application updates are scattered over time to a much greater extent than Web

application updates. The most painful aspect is that *some clients will never be up to date*, due to one of three reasons:

- Developers simply don't want to update the app, i.e., its development has stopped.
- Users do not want to get updates (sometimes because they believe that developers “spoiled” the app in new versions)
- Users cannot get updates because their devices are no longer supported.

In modern times these three categories combined could easily constitute a significant portion (tens of percent) of the audience. This implies that discontinuing support for any API version could be a nightmare experience — especially if partners' apps continue supporting a broader range of platforms than the API does.

If you have never issued an SDK, providing only server-side APIs, for example in the form of HTTP endpoints, you might think that the backward compatibility problem is mitigated, although your API is less competitive on the market due to the lack of SDKs. However, that's not what will happen. If you don't provide an SDK, developers will either adopt an unofficial one (if someone bothered to create it) or write a framework themselves, independently. The “your framework — your problems” strategy, fortunately or unfortunately, works poorly. If developers write low-quality code on top of your API, then your API itself is of low quality — definitely in the view of developers and possibly in the view of end-users if the API's performance within the app is visible to them.

Certainly, if you provide stateless APIs that don't require client SDKs or can be auto-generated from the spec, these problems will be much less noticeable. However, they are not fully avoidable unless you never issue any new API versions. If you do, you will still have to deal with some fragmentation of users by API and SDK versions.

2. Subject Area Evolution

The other side of the canyon is the underlying functionality that you expose via the API. It is, of course, not static and evolves in the following ways:

- New functionality emerges
- Older functionality shuts down
- Interfaces change.

As usual, the API provides an abstraction to a much more granular subject area. In the case of our coffee machine API example, one might reasonably expect new machine models to emerge, which will need to be supported by the platform. New models often come with new APIs, making it challenging to ensure their adoption while preserving the same high-level API. In any case, the API's code needs to be altered, which may lead to incompatibility, albeit unintentionally.

Let us also emphasize that vendors of low-level APIs are not always as committed to maintaining backward compatibility for their APIs (or any software they provide) as we hope you are. It is important to be aware that keeping your API in an operational state, which involves writing and supporting facades to the shifting subject area landscape, will be your responsibility, sometimes posing quite sudden challenges.

3. Platform Drift

Finally, there is a third aspect to consider — the “canyon” you are crossing over with a bridge of your API. Developers write code that is executed in an environment beyond your control, and it evolves. New versions of operating systems, browsers, protocols, and programming language SDKs emerge. New standards are being developed and new arrangements are made, some of which are backward-incompatible, and there is nothing that can be done about that.

Older platform versions contribute to fragmentation just like older app versions as developers (including the API developers) struggle to support older platforms. At the same time, users face challenges with platform updates. In many cases, they are unable to update their devices

to newer platform versions since newer platform versions require newer devices.

The most challenging aspect here is that not only does incremental progress, in the form of new platforms and protocols, necessitate changes to the API, but also the vulgar influence of trends. Several years ago realistic 3D icons were popular, but since then, public taste has changed in favor of flat and abstract ones. UI component developers had to follow the fashion, rebuilding their libraries by either shipping new icons or replacing the old ones. Similarly, the current trend of integrating the "night mode" feature has become widespread, demanding changes in a wide range of APIs.

Backward-Compatible Specifications

In the case of the API-first approach, the backward compatibility problem adds another dimension: the specification and code generation based on it. It becomes possible to break backward compatibility without breaking the spec (for example, by introducing eventual consistency instead of strict consistency) — and vice versa, modify the spec in a backward-incompatible manner without changing anything in the protocol and therefore not affecting existing integrations at all (for example, by replacing `additionalProperties: false` with `true` in OpenAPI).

The question of whether two specification versions are backward-compatible or not belongs to a gray zone, as specification standards themselves do not define this. Generally speaking, the statement "specification change is backward-compatible" is equivalent to "any client code written or generated based on the previous version of the spec continues to work correctly after the API vendor releases the new API version implementing the new version of the spec." Practically speaking, following this definition seems quite unrealistic for two reasons: it is impossible to learn the behavior of every piece of code-generating software out there (for instance, it's rather hard to say whether code generated based on a specification that includes the parameter `additionalProperties: false` will still function properly if the server starts returning additional fields).

Thus, using IDLs to describe APIs with all the advantages they undeniably bring to the field, leads to having one aspect of the technology drift problem: the IDL version and, more importantly, versions of helper software based on it, are constantly and sometimes unpredictably evolving. If an API vendor employs the "code-first" approach, meaning that the spec is generated based on the actual API code, the occurrence of backward-incompatible changes in the server code — spec — code-generated SDK — client app chain is only a matter of time.

NB: We recommend sticking to reasonable practices such as not using functionality that is controversial from a backward compatibility point of view (including the above-mentioned `additionalProperties: false`) and when evaluating the safety of changes, considering spec-generated code behaves just like manually written code. If you find yourself in a situation of unresolvable doubts, your only option is to manually check every code generator to determine whether its output continues to work with the new version of the API.

Backward Compatibility Policy

To summarize the points discussed above:

- You will have to deploy new API versions because of the evolution of apps, platforms, and subject areas. Different areas evolve at different paces but never stop doing so.
- This will result in the fragmentation of the API versions across different platforms and apps.
- You have to make decisions that greatly affect the sustainability of your API from your customers' perspective.

Let's briefly describe these decisions and the key factors to consider while making them.

1. How often should new major API versions be released?

This is primarily a *product* question. A new major API version should be released when a critical mass of functionality is reached, meaning a critical mass of features that couldn't be introduced in the previous API versions or would be too expensive to introduce. In stable markets, this situation typically occurs once every several years. In emerging markets, new major API versions might be shipped more frequently, depending only on your ability to support the zoo of the previous versions. However, it is important to note that deploying a new version before stabilizing the previous one, which commonly takes several months up to a year, is always a troubling sign to developers, as it means they risk dealing with API glitches permanently.

2. How many *major* versions should be supported simultaneously?

   Theoretically, all of them. *Practically*, you should look at the size of the audience that continues to use older versions and develop guidelines on when the support for those versions will end.

3. How many *minor* versions (within one major version) should be supported simultaneously?

   Regarding minor versions, there are two options:

   - If you provide server-side APIs and compiled SDKs only, you may basically choose not to expose minor versions at all (see below). However, at some maturity stage, providing access to at least the two latest versions becomes necessary.
   - If you provide code-on-demand SDKs, it is considered good practice to provide access to previous minor versions of the SDK for a period of time sufficient for developers to test their applications and address issues if necessary. Since minor changes do not require rewriting large portions of code, it is acceptable to align the lifecycle of a minor version with the app release cycle duration in your industry, which in the worst cases may comprise several months.

Keeping Several API Versions

In modern professional software development, especially when talking about internal APIs, a new API version usually fully replaces the previous one. If any problems are found, it might be rolled back by releasing the previous version, but the two builds never coexist. However, in the case of public APIs, the more partner integrations there are, the more dangerous this approach becomes.

Indeed, with the growth in the number of users, the "rollback the API version in case of problems" paradigm becomes increasingly destructive. For partners, the optimal solution is rigidly referencing the specific API version — the one that had been tested (ideally, while also having the API vendor seamlessly address security concerns and make their software compliant with newly introduced legislation).

NB: Based on the same considerations, providing beta (or maybe even alpha) versions of popular APIs becomes more and more desirable as well, allowing partners to test upcoming versions and address possible issues in advance.

The important (and undeniable) advantage of the *semver* system is that it provides proper version granularity:

- Stating the first digit (major version) allows obtaining a backward-compatible version of the API.
- stating two digits (major and minor versions) guarantees that functionality added after the initial release will be available.
- Finally, stating all three numbers (major version, minor version, and patch) allows fixing a concrete API release with all its specificities (and errors), which — theoretically — means that the integration will remain operational until this version becomes physically unavailable.

Of course, preserving minor versions indefinitely is not possible (partly because of security and compliance issues that tend to accumulate). However, providing such access for a reasonable period of time is considered a hygienic norm for popular APIs.

NB: Sometimes to defend the concept of a single accessible API version, the following argument is put forward: preserving the SDK or API application server code is not enough to maintain strict backward compatibility as it might rely on some unversioned services (for example, data in the DB shared between all API versions). However, we consider this an additional reason to isolate such dependencies (see "The Serenity Notepad" chapter) as it means that changes to these subsystems might result in the API becoming inoperable.

Chapter 27. On the Waterline of the Iceberg

Before we start talking about extensible API design, we should discuss the hygienic minimum. Many problems would have never occurred if API vendors had paid more attention to clearly marking their area of responsibility.

1. Provide a Minimal Amount of Functionality

At any given moment, your API is like an iceberg: it comprises an observable (i.e., documented) part and a hidden undocumented one. If the API is properly designed, these two parts correspond to each other just like the above-water and under-water parts of a real iceberg do, i.e. one to ten. Why so? Because of two obvious reasons.

- Computers exist to make complicated things easy, not the other way around. The code that developers write using your API should describe a complicated problem's solution in neat and straightforward sentences. If developers have to write more code than the API itself comprises, then there is something rotten here. It's possible that this API isn't needed at all.
- Revoking API functionality causes losses. If you have promised to provide certain functionality, you will have to do so “forever” (or at least until the maintenance period for that API version is over). Pronouncing some functionality as deprecated can be tricky and may alienate your customers.

The rule of thumb is very simple: if some functionality might be withheld, then never expose it until you really need to. It might be reformulated as follows: every entity, every field, and every public API method is a *product decision*. There must be solid *product* reasons why certain functionality is exposed.

2. Avoid Gray Zones and Ambiguities

Your obligations to maintain some functionality must be stated as clearly as possible, especially when provided in environments and platforms where there is no native capability to restrict access to undocumented functionality. Unfortunately, developers often consider some private features they “discover” as eligible for use, assuming the API vendor shall maintain them intact. The policy regarding such “findings” must be explicitly articulated. At the very least, in the case of unauthorized usage of undocumented functionality, you can refer to the documentation and be within your rights in the eyes of the community.

However, API developers often legitimize these gray zones themselves. For example, by:

- Returning undocumented fields in endpoint responses
- Using private functionality in code samples: in the documentation, responses to support inquiries, conference talks, etc.

One cannot make a partial commitment. Either you guarantee that the code will always work or do not slip the slightest note that such functionality exists.

3. Codify Implicit Agreements

The third principle is much less obvious. Pay close attention to the code that you're suggesting developers write: are there any conventions that you consider self-evident but never wrote down?

Example #1. Let's take a look at this order processing SDK example:

```
// Creates an order
let order = api.createOrder();
// Returns the order status
let status = api.getStatus(order.id);
```

Let's imagine that you're struggling with scaling your service, and at some point switched to eventual consistency, as we discussed in the corresponding chapter. What would be the result? The code above will stop working. A user creates an order, then tries to get its status but receives an error instead. It's very hard to predict what approach developers would implement to tackle this error. They probably would not expect this to happen at all.

You may say something like, “But we've never promised strict consistency in the first place” — and that is obviously not true. You may say that if, and only if, you have really described the eventual consistency in the `createOrder` docs, and all your SDK examples look like this:

```
let order = api.createOrder();
let status;
while (true) {
  try {
    status = api.getStatus(order.id);
  } catch (e) {
    if (e.httpStatusCode != 404 ||
      timeoutExceeded()) {
      break;
    }
  }
}
if (status) {
  …
}
```

We presume we may skip the explanations of why such code must never be written under any circumstances. If you're really providing a non-strictly consistent API, then either the `createOrder` operation must be asynchronous and return the result when all replicas are synchronized, or the retry policy must be hidden inside the `getStatus` operation implementation.

If you failed to describe the eventual consistency in the first place, then you simply couldn't make these changes in the API. You will effectively break backward compatibility, which will lead to huge problems with your customers' apps, intensified by the fact that they can't be simply reproduced by QA engineers.

Example #2. Take a look at the following code:

```
let resolve;
let promise = new Promise(
    function (innerResolve) {
        resolve = innerResolve;
    }
);
resolve();
```

This code presumes that the callback function passed to a `new Promise` will be executed synchronously, and the `resolve` variable will be initialized before the `resolve()` function call is executed. But this assumption is based on nothing: there are no clues indicating that the `new Promise` constructor executes the callback function synchronously.

Of course, the developers of the language standard can afford such tricks; but you as an API developer cannot. You must at least document this behavior and make the signatures point to it. Actually, the best practice is to avoid such conventions since they are simply not obvious when reading the code. And of course, under no circumstances dare you change this behavior to an asynchronous one.

Example #3. Imagine you're providing an animations API, which includes two independent functions:

```
// Animates object's width,
// beginning with the first value,
// ending with the second
// in the specified time frame
object.animateWidth(
  '100px', '500px', '1s'
);
// Observes the object's width changes
object.observe(
  'widthchange', observerFunction
);
```

A question arises: how frequently and at what time fractions will the `observerFunction` be called? Let's assume in the first SDK version we emulated step-by-step animation at 10 frames per second. Then the `observerFunction` will be called 10 times, getting values `'140px'`, `'180px'`, etc., up to `'500px'`. But then, in a new API version, we have switched to implementing both functions atop the operating system's native functionality. Therefore, you simply don't know when and how frequently the `observerFunction` will be called.

Just changing the call frequency might result in making some code dysfunctional. For example, if the callback function performs some complex calculations and no throttling is implemented because developers relied on your SDK's built-in throttling. Additionally, if the `observerFunction` ceases to be called exactly when the `'500px'` value is reached due to system algorithm specifics, some code will be broken beyond any doubt.

In this example, you should document the concrete contract (how often the observer function is called) and stick to it even if the underlying technology is changed.

Example #4. Imagine that customer orders are passing through a specific pipeline:

```
GET /v1/orders/{id}/events/history
→
{ "event_history": [
  {
    "iso_datetime":
      "2020-12-29T00:35:00+03:00",
    "new_status": "created"
  }, {
    "iso_datetime":
      "2020-12-29T00:35:10+03:00",
    "new_status": "payment_approved"
  }, {
    "iso_datetime":
      "2020-12-29T00:35:20+03:00",
    "new_status": "preparing_started"
  }, {
    "iso_datetime":
      "2020-12-29T00:35:30+03:00",
    "new_status": "ready"
  }
]}
```

Suppose at some moment we decided to allow trustworthy clients to get their coffee in advance before the payment is confirmed. So an order will jump straight to `"preparing_started"` or even `"ready"` without a `"payment_approved"` event being emitted. It might appear to you that this modification *is* backward-compatible since you've never really promised any specific event order being maintained, but it is not.

Let's assume that a developer (probably your company's business partner) wrote some code implementing valuable business procedures, for example, gathering income and expenses analytics. It's quite logical to expect this code operates a state machine that switches from one state to another depending on specific events. This analytical code will be broken if the event order changes. In the best-case scenario, a developer will get some exceptions and will have to cope with the error's cause. In the worst case, partners will operate incorrect statistics for an indefinite period of time until they find the issue.

A proper decision would be, first, documenting the event order and the allowed states; second, continuing to generate the "payment_approved" event before the "preparing_started" one (since you're making a decision to prepare that order, so you're in fact approving the payment) and add extended payment information.

This example leads us to the last rule.

4. Product Logic Must Be Backward-Compatible as Well

The state transition graph, event order, possible causes of status changes, etc. — such critical things must be documented. However, not every piece of business logic can be defined in the form of a programmable contract; some cannot be represented in a machine-readable form at all.

Imagine that one day you start taking phone calls. A client may contact the call center to cancel an order. You might even make this functionality *technically* backward-compatible by introducing new fields to the “order” entity. But the end-user might simply *know* the number and call it even if the app wasn't suggesting anything like that. The partner's business analytical code might be broken as well or start displaying weather on Mars since it was written without knowing about the possibility of canceling orders in circumvention of the partner's systems.

A *technically* correct decision would be to add a “canceling via call center allowed” parameter to the order creation function. Conversely, call center operators might only cancel those orders that were created with this flag set. But that would be a bad decision from a *product* point of view because it is not obvious to users that they can cancel some orders by phone and not others. The only “good” decision in this situation is to foresee the possibility of external order cancellations in the first place. If you haven't foreseen it, your only option is the “Serenity Notepad” that will be discussed in the last chapter of this Section.

Chapter 28. Extending through Abstracting

In the previous chapters, we have attempted to outline theoretical rules and illustrate them with practical examples. However, understanding the principles of designing change-proof APIs requires practice above all else. The ability to anticipate future growth problems comes from a handful of grave mistakes once made. While it is impossible to foresee everything, one can develop a certain technical intuition.

Therefore, in the following chapters, we will test the robustness of our study API from the previous Section, examining it from various perspectives to perform a "variational analysis" of our interfaces. More specifically, we will apply a "What If?" question to every entity, as if we are to provide a possibility to write an alternate implementation of every piece of logic.

NB: In our examples, the interfaces will be constructed in a manner allowing for dynamic real-time linking of different entities. In practice, such integrations usually imply writing *ad hoc* server-side code in accordance with specific agreements made with specific partners. But for educational purposes, we will pursue more abstract and complicated ways. Dynamic real-time linking is more typical in complex program constructs like operating system APIs or embeddable libraries; giving educational examples based on such sophisticated systems would be too inconvenient.

Let's start with the basics. Imagine that we haven't exposed any other functionality but searching for offers and making orders, thus providing an API with two methods: `POST /offers/search` and `POST /orders`.

Let us take the next logical step and suppose that partners will wish to dynamically plug their own coffee machines (operating some previously unknown types of API) into our platform. To allow doing so, we have to negotiate a callback format that would allow us to call partners' APIs and expose two new endpoints providing the following capabilities:

- Registering new API types in the system
- Providing the list of the coffee machines and their API types.

For example, we might provide a second API family (the partner-bound one) with the following methods:

```
// 1. Register a new API type
PUT /v1/api-types/{api_type}
{
  "order_execution_endpoint": {
    // Callback function description
  }
}
```

```
// 2. Provide a list of coffee machines
// with their API types
PUT /v1/partners/{partnerId}/coffee-machines
{
  "coffee_machines": [{
    "api_type",
    "location",
    "supported_recipes"
  }, …]
}
```

So the mechanics are like this:

- A partner registers their API types, coffee machines, and supported recipes.
- With each incoming order, our server will call the callback function, providing the order data in the stipulated format.

Now the partners might dynamically plug their coffee machines in and get the orders. But now we will do the following exercise:

- Enumerate all the implicit assumptions we have made
- Enumerate all the implicit coupling mechanisms we need to have the platform functioning properly.

It may seem like there are no such things in our API since it's quite simple and basically just describes making some HTTP calls, but that's not true.

1. It is implied that every coffee machine supports every order option like varying the beverage volume.
2. There is no need to display additional data to the end-user regarding coffee being brewed on these new coffee machines.
3. The price of the beverage doesn't depend on the selected partner or coffee machine type.

We have written down this list having one purpose in mind: we need to understand how exactly we will make these implicit arrangements explicit if we need to. For example, if different coffee machines provide different functionality — let's say, some of them are capable of brewing fixed beverage volumes only — what would change in our API?

The universal approach to making such amendments is to consider the existing interface as a reduction of some more general one, as if some parameters were set to defaults and therefore omitted. So making a change is always a three-step process:

1. Explicitly define the programmatical contract *as it works right now*.
2. Extend the functionality: add a new method that allows for tackling the restrictions set in the previous paragraph.
3. Pronounce the existing interfaces (those defined in #1) as “helpers” to the new ones (those defined in #2) that pre-fill some options with default values.

More specifically, if we talk about changing available order options, we should do the following:

1. Describe the current state. All coffee machines, plugged via the API, must support three options: sprinkling with cinnamon, changing the volume, and contactless delivery.

2. Add a new “with options” endpoint:

```
PUT /v1/partners/{partner_id}↵
  /coffee-machines-with-options
{
  "coffee_machines": [{
    "id",
    "api_type",
    "location",
    "supported_recipes",
    "supported_options": [
      {"type": "volume_change"}
    ]
  }, …]
}
```

3. Pronounce the `PUT /coffee-machines` endpoint as it currently stands in the protocol as equivalent to calling `PUT /coffee-machines-with-options` if we pass those three options to it (sprinkling with cinnamon, changing the volume, contactless delivery) and therefore being a partial case — a helper to a more general call.

Usually, just adding a new optional parameter to the existing interface is enough; in our case, adding non-mandatory `options` to the `PUT /coffee-machines` endpoint.

NB: When we talk about defining the contract as it works right now, we're referring to *internal* agreements. We must have asked partners to support those three options while negotiating the interaction format. If we had failed to do so from the very beginning and are now defining them during the expansion of the public API, it's a very strong claim to break backward compatibility, and we should never do that (see the previous chapter).

Limits of Applicability

Though this exercise appears to be simple and universal, its consistent usage is only possible if the hierarchy of entities is well-designed from the very beginning and, more importantly, if the direction of further API expansion is clear. Imagine that after some time has passed, the options list has new items, such as adding syrup or a second espresso shot. We are fully capable of expanding the list, but not the defaults. As a result,

the “default” `PUT /coffee-machines` interface will eventually become completely useless because the default set of three options will no longer be useful and will appear ridiculous: why these three options, what are the selection criteria? In fact, the defaults and the method list reflect the historical stages of our API development, which is not what one would expect from the helpers and defaults nomenclature.

Alas, this dilemma can't be easily resolved. On one hand, we want developers to write neat and concise code, so we must provide useful helpers and defaults. On the other hand, we can't know in advance which sets of options will be the most useful after several years of API evolution.

NB: We might conceal this problem in the following manner: one day gather all these oddities and re-define all the defaults with a single parameter. For example, introduce a special method like `POST /use-defaults {"version": "v2"}` that would overwrite all the defaults with more suitable values. This would ease the learning curve, but it would make your documentation even worse.

In the real world, the only viable approach to somehow tackle the problem is weak entity coupling, which we will discuss in the next chapter.

Chapter 29. Strong Coupling and Related Problems

To demonstrate the problems of strong coupling, let's move on to *interesting* topics. Let's continue our "variation analysis": what if partners wish to offer their own unique coffee recipes to end users in addition to the standard beverages? The challenge is that the partner API, as described in the previous chapter, does not expose the very existence of the partner network to the end user, thus presenting a simple case. However, once we start providing methods to modify the core functionality, not just API extensions, we will soon face next-level problems.

So, let's add one more endpoint for registering the partner's own recipe:

```
// Adds new recipe
POST /v1/recipes
{
  "id",
  "product_properties": {
    "name",
    "description",
    "default_volume"
    // Other properties to describe
    // the beverage to an end user
    …
  }
}
```

At first glance, this appears to be a reasonably simple interface, explicitly decomposed into abstraction levels. But let's imagine the future and consider what would happen to this interface as our system evolves further.

The first problem is obvious to those who thoroughly read the "Describing Final Interfaces" chapter: product properties must be localized. This leads us to the first change:

```
"product_properties": {
  // "l10n" is the standard abbreviation
  // for "localization"
  "l10n": [{
    "language_code": "en",
    "country_code": "US",
    "name",
    "description"
  }, /* other languages and countries */ … ]
}
```

And here arises the first big question: what should we do with the `default_volume` field? On one hand, it's an objective property measured in standardized units to be passed to the program execution engine. On the other hand, in countries like the United States, beverage volumes are specified as “10 fl oz” rather than “300 ml.” We can propose two solutions:

- Either the partner provides only the corresponding number and we will make readable descriptions ourselves, or
- The partner provides both the number and all its localized representations.

The flaw in the first option is that a partner might be willing to use the service in a new country or language, but they will be unable to do so until the API is localized to support these new territories. The flaw in the second option is that it only works with predefined volumes, so ordering an arbitrary beverage volume will not be possible. The very first step we've taken effectively has had us trapped.

The localization flaws are not the only problem with this API. We should ask ourselves a question: *why* do we really need these `name` and `description` fields? They are simply non-machine-readable strings with no specific semantics. At first glance, we need them to return in the `/v1/search` method response, but that's not a proper answer as it only leads to another question: why do we actually return these strings from `search`?

The correct answer lies beyond this specific interface. We need them *because some representation exists*. There is a UI for choosing a beverage type. The `name` and `description` fields are probably two designations of the beverage for the user to read, a short one (to be displayed on the search results page) and a long one (to be displayed in the extended product specification block). This means we are setting the API requirements based on some specific visual design. But *what if* a partner is creating their own UI for their own app? Not only might they not actually need two descriptions, but we are also *deceiving* them. The `name` is not "just a name" as it implies certain restrictions: it has a recommended length that is optimal for a specific UI, and it must look consistent on the search results page. Indeed, designations like "our best quality™ coffee" or "Invigorating Morning Freshness®" would look out of place among "Cappuccino," "Lungo," and "Latte."

There is also another aspect to consider. As UIs (both ours and partners') evolve, new visual elements will eventually be introduced. For example, a picture of the beverage, its energy value, allergen information, etc. The `product_properties` entity will become a scrapyard for numerous optional fields, and learning how to set each field and its effects in the UI will be an interesting journey filled with trial and error.

The problems we're facing are the problems of *strong coupling*. Each time we offer an interface as described above, we effectively dictate the implementation of one entity (recipe) based on the implementations of other entities (UI layout, localization rules). This approach disregards the fundamental principle of "top to bottom" API design because **low-level entities should not define high-level ones**.

The Rule of Contexts

To exacerbate matters, let us state that the inverse principle is also true: high-level entities should not define low-level ones as well since it is not their responsibility. The way out of this logical labyrinth is that high-level entities should *define a context* for other objects to interpret. To properly design the interfaces for adding a new recipe we should not attempt to find a better data format. Instead, we need to understand the explicit and implicit contexts that exist in our subject area.

We have already identified a localization context. There is a set of languages and regions supported by our API, and there are requirements for what partners must provide to make the API work in a new region. Specifically, there must be a formatting function to represent beverage volume somewhere in our API code, either internally or within an SDK:

```
l10n.volume.format = function(
  value, language_code, country_code
) { … }
/*
  l10n.formatVolume(
   '300ml', 'en', 'UK'
  ) → '300 ml'
  l10n.formatVolume(
    '300ml', 'en', 'US'
  ) → '10 fl oz'
*/
```

To ensure our API works correctly with a new language or region, the partner must either define this function or indicate which pre-existing implementation to use through the partner API, like this:

```
// Add a general formatting rule
// for the Russian language
PUT /formatters/volume/ru
{
  "template": "{volume} мл"
}
// Add a specific formatting rule
// for the Russian language
// in the “US” region
PUT /formatters/volume/ru/US
{
  // In the US, we need to recalculate
  // the number and add a postfix
  "value_transform": {
    "action": "divide",
    "divisor": 30
  },
  "template": "{volume} ун."
}
```

so the aforementioned `l10n.volume.format` function implementation can retrieve the formatting rules for the new language-region pair and utilize them.

NB: We are well aware that such a simple format is not sufficient to cover real-world localization use cases, and one would either rely on existing libraries or design a sophisticated format for such templating, which takes into account various aspects such as grammatical cases and rules for rounding numbers or allows defining formatting rules in the form of function code. The example above is simplified for purely educational purposes.

Let's address the `name` and `description` problem. To reduce the coupling level, we need to formalize (probably just for ourselves) a “layout” concept. We request the provision of the `name` and `description` fields not because we theoretically need them but to present them in a specific user interface. This particular UI might have an identifier or a semantic name associated with it:

```
GET /v1/layouts/{layout_id}
{
  "id",
  // Since we will likely have numerous
  // layouts, it's better to enable
  // extensibility from the beginning
  "kind": "recipe_search",
  // Describe every property we require
  // to have this layout rendered properly
  "properties": [{
    // Since we learned that `name`
    // is actually a title for a search
    // result snippet, it's much more
    // convenient to have an explicit
    // `search_title` instead
    "field": "search_title",
    "view": {
      // A machine-readable description
      // of how this field is rendered
      "min_length": "5em",
      "max_length": "20em",
      "overflow": "ellipsis"
    }
  }, …],
  // Which fields are mandatory
  "required": [
    "search_title",
    "search_description"
  ]
}
```

Thus, the partner can decide which option better suits their needs. They can provide mandatory fields for the standard layout:

```
PUT /v1/recipes/{id}/properties/l10n/{lang}
{
  "search_title", "search_description"
}
```

Alternatively, they can create their own layout and provide the data fields it requires, or they may choose to design their own UI and not use this functionality at all, thereby defining neither layouts nor corresponding data fields.

Ultimately, our interface would look like this:

```
POST /v1/recipes
{ "id" }
→
{ "id" }
```

This conclusion might seem highly counter-intuitive, but the absence of fields in a `Recipe` simply tells us that this entity possesses no specific semantics of its own. It serves solely as an identifier of a context, a way to indicate where to find the data needed by other entities. In the real world, we should implement a builder endpoint capable of creating all the related contexts with a single request:

```
POST /v1/recipe-builder
{
  "id",
  // Recipe's fixed properties
  "product_properties": {
    "default_volume",
    "l10n"
  },
  // Create all the desired layouts
  "layouts": [{
    "id", "kind", "properties"
  }],
  // Add all the required formatters
  "formatters": {
    "volume": [
      {
        "language_code",
        "template"
      }, {
        "language_code",
        "country_code",
        "template"
      }
    ]
  },
  // Other actions needed
  // to register a new recipe
  // in the system
  …
}
```

We should also note that providing a newly created entity identifier from the requesting side is not the best practice. However, since we decided from the very beginning to keep recipe identifiers semantically meaningful, we have to live on with this convention. Obviously, there is a risk of encountering collisions with recipe names used by different partners. Therefore, we actually need to modify this operation: either a partner must always use a pair of identifiers (e.g., the recipe id plus the

partner's own id), or we need to introduce composite identifiers, as we recommended earlier in the "Describing Final Interfaces" chapter.

```
POST /v1/recipes/custom
{
  // The first part of the composite
  // identifier, for example,
  // the partner's own id
  "namespace": "my-coffee-company",
  // The second part of the identifier
  "id_component": "lungo-customato"
}
→
{
  "id":
    "my-coffee-company:lungo-customato"
}
```

Also note that this format allows us to maintain an important extensibility point: different partners might have both shared and isolated namespaces. Furthermore, we might introduce special namespaces (like `common`, for example) to allow editing standard recipes (and thus organizing our own recipes backoffice).

NB: A mindful reader might have noticed that this technique was already used in our API study much earlier in the "Separating Abstraction Levels" chapter regarding the "program" and "program run" entities. Indeed, we can propose an interface for retrieving commands to execute a specific recipe without the `program-matcher` endpoint, and instead, do it this way:

```
GET /v1/recipes/{id}/run-data/{api_type}
→
{ /* A description of how to
     execute a specific recipe
     using a specified API type */ }
```

Then developers would have to make this trick to get the beverage prepared:

- Learn the API type of the specific coffee machine.
- Retrieve the execution description as described above.
- Based on the API type, execute specific commands.

Obviously, such an interface is completely unacceptable because, in the majority of use cases, developers do not care at all about which API type the specific coffee machine exposes. To avoid the need for introducing such poor interfaces we created a new “program” entity, which serves solely as a context identifier, just like a “recipe” entity does. Similarly, the `program_run_id` entity is also organized in the same manner, without possessing any specific properties and representing *just* a program run identifier.

Chapter 30. Weak Coupling

In the previous chapter, we demonstrated how breaking strong coupling of components leads to decomposing entities and collapsing their public interfaces down to a reasonable minimum. But let us return to the question we previously mentioned in the "Extending through Abstracting" chapter: how should we parametrize the order preparation process implemented via a third-party API? In other words, what *is* the `order_execution_endpoint` required in the API type registration handler?

```
PUT /v1/api-types/{api_type}
{
  …
  "order_execution_endpoint": {
    // ???
  }
}
```

From general considerations, we may assume that every such API would be capable of executing three functions: running a program with specified parameters, returning the current execution status, and finishing (canceling) the order. An obvious way to provide the common interface is to require these three functions to be executed via a remote call, let's say, like this:

```
PUT /v1/api-types/{api_type}
{
  …
  "order_execution_endpoint": {
    "program_run_endpoint": {
      /* Some description of
          the remote function call */
      "type": "rpc",
      "endpoint": <URL>,
      "parameters"
    },
    "program_get_state_endpoint",
    "program_cancel_endpoint"
  }
}
```

NB: By doing so, we transfer the complexity of developing the API onto the plane of developing appropriate data formats, i.e., developing formats for order parameters to the `program_run_endpoint`, determining what format the `program_get_state_endpoint` shall return, etc. However, in this chapter, we're focusing on different questions.

Though this API looks absolutely universal, it's quite easy to demonstrate how a once simple and clear API ends up being confusing and convoluted. This design presents two main problems:

1. It nicely describes the integrations we've already implemented (it costs almost nothing to support the API types we already know), but it brings no flexibility to the approach. In fact, we simply described what we had already learned, without even trying to look at the larger picture.
2. This design is ultimately based on a single principle: every order preparation might be codified with these three imperative commands.

We can easily disprove the second statement, which will uncover the implications of the first. Let's imagine, for example, that as the service grows further, we decide to allow end-users to change the order after the execution has started. For example, they may request a contactless takeout. This would lead us to the creation of a new endpoint, let's say, `program_modify_endpoint`, and new difficulties in data format development (as new fields for contactless delivery requested and satisfied flags need to be passed in both directions). What *is* important is that both the endpoint and the new data fields would be optional due to the backward compatibility requirement.

Now let's try to imagine a real-world example that doesn't fit into our "three imperatives to rule them all" picture. That's quite easy as well: what if we're plugging in a vending machine via our API instead of a coffee house? On one hand, it means that the `modify` endpoint and all related stuff are simply meaningless: the contactless takeout requirement means nothing to a vending machine. On the other hand, the machine, unlike the people-operated café, requires *takeout approval*: the end-user places an order while being somewhere else and then walks to the machine and pushes the "get the order" button in the app. We might, of course, require the user to stand up in front of the machine when placing an order, but that would contradict the entire product concept of users selecting and ordering beverages and then walking to the takeout point.

Programmable takeout approval requires one more endpoint, let's say, `program_takeout_endpoint`. And so we've lost our way in a forest of five endpoints:

- To have vending machines integrated a partner must implement the `program_takeout_endpoint` but doesn't need the `program_modify_endpoint`.
- To have regular coffee houses integrated a partner must implement the `program_modify_endpoint` but doesn't need the `program_takeout_endpoint`.

Furthermore, we have to describe both endpoints in the documentation. It's quite natural that the `takeout` endpoint is very specific; unlike requesting contactless delivery, which we hid under the pretty general `modify` endpoint, operations like takeout approval will require introducing a new unique method every time. After several iterations, we would have a scrapyard full of similarly looking methods, mostly optional. However, developers would still need to study the documentation to understand which methods are needed in their specific situation and which are not.

NB: In this example, we assumed that having the optional `program_takeout_endpoint` value filled serves as a flag to the application to display the "get the order" button. It would be better to add something like a `supported_flow` field to the `PUT /api-types/` endpoint to provide an explicit flag instead of relying on this implicit convention. However, this wouldn't change the problematic nature of stockpiling optional methods in the interface, so we skipped it to keep the examples concise.

We actually don't know whether in the real world of coffee machine APIs this problem will occur or not. But we can say with confidence that regarding "bare metal" integrations, the processes we described *always* happen. The underlying technology shifts; an API that seemed clear and straightforward becomes a trash bin full of legacy methods, half of which bear no practical sense under any specific set of conditions. If we add technical progress to the situation, i.e., imagine that after a while all coffee houses have become automated, we will finally end up in a situation where most methods *aren't actually needed at all*, such as requesting a contactless takeout.

It is also worth mentioning that we unwittingly violated the principle of isolating abstraction levels. At the vending machine API level, there is no such thing as "contactless takeout" as it is actually a product concept.

So, how would we tackle this issue? We can use one of two possible approaches: either thoroughly study the entire subject area and its upcoming improvements for at least several years ahead or abandon strong coupling in favor of a weak one. How would the *ideal* solution look for both parties? Something like this:

- The higher-level program API level doesn't actually know how the execution of its commands works. It formulates the tasks at its own level of understanding: brew this recipe, send the user's requests to a partner, allow the user to collect their order.
- The underlying program execution API level doesn't care about what other same-level implementations exist. It just interprets those parts of the task that make sense to it.

If we take a look at the principles described in the previous chapter, we would find that this principle was already formulated: we need to describe *informational contexts* at every abstraction level and design a mechanism to translate them between levels. Furthermore, in a more general sense, we formulated it as early as in the "Data Flow" paragraph of the "Separating Abstraction Levels" chapter.

In our case we need to implement the following mechanisms:

- Running a program creates a corresponding context comprising all the essential parameters.
- There is an information stream regarding the state modifications: the execution level may read the context, learn about all the changes and report back its own changes.

There are different techniques to organize this data flow (see the corresponding chapter of the "API Patterns" Section of this book). Basically, we always have two contexts and a two-way data pipe in between. If we were developing an SDK, we would express the idea with emitting and listening events, like this:

```
/* Partner's implementation of the program
   run procedure for a custom API type */
registerProgramRunHandler(
  apiType,
  (program) => {
    // Initiating an execution
    // on the partner's side
    let execution = initExecution(…);
    // Listen to parent context changes
    program.context.on(
      'takeout_requested',
      () => {
        // If a takeout is requested, initiate
        // required procedures
        await execution.prepareTakeout();
        // When the cup is ready for takeout,
        // emit the corresponding event for
        // a higher-level entity to catch it
        execution.context.emit('takeout_ready');
      }
    );
    program.context.on(
      'order_canceled',
      () => {
        await execution.cancel();
        execution.context.emit('canceled');
      }
    );

    return execution.context;
  }
);
```

NB: In the case of an HTTP API, a corresponding example would look rather bulky as it would require implementing several additional endpoints for the message exchange like `GET /program-run/events` and `GET /partner/{id}/execution/events`. We would leave this exercise to the reader.

At this point, a mindful reader might begin protesting because if we take a look at the nomenclature of the new entities, we will find that nothing changed in the problem statement. It actually became even more complicated:

- Instead of calling the `takeout` method, we're now generating a pair of `takeout_requested` / `takeout_ready` events
- Instead of a long list of methods that shall be implemented to integrate the partner's API, we now have a long list of context entities and events they generate
- And with regards to technological progress, we've changed nothing: now we have deprecated fields and events instead of deprecated methods.

And this remark is totally correct. Changing API formats doesn't solve any problems related to the evolution of functionality and underlying technology. Changing API formats serves another purpose: to make the code written by developers stay clean and maintainable. Why would strong-coupled integration (i.e., making entities interact via calling methods) render the code unreadable? Because both sides *are obliged* to implement functionality that is meaningless in their corresponding subject areas. Code that integrates vending machines into the system *must* respond “ok” to the contactless delivery request — so after a while, these implementations would comprise a handful of methods that just always return `true` (or `false`).

The difference between strong coupling and weak coupling is that the field-event mechanism *isn't obligatory for both actors*. Let us remember what we sought to achieve:

- A higher-level context doesn't know how the low-level API works — and it really doesn't. It describes the changes that occur within the context itself and reacts only to those events that mean something to it.
- A low-level context doesn't know anything about alternative implementations — and it really doesn't. It handles only those events which mean something at its level and emits only those events that could happen under its specific conditions.

It's ultimately possible that both sides would know nothing about each other and wouldn't interact at all, and this might happen with the evolution of underlying technologies.

NB: In the real world, this might not be the case as we might *want* the application to know whether the takeout request was successfully served or not, i.e., listen to the `takeout_ready` event and require the `takeout_ready` flag in the state of the execution context. Still, the general possibility of *not caring* about the implementation details is a very powerful technique that makes the application code much less complex — of course, unless this knowledge is important to the user.

One more important feature of weak coupling is that it allows an entity to have several higher-level contexts. In typical subject areas, such a situation would look like an API design flaw, but in complex systems, with several system state-modifying agents present, such design patterns are not that rare. Specifically, you would likely face it while developing user-facing UI libraries. We will cover this issue in detail in the “SDK and UI Libraries” section of this book.

The Inversion of Responsibility

It becomes obvious from what was said above that two-way weak coupling means a significant increase in code complexity on both levels, which is often redundant. In many cases, two-way event linking might be replaced with one-way linking without significant loss of design quality. That means allowing a low-level entity to call higher-level methods directly instead of generating events. Let's alter our example:

```
/* Partner's implementation of the program
   run procedure for a custom API type */
registerProgramRunHandler(
  apiType,
  (program) => {
    // Initiating an execution
    // on the partner's side
    let execution = initExecution(…);
    // Listen to parent context changes
    program.context.on(
      'takeout_requested',
      () => {
        // If a takeout is requested, initiate
        // corresponding procedures
        await execution.prepareTakeout();
        /* When the order is ready
           for takeout, signalize that
           by calling the parent context
           method, not with event emitting */
        // execution.context
        //   .emit('takeout_ready')
        program.context
          .set('takeout_ready');
        // Or even more rigidly
        // program.setTakeoutReady();
      }
    );
    /* Since we're modifying the parent
       context instead of emitting events,
       we don't actually need
       to return anything */
    // return execution.context;
  }
);
```

Again, this solution might look counter-intuitive, since we efficiently returned to strong coupling via strictly defined methods. But there is an important difference: we're bothering ourselves with weak coupling because we expect alternative implementations of the *lower* abstraction level to pop up. Situations with different realizations of *higher*

abstraction levels emerging are, of course, possible but quite rare. The tree of alternative implementations usually grows from root to leaves.

Another reason to justify this solution is that major changes occurring at different abstraction levels have different weights:

- If the technical level is under change, that must not affect product qualities and the code written by partners.
- If the product is changing, e.g., we start selling flight tickets instead of preparing coffee, there is literally no sense in preserving backward compatibility at technical abstraction levels. Ironically, we may actually make our API sell tickets instead of brewing coffee without breaking backward compatibility, but the partners' code will still become obsolete.

In conclusion, as higher-level APIs are evolving more slowly and much more consistently than low-level APIs, reverse strong coupling might often be acceptable or even desirable, at least from the price-quality ratio point of view.

NB: Many contemporary frameworks explore a shared state approach, Redux being probably the most notable example. In the Redux paradigm, the code above would look like this:

```
program.context.on(
  'takeout_requested',
  () => {
    await execution.prepareTakeout();
    // Instead of generating events
    // or calling higher-level methods,
    // an `execution` entity calls
    // a global or quasi-global `dispatch`
    // callback to change a global state
    dispatch(takeoutReady());
  }
);
```

Let us note that this approach *in general* doesn't contradict the weak coupling principle but violates another one — abstraction levels isolation — and therefore isn't very well suited for writing branchy APIs with high hierarchy trees. In such systems, it's still possible to use a global or quasi-global state manager, but you need to implement event or method call propagation through the hierarchy, i.e., ensure that a low-level entity always interacts with its closest higher-level neighbors only, delegating the responsibility of calling high-level or global methods to them.

```
program.context.on(
  'takeout_requested',
  () => {
    await execution.prepareTakeout();
    // Instead of calling the global
    // `dispatch` method,  an `execution`
    // entity invokes its superior's
    // dispatch functionality
    program.context.dispatch(takeoutReady());
  }
);
```

```
// program.context.dispatch implementation
ProgramContext.dispatch = (action) => {
  // program.context calls its own
  // superior or global object
  // if there are no superiors
  globalContext.dispatch(
    // The action itself may and
    // must be reformulated
    // in appropriate terms
    this.generateAction(action)
  );
}
```

Delegate!

Based on what was said, one more important conclusion follows: doing a real job, i.e., implementing concrete actions (making coffee, in our case) should be delegated to the lower levels of the abstraction hierarchy. If the upper levels try to prescribe implementation algorithms, then (as demonstrated in the example of `order_execution_endpoint`) we will soon face a situation of inconsistent methods, most of which have no specific meaning when applied to a particular hardware context.

On the other hand, by following the paradigm of concretizing the contexts at each new abstraction level, we will eventually fall into the bunny hole deep enough to have nothing more to concretize: the context itself unambiguously matches the functionality we can programmatically control. At that level, we should stop detailing contexts further and focus on implementing the necessary algorithms. It's worth mentioning that the depth of abstraction may vary for different underlying platforms.

NB: In the "Separating Abstraction Levels" chapter we illustrated exactly this: when we talk about the first coffee machine API type, there is no need to extend the tree of abstractions beyond running programs. However, with the second API type, we need one more intermediary abstraction level, namely the runtimes API.

Chapter 31. Interfaces as a Universal Pattern

Let us summarize what we have written in the three previous chapters:

1. Extending API functionality is implemented through abstracting: the entity nomenclature is to be reinterpreted so that existing methods become partial simplified cases of more general functionality, ideally representing the most frequent scenarios.
2. Higher-level entities are to be the informational contexts for low-level ones, meaning they don't prescribe any specific behavior but rather translate their state and expose functionality to modify it, either directly through calling some methods or indirectly through firing events.
3. Concrete functionality, such as working with “bare metal” hardware or underlying platform APIs, should be delegated to low-level entities.

NB: There is nothing novel about these rules: one might easily recognize them as the SOLID architecture principles. This is not surprising either, as SOLID focuses on contract-oriented development, and APIs are contracts by definition. We have simply introduced the concepts of “abstraction levels” and “informational contexts” to these principles.

However, there remains an unanswered question: how should we design the entity nomenclature from the beginning so that extending the API won't result in a mess of assorted inconsistent methods from different stages of development? The answer is quite obvious: to avoid clumsy situations during abstracting (as with the recipe properties), all the entities must be originally considered as specific implementations of a more general interface, even if there are no planned alternative implementations for them.

For example, while designing the `POST /search` API, we should have asked ourselves a question: what is a “search result”? What abstract interface does it implement? To answer this question we need to decompose this entity neatly and identify which facet of it is used for interacting with which objects.

Then we would have come to the understanding that a “search result” is actually a composition of two interfaces:

- When creating an order, we need the search result to provide fields that describe the order itself, which could be a structure like:

  `{coffee_machine_id, recipe_id, volume, currency_code, price}`,

  or we can encode this data in the single `offer_id`.

- When displaying search results in the app, we need a different data set: `name`, `description`, and formatted and localized prices.

So our interface (let's call it `ISearchResult`) is actually a composition of two other interfaces: `IOrderParameters` (an entity that allows for creating an order) and `ISearchItemViewParameters` (an abstract representation of the search result in the UI). This interface split should naturally lead us to additional questions:

1. How will we couple the former and the latter? Obviously, these two sub-interfaces are related: the machine-readable price must match the human-readable one, for example. This will naturally lead us to the “formatter” concept described in the “Strong Coupling and Related Problems” chapter.

2. And what constitutes the “abstract representation of a search result in the UI”? Do we have other types of search? Should the `ISearchItemViewParameters` interface be a subtype of some even more general interface, or maybe a composition of several such interfaces?

Replacing specific implementations with interfaces not only allows us to respond more clearly to many concerns that arise during the API design phase but also helps us outline many possible directions for API evolution. This approach should assist us in avoiding API inconsistency problems in the future.

Apart from the abovementioned abstract principles, let us give a list of concrete recommendations on how to make changes in existing APIs to maintain backward compatibility

1. Remember the Iceberg's Waterline

If you haven't given any formal guarantee, it doesn't mean that you can violate informal ones. Often, just fixing bugs in APIs might render some developers' code inoperable. We can illustrate this with a real-life example that the author of this book actually faced once:

- There was an API to place a button into a visual container. According to the docs, it was taking its position (offsets to the container's corner) as a mandatory argument.
- In reality, there was a bug: if the position was not supplied, no exception was thrown. Buttons were simply stacked in the corner one after another.
- After the error had been fixed, we received a bunch of complaints: clients had really used this flaw to stack the buttons in the container's corner.

If fixing an error might somehow affect real customers, you have no other choice but to emulate this erroneous behavior until the next major release. This situation is quite common when you develop a large API with a huge audience. For example, operating system developers literally have to transfer old bugs to new OS versions.

2. Test the Formal Interface

Any software must be tested, and APIs are no exception. However, there are some subtleties involved: as APIs provide formal interfaces, it's the formal interfaces that need to be tested. This leads to several kinds of mistakes:

1. Often, requirements like “the `getEntity` function returns the value previously set by the `setEntity` function” appear to be too trivial for both developers and QA engineers to have a proper test. But it's quite possible to make a mistake there, and we have actually encountered such bugs several times.
2. The interface abstraction principle must also be tested. In theory, you might have considered each entity as an implementation of some interface; in practice, it might happen that you have forgotten something and alternative implementations aren't actually possible. For testing purposes, it's highly desirable to have an alternative realization, even a provisional one, for every interface.

3. Isolate the Dependencies

In the case of a gateway API that provides access to some underlying API or aggregates several APIs behind a single façade, there is a strong temptation to proxy the original interface as is, thus not introducing any changes to it and making life much simpler by sparing the effort needed to implement the weak-coupled interaction between services. For example, while developing program execution interfaces as described in the “Separating Abstraction Levels” chapter we might have taken the existing first-kind coffee-machine API as a role model and provided it in our API by just proxying the requests and responses as is. Doing so is highly undesirable because of several reasons:

- Usually, you have no guarantees that the partner will maintain backward compatibility or at least keep new versions more or less conceptually akin to the older ones.
- Any partner's problem will automatically ricochet into your customers.

The best practice is quite the opposite: isolate the third-party API usage, i.e., develop an abstraction level that will allow for:

- Keeping backward compatibility intact because of extension capabilities incorporated in the API design.

- Negating partner's problems by technical means:
  - Limiting the partner's API usage in case of load surges
  - Implementing retry policies or other methods of recovering after failures
  - Caching some data and states to have the ability to provide some (at least partial) functionality even if the partner's API is fully unreachable
  - Finally, configuring an automatic fallback to another partner or alternative API.

4. Implement Your API Functionality Atop Public Interfaces

There is an antipattern that occurs frequently: API developers use some internal closed implementations of some methods that exist in the public API. It happens because of two reasons:

- Often the public API is just an addition to the existing specialized software, and the functionality, exposed via the API, isn't being ported back to the closed part of the project, or the public API developers simply don't know the corresponding internal functionality exists.
- In the course of extending the API, some interfaces become abstract, but the existing functionality isn't affected. Imagine that while implementing the `PUT /formatters` interface described in the "Strong Coupling and Related Problems" chapter API developers have created a new, more general version of the volume formatter but haven't changed the implementation of the existing one, so it continues working for pre-existing languages.

There are obvious local problems with this approach (like the inconsistency in functions' behavior or the bugs that were not found while testing the code), but also a bigger one: your API might be simply unusable if a developer tries any non-mainstream approach because of performance issues, bugs, instability, etc., as the API developers themselves never tried to use this public interface for anything important.

NB: The perfect example of avoiding this anti-pattern is the development of compilers. Usually, the next compiler's version is compiled with the previous compiler's version.

5. Keep a Notepad

Whatever tips and tricks described in the previous chapters you use, it's often quite probable that you can't do *anything* to prevent API inconsistencies from piling up. It's possible to reduce the speed of this stockpiling, foresee some problems, and have some interface durability reserved for future use. But one can't foresee *everything*. At this stage, many developers tend to make some rash decisions, e.g., releasing a backward-incompatible minor version to fix some design flaws.

We highly recommend never doing that. Remember that the API is also a multiplier of your mistakes. What we recommend is to keep a serenity notepad — to write down the lessons learned and not to forget to apply this knowledge when a new major API version is released.

SECTION IV. HTTP APIS & THE REST ARCHITECTURAL PRINCIPLES

Chapter 33. On the HTTP API Concept and Terminology

The problem of designing HTTP APIs is, unfortunately, one of the most "holywar"-inspiring issues. On one hand, it is one of the most popular technologies but, on the other hand, it is quite complex and difficult to comprehend due to the large and fragmented standard split into many RFCs. As a result, the HTTP specification is doomed to be poorly understood and imperfectly interpreted by millions of software engineers and thousands of textbook writers. Therefore, before proceeding to the useful part of this Section, we must clarify exactly what we are going to discuss.

It has somehow happened that the entire modern network stack used for developing client-server APIs has been unified in two important points. One of them is the Internet Protocol Suite, which comprises the IP protocol as a base and an additional layer on top of it in the form of either the TCP or UDP protocol. Today, alternatives to the TCP/IP stack are used for a very limited subset of engineering tasks.

However, from a practical standpoint, there is a significant inconvenience that makes using raw TCP/IP protocols much less practical. They operate over IP addresses which are poorly suited for organizing distributed systems:

- Firstly, humans are not adept at remembering IP addresses and prefer readable names
- Secondly, an IP address is a technical entity bound to a specific network node while developers require the ability to add or modify nodes without having to modify the code of their applications.

The domain name system, which allows for assigning human-readable aliases to IP addresses, has proved to be a convenient abstraction with almost universal adoption. Introducing domain names necessitated the development of new protocols at a higher level than TCP/IP. For text (hypertext) data this protocol happened to be HTTP 0.9 developed by Tim Berners-Lee and published in 1991. Besides enabling the use of

network node names, HTTP also provided another useful abstraction: assigning separate addresses to endpoints working on the same network node.

Initially, the protocol was very simple and merely described a method of retrieving a document by establishing a TCP/IP connection to the server and passing a string in the `GET document_address` format. Subsequently, the protocol was enhanced by the URL standard for document addresses. After that, the protocol evolved rapidly: new verbs, response statuses, headers, data types, and other features emerged in a short time.

HTTP was developed to transfer hypertext which poorly fits for developing program interfaces. However, loose HTML quickly evolved into strict and machine-readable XML, which became one of the most widespread standards for describing API calls. Starting from the 2000s, XML was gradually replaced by much simpler and interoperable JSON. Today, when we talk about HTTP APIs, we usually refer to interfaces for transmitting data and executing remote calls in JSON format over the HTTP protocol.

On one hand, HTTP was a simple and easily understandable protocol to make arbitrary calls to remote servers using their domain names. On the other hand, it quickly gained a wide range of extensions beyond its base functionality. Eventually, HTTP became another “attractor” where all the network technology stacks converge. Most API calls within TCP/IP networks are made through the HTTP protocol. However, unlike the TCP/IP case, it is each developer's own choice which parts of the functionality provided by the HTTP protocol and its numerous extensions they are going to use. For example, gRPC and GraphQL work on top of HTTP but employ a limited subset of its capabilities.

However, the term “HTTP API” is not always a synonym for “any API that utilizes the HTTP protocol.” When we refer to HTTP APIs, we *rather* imply it is used not as a third additional quasi-transport layer protocol (as it happens in the case of gRPC and GraphQL) but as an application-level protocol, meaning its components (such as URL, headers, HTTP verbs, status codes, caching policies, etc.) are used according to their respective semantics. We also likely imply that some textual data format (JSON or XML) is used to describe procedure calls.

In this Section, we will discuss client-server APIs with the following properties:

- The interaction protocol is HTTP version 1.1 or higher
- The data format is JSON (excluding endpoints specifically designed to provide data in other formats, usually files)
- The endpoints (resources) are identified by their URLs in accordance with the standard
- The semantics of HTTP calls match the specification
- None of the Web standards are intentionally violated.

We will refer to such APIs as “HTTP APIs” or “JSON-over-HTTP APIs.” We understand that this is a loose interpretation of the term, but we prefer to live with that rather than using the phrase “JSON-over-HTTP endpoints utilizing the semantics described in the HTTP and URL standards” each time.

Chapter 34. Advantages and Disadvantages of HTTP APIs Compared to Alternative Technologies

After reviewing the previous chapter, the reader may wonder why this dichotomy exists in the first place, i.e., why do some HTTP APIs rely on HTTP semantics, while others reject it in favor of custom arrangements, and still others are stuck somewhere in between? For example, if we consider the JSON-RPC response format, we quickly notice that it could be replaced with standard HTTP protocol functionality. Instead of this:

```
HTTP/1.1 200 OK

{
  "jsonrpc": "2.0",
  "id",
  "error": {
    "code": -32600,
    "message": "Invalid request"
  }
}
```

the server could have simply responded with a `400 Bad Request`, passing the request identifier as a custom header like `X-OurCoffeeAPI-RequestId`. Nevertheless, protocol designers decided to introduce their own custom format.

This situation (not only with JSON-RPC but with essentially every high-level protocol built on top of HTTP) has developed due to various reasons. Some of them are historical (such as the inability to use many HTTP protocol features in early implementations of the `XMLHttpRequest` functionality in web browsers). However, new RPC protocols relying on the bare minimum of HTTP capabilities continue to emerge today. We can enumerate at least four groups of reasons leading to this situation.

1. Metadata readability

Let us emphasize a very important distinction between application-level protocols (such as JSON-RPC in our case) and pure HTTP. In the example above, a `400 BadRequest` error is a transparent status for every intermediary network agent but a JSON-RPC custom error is not. Firstly, only a JSON-RPC-enabled client can read it. Secondly, and more importantly, in JSON-RPC, the request status *is not metadata*. In pure HTTP, the details of the operation, such as the method, requested URL, execution status, and request / response headers are readable *without the necessity to parse the entire body*. In most higher-level protocols, including JSON-RPC, this is not the case: even a protocol-enabled client must read a body to retrieve that information.

How does an API developer benefit from the capability of reading request and response metadata? The modern client-server communication stack is multi-layered. We can enumerate a number of intermediary agents that process network requests and responses:

- Frameworks that developers use to write code
- Programming language APIs that frameworks are built on, and operating system APIs that compilers / interpreters of these languages rely on
- Intermediary proxy servers between a client and a server
- Various abstractions used in server programming, including server frameworks, programming languages, and operating systems
- Web server software that is typically placed in front of backend handlers
- Additional modern microservice-oriented tools such as API gateways and proxies

The main advantage that following the letter of the HTTP standard offers is the possibility to rely on intermediary agents, from client frameworks to API gateways, to read the request metadata and perform actions based on it. This includes regulating timeouts and retry policies, logging, proxying, and sharding requests, among other things, without the necessity to write additional code to achieve these functionalities. If we try to formulate the main principle of designing HTTP APIs, it will be: **you would rather design an API in a way that intermediary agents can read and interpret request and response metadata**.

The main disadvantage of HTTP APIs is that you have to rely on intermediary agents, from client frameworks to API gateways, to read the request metadata and perform actions based on it *without your consent*. This includes regulating timeouts and retry policies, logging, proxying, and sharding requests, among other things. Since HTTP-related specifications are complex and the concepts of REST can be challenging to comprehend, and software engineers do not always write perfect code, these intermediary agents (including partners' developers!) will sometimes interpret HTTP metadata *incorrectly*, especially when dealing with exotic and hard-to-implement standards. Usually, one of the stated reasons for developing new RPC frameworks is the desire to make working with the protocol simple and consistent, thereby reducing the likelihood of errors when writing integration code.

2. Quality of Solutions

The ability to read and interpret the metadata of requests and responses leads to the fragmentation of available software for working with HTTP APIs. There are plenty of tools on the market, being developed by many different companies and collaborations, and many of them are free to use:

- Proxies and gateways (nginx, Envoy, etc.)
- Different IDLs (first of all, OpenAPI) and related tools for working with specifications (Redoc, Swagger UI, etc.) and auto-generating code
- Programmer-oriented software that allows for convenient development and debugging of API clients (Postman, Insomnia), etc.

Of course, most of these instruments will work with APIs that utilize other paradigms. However, the ability to read HTTP metadata and interpret it *uniformly* makes it possible to easily design complex pipelines such as exporting nginx access logs to Prometheus and generating response status code monitoring dashboards in Grafana that work out of the box.

The downside of this versatility is the quality of these solutions and the amount of time one needs to integrate them, especially if one's technological stack is not common. On the other hand, the development of alternative technologies is usually driven by a single large IT company (such as Facebook, Google, or Apache Software Foundation). Such a framework might be less functional, but it will certainly be more homogeneous and qualitative in terms of convenience for developers, supporting users, and the number of known issues.

This observation applies not only to software but also to its creators. Developers' knowledge of HTTP APIs is fragmented as well. Almost every programmer is capable of working with HTTP APIs to some extent, but a significant number of them lack a thorough understanding of the standards and do not consult them while writing code. As a result, implementing business logic that effectively and consistently works with HTTP APIs can be more challenging than integrating alternative technologies. This statement holds true for both partner integrators and API providers themselves.

Additionally, let's emphasize that the HTTP API paradigm is currently the default choice for *public* APIs. Because of the aforementioned reasons, partners can integrate an HTTP API without significant obstacles, regardless of their technological stack. Moreover, the prevalence of the technology lowers the entry barrier and the requirements for the qualification of partners' engineers.

3. The Design Paradigm

Modern HTTP APIs inherited the design paradigm from the times when the HTTP protocol was mainly used to transfer hypertext. It implies that an HTTP request constitutes an operation performed on some object (*a resource*) identified by a URL. Many alternative solutions stick to other concepts; notably, in these technologies, URLs identify *a function* to call with the given parameters. This semantics doesn't exactly contradict the HTTP architectural principles, as making remote procedure calls is covered by the protocol pretty well, but it makes using some HTTP capabilities (such as, let's say, the `Range-*` headers) meaningless, and

some even dangerous as ambivalences of interpretations of some fields (such as, let's say, `ETag`) arise.

From the client developers' perspective, following the HTTP paradigms implies implementing an additional layer of logic that transforms calling methods on objects to HTTP operations on corresponding resources. RPC technologies are more convenient to integrate in this sense. (Although, any complex RPC API will require such an adapter level, and GraphQL requires it from the very beginning.)

4. The Question of Performance

When discussing the advantages of alternative technologies such as GraphQL, gRPC, Apache Thrift, etc., the argument of lower performance of JSON-over-HTTP APIs is often presented. Specifically, the following issues with the technology are commonly mentioned:

1. The verbosity of the JSON format:
    - Mandatory field names in every object, even for an array of similar entities
    - The large proportion of technical symbols (quotes, braces, commas, etc.) and the necessity to escape them in string values
2. The common approach of returning a full resource representation on resource retrieval requests, even if the client only needs a subset of the fields
3. The lower performance of data serializing and deserializing operations
4. The need to introduce additional encoding, such as Base64, to handle binary data
5. Performance quirks of the HTTP protocol itself, particularly the inability to serve multiple simultaneous requests through one connection.

Let's be honest: HTTP APIs do suffer from the listed problems. However, we can confidently say that the impact of these factors is often overestimated. The reason API vendors care little about HTTP API performance is that the actual overhead is not as significant as

perceived. Specifically:

1. Regarding the verbosity of the format, it is important to note that these issues are mainly relevant when compresiion algorithms are not utilized. Comparisons have shown that enabling compression algorithms such as gzip largely reduces the difference in sizes between JSON documents and alternative binary formats (and there are compression algorithms specifically designed for processing text data, such as brotli).

2. If necessary, API designers can customize the list of returned fields in HTTP APIs. It aligns well with both the letter and the spirit of the standard. However, as we already explained to the reader in the "Partial Updates" chapter, trying to minimize traffic by returning only subsets of data is rarely justified in well-designed APIs.

3. If standard JSON deserializers are used, the overhead compared to binary standards might indeed be significant. However, if this overhead is a real problem, it makes sense to consider alternative JSON serializers such as simdjson. Due to their low-level and highly optimized code, simdjson demonstrates impressive throughput which would be suitable for all APIs except some corner cases.

4. Generally speaking, the HTTP API paradigm implies that binary data (such as images or video files) is served through separate endpoints. Returning binary data in JSON is only necessary when a separate request for the data is a problem from the performance perspective. These situations are virtually non-existent in server-to-server interactions and/or if HTTP/2 or a higher protocol version is used.

5. The HTTP/1.1 protocol version is indeed a suboptimal solution if request multiplexing is needed. However, alternate approaches to tackling the problem usually rely on... HTTP/2. Of course, an HTTP API can also be served over HTTP/2.

Let us reiterate once more: JSON-over-HTTP APIs are *indeed* less performative than binary protocols. Nevertheless, we take the liberty to say that for a well-designed API in common subject areas switching to alternative protocols will generate quite a modest profit.

Advantages and Disadvantages of the JSON Format

It's not hard to notice that most of the claims regarding HTTP API performance are actually not about the HTTP protocol but the JSON format. There is no problem in developing an HTTP API that will utilize any binary format (including, for instance, Protocol Buffers). Then the difference between a Protobuf-over-HTTP API and a gRPC API would be just about using granular URLs, status codes, request / response headers, and the ensuing (in)ability to use integrated software tools out of the box.

However, on many occasions (including this book) developers prefer the textual JSON over binary Protobuf (Flatbuffers, Thrift, Avro, etc.) for a very simple reason: JSON is easy to read. First, it's a text format and doesn't require additional decoding. Second, it's self-descriptive, meaning that property names are included. Unlike Protobuf-encoded messages which are basically impossible to read without a `.proto` file, one can make a very good guess as to what a JSON document is about at a glance. Provided that request metadata in HTTP APIs is readable as well, we ultimately get a communication format that is easy to parse and understand with just our eyes.

Apart from being human-readable, JSON features another important advantage: it is strictly formal meaning it does not contain any constructs that can be interpreted differently in different architectures (with a possible exception of the sizes of numbers and strings), and the deserialization result aligns very well with native data structures (i.e., indexed and associative arrays) of almost every programming language. From this point of view, we actually had no other choice when selecting a format for code samples in this book.

If you happen to design a less general API for a specific subject area, we still recommend the same approach for choosing a format:

- Estimate the overhead of preparing and introducing tools to decipher binary protocols versus the overhead of using not the most optimal data transfer protocols.
- Make an assessment of what is more important to you: having a quality but restricted in its capabilities set of bundled software or having the possibility of using a broad range of tools that work with HTTP APIs, even though their quality is not that high.
- Evaluate the cost of finding developers proficient with the format.

Chapter 35. The REST Myth

Before we proceed to discuss HTTP API design patterns, we feel obliged to clarify one more important terminological issue. Often, an API matching the description we gave in the "On the HTTP API Concept and Terminology" chapter is called a "REST API" or a "RESTful API." In this Section, we don't use any of these terms as it makes no practical sense.

What is "REST"? In 2000, Roy Fielding, one of the authors of the HTTP and URI specifications, published his doctoral dissertation titled "Architectural Styles and the Design of Network-based Software Architectures," the fifth chapter of which was named "Representational State Transfer (REST)."

As anyone can attest by reading this chapter, it features a very much abstract overview of a distributed client-server architecture that is not bound to either HTTP or URL. Furthermore, it does not discuss any API design recommendations. In this chapter, Fielding methodically *enumerates restrictions* that any software engineer encounters when developing distributed client-server software. Here they are:

- The client and the server do not know how each of them is implemented
- Sessions are stored on the client (the "stateless" constraint)
- Data must be marked as cacheable or non-cacheable
- Interaction interfaces between system components must be uniform
- Network-based systems are layered, meaning every server may just be a proxy to another server
- The functionality of the client might be enhanced by the server providing code on demand.

That's it. With this, the REST definition is over. Fielding further concretizes some implementation aspects of systems under the stated restrictions. However, all these clarifications are no less abstract. Literally, the key abstraction for the REST architectural style is "resource"; any data that can have a name may be a resource.

The key conclusion that we might draw from the Fielding-2000 definition of REST is, generally speaking, that *any networking software in the world complies with the REST constraints*. The exceptions are very rare.

Consider the following:

- It is very hard to imagine any system that does not feature *any* level of uniformity of inter-component communication as it would be impossible to develop such a system. Ultimately, as we mentioned in the previous chapter, almost all network interactions are based on the IP protocol, which *is* a uniform interface.
- If there is a uniform communication interface, it can be mimicked if needed, so the requirement of client and server implementation independence can always be met.
- If we can create an alternative server, it means we can always have a layered architecture by placing an additional proxy between the client and the server.
- As clients are computational machines, they *always* store some state and cache some data.
- Finally, the code-on-demand requirement is a sly one as in a von Neumann architecture, we can always say that the data the client receives actually comprises instructions in some formal language.

Yes, of course, the reasoning above is a sophism, a reduction to absurdity. Ironically, we might take the opposite path to absurdity by proclaiming that REST constraints are never met. For instance, the code-on-demand requirement obviously contradicts the requirement of having an independently-implemented client and server as the client must be able to interpret the instructions the server sends written in a specific language. As for the "S" rule (i.e., the "stateless" constraint), it is very hard to find a system that does not store *any* client context as it's close to impossible to make anything *useful* for the client in this case. (And, by the way, Fielding explicitly requires that: "communication … cannot take advantage of any stored context on the server.")

Finally, in 2008, Fielding himself increased the entropy in the understanding of the concept by issuing a clarification explaining what he actually meant. In this article, among other things, he stated that:

- REST API development must focus on describing media types representing resources
- The client must be agnostic of these media types
- There must not be fixed resource names and operations with resources. Clients must extract this information from the server's responses.

The concept of "Fielding-2008 REST" implies that clients, after somehow obtaining an entry point to the API, must be able to communicate with the server having no prior knowledge of the API and definitely must not contain any specific code to work with the API. This requirement is much stricter than the ones described in the dissertation of 2000. Particularly, REST-2008 implies that there are no fixed URL templates; actual URLs to perform operations with the resource are included in the resource representation (this concept is known as HATEOAS). The dissertation of 2000 does not contain any definitions of "hypermedia" that contradict the idea of constructing such links based on the prior knowledge of the API (such as a specification).

NB: Leaving out the fact that Fielding rather loosely interpreted his own dissertation, let us point out that no system in the world complies with the Fielding-2008 definition of REST.

We have no idea why, out of all the overviews of abstract network-based software architecture, Fielding's concept gained such popularity. It is obvious that Fielding's theory, reflected in the minds of millions of software developers, became a genuine engineering subculture. By reducing the REST idea to the HTTP protocol and the URL standard, the chimera of a "RESTful API" was born, of which nobody knows the definition.

Do we want to say that REST is a meaningful concept? Definitely not. We only aimed to explain that it allows for quite a broad range of interpretations, which is simultaneously its main power and its main weakness.

On one hand, thanks to the multitude of interpretations, the API developers have built a perhaps vague but useful view of “proper” HTTP API architecture. On the other hand, the lack of concrete definitions has made REST API one of the most “holywar”-inspiring topics, and these holywars are usually quite meaningless as the popular REST concept has nothing to do with the REST described in Fielding's dissertation (and even more so, with the REST described in Fielding's manifesto of 2008).

The terms “REST architectural style” and its derivative “REST API” will not be used in the following chapters since it makes no practical sense as we explained above. We referred to the constraints described by Fielding many times in the previous chapters because, let us emphasize it once more, it is impossible to develop distributed client-server APIs without taking them into account. However, HTTP APIs (meaning JSON-over-HTTP endpoints utilizing the semantics described in the HTTP and URL standards) as we will describe them in the following chapter align well with the “average” understanding of “REST / RESTful API” as per numerous tutorials on the Web.

Chapter 36. Components of an HTTP Request and Their Semantics

The important exercise we must conduct is to describe the format of an HTTP request and response and explain the basic concepts. Many of these may seem obvious to the reader. However, the situation is that even the basic knowledge we require to move further is scattered across vast and fragmented documentation, causing even experienced developers to struggle with some nuances. Below, we will try to compile a structured overview that is sufficient to design HTTP APIs.

To describe the semantics and formats, we will refer to the brand-new RFC 9110, which replaces no fewer than nine previous specifications dealing with different aspects of the technology. However, a significant volume of additional functionality is still covered by separate standards. In particular, the HTTP caching principles are described in the standalone RFC 9111, while the popular `PATCH` method is omitted in the main RFC and is regulated by RFC 5789.

An HTTP request consists of (1) applying a specific verb to a URL, stating (2) the protocol version, (3) additional meta-information in headers, and (4) optionally, some content (request body):

```
POST /v1/orders HTTP/1.1
Host: our-api-host.tld
Content-Type: application/json

{
  "coffee_machine_id": 123,
  "currency_code": "MNT",
  "price": "10.23",
  "recipe": "lungo",
  "offer_id": 321,
  "volume": "800ml"
}
```

An HTTP response to such a request includes (1) the protocol version, (2) a status code with a corresponding message, (3) response headers, and (4) optionally, response content (body):

```
HTTP/1.1 201 Created
Location: /v1/orders/123
Content-Type: application/json

{
  "id": 123
}
```

NB: In HTTP/2 (and future HTTP/3), separate binary frames are used for headers and data instead of the holistic text format. However, this doesn't affect the architectural concepts we will describe below. To avoid ambiguity, we will provide examples in the HTTP/1.1 format. You can find detailed information about the HTTP/2 format here.

1. A URL

A Uniform Resource Locator (URL) is an addressing unit in an HTTP API. Some evangelists of the technology even use the term “URL space” as a synonym for “The World Wide Web.” It is expected that a proper HTTP API should employ an addressing system that is as granular as the subject area itself; in other words, each entity that the API can manipulate should have its own URL.

The URL format is governed by a separate standard developed by an independent body known as the Web Hypertext Application Technology Working Group (WHATWG). The concepts of URLs and Uniform Resource Names (URNs) together constitute a more general entity called Uniform Resource Identifiers (URIs). (The difference between the two is that a URL allows for *locating* a resource within the framework of some protocol whereas a URN is an “internal” entity name that does not provide information on how to find the resource.)

URLs can be decomposed into sub-components, each of which is optional. While the standard enumerates a number of legacy practices, such as passing logins and passwords in URLs or using non-UTF encodings, we will skip discussing those. Instead, we will focus on the following components that are relevant to the topic of HTTP API design:

- A scheme: a protocol to access the resource (in our case it is always `https:`)
- A host: a top-level address unit in the form of either a domain name or an IP address. A host might contain subdomains.
- A port.
- A path: a URL part between the host (including port) and the `?` or `#` symbols or the end of the line.
    - The path itself is usually decomposed into parts using the `/` symbol as a delimiter. However, the standard does not define any semantics for it.
    - Two paths, one ending with `/` and one without it (for example, `/root/leaf` and `/root/leaf/`), are considered different paths according to the standard. Conversely, two URLs that differ only in trailing slashes in their paths are considered different as well. However, we are unaware of a single argument to differentiate such URLs in practice.
    - Paths may contain `.` and `..` parts, which are supposed to be interpreted similarly to analogous symbols in file paths (meaning that `/root/leaf`, `/root/./leaf`, and `/root/branch/../leaf` are equivalent).
- A query: a URL part between the `?` symbol and either `#` or the end of the line.
    - A query is usually decomposed into `key=value` pairs split by the `&` character. Again, the standard does not require this or define the semantics.
    - Nor does the standard imply any normalization of the ordering. URLs that differ only in the order of keys in the queries are considered different.
- A fragment (also known as an anchor): a part of a URL that follows the `#` sign.

- Fragments are usually treated as addresses within the requested document and because of that are often omitted by user agents while executing the request.
- Two URLs that only differ in fragment parts may be considered equal or not, depending on the context.

In HTTP requests, the scheme, host, and port are usually (but not always) omitted and presumed to be equal to the connection parameters. (Fielding actually names this arrangement one of the biggest flaws in the protocol design.)

Traditionally, it is implied that paths describe a strict hierarchy of resource subordination (for example, the URL of a specific coffee machine in our API could look like `places/{id}/coffee-machines/{id}`, since a coffee machine strictly belongs to one coffee shop), while query parameters express non-strict hierarchies and operation parameters (for example, the URL for searching listings could look like `search?location=<map point>`).

Additionally, the standard contains rules for serializing, normalizing, and comparing URLs, knowing which can be useful for an HTTP API developer.

2. Headers

Headers contain *metadata* associated with a request or a response. They might describe properties of entities being passed (e.g., `Content-Length`), provide additional information regarding a client or a server (e.g., `User-Agent`, `Date`, etc.) or simply contain additional fields that are not directly related to the request/response semantics (such as `Authorization`).

The important feature of headers is the possibility to read them before the message body is fully transmitted. This allows for altering request or response handling depending on the headers, and it is perfectly fine to manipulate headers while proxying requests. Many network agents actually do this, i.e., add, remove, or modify headers while proxying requests. In particular, modern web browsers automatically add a number of technical headers, such as `User-Agent`, `Origin`, `Accept-`

`Language`, `Connection`, `Referer`, `Sec-Fetch-*`, etc., and modern server software automatically adds or modifies such headers as `X-Powered-By`, `Date`, `Content-Length`, `Content-Encoding`, `X-Forwarded-For`, etc.

This freedom in manipulating headers can result in unexpected problems if an API uses them to transmit data as the field names developed by an API vendor can accidentally overlap with existing conventional headers, or worse, such a collision might occur in the future at any moment. To avoid this issue, the practice of adding the prefix `X-` to custom header names was frequently used in the past. More than ten years ago this practice was officially discouraged (see the detailed overview in RFC 6648). Nevertheless, the prefix has not been fully abolished, and many semi-standard headers still contain it (notably, `X-Forwarded-For`). Therefore, using the `X-` prefix reduces the probability of collision but does not eliminate it. The same RFC reasonably suggests using the API vendor name as a prefix instead of `X-`. (We would rather recommend using both, i.e., sticking to the `X-ApiName-FieldName` format. Here `X-` is included for readability [to distinguish standard fields from custom ones], and the company or API name part helps avoid collisions with other non-standard header names).

Additionally, headers are used as control flow instructions for so-called "content negotiation," which allows the client and server to agree on a response format (through `Accept*` headers) and to perform conditional requests that aim to reduce traffic by skipping response bodies, either fully or partially (through `If-*` headers, such as `If-Range`, `If-Modified-Since`, etc.)

3. HTTP Verbs

One important component of an HTTP request is a method (verb) that describes the operation being applied to a resource. RFC 9110 standardizes eight verbs — namely, `GET`, `POST`, `PUT`, `DELETE`, `HEAD`, `CONNECT`, `OPTIONS`, and `TRACE` — of which we as API developers are interested in the former four. The `CONNECT`, `OPTIONS`, and `TRACE` methods are technical and rarely used in HTTP APIs (except for `OPTIONS`, which needs to be implemented to ensure access to the API from a web browser). Theoretically, the `HEAD` verb, which allows for requesting *resource*

metadata only, might be quite useful in API design. However, for reasons unknown to us, it did not take root in this capacity.

Apart from RFC 9110, many other specifications propose additional HTTP verbs, such as `COPY`, `LOCK`, `SEARCH`, etc. — the full list can be found in the registry. However, only one of them gained widespread popularity — the `PATCH` method. The reasons for this state of affairs are quite trivial: the five methods (`GET`, `POST`, `PUT`, `DELETE`, and `PATCH`) are enough for almost any API.

HTTP verbs define two important characteristics of an HTTP call:

- Semantics: what the operation *means*
- Side effects:
  - Whether the request modifies any resource state or if it is safe (and therefore, could it be cached)
  - Whether the request is idempotent or not.

Verb	Semantics	Is safe (non-modifying)	Is idempotent	Can have a body
GET	Returns a representation of a resource	Yes	Yes	Should not
PUT	Replaces (fully overwrites) a resource with a provided entity	No	Yes	Yes
DELETE	Deletes a resource	No	Yes	Should not

Verb	Semantics	Is safe (non-modifying)	Is idempotent	Can have a body
POST	Processes a provided entity according to its internal semantics	No	No	Yes
PATCH	Modifies (partially overwrites) a resource with a provided entity	No	No	Yes

NB: Contrary to a popular misconception, the `POST` method is not limited to creating new resources.

The most important property of modifying idempotent verbs is that **the URL serves as an idempotency key for the request**. The `PUT /url` operation fully overwrites a resource, so repeating the request won't change the resource. Conversely, retrying a `DELETE /url` request must leave the system in the same state where the `/url` resource is deleted. Regarding the `GET /url` method, it must semantically return the representation of the same target resource `/url`. If it exists, its implementation must be consistent with prior `PUT` / `DELETE` operations. If the resource was overwritten via `PUT /url`, a subsequent `GET /url` call must return a representation that matches the entity enclosed in the `PUT /url` request. In the case of JSON-over-HTTP APIs, this simply means that `GET /url` returns the same data as what was passed in the preceding `PUT /url`, possibly normalized and equipped with default values. On the other hand, a `DELETE /url` call must remove the resource, resulting in subsequent `GET /url` requests returning a `404` or `410` error.

The idempotency and symmetry of the `GET` / `PUT` / `DELETE` methods imply that neither `GET` nor `DELETE` can have a body as no reasonable meaning could be associated with it. However, most web server software allows these methods to have bodies and transmits them further to the endpoint handler, likely because many software engineers are unaware of the semantics of the verbs (although we strongly discourage relying on this behavior).

For obvious reasons, responses to modifying endpoints are not cached (though there are some conditions to use a response to a `POST` request as cached data for subsequent `GET` requests). This ensures that repeating `POST` / `PUT` / `DELETE` / `PATCH` requests will hit the server as no intermediary agent can respond with a cached result. In the case of a `GET` request, it is generally not true. Only the presence of `no-store` or `no-cache` directives in the response guarantees that the subsequent `GET` request will reach the server.

One of the most widespread HTTP API design antipatterns is violating the semantics of HTTP verbs:

- Placing modifying operations in a `GET` handler. This can lead to the following problems:
    - Interim agents might respond to such a request using a cached value if a required caching directive is missing, or vice versa, automatically repeat a request upon receiving a network timeout.
    - Some agents consider themselves eligible to traverse hyper-references (i.e., making HTTP `GET` requests) without the explicit user's consent. For example, social networks and messengers perform such calls to generate a preview for a link when a user tries to share it.
- Placing non-idempotent operations in `PUT` / `DELETE` handlers. Although interim agents do not typically repeat modifying requests regardless of their alleged idempotency, a client or server framework can easily do so. This mistake is often coupled with requiring passing a body alongside a `DELETE` request to discern the specific object that needs to be deleted, which per se is a problem as any interim agent might discard such a body.

- Ignoring the `GET` / `PUT` / `DELETE` symmetry requirement. This can manifest in different ways, such as:
    - Making a `GET /url` operation return data even after a successful `DELETE /url` call
    - Making a `PUT /url` operation take the identifiers of the entities to modify from the request body instead of the URL, resulting in the `GET /url` operation's inability to return a representation of the entity passed to the `PUT /url` handler.

4. Status Codes

A status code is a machine-readable three-digit number that describes the outcome of an HTTP request. There are five groups of status codes:

- `1xx` codes are informational. Among these, the `100 Continue` code is probably the only one that is commonly used.
- `2xx` codes indicate that the operation was successful.
- `3xx` codes are redirection codes, implying that additional actions must be taken to consider the operation fully successful.
- `4xx` codes represent client errors
- `5xx` codes represent server errors.

NB: The separation of codes into groups by the first digit is of practical importance. If the client is unaware of the meaning of an `xyz` code returned by the server, it must conduct actions as if an `x00` code was received.

The idea behind status codes is obviously to make errors machine-readable so that all interim agents can detect what has happened with a request. The HTTP status code nomenclature effectively describes nearly every problem applicable to an HTTP request, such as invalid `Accept-*` header values, missing `Content-Length`, unsupported HTTP verbs, excessively long URIs, etc.

Unfortunately, the HTTP status code nomenclature is not well-suited for describing errors in *business logic*. To return machine-readable errors related to the semantics of the operation, it is necessary either to use status codes unconventionally (i.e., in violation of the standard) or to

enrich responses with additional fields. Designing custom errors in HTTP APIs will be discussed in the corresponding chapter.

NB: Note the problem with the specification design. By default, all `4xx` codes are non-cacheable, but there are several exceptions, namely the `404`, `405`, `410`, and `414` codes. While we believe this was done with good intentions, the number of developers aware of this nuance is likely to be similar to the number of HTTP specification editors.

One Important Remark Regarding Caching

Caching is a crucial aspect of modern microservice architecture design. It can be tempting to control caching at the protocol level, and the HTTP standard provides various tools to facilitate this. However, the author of this book must warn you: if you decide to utilize these tools, it is essential to thoroughly understand the standard. Flaws in the implementation of certain techniques can result in disruptive behavior. The author personally experienced a major outage caused by the aforementioned lack of knowledge regarding the default cacheability of `404` responses. In this incident, some settings for an important geographical area were mistakenly deleted. Although the problem was quickly localized and the settings were restored, the service remained inoperable in the area for several hours because clients had cached the `404` response and did not request it anew until the cache had expired.

One Important Remark Regarding Consistency

One parameter might be placed in different components of an HTTP request. For example, an identifier of a partner making a request might be passed as part of:

- A domain name, e.g., `{partner_id}.domain.tld`
- A path, e.g., `/v1/{partner_id}/orders`
- A query parameter, e.g. `/v1/orders?partner_id=<partner_id>`
- A header value, e.g.

```
GET /v1/orders HTTP/1.1
X-ApiName-Partner-Id: <partner_id>
```

- A field within the request body, e.g.

```
POST /v1/orders/retrieve HTTP/1.1

{
  "partner_id": <partner_id>
}
```

There are also more exotic options, such as placing a parameter in the scheme of a request or in the `Content-Type` header.

However, when we move a parameter around different components, we face three annoying issues:

- Some tokens are case-sensitive (path, query parameters, JSON field names), while others are not (domain and header names)
    - With header *values*, there is even more chaos: some of them are required to be case-insensitive (e.g., `Content-Type`), while others are prescribed to be case-sensitive (e.g., `ETag`)
- Allowed symbols and escaping rules differ as well:
    - Notably, there is no widespread practice for escaping the `/`, `?`, and `#` symbols in a path
    - Unicode symbols in domain names are allowed (though not universally supported) through a peculiar encoding technique called "Punycode"
- Traditionally, different casings are used in different parts of an HTTP request:
    - `kebab-case`in domains, headers, and paths
    - `snake_case` in query parameters
    - `snake_case` or `camelCase` in request bodies.

    Furthermore, using both `snake_case` and `camelCase` in domain names is impossible as the underscore sign is not allowed and capital letters will be lowercased during URL normalization.

Theoretically, it is possible to use `kebab-case` everywhere. However, most programming languages do not allow variable names and object fields in `kebab-case`, so working with such an API would be quite inconvenient.

To wrap this up, the situation with casing is so spoiled and convoluted that there is no consistent solution to employ. In this book, we follow this rule: tokens are cased according to the common practice for the corresponding request component. If a token's position changes, the casing is changed as well. (However, we're far from recommending following this approach unconditionally. Our recommendation is rather to try to avoid increasing the entropy by choosing a solution that minimizes the probability of misunderstanding.)

NB: Strictly speaking, JSON stands for "JavaScript Object Notation," and in JavaScript, the default casing is `camelCase`. However, we dare to say that JSON ceased to be a format bound to JavaScript long ago and is now a universal format for organizing communication between agents written in different programming languages. Employing `camel_case` allows for easily moving a parameter from a query to a body, which is the most frequent case. Although the inverse solution (i.e., using `camelCase` in query parameter names) is also possible.

Chapter 37. Organizing HTTP APIs Based on the REST Principles

Now let's discuss the specifics: what does it mean exactly to “follow the protocol's semantics” and “develop applications in accordance with the REST architectural style”? Remember, we are talking about the following principles:

- Operations must be stateless
- Data must be marked as cacheable or non-cacheable
- There must be a uniform interface of communication between components
- Network systems are layered.

We need to apply these principles to an HTTP-based interface, adhering to the letter and spirit of the standard:

- The URL of an operation must point to the resource the operation is applied to, serving as a cache key for `GET` operations and an idempotency key for `PUT` and `DELETE` operations.
- HTTP verbs must be used according to their semantics.
- Properties of the operation, such as safety, cacheability, idempotency, as well as the symmetry of `GET` / `PUT` / `DELETE` methods, request and response headers, response status codes, etc., must align with the specification.

NB: We're deliberately skipping many nuances of the standard:

- A caching key might be composite (i.e., include request headers) if the response contains the `Vary` header.

- An idempotency key might also be composite if the request contains the `Range` header.

- If there are no explicit cache control headers, the caching policy will not be defined by the HTTP verb alone. It will also depend on the response status code, other request and response headers, and platform policies.

> To keep the chapter size reasonable, we will not discuss these details, but we highly recommend reading the standard thoroughly.

Let's talk about organizing HTTP APIs based on a specific example. Imagine an application start procedure: as a rule of thumb, the application requests the current user profile and important information regarding them (in our case, ongoing orders), using the authorization token saved in the device's memory. We can propose a straightforward endpoint for this purpose:

```
GET /v1/state HTTP/1.1
Authorization: Bearer <token>
→
HTTP/1.1 200 OK

{ "profile", "orders" }
```

Upon receiving such a request, the server will check the validity of the token, fetch the identifier of the user `user_id`, query the database, and return the user's profile and the list of their orders.

This simple monolith API service violates several REST architectural principles:

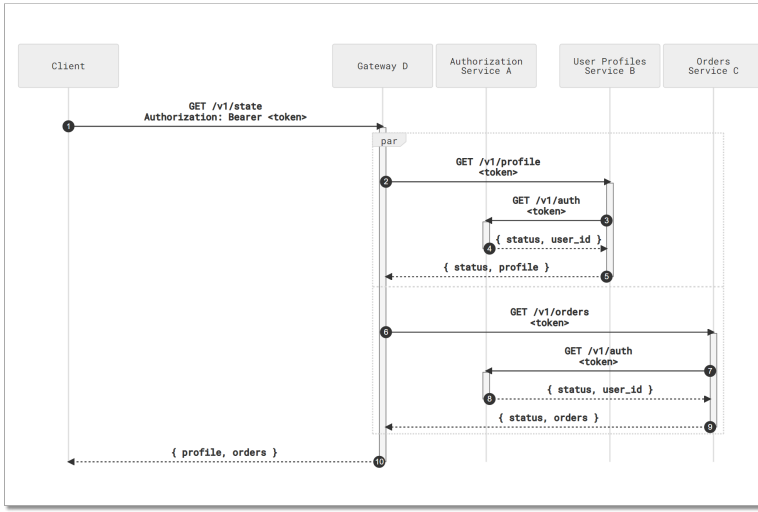
- There is no obvious solution for caching responses on the client side (the order state is being frequently updated and there is no sense in saving it)
- The operation is stateful as the server must keep tokens in memory to retrieve the user identifier, to which the requested data is bound.
- The system comprises a single layer, and therefore, the question of a uniform interface is meaningless.

While scaling the backend is not a problem, this approach works. However, with the audience and the service's functionality (and the number of software engineers working on it) growing, we sooner or later face the fact that this monolith architecture costs too much in overhead charges. Imagine we decided to decompose this single backend into four microservices:

- Service A checks authentication tokens
- Service B stores user accounts
- Service C stores orders
- Gateway service D routes incoming requests to other microservices.

This implies that a request traverses the following path:

- Gateway D receives the request and sends it to both Service C and Service D.
- C and D call Service A to check the authentication token (passed as a proxied `Authorization` header or as an explicit request parameter) and return the requested data — the user's profile and the list of their orders.
- Service D merges the responses and sends them back to the client.



The original microservice mesh. Click to enlarge

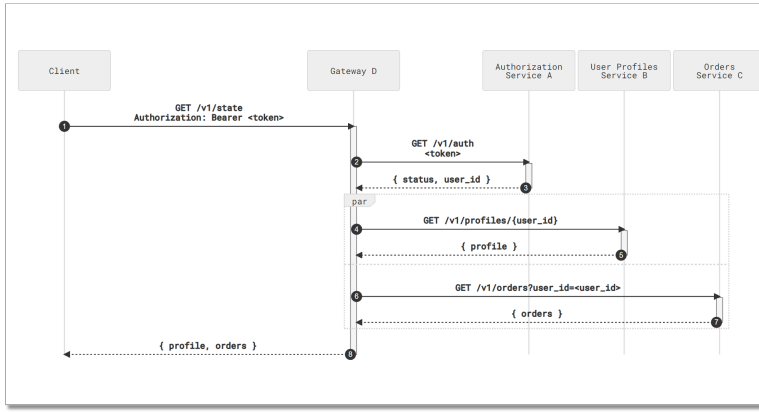
It is quite obvious that in this setup, we put excessive load on the authorization service as every nested microservice now needs to query it. Even if we abolish checking the authenticity of internal requests, it won't help as services B and C can't know the identifier of the user. Naturally, this leads to the idea of propagating the once-retrieved `user_id` through the microservice mesh:

- Gateway D receives a request and exchanges the token for `user_id` through service A
- Gateway D queries service B:

  ```
  GET /v1/profiles/{user_id}
  ```

  and service C:

  ```
  GET /v1/orders?user_id=<user id>
  ```



Step 2. Adding explicit user identifiers. Click to enlarge

NB: We used the `/v1/orders?user_id` notation and not, let's say, `/v1/users/{user_id}/orders`, because of two reasons:

- The orders service stores orders, not users, and it would be logical to reflect this fact in URLs
- If in the future, we require allowing several users to share one order, the `/v1/orders?user_id` notation will better reflect the relations between entities.

  We will discuss organizing URLs in HTTP APIs in more detail in the next chapter.

Now both services A and B receive the request in a form that makes it redundant to perform additional actions (identifying the user through service A) to obtain the result. By doing so, we refactored the interface *allowing a microservice to stay within its area of responsibility*, thus making it compliant with the stateless constraint.

Let us emphasize that the difference between **stateless** and **stateful** approaches is not clearly defined. Microservice B stores the client state (i.e., the user profile) and therefore is stateful according to Fielding's dissertation. However, we rather intuitively agree that storing profiles

and just checking token validity is a better approach than doing all the same operations plus having the token cache. In fact, we rather embrace the *logical* principle of separating abstraction levels which we discussed in detail in the corresponding chapter:

- **Microservices should be designed to clearly outline their responsibility area and to avoid storing data belonging to other abstraction levels**

- External entities should be just context identifiers, and microservices should not interpret them

- If operations with external data are unavoidable (for example, the authority making a call must be checked), the **operations must be organized in a way that reduces them to checking the data integrity**.

  In our example, we might get rid of unnecessary calls to service A in a different manner — by using stateless tokens, for example, employing the JWT standard. Then services B and C would be capable of deciphering tokens and extracting user identifiers on their own.

Let us take a step further and notice that the user profile rarely changes, so there is no need to retrieve it each time as we might cache it at the gateway level. To do so, we must form a cache key which is essentially the client identifier. We can do this by taking a long way:

- Before requesting service B, generate a cache key and probe the cache
- If the data is in the cache, respond with the cached snapshot; if it is not, query service B and cache the response.

Alternatively, we can rely on HTTP caching which is most likely already implemented in the framework we use or easily added as a plugin. In this scenario, gateway D requests the `/v1/profiles/{user_id}` resource in service B, retrieves the data alongside the cache control headers, and caches it locally.

Now let's shift our attention to service C. The results retrieved from it might also be cached. However, the state of an ongoing order changes more frequently than the user's profiles, and returning an invalid state might entail objectionable consequences. However, as discussed in the "Synchronization Strategies" chapter, we need optimistic concurrency control (i.e., the resource revision) to ensure the functionality works correctly, and nothing could prevent us from using this revision as a cache key. Let service C return a tag describing the current state of the user's orders:

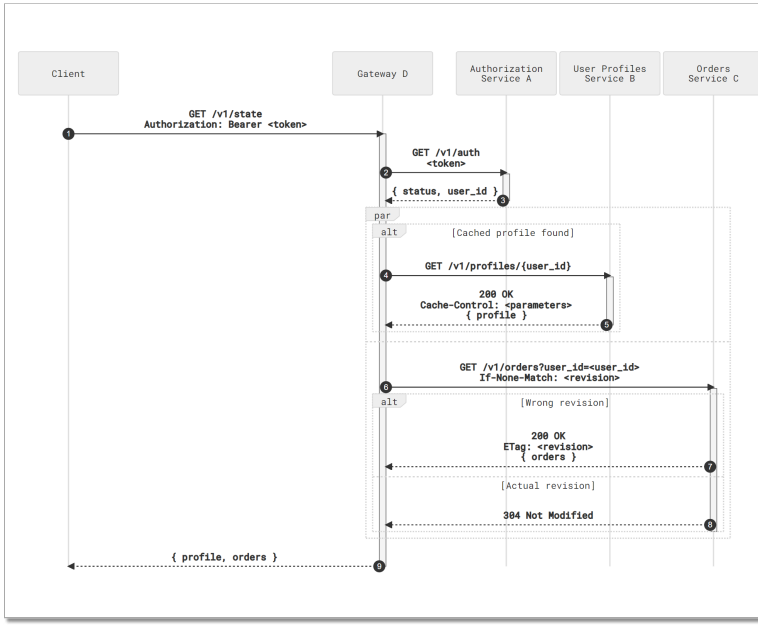
```
GET /v1/orders?user_id=<user_id> HTTP/1.1
→
HTTP/1.1 200 OK
ETag: <revision>
…
```

Then gateway D can be implemented following this scenario:

1. Cache the response of `GET /v1/orders?user_id=<user_id>` using the URL as a cache key
2. Upon receiving a subsequent request:
    - Fetch the cached state, if any
    - Query service C passing the following parameters:

      ```
      GET /v1/orders?user_id=<user_id> HTTP/1.1
      If-None-Match: <revision>
      ```

    - If service C responds with a `304 Not Modified` status code, return the cached state
    - If service C responds with a new version of the data, cache it and then return it to the client.



Step 2. Adding server-side caches. Click to enlarge

By employing this approach [using `ETags` to control caching], we automatically get another pleasant bonus. We can reuse the same data in the order creation endpoint design. In the optimistic concurrency control paradigm, the client must pass the actual revision of the `orders` resource to change its state:

```
POST /v1/orders HTTP/1.1
If-Match: <revision>
```

Gateway D will add the user's identifier to the request and query service C:

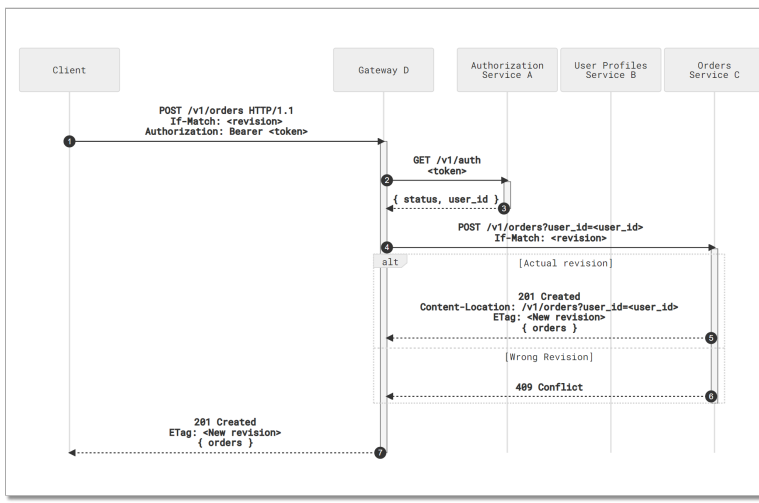
```
POST /v1/orders?user_id=<user_id> HTTP/1.1
If-Match: <revision>
```

If the revision is valid and the operation is executed, service C might return the updated list of orders alongside the new revision:

```
HTTP/1.1 201 Created
Content-Location: /v1/orders?user_id=<user_id>
ETag: <new revision>

{ /* The updated list of orders */ }
```

and gateway D will update the cache with the current data snapshot.



Creating a new order. Click to enlarge

Importantly, after this API refactoring, we end up with a system in which we can *remove gateway D* and make the client itself perform its duty. Nothing prevents the client from:

- Storing `user_id` on its side (or retrieving it from the token, if the format allows it) as well as the last known `ETag` of the order list
- Instead of a single `GET /v1/state` request performing two HTTP calls (`GET /v1/profiles/{user_id}` and `GET /v1/orders?user_id=<user_id>`) which might be multiplexed thanks to HTTP/2
- Caching the result on its side using standard libraries and/or plugins.

From the perspective of implementing services B and C, the presence of a gateway affects nothing, with the exception of security checks. Vice versa, we might add a nested gateway to, let's say, split order storage into “cold” and “hot” ones, or make either service B or C work as a gateway themselves.

If we refer to the beginning of the chapter, we will find that we designed a system fully compliant with the REST architectural principles:

- Requests to services contain all the data needed to process the request
- The interaction interface is uniform to the extent that we might freely transfer gateway functions to the client or another intermediary agent
- Every resource is marked as cacheable

Let us reiterate once more that we can achieve exactly the same qualities with RPC protocols by designing formats for describing caching policies, resource versions, reading and modifying operation metadata, etc. However, the author of this book would firstly, express doubts regarding the quality of such a custom solution and secondly, emphasize the considerable amount of code needed to be written to realize all the functionality stated above.

Authorizing Stateless Requests

Let's elaborate a bit on the no-authorizing service solution (or, to be more precise, the solution with the authorizing functionality being implemented as a library or a local daemon inside services B, C, and D) with all the data embedded in the authorization token itself. In this scenario, every service performs the following actions:

1. Receives a request like this:

```
GET /v1/profiles/{user_id}
Authorization: Bearer <token>
```

2. Deciphers the token and retrieves a payload. For example, in the following format:

```
{
  // The identifier of a user
  // who owns the token
  "user_id",
  // Token creation timestamp
  "iat"
}
```

3. Checks that the permissions stated in the token payload match the operation parameters (in our case, compares `user_id` passed as a query parameter with `user_id` encrypted in the token itself) and decides on the validity of the operation.

The necessity to compare two `user_id`s might appear illogical and redundant. However, this opinion is invalid; it originates from the widespread (anti)pattern we started the chapter with, namely the stateful determining of operation parameters:

```
GET /v1/profile
Authorization: Bearer <token>
```

Such an endpoint effectively performs all three access control operations in one place:

- *Authenticates* the user by searching the passed token in the token storage
- *Identifies* the user by retrieving the identifier bound to the token
- *Authorizes* the operation by enriching its parameters and *implicitly* stipulating that users always have access to their own data.

The problem with this approach is that *splitting* these three operations is not possible. Let us remind the reader about the authorization options we described in the "Authenticating Partners and Authorizing API Calls" chapter: in a complex enough system we will have to solve the problem of allowing user X to make actions on behalf of user Y. For example, if we sell the functionality of ordering beverages as a B2B API, the CEO of the partner company might want to control (personally or programmatically) the orders the employees make.

In the case of the "triple-stacked" access checking endpoint, our only option is implementing a new endpoint with a new interface. With stateless tokens, we might do the following:

1. Include in the token *a list* of the users that the token allows access to:

   ```
   {
     // The list of identifiers
     // of user profiles accessible
     // with the token
     "user_ids",
     // Token creation timestamp
     "iat"
   }
   ```

2. Modify the permission-checking procedure (i.e., make changes in the code of a local SDK or a daemon) so that it allows performing the action if the `user_id` query parameter value is included in the `user_ids` list from the token payload.

   This approach might be further enhanced by introducing granular permissions to carry out specific actions, access levels, additional ACL service calls, etc.

Importantly, the visible redundancy of the format ceases to exist: `user_id` in the request is now not duplicated in the token payload as these identifiers carry different semantics: *on which resource* the operation is performed against *who* performs it. The two often coincide, but this coincidence is just a special case. Unfortunately, this doesn't negate the fact that it's quite easy simply to forget to implement this unobvious check in the code. This is the way.

Chapter 38. Designing a Nomenclature of URLs. The CRUD Operations

As we noted on several occasions in the previous chapters, neither the HTTP and URL standards nor REST architectural principles prescribe concrete semantics for the meaningful parts of a URL (notably, path fragments and key-value pairs in the query). **The rules for organizing URLs in an HTTP API exist *only* to improve the API's readability and consistency from the developers' perspective**. However, this doesn't mean they are unimportant. Quite the opposite: URLs in HTTP APIs are a means of describing abstraction levels and entities' responsibility areas. A well-designed API hierarchy should be reflected in a well-designed URL nomenclature.

NB: The lack of specific guidance from the specification editors naturally led to developers inventing it themselves. Many of these spontaneous practices can be found on the Internet, such as the requirement to use only nouns in URLs. They are often claimed to be a part of the standards or REST architectural principles (which they are not). Nevertheless, deliberately ignoring such self-proclaimed "best practices" is a rather risky decision for an API vendor as it increases the chances of being misunderstood.

Traditionally, the following semantics are considered to be the default:

- Path components (i.e., fragments between `/` symbols) are used to organize nested resources, such as `/partner/{id}/coffee-machines/{id}`. A path can be further extended by adding new suffixes to indicate subordinate sub-resources.
- Query parameters are used to indicate non-strict connections (i.e., "many-to-many" relations such as `/recipes/?partner=<partner_id>`) or as a means to pass operation parameters (`/search/?recipe=lungo`).

This convention allows for reflecting almost any API's entity nomenclature decently and it is more than reasonable to follow it (and it's unreasonable to defiantly neglect it). However, this indistinctly defined logic inevitably leads to numerous variants of interpreting it:

1. Where do metadata of operations end and "regular" data begin, and how acceptable is it to duplicate fields in both places? For example, one common practice in HTTP APIs is to indicate the type of returned data by adding an "extension" to the URL, similar to file names in file systems (i.e., when accessing the resource `/v1/orders/{id}.xml`, the response will be in XML format, and when accessing `/v1/orders/{id}.json`, it will be in JSON format). On the one hand, `Accept*` headers are intended to determine the data format. On the other hand, code readability is clearly improved by introducing a "format" parameter directly in the URL.

2. As a consequence of the previous point, how should the version of the API itself be correctly indicated in an HTTP API? At least three options can be suggested, each of which fully complies with the letter of the standard:

    - A path parameter: `/v1/orders/{id}`
    - A query parameter: `/orders/{id}?version=1`
    - A header:

      ```
      GET /orders/{id} HTTP/1.1
      X-OurCoffeeAPI-Version: 1
      ```

    Even more exotic options can be added here, such as specifying the schema in a customized media type or request protocol.

3. How exactly should the endpoints connecting two entities lacking a clear relation between them be organized? For example, how should a URL for preparing a lungo on a specific coffee machine look?

    - `/coffee-machines/{id}/recipes/lungo/prepare`
    - `/recipes/lungo/coffee-machines/{id}/prepare`
    - `/coffee-machines/{id}/prepare?recipe=lungo`
    - `/recipes/lungo/prepare?coffee_machine_id=<id>`
    - `/prepare?coffee_machine_id=<id>&recipe=lungo`
    - `/action=prepare&coffee_machine_id=<id>&recipe=lungo`

All these options are semantically viable and generally speaking equitable.

4. How strictly should the literal interpretation of the `VERB /resource` construct be enforced? If we agree to follow the "only nouns in the URLs" rule (logically, a verb cannot be applied to a verb, right?) then we should use `preparer` or `preparator` in the examples above (and the `/action=prepare&coffee_machine_id=<id>&recipe=lungo` is unacceptable at all as there is no object to act upon). However, this adds visual noise in the form of "ator" suffixes but definitely doesn't make the code more concise or readable.

5. If the call signature implies that the operation is by default unsafe or non-idempotent, does it mean that the operation *must* be unsafe or non-idempotent? As HTTP verbs bear double semantics (the meaning of the operation vs. possible side effects), it implies ambiguity in organizing APIs. Let's consider the `/v1/search` resource from our study API. Which verb should be used to request it?

    - On one hand, `GET /v1/search?query=<search query>` explicitly declares that there are no side effects (no state is overwritten) and the results can be cached (given that all significant parameters are passed as parts of the URL).
    - On the other hand, a response to a `GET /v1/search` request must contain *a representation of the `/search` resource*. Are search results a representation of a search engine? The meaning of a "search" operation is much better described as "processing the representation enclosed in the request according to the resource's own specific semantics," which is exactly the definition of the `POST` method. Additionally, how could we cache search requests? The results page is dynamically formed from a plethora of various sources, and a subsequent request with the same query might yield a different result.

In other words, with any operation that runs an algorithm rather than returns a predefined result (such as listing offers relevant to a search phrase), we will have to decide what to choose: following verb semantics or indicating side effects? Caching the results or hinting that the results are generated on the fly?

NB: The authors of the standard are also concerned about this dichotomy and have finally proposed the `QUERY` HTTP method, which is basically a safe (i.e., non-modifying) version of `POST`. However, we do not expect it to gain widespread adoption just as the existing `SEARCH` verb did not.

Unfortunately, we don't have simple answers to these questions. Within this book, we adhere to the following approach: the call signature should, first and foremost, be concise and readable. Complicating signatures for the sake of abstract concepts is undesirable. In relation to the mentioned issues, this means that:

1. Operation metadata should not change the meaning of the operation. If a request reaches the final microservice without any headers at all, it should still be executable, although some auxiliary functionality may degrade or be absent.

2. We use versioning in the path for one simple reason: all other methods make sense only if, when changing the major version of the protocol, the URL nomenclature remains the same. However, if the resource nomenclature can be preserved, there is no need to break backward compatibility.

3. Hierarchies are indicated if they are unequivocal. If a low-level entity is a full subordinate of a higher-level entity, the relation will be expressed with nested path fragments.

    - If there are doubts about the hierarchy persisting during further development of the API, it is more convenient to create a new root path prefix rather than employ nested paths.

4. For “cross-domain” operations (i.e., when it is necessary to refer to entities of different abstraction levels within one request) it is better to have a dedicated resource specifically for this operation (e.g., in the example above, we would prefer the `/prepare?coffee_machine_id=<id>&recipe=lungo` signature).

5. The semantics of the HTTP verbs take priority over false non-safety / non-idempotency warnings. Furthermore, the author of this book prefers using `POST` to indicate any unexpected side effects of an operation, such as high computational complexity, even if it is fully safe.

NB: Passing variables as either query parameters or path fragments affects not only readability. Let's consider the example from the previous chapter and imagine that gateway D is implemented as a stateless proxy with a declarative configuration. Then receiving a request like this:

- `GET /v1/state?user_id=<user_id>`

  and transforming it into a pair of nested sub-requests:

- `GET /v1/profiles?user_id=<user_id>`

- `GET /v1/orders?user_id=<user_id>`

  would be much more convenient than extracting identifiers from the path or some header and putting them into query parameters. The former operation [replacing one path with another] is easily described declaratively and is supported by most server software out of the box. On the other hand, retrieving data from various components and rebuilding requests is a complex functionality that most likely requires a gateway supporting scripting languages and/or plugins for such manipulations. Conversely, the automated creation of monitoring panels in services like the Prometheus+Grafana bundle (or basically any other log analyzing tool) is much easier to organize by path prefix than by a synthetic key computed from request parameters.

All this leads us to the conclusion that maintaining an identical URL structure when paths are fixed and all the parameters are passed as query parameters will result in an even more uniform interface, although less readable and semantic. In internal systems, preferring the convenience of usage over readability is sometimes an obvious decision. In public APIs, we would rather discourage implementing this approach.

The CRUD Operations

One of the most popular tasks solved by exposing HTTP APIs is implementing CRUD interfaces. The "CRUD" acronym (which stands for **C**reate, **R**ead, **U**pdate, **D**elete) was popularized in 1983 by James Martin and gained a second wind with HTTP APIs gaining widespread acclaim. The key concept is that every CRUD operation matches a specific HTTP verb:

- The "create" operation corresponds to the HTTP `POST` method.
- The "read" operation corresponds to returning a representation of the resource via the `GET` method.
- The "update" operation corresponds to overwriting a resource with either the `PUT` or `PATCH` method.
- The "delete" operation corresponds to deleting a resource with the `DELETE` method.

NB: In fact, this correspondence serves as a mnemonic to choose the appropriate HTTP verb for each operation. However, we must warn the reader that verbs should be chosen according to their definition in the standards, not based on mnemonic rules. For example, it might seem like deleting the third element in a list should be organized via the `DELETE` method:

- `DELETE /v1/list/{list_id}/?position=3`

  However, as we remember, doing so is a grave mistake: first, such a call is non-idempotent, and second, it violates the `GET` / `DELETE` consistency principle.

The CRUD/HTTP correspondence might appear convenient as every resource is forced to have its own URL and each operation has a suitable verb. However, upon closer examination, we will quickly understand that the correspondence presents resource manipulation in a very simplified, and, even worse, poorly extensible way.

1. Creating

Let's start with the resource creation operation. As we remember from the "Synchronization Strategies" chapter, in any important subject area, creating entities must be an idempotent procedure that ideally allows for controlling concurrency. In the HTTP API paradigm, idempotent creation could be implemented using one of the following three approaches:

1. Through the `POST` method with passing an idempotency token (in which capacity the resource `ETag` might be employed):

```
POST /v1/orders/?user_id=<user_id> HTTP/1.1
If-Match: <revision>

{ … }
```

2. Through the `PUT` method, implying that the entity identifier is generated by the client. Revision still could be used for controlling concurrency; however, the idempotency token is the URL itself:

```
PUT /v1/orders/{order_id} HTTP/1.1
If-Match: <revision>

{ … }
```

3. By creating a draft with the `POST` method and then committing it with the `PUT` method:

```
POST /v1/drafts HTTP/1.1

{ … }
→
HTTP/1.1 201 Created
Location: /v1/drafts/{id}
```

```
PUT /v1/drafts/{id}/commit
If-Match: <revision>

{"status": "confirmed"}
→
HTTP/1.1 200 OK
Location: /v1/orders/{id}
```

Approach #2 is rarely used in modern systems as it requires trusting the client to generate identifiers properly. If we consider options #1 and #3, we must note that the latter conforms to HTTP semantics better as `POST` requests are considered non-idempotent by default and should not be repeated in case of a timeout or server error. Therefore, repeating a request would appear as a mistake from an external observer's perspective, and it could indeed become one if the server changes the `If-Match` header check policy to a more relaxed one. Conversely, repeating a `PUT` request (assuming that getting a timeout or an error while performing a "heavy" order creation operation is much more probable than in the case of a "lightweight" draft creation) could be automated and would not result in order duplication even if the revision check is disabled.

2. Reading

Let's continue. The reading operation is at first glance straightforward:

- `GET /v1/orders/{id}`.

However, upon closer inspection, it becomes less simple. First, the client should have a method to retrieve the ongoing orders executed on behalf of the user, which requires creating a separate enumerator resource:

- `GET /v1/orders/?user_id=<user_id>`.

Returning potentially long lists in a single response is a bad idea, so we will need pagination:

- `GET /v1/orders/?user_id=<user_id>&cursor=<cursor>`.

If there is a long list of orders, the user will require filters to navigate it. Let's say we introduce a beverage type filter:

- `GET /v1/orders/?user_id=<user_id>&recipe=lungo`.

However, if the user needs to see a single list containing both latte and lungo orders, this interface becomes much less viable as there is no universally adopted technique for passing structures in that are more complex than key-value pairs. Soon, we will face the need to have a search endpoint with rich semantics, which naturally should be represented as a `POST` request body.

Additionally, if some media data could be attached to an order (such as photos), a separate endpoint to expose them should be developed:

- `GET /v1/orders/{order_id}/attachments/{id}`.

3. Updating

The problem of partial updates was discussed in detail in the corresponding chapter of “The API Patterns” section. To quickly recap:

- The concept of fully overwriting resources with `PUT` is viable but soon faces problems when working with calculated or immutable fields and organizing collaborative editing. It is also suboptimal in terms of traffic consumption.
- Partially updating a resource using the `PATCH` method is potentially non-idempotent (and likely non-transitive), and the aforementioned concerns regarding automatic retries are applicable to it as well.

If we need to update a complex entity, especially if collaborative editing is needed, we will soon find ourselves leaning towards one of the following two approaches:

- Decomposing the `PUT` functionality into a set of atomic nested handlers (like `PUT /v1/orders/{id}/address`, `PUT /v1/orders/{id}/volume`, etc.), one for each specific operation.
- Introducing a resource to process a list of changes encoded in a specially designed format. Likely, this resource will also require implementing a draft/commit scheme via a `POST` + `PUT` pair of methods.

If media data is attached to an entity, we will additionally require more endpoints to amend this metadata.

4. Deleting

Finally, with deleting resources the situation is simple: in modern services, data is never deleted, only archived or marked as deleted. Therefore, instead of a `DELETE /v1/orders/{id}` endpoint there should be `PUT /v1/orders/{id}/archive` or `PUT /v1/archive?order=<order_id>`.

Real-Life CRUD Operations

This discourse is not intended to be perceived as criticizing the idea of CRUD operations itself. We simply point out that in complex subject areas, cutting edges and sticking to some mnemonic rules rarely play out. We started with the idea of having two URLs and four or five methods to apply to them:

- `/v1/orders/` to be acted upon with `POST`
- `/v1/orders/{id}` to be acted upon with `GET` / `PUT` / `DELETE` / optionally `PATCH`.

However, if we add the following requirements:

- Concurrency control in entity creation

- Collaborative editing
- Archiving entities
- Searching entities with filters then we end up with the following nomenclature of 8 URLs and 9-10 methods:
- `GET /v1/orders/?user_id=<user_id>` to retrieve the ongoing orders, perhaps with additional simple filters
- `/v1/orders/drafts/?user_id=<user_id>` to be acted upon with `POST` to create an order draft and with `GET` to retrieve existing drafts and the revision
- `PUT /v1/orders/drafts/{id}/commit` to commit the draft
- `GET /v1/orders/{id}` to retrieve the newly created order
- `POST /v1/orders/{id}/drafts` to create a draft for applying partial changes
- `PUT /v1/orders/{id}/drafts/{id}/commit` to apply the drafted changes
- `/v1/orders/search?user_id=<user_id>` to be acted upon with either `GET` (for simple cases) or `POST` (in more complex scenarios) to search for orders
- `PUT /v1/orders/{id}/archive` to archive the order

plus presumably a set of operations like `POST /v1/orders/{id}/cancel` for executing atomic actions on entities. This is what is likely to happen in real life: the idea of CRUD as a methodology for describing typical operations applied to resources with a small set of uniform verbs quickly evolves towards a bucket of different endpoints, each covering a specific aspect of working with the entity during its lifecycle. This only proves that mnemonics are just helpful starting points; each situation requires a thorough understanding of the subject area and designing an API that fits it. However, if your task is to develop a “universal” interface that fits every kind of entity, we would strongly suggest starting with something like the ten-method nomenclature described above.

Chapter 39. Working with HTTP API Errors

The examples of organizing HTTP APIs discussed in the previous chapters were mostly about “happy paths,” i.e., the direct path of working with an API in the absence of obstacles. It's now time to talk about the opposite case: how HTTP APIs should work with errors and how the standard and the REST architectural principles can help us.

Imagine that some actor (a client or a gateway) tries to create a new order:

```
POST /v1/orders?user_id=<user_id> HTTP/1.1
Authorization: Bearer <token>
If-Match: <revision>

{ /* order parameters */ }
```

What problems could potentially happen while handling the request? Off the top of the mind, it might be:

1. The request cannot be parsed (invalid symbols, syntax violation, etc.)
2. The authorization token is missing
3. The authorization token is invalid
4. The token is valid, but the user is not permitted to create new orders
5. The user is deleted or deactivated
6. The user identifier is invalid or does not exist
7. The revision is missing
8. The revision does not match the actual one
9. Some required fields are missing in the request body
10. A value of a field exceeds the allowed boundaries
11. The limit for the number of requests reached
12. The server is overloaded and cannot respond
13. Unknown server error (i.e., the server is broken to the extent that it's impossible to understand why the error happened).

From general considerations, the natural idea is to assign a status code for each mistake. Obviously, the `403 Forbidden` code fits well for mistake #4, and the `429 Too Many Requests` for #11. However, let's not be rash and ask first *for what purpose* are we assigning codes to errors?

Generally speaking, there are three kinds of actors in the system: the user, the application (a client), and the server. Each of these actors needs to understand several important things about the error (and the answers could actually differ for each of them):

1. Who made the mistake: the end user, the developer of the client, the backend developer, or another interim agent such as the network stack programmer?
    - And let's not forget about the possibility of the mistake being *deliberately* made by either an end user or a client developer while trying to blunt-force hijack the account of another user.
2. Is it possible to fix the error by just repeating the request?
    - If yes, then after what period of waiting?
3. If it is not the case, is it still possible to fix it by reformulating the request?
4. If the error cannot be resolved, what should be done about it?

One of these questions is easily answered in the HTTP API paradigm: the desired interval of repeating the request might be indicated in a `Retry-After` header. Also, HTTP helps with question #1: to understand which side is the cause of the error, the first digit in the HTTP status code is used (see below).

With the other questions, the situation is unfortunately much more complicated.

Client Errors

Status codes that start with the digit `4` indicate that it was the user or the client who made a mistake, or at least the server decided so. *Usually*, repeating a request that resulted in a `4xx` error is meaningless: the request will never be fulfilled unless some additional actions are

performed. However, there are notable exceptions, most importantly `429 Too Many Requests` and `404 Not Found`. The latter implies some "uncertainty state" according to the standard: the server could use it if exposing the real cause of the error is undesirable. After receiving a `404`, the request might be repeated, possibly yielding a different outcome. To indicate the *persistent* non-existence of a resource, a separate `410 Gone` status is used.

A more interesting question is what the client can (or must) do if such an error is received. As we discussed in the "Isolating Responsibility Areas" chapter, if the error can be resolved, there must be a machine-readable description for the client to interpret. In the case it cannot, human-readable instructions should be provided for the user (even "Try restarting the application" is a better user experience than "Unknown error happened") and for the client developer.

If we try to apply this principle to HTTP APIs, we will soon learn that the situation is complicated. On one hand, the protocol includes a lot of codes that indicate specific problems with using the protocol, such as `405 Method Not Allowed` (indicates that the verb in the request cannot be applied to the requested resource), `406 Not Acceptable` (the server cannot return a representation that satisfies the `Accept*` headers in the request), `411 Length Required`, `414 URI Too Long`, etc. The client code might process these errors and sometimes even perform some actions to mitigate them (for example, add a `Content-Length` header in case of a `411` error). However, this is hardly applicable to business logic. If the server returns a `429 Too Many Requests` if some limit is exceeded, there are no standardized means of indicating *which exact limit* was hit.

Sometimes, the absence of a common approach to describing business logic errors is circumvented by using different codes with almost identical semantics (or just randomly chosen codes) to distinguish between different causes of the error. One notable example is the widely adopted usage of the `401 Unauthorized` status code to indicate the absence or the invalid value of authorization headers, which is a signal for an application to ask the user to log in. This usage contradicts the standard (which requires that a `401` response must contain the `WWW-Authenticate` header that describes the methods of authorization; we are

unaware of a single API that follows this requirement), but it has become a *de facto* standard itself.

Even if we choose this approach, there are very few status codes that can reflect different aspects of the same error type. In fact, we face the situation that all the multiplicity of business-bound errors is to be returned using a very limited set of status codes:

- `400 Bad Request` for all the errors related to request validation issues. (Some purists insist that `400` corresponds to format violations such as invalid JSON. For logical errors, the `422 Unprocessable Content` code is to be used. This actually changes nothing regarding the discussed problem.)
- `403 Forbidden` for any problems related to authorizing the user's actions.
- `404 Not Found` if any of the entities referred to in the request are non-existent *or* if exposing the real cause of the error is undesirable.
- `409 Conflict` if data integrity is violated.
- `410 Gone` if the resource was deleted.
- `429 Too Many Requests` if some quotas are exceeded.

The editors of the specification are very well aware of this problem as they state that "the server SHOULD send a representation containing an explanation of the error situation, and whether it is a temporary or permanent condition." This, however, contradicts the entire idea of a uniform machine-readable interface (and so does the idea of using arbitrary status codes). (Let us additionally emphasize that this lack of standard tools to describe business logic-bound errors is one of the reasons we consider the REST architectural style as described by Fielding in his 2008 article non-viable. The client *must* possess prior knowledge of error formats and how to work with them. Otherwise, it could restore its state after an error only by restarting the application.)

Additionally, there is a third dimension to this problem in the form of webserver software for monitoring system health that often relies on status codes to plot charts and emit notifications. However, two errors represented with the same status code — let's say, wrong password and

expired token — might be very different. The increased rate of the former might indicate brute-forcing of accounts, while an unusually high frequency of the latter could be a result of a client error if a new version of an application wrongly caches authorization tokens.

All these observations naturally lead us to the following conclusion: if we want to use errors for diagnostics and (possibly) helping clients to recover, we need to include machine-readable metadata about the error subtype and, possibly, additional properties to the error body with a detailed description of the error. For example, as we proposed in the "Describing Final Interfaces" chapter:

```
POST /v1/coffee-machines/search HTTP/1.1

{
  "recipes": ["lngo"],
  "position": {
    "latitude": 110,
    "longitude": 55
  }
}
→
HTTP/1.1 400 Bad Request
X-OurCoffeeAPI-Error-Kind:↵
  wrong_parameter_value

{
  "reason": "wrong_parameter_value",
  "localized_message":
    "Something is wrong.↵
     Contact the developer of the app.",
  "details": {
    "checks_failed": [
      {
        "field": "recipe",
        "error_type": "wrong_value",
        "message":
          "Unknown value: 'lngo'.↵
           Did you mean 'lungo'?"
      },
      {
        "field": "position.latitude",
        "error_type":
          "constraint_violation",
        "constraints": {
          "min": -90,
          "max": 90
        },
        "message":
          "'position.latitude' value↵
            must fall within↵
            the [-90, 90] interval"
      }
```

```
      ]
    }
  }
```

Let us also remind the reader that the client must treat unknown `4xx` status codes as a `400 Bad Request` error. Therefore, the (meta)data format for the `400` error must be as general as possible.

Server Errors

`5xx` errors indicate that the client did everything right, and the problem is server-bound. For the client, the only important thing about the server error is whether it makes sense to repeat the request (and if yes, then when). Keeping in mind that in publicly available APIs, the real reason for the error is usually not exposed, having just the `500 Internal Server Error` and `503 Service Unavailable` codes is enough for most subject areas. (The latter is needed to indicate that the denial of service state is temporary and it might be replaced with just a `Retry-After` header to the `500` error.)

However, for internal systems, this argumentation is wrong. To build proper monitoring and notification systems, server errors must contain machine-readable error subtypes, just like the client errors. The same approaches are applicable (either using arbitrary status codes and/or passing error kind as a header); however, this data must be stripped off by a gateway that marks the border between external and internal systems and replaced with general instructions for both developers and end users, describing actions that need to be performed upon receiving an error.

```
POST /v1/orders/?user_id=<user id> HTTP/1.1
If-Match: <revision>

{ parameters }
→
// The response the gateway received
// from the server, the metadata
// of which will be used for
// monitoring and diagnostics
HTTP/1.1 500 Internal Server Error
// Error kind: timeout from the DB
X-OurCoffeAPI-Error-Kind: db_timeout
{ /*
   * Additional data, such as
   * which host returned an error
   */ }
```

```
// The response as returned to
// the client. The details regarding
// the server error are removed
// and replaced with instructions
// for the client. As at the gateway
// level it is unknown whether
// order creation succeeded, the client
// is advised to repeat the request
// and/or retrieve the actual state.
HTTP/1.1 500 Internal Server Error
Retry-After: 5

{
  "reason": "internal_server_error",
  "localized_message": "Cannot get↵
    a response from the server.↵
    Please try repeating the operation
    or reload the page.",
  "details": {
    "can_be_retried": true,
    "is_operation_failed": "unknown"
  }
}
```

However, we go on a slippery slope here. The contemporary practice of implementing HTTP API clients allows for repeating safe requests (e.g., `GET`, `HEAD`, and `OPTIONS` methods). In the case of unsafe methods, *developers need to write code* to repeat the request, and to do so they need to read the documentation very carefully to check if it is the desired behavior and if it is actually safe.

Theoretically, with idempotent `PUT` and `DELETE` it should be more convenient. Practically, as many developers let this knowledge pass them, frameworks for working with HTTP APIs will likely not repeat these requests. Still, we can get *some* benefit from following the standards as the signature itself indicates that the request can be retried.

As for more complex operations, to make developers aware that they can repeat a potentially unsafe operation, we could introduce a format describing the possible actions in the error response itself… However, developers seldom expect to find such instructions in the error body, probably because programmers rarely see `5xx` errors during development, unlike their `4xx` counterparts, and testing environments usually do not provide capabilities to emulate server errors. All in all, you will have to describe the desirable actions in the documentation. (Be aware that this instruction will likely be ignored. This is the way.)

Organizing HTTP API Error Nomenclature in Practice

As it is obvious from what was discussed above, there are essentially three approaches to working with errors in HTTP APIs:

1. Applying an "extended interpretation" to the status code nomenclature, or in plain words, selecting or inventing a new status code for each new type of error introduced. (The author of this book has frequently observed an approach to API development that included choosing a status code based on wording resembling the error cause, disregarding its description in the standard completely.)

2. Abolishing the use of status codes and developing a format for errors enveloped in a `200` HTTP response. Most RPC frameworks choose this direction.

    - 2a. A subvariant of this strategy is using just two status codes (`400` for every client error, `500` for every server error), optionally complemented by a third one (`404` to indicate situations of uncertainty).

3. Employing a mixed approach, i.e., using status codes in accordance with their semantics to indicate an *error family* with additional (meta)data being passed in a specially developed format (similar to the code samples we gave above).

Obviously, only approach #3 could be considered compliant with the standard. Let us be honest and say that the benefits of following it (especially compared to option #2a) are not very significant and only comprise better readability of logs and transparency for intermediate proxies.

Chapter 40. Final Provisions and General Recommendations

Let's summarize what was discussed in the previous chapters. To design a fine HTTP API one needs to:

1. Describe a happy path, i.e. draw a diagram of all HTTP calls that occur during a normal work cycle of an application.
2. Interpret every call as an operation executed on a resource and assemble a nomenclature of URLs and applicable methods accordingly.
3. Enumerate errors that might occur during operation execution and determine paths to restore the application state for clients after receiving an error.
4. Decide which functionality will be communicated at the HTTP protocol level, i.e., which standard protocol capabilities to use in conjunction with what tools and software and the extent of their usage.
5. Develop a detailed specification regarding the aforementioned list points.
6. Check yourselves: elaborate on paragraphs 1-3 to write pseudo-code for the application's business logic in accordance with the specification, and evaluate the convenience, understandability and readability of your API.

Additionally, we'd like to provide some code style advice:

1. Do not differentiate paths with trailing `/` and without it. Employ a default policy (we would rather recommend ending paths with `/` for a simple reason: it allows for referring to operations on the domain root resource in a readable manner as `VERB /`). If you decide to prohibit one of the variants (let's say, all URLs must end with a trailing slash), make a redirect or provide a very readable error message if a developer tries to call a URL formatted otherwise.

2. Include common headers (such as `Date`, `Content-Type`, `Content-Encoding`, `Content-Length`, `Cache-Control`, `Retry-After`, etc.) in the responses and generally avoid relying on clients to guess default protocol parameters correctly.

3. Support the `OPTIONS` method and the CORS protocol just in case your API needs to be accessed from a Web browser.

4. Choose a casing rule and a rule for transforming casing while moving a parameter from one part of an HTTP request to another.

5. Always leave an opportunity for backward-compatible extension of an API method. In particular, always return a JSON object as the endpoint response root as objects can always be extended with a new field, unlike arrays and primitives.

    - Let us also note that an empty string is invalid JSON, so you need to return an empty object `{}` in `200` responses even if it doesn't have a specific meaning. Alternatively, you can use the `204 No Content` status code with an empty body, which is not extensible.

6. For every `GET` response, provide explicit caching parameters (otherwise, there is always a chance that a client or an intermediate agent invents them on their own).

7. Do not employ known possibilities to serve requests in violation of the standard and avoid exploiting “gray zones” of the protocol. In particular:

    - Do not place unsafe operations behind the `GET` verb, and do not place non-idempotent operations behind the `PUT` / `DELETE` methods.
    - Maintain the `GET` / `PUT` / `DELETE` operations symmetry.
    - Do not allow `GET` / `HEAD` / `DELETE` requests to have a body and do not provide bodies in response to `HEAD` requests or alongside the `204` status code.

    - Do not invent your own standards for passing arrays and nested objects as query parameters. It is better to use an HTTP verb that allows having a body, or as a last resort pass the parameter as a Base64-encoded JSON-stringified value.
    - Do not put parameters that require escaping (i.e., non-alphanumeric ones) in a path or a domain of a URL. Use query or body parameters for this purpose.

8. Familiarize yourself with at least the basics of typical vulnerabilities in HTTP APIs used by attackers, such as:

    - CSRF
    - SSRF
    - HTTP Response Splitting
    - Unvalidated Redirects and Forwards

    and include protection against these attack vectors at the webserver software level. The OWASP community provides a good cheatsheet on the best HTTP API security practices.

In conclusion, we would like to make the following statement: building an HTTP API is relying on the common knowledge of HTTP call semantics and drawing benefits from it by leveraging various software built upon this paradigm, from client frameworks to server gateways, and developers reading and understanding API specifications. In this sense, the HTTP ecosystem provides probably the most comprehensive vocabulary, both in terms of profoundness and adoption, compared to other technologies, allowing for describing many different situations that may arise in client-server communication. While the technology is not perfect and has its flaws, for a *public API* vendor, it is the default choice, and opting for other technologies rather needs to be substantiated as of today.

[WORK IN PROGRESS] SECTION V. SDKS & UI LIBRARIES

Chapter 41. On the Content of This Section

As we mentioned in the Introduction, the term "SDK" (which stands for "Software Development Kit") lacks concrete meaning. The common understanding is that an SDK differs from an API as it provides both program interfaces and tools to work with them. This definition is hardly satisfactory as today any technology is likely to be provided with a bundled toolset.

However, there is a very specific narrow definition of an SDK: it is a client library that provides a high-level interface (usually a native one) to some underlying platform (such as a client-server API). Most often, we talk about libraries for mobile OSes or Web browsers that work on top of a general-purpose HTTP API.

Among such client SDKs, one case is of particular interest to us: those libraries that not only provide programmatic interfaces to work with an API but also offer ready-to-use visual components for developers. A classic example of such an SDK is the UI libraries provided by cartographical services. Since developing a map engine, especially a vector one, is a very complex task, maps API vendors provide both "wrappers" to their HTTP APIs (such as a search function) and visual components to work with geographical entities. The latter often include general-purpose elements (such as buttons, placemarks, context menus, etc.) that can be used independently from the main functionality of the API.

This Section will be dedicated to these two types of program toolkits:

- Client "wrappers" that work with client-server APIs
- Client libraries that contain visual components with which end users might interact.

To avoid being wordy, we will use the term "SDK" for the former and "UI libraries" for the latter.

NB: Strictly speaking, a UI library might either include a client-server API "wrapper" or not (i.e., just provide a "pure" API to some underlying system engine). In this Section, we will mostly talk about the first option as it is the most general case and the most challenging one in terms of API design. Most SDK development patterns we will discuss are also applicable to "pure" libraries.

Chapter 42. SDKs: Problems and Solutions

The first question we need to clarify about SDKs (let us reiterate that we use this term to denote a native client library that allows for working with a technology-agnostic underlying client-server API) is why SDKs exist in the first place. In other words, why is using “wrappers” more convenient for frontend developers than working with the underlying API directly?

Several reasons are obvious:

1. Client-server protocols are usually designed to allow for the implementation of clients in any programming language. This implies that the data received from such an API will not be in the most convenient format. For example, there is no “datetime” type in JSON, and the dates need to be passed as strings. Similarly, most mainstream protocols don't support (de)serializing hash tables.

2. Most programming languages are imperative (and many of them are object-oriented) while most data formats are declarative. Working with raw data received from an API endpoint is inconvenient in terms of writing code. Client developers would prefer this data to be represented as objects (class instances).

3. Different programming languages imply different code styles (casing, namespace organization, etc.), while the practice of tailoring data formats in APIs to match the client's code style is very rare.

4. Platforms and/or programming languages usually prescribe how error handling should be organized — typically, through throwing exceptions or employing defer/panic techniques — which is hardly applicable to the concept of uniform APIs.

5. APIs are provided with instructions (human- or machine-readable) on how to repeat requests if the API endpoints are unavailable. This logic needs to be implemented by a client developer as client frameworks rarely provide it (and it would be

very dangerous to automatically repeat potentially non-idempotent requests). Though this point might appear insignificant, it is in fact very important for every vendor of a popular API, as safeguards need to be installed to prevent API servers from overloading due to an uncontrollable spike of request repeats. This is achieved through:

    - Reading the `Retry-After` header and avoiding retrying the endpoint earlier than the time stated by the server
    - Introducing exponentially growing intervals between consecutive requests.

    This is what client developers should do regarding server errors, but they often skip this part, especially if they work for external partners.

Having an SDK would resolve these issues as they are, so to say, trivial: to fix them, the principles of working with the API aren't changed. For every request and response, we construct the corresponding SDK method, and we only need to set rules for doing this transformation, i.e., adapting platform-independent API formats to specific platforms. Additionally, this transformation usually could be automated.

However, there are also non-trivial problems we face while developing an SDK for a client-server API:

1. In client-server APIs, data is passed by value. To refer to some entities, specially designed identifiers need to be used. For example, if we have two sets of entities — recipes and offers — we need to build a map to understand which recipe corresponds to which offer:

```
// Request 'lungo' and 'latte' recipes
const recipes = await api
  .getRecipes(['lungo', 'latte']);
// Build a map that allows to quickly
// find a recipe by its identifier
const recipeMap = new Map();
recipes.forEach((recipe) => {
  recipeMap.set(recipe.id, recipe);
});
// Request offers for latte and lungo
// in the vicinity
const offers = await api.search({
  recipes: ['lungo', 'latte'],
  location
});
// To show offers to the user, we need
// to take the `recipe_id` in the offer,
// find the recipe description in the map
// and enrich the offer data with
// the recipe data
promptUser(
  'What we have found',
  offers.map((offer) => {
    const recipe = recipeMap
      .get(offer.recipe_id);
    return {offer, recipe};
  }));
```

This piece of code would be half as long if we received offers from the `api.search` SDK method with a *reference* to a recipe:

```
// Request 'lungo' and 'latte' recipes
const recipes = await api
  .getRecipes(['lungo', 'latte']);
// Request offers for latte and lungo
// in the vicinity
const offers = await api.search({
  // Pass the references to the recipes,
  // not their identifiers
  recipes,
  location
});

promptUser(
  'What we have found',
  // Offer already contains a reference
  // to the recipe
  offers
);
```

2. Client-server APIs are typically decomposed so that one response contains data regarding one kind of entity. Even if the endpoint is composite (i.e., allows for combining data from different sources depending on parameters), it is still the developer's responsibility to use these parameters. The code sample from the previous example would be even shorter if the SDK allowed for the initialization of all related entities:

```
// Request offers for latte and lungo
// in the vicinity
const offers = await api.search({
  recipes: ['lungo', 'latte'],
  location
});

// The SDK requested latte and lungo
// data from the `getRecipes` endpoint
// under the hood
promptUser(
  'What we have found',
  offers
);
```

   The SDK can also populate program caches for the entities (if we do not rely on protocol-level caching) and/or allow for the “lazy” initialization of objects.

   Generally speaking, storing pieces of data (such as authorization tokens, idempotency keys, draft identifiers in two-phase commits, etc.) between requests is the client's responsibility, and it is rather hard to formalize the rules. If an SDK takes responsibility for managing the data, there will be far fewer mistakes in application code.

3. Receiving callbacks in client-server APIs, even if it is a duplex communication channel, is rather inconvenient to work with and requires object mapping as well. Even if a push model is implemented, the resulting client code will be rather bulky:

```
// Retrieve ongoing orders
const orders = await api
  .getOngoingOrders();
// Build order map
const orderMap = new Map();
orders.forEach((order) => {
  orderMap.set(order.id, order);
});
// Subscribe to state change
// events
api.subscribe(
  'order_state_change',
  (event) => {
    // Find the corresponding order
    const order = orderMap
      .get(event.order_id);
    // Take some actions, like
    // updating the UI
    // in the application
    UI.update(order);
  }
);
```

If the API requires polling a state change endpoint, developers will additionally need to implement periodic requesting of the endpoint (and install safeguards to avoid overloading the server).

Meanwhile, this code sample already contains several mistakes:

- First, the list of orders is requested, and then the state change listener is added. If some order changes its state between those two calls, the application would never learn about this fact.
- If an event comes with an identifier of some unknown order (created from a different device or in a different execution thread), the map lookup operation will return an empty result, and the listener will throw an exception that is not properly handled anywhere.

Once again, we face a situation where an SDK lacking important features leads to mistakes in applications that use it. It would be much more convenient for a developer if an order object allowed for subscribing to its status updates without the need to learn how it works at the transport level and how to avoid missing an event.

```
const order = await api
  .createOrder(…)
  // No need to subscribe to
  // the entire status change
  // stream and filter it
  .subscribe(
    'state_change',
    (event) => { … }
  );
```

NB: This code relies on the idea that the `order` is being updated in a consistent manner. Even if the state changes on the server side between the `createOrder` and `subscribe` calls, the `order` object's state will be consistent with the `state_change` events fired. Organizing this technically is the responsibility of the SDK developers.

4. Restoring after encountering business logic-bound errors is typically a complex procedure. As it can hardly be described in a machine-readable manner, client developers have to elaborate on the scenarios on their own.

```
// Request offers
const offers = await api
  .search(…);
// The user selects an offer
const selectedOffer =
  await promptUser(offers);

let order;
let offer = selectedOffer;
let numberOfTries = 0;
do {
  // Trying to create an order
  try {
    numberOfTries++;
    order = await api
      .createOrder(offer, …);
  } catch (e) {
    // If the number of tries exceeded
    // some reasonable limit, it's
    // better to give up
    if (numberOfTries > TRY_LIMIT) {
      throw new NoRetriesLeftError();
    }
    // If the error is of the “offer
    // expired” kind…
    if (e.type ==
      api.OfferExpiredError) {
      // Trying to get a new offer
      offer = await api
        .renewOffer(offer);
    } else {
      // Other errors
      …
    }
  }
} while (!order);
```

As we can see, the simple operation “try to renew an offer if needed” results in a bulky piece of code that is simultaneously error-prone and totally unnecessary as it doesn't introduce any new functionality visible to end users. For an application developer, it would be more convenient if this error (“offer expired”) *were not exposed in the SDK*, i.e., the SDK automatically renewed offers if needed.

Such situations also occur when working with APIs featuring eventual consistency or optimistic concurrency control — generally speaking, with any API where background errors are expected (which is rather a norm of life for client-server APIs). For

frontend developers, writing code to implement policies like “read your writes” (i.e., passing tokens of the last known operation to subsequent queries) is essentially a waste of time.

5. Finally, one more important function that a customized SDK might fulfill is isolating the low-level API and changing the versioning paradigm. It is possible to fully conceal the underlying functionality (i.e., developers won't have direct access to the API) and ensure a certain freedom of working with the API inside the SDK up to seamlessly switching a major version. This approach for sure provides API vendors with more control over how partners' applications work. However, it requires investing more in developing the SDK and, more importantly, properly designing the SDK so that developers won't need to call the API directly due to the lack of access to some functionality or the inconvenience of working with the SDK. Moreover, the SDK itself should be robust enough to be able to handle the transition to a new major version of the API.

To summarize the above, a properly designed SDK, apart from maintaining consistency with the platform guidelines and providing “syntactic sugar,” serves three important purposes:

- Lowering the number of mistakes in client code by implementing helpers that cover unobvious and poorly formalizable aspects of working with the API
- Relieving client developers of the duty to write code that is absolutely irrelevant to the tasks they are solving
- Giving an API vendor more control over integrations.

Code Generation

As we have seen, the list of tasks that an SDK developer faces (if they aim to create a quality product, of course) is quite considerable. Given that every target platform requires a separate SDK, it is not surprising that many API vendors seek to replace manual labor with automation.

One of the most potent approaches to such automation is code generation. It involves developing a technology that allows generating the SDK code in the target programming language for the target platform based on the API specification. Many modern data interchange protocols (gRPC, for instance) are shipped with generators of such ready-to-use clients in different programming languages. For other technologies (OpenAPI/Swagger, for instance) generators are being developed by the community.

Code generation allows for solving trivial problems such as adapting code style, (de)serializing complex types, etc. — in other words, resolving issues that are bound to the specifics of the platform, not the high-level business logic. The SDK developers can relatively inexpensively complement this machine "translation" with convenient helpers: realize automated repeating idempotent requests (with some retry time management policy), caching results, storing persistent data (such as authorization tokens) in the system storage, etc.

Such a generated SDK is usually referred to as a "client to an API." The convenience of usage and the functional capabilities of code generation are so formidable that many API vendors restrict themselves to this technology, only providing their SDKs as generated clients.

However, for the aforementioned reasons, higher-level problems (such as receiving callbacks, dealing with business logic-bound errors, etc.) cannot be solved with code generation without writing some arbitrary code for the specific API. In the case of complex APIs with a non-trivial workcycle it is highly desirable that an SDK also solves high-level problems. Otherwise, an API vendor will end up with a bunch of applications using the API and making all the same "rookie mistakes." This is, of course, not a reason to fully abolish code generation as it's quite convenient to use a generated client as a basis for developing a high-level SDK.

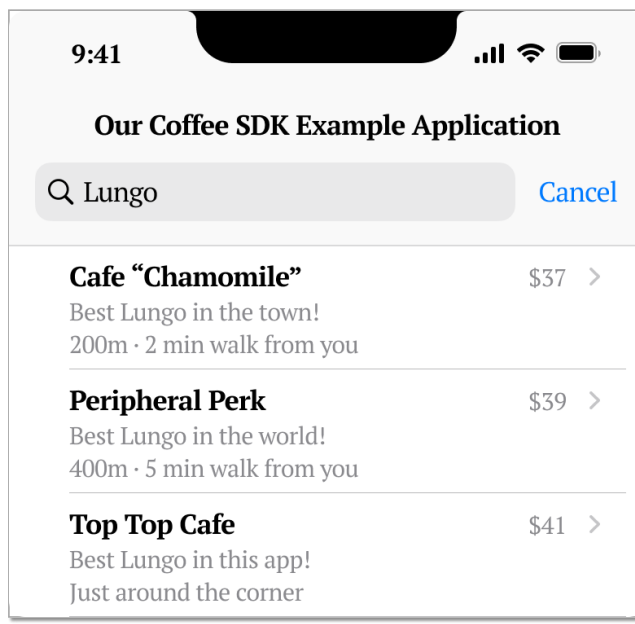
Other Tooling

The word “Kit” in “Software Development Kit” implies that the technology comes with auxiliary tools such as emulators / simulators, sandboxes, plugins for IDEs, etc. In this Section, we will not delve into this topic and will discuss it in more detail in the “API Product” section.

Chapter 43. Problems of Introducing UI Components

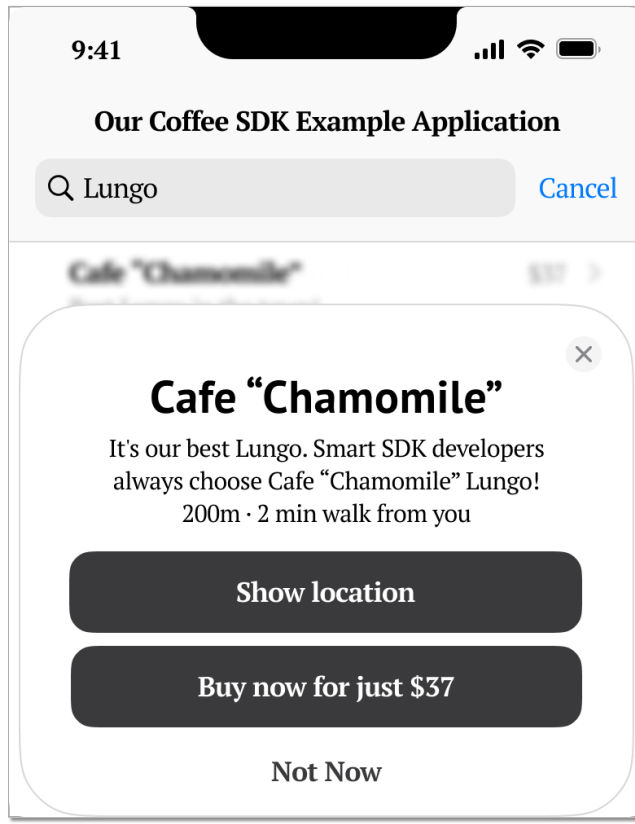
Introducing UI components to an SDK brings an additional dimension to an already complex setup comprising a low-level API and a client wrapper on top of it. Now both developers (who write the application) and end users (who use the application) interact with your API. This might not appear as a game changer at first glance; however, we assure you that it is. Involving an end-user has significant consequences from the API / SDK design point of view as it requires much more careful and elaborate program interfaces compared to a "pure" client-server API. Let us explain this statement with a concrete example.

Imagine that we decided to provide a client SDK for our API that features ready-to-use components for application developers. The functionality is simple: the user enters a search phrase and observes the results in the form of a list.



The main screen of an application with search results. Click to enlarge

The user can select an item and view the offer details with available actions.



Offer view panel. Click to enlarge

To implement this scenario, we provide an object-oriented API in the form of, let's say, a class named `SearchBox` that realizes the aforementioned functionality by utilizing the `search` method in our client-server API.

The Problems

At first glance, it might appear that this UI is a superstructure atop the `search` method that simply visualizes the results. Unfortunately, that is not the case. Let us enumerate the problems we have never encountered while developing APIs without visual components.

1. Coupling Heterogeneous Functionality in One Entity

We have placed two buttons (to make an order and to show the coffee shop's location) plus a cancel action onto the offer view panel. These buttons may look identical and they react to the user's actions in the same way, but the way the `SearchBox` component handles pressing each of them is completely different.

Imagine if we allow developers to add their own action buttons onto the panel, for which purpose we introduce a `Button` class. We will soon learn that this functionality will be used to cover two diametrically opposite scenarios:

- Adding extra buttons to the panel, such as “Call the coffee shop,” while *sharing the design with the standard ones*
- Changing the appearance of the standard buttons to match the partner's corporate design guidelines *while preserving the functionality intact*.

Furthermore, a third scenario is possible: developers might want to create a “Call” button that both looks different and performs different actions but *inherits the UX* of the standard button, such as animating button presses, stacking with other buttons, etc.

From the developers' perspective, this means that the `Button` class should allow redefining the appearance of the button, the actions it performs, and the UX elements — in other words, each of these three subsystems might be replaced with an alternative implementation so that the other two subsystems continue working normally.

2. Shared Resources

Imagine that we need to allow developers to programmatically create a `SearchBox` to process a query. This functionality seems reasonable as it would allow displaying a "find lungo nearby" banner in the application, clicking on which would show a `SearchBox` with the pre-entered "lungo" query. Developers will just need to open the corresponding screen in the app and call a method that we are to design. Let's simply name it `search`.

Two of our `search` methods (the "pure" client-server one and the component-bound `SearchBox.search`) accept the same parameters and emit the same results. However, their *behavior* is totally different:

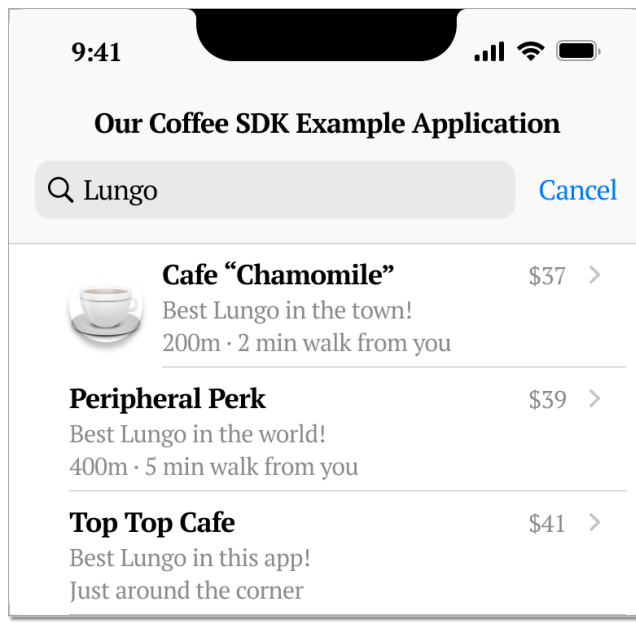
- If requested several times, `SearchBox.search` must discard all server responses except for the one corresponding to the latest request (even if it is not the one received last).
    - Additional question: What should `SearchBox.search` return if it is interrupted by another search? If an error, then what was the error of the caller? If a success, then why are the results not displayed?
- This leads to another problem: What should happen if `SearchBox.search` was called when it was processing a request by an end user? Which of the callers is more important — a developer or a user?

While implementing a client-server API, we don't typically face this issue. Every actor calling a search function will receive the response independently. With UI components this approach doesn't work as all the components ultimately share one common resource: the screen of the application and the user's attention.

Any asynchronous operation in a UI component, especially if it is visibly indicated with animation or other continuous action, could disrupt other visual operations, including cases when the disruption happened because of the user's actions.

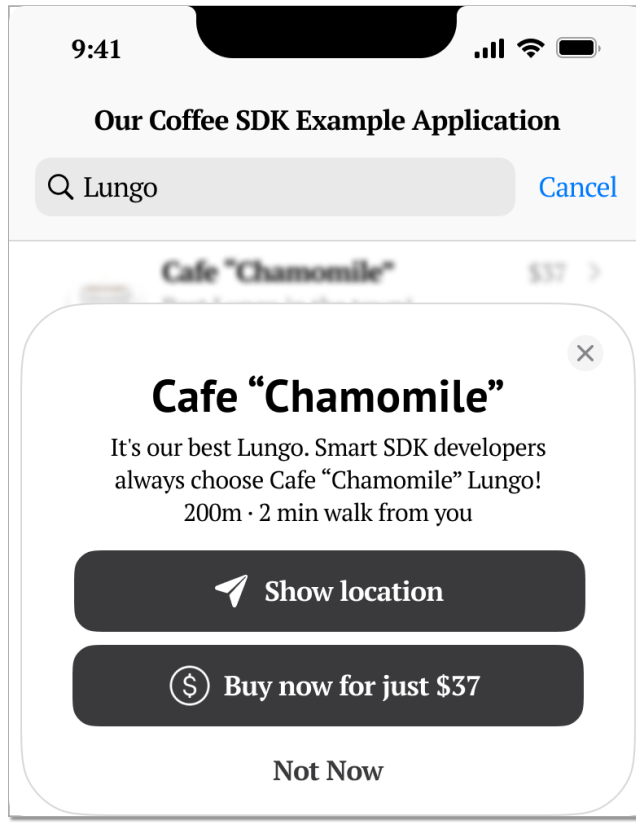
3. Multiple Inheritance in Entity Hierarchies

Imagine that a developer decided to enhance the design of the offer list with icons of coffee shop chains. If the icon is set, it should be shown in every place related to a specific coffee shop's offer.



Search results with a coffee shop chain icon. Click to enlarge

Now let's also imagine that the developer additionally customized all buttons in the SDK by adding action icons.



The offer view panel with action icons. Click to enlarge

A question arises: If an offer of the coffee chain is shown in the panel, which icon should be featured on the order creation button: the one inherited from the offer properties (the coffee chain logo) or the one inherited from the action type of the button itself? The order creation control element is incorporated into two entity hierarchies (visual one and data-bound one) and inherits from both equally.

It is very easy to demonstrate how coupling several subject areas in one entity leads to highly sophisticated and unobvious logic. As the same arguments are applicable to the “Show location” button as well, it is kind of obvious that specialized options should take precedence over general ones. In our case, the type of a button should have more priority than some abstract “icon” data property.

But it is not the end of the story. If the developer still wants exactly this, i.e., to show a coffee shop chain icon (if any) on the order creation button, then what should they do? Following the same logic, we should provide an even more specialized possibility to do so. For example, we can adopt the following logic: if there is a `checkoutButtonIconUrl` property in the data, the icon will be taken from this field. Developers could customize the order creation button by overwriting this `checkoutButtonIconUrl` field for every search result:

```
const searchBox = new SearchBox({
  // For simplicity, let's allow
  // to override the search function
  searchFunction: function (params) {
    const res = await api.search(params);
    res.forEach(function (item) {
        item.checkoutButtonIconUrl =
          <the URL of the icon>;
    });
    return res;
  }
})
```

Formally speaking, this code is correct and does not violate any agreements. However, the readability and maintainability of this code are a catastrophe. The last place the next developer asked to change the *button icon* will look is the *offer search function*.

This functionality would appear more maintainable if no such customization opportunity was provided at all. Developers will be unhappy as they would need to implement their own search control from scratch just to replace an icon, but this implementation would be at least

logical with icons defined somewhere in the rendering function.

NB: There are many other possibilities to allow developers to customize a button nested deeply within a component, such as exposing dependency injection or sub-component class factories, giving direct access to a rendered view, allowing to provide custom button layouts, etc. All of them are inherently subject to the same problem: it is a very complicated task to consistently define the order and the priority of injections / rendering callbacks / custom layouts.

Consistently solving all the problems listed above is unfortunately a very complex task. In the following chapters, we will discuss design patterns that allow for splitting responsibility areas between the component's sub-entities. However, it is important to understand one thing: full separation of concerns, meaning developing a functional SDK+UI that allows developers to independently overwrite the look, business logic, and UX of the components, is extremely expensive. In the best-case scenario, the nomenclature of entities will be tripled. So the universal advice is: *think thrice before exposing the functionality of customizing UI components*. Though the price of design mistakes in UI library APIs is typically not very high (customers rarely request a refund if button press animation is broken), a badly structured, unreadable and buggy SDK could hardly be viewed as a competitive advantage of your API.

Chapter 44. Decomposing UI Components

Chapter 45. The MV* Frameworks

Chapter 46. The Backend-Driven UI

Chapter 47. Shared Resources and Asynchronous Locks

Chapter 48. Computed Properties

Chapter 49. Conclusion

SECTION VI. THE API PRODUCT

Chapter 50. The API as a Product

There are two important statements regarding APIs viewed as products.

1. APIs are *proper products*, just like any other kind of software. You are "selling" them in the same manner, and all the principles of product management are fully applicable to them. It is quite doubtful that you would be able to develop APIs well unless you conduct proper market research, learn customers' needs, and study competitors, supply, and demand.

2. However, APIs are *quite special products*. You are selling the possibility to perform some actions programmatically by writing code, and this fact puts some restrictions on product management.

To properly develop the API product, you must be able to answer this question precisely: why would your customers prefer to perform those actions programmatically? It is not an idle question: based on this book's author's experience, the lack of expertise by product owners in working with APIs *is exactly* the biggest problem in API product development.

End users interact with your API indirectly, through applications built upon it by software engineers acting on behalf of some company (and sometimes there is more than one engineer in between you and an end user). From this perspective, the API's target audience resembles a Maslow-like pyramid:

- Users constitute the base of the pyramid; they seek the fulfillment of their needs and do not think about technicalities.
- Business owners form the middle level; they match users' needs against technical capabilities declared by developers and build products.

- Developers make up the apex of the pyramid; it is developers who work with APIs directly, and they decide which of the competing APIs to choose.

The obvious conclusion of this model is that you must advertise the advantages of your API to developers. They will select the technology, and business owners will translate the concept to end users. If former or acting developers manage the API product, they often tend to evaluate the API's market competitiveness in this dimension only and mainly focus the product promotion efforts on the developer audience.

“Stop!” the mindful reader must yell at this moment. “The actual order of things is exactly the opposite!”

- Developers are typically a hired workforce implementing the tasks set by business owners (and even if a developer implements their own project, they still choose an API that best fits the project, thus being employers of themselves).
- Business leaders do not set tasks based on their sense of style or code elegance; they need certain functionality to be implemented — functionality that solves their customers' problems.
- Finally, customers do not concern themselves with the technical aspects of the solution; they choose the product they need and request specific functionality to be implemented.

So it turns out that customers are at the apex of the pyramid: it is customers whom you need to convince that they need not just *any* cup of coffee, but a cup of coffee brewed using our API (an interesting question arises: how will we convey the knowledge of which API works under the hood and why customers should pay for it!); then business owners will set the task to integrate the API, and developers will have no other choice but to implement it (which, by the way, means that investing in the readability and consistency of the API is not *that* important).

The truth, of course, lies somewhere in between. In some markets and subject areas, it is developers who make decisions (e.g., which framework to choose); in other markets and areas, it might be business owners or customers. It also depends on the competitiveness of the

market: introducing a new frontend framework does not encounter much resistance while developing, let's say, a new mobile operating system requires million-dollar investments in promotions and strategic partnerships.

Hereafter, we will describe some “averaged” situations, meaning that all three levels of the pyramid are important: customers choose the product that best fits their needs, business owners seek quality guarantees and lower development costs, and software engineers care about the API capabilities and the convenience of working with it.

Chapter 51. API Business Models

Before we proceed to the API product management principles, let us draw your attention to the matter of profits that the API vendor company might extract from it. As we will demonstrate in the next chapters, this is not an idle question as it directly affects making product decisions and setting KPIs for the API team. In this chapter, we will enumerate the main API monetization models. [In brackets, we will provide examples of such models applicable to our coffee-machine API study.]

1. Developers = End Users

The easiest and most understandable case is that of providing a service for developers, with no end users involved. First of all, we talk about software engineering tools: APIs of programming languages, frameworks, operating systems, UI libraries, game engines, etc. — general-purpose interfaces, in other words. [In our coffee API case, it means the following: we've developed a library for ordering a cup of coffee, possibly furnished with UI components, and now selling it to coffeeshop chains owners who are willing to buy it to ease the development of their own applications.] In this case, the answer to the "why have an API" question is self-evident.

There is also a plethora of monetizing techniques; in fact, we're just talking about monetizing software for developers.

1. The framework / library / platform might be paid per se, e.g., distributed under a commercial license. Nowadays such models are becoming less and less popular with the rise of free and open-source software but are still quite common.

2. The API may be licensed under an open license with some restrictions that might be lifted by buying an extended license. It might be either functional limitations (an inability to publish the app in the app store or an incapacity to build the app in the production mode) or usage restrictions (for example, using the

API for some purposes might be prohibited or an open license might be "contagious," i.e., require publishing the derived code under the same license).

3. The API itself might be free, but the API vendor might provide additional paid services (for example, consulting or integrating ones), or just sell the extended technical support.

4. The API development might be sponsored (explicitly or implicitly) by the platform or operating system owners [in our coffee case — by the vendors of smart coffee machines] who are interested in providing a wide range of convenient tools for developers to work with the platform.

5. Finally, by publishing the API under a free license, the API vendor might be attracting attention to other programming tools it makes to increase sales.

Remarkably, such APIs are probably the only "pure" case when developers choose the solution solely because of its clean design, elaborate documentation, thought-out use cases, etc. There are examples of copying the API design (which is the sincerest form of flattery, as we all know!) by other companies or even enthusiastic communities — that happened, for example, with the Java language API (an alternate implementation by Google) and the C# one (the Mono project) — or just borrowing apt solutions — as it happened with the concept of selecting DOM elements with CSS selectors, initially implemented in the *cssQuery* project, then adopted by *jQuery*, and after the latter became popular, incorporated as a part of the DOM standard itself.

2. API = the Main and/or the Only Access to the Service

This case is close to the previous one as developers again, not end users, are API consumers. The difference is that the API is not a product per se, but the service exposed via the API is. The purest examples are cloud platform APIs like Amazon AWS or Braintree API. Some operations are possible through end-user interfaces, but generally speaking, the

services are useless without APIs. [In our coffee example, imagine we are an operator of “cloud” coffee machines equipped with drone-powered delivery, and the API is the only means of making an order.]

Usually, customers pay for the service usage, not for the API itself, though frequently the tariffs depend on the number of API calls.

3. API = a Partner Program

Many commercial services provide access to their platforms for third-party developers to increase sales or attract additional audiences. Examples include the Google Books partner program, Skyscanner Travel APIs, and Uber API. [In our case study, it might be the following model: we are a large chain of coffee shops, and we encourage partners to sell our coffee through their websites or applications.] Such partnerships are fully commercial: partners monetize their own audience, and the API provider company yearns to get access to a broader public and additional advertising channels. As a rule, the API provider company pays for users reaching target goals and sets requirements for the integration performance level (for example, in a form of a minimum acceptable click-target ratio) to avoid misusing the API.

4. API = Additional Access to the Service

If a company possesses some unique expertise, usually in the form of a dataset that couldn't be easily gathered if needed, quite logically a demand for the API exposing this expertise arises. The most classical examples of such APIs are cartographical APIs: collecting detailed and precise geodata and keeping it up-to-date are extremely expensive, while a wide range of services would become much more useful if they featured an integrated map. [Our coffee example hardly matches this pattern as the data we accumulate — coffee machine locations, beverage types — is something useless in any other context but ordering a cup of coffee.]

This case is the most interesting one from the API developers' point of view as the existence of the API does really serve as a multiplier to the opportunities as the expertise owner could not physically develop all imaginable services utilizing the expertise but might help others to do it. Providing the API is a win-win: third-party services got their functionality improved, and the API provider got some profits.

Access to the API might be unconditionally paid. However, hybrid models are more common: the API is free until some threshold is reached, such as usage limits or constraints (for example, only non-commercial projects are allowed). Sometimes the API is provided for free with minimal restrictions to popularize the platform (for example, Apple Maps).

B2B services are a special case. As B2B service providers benefit from offering diverse capabilities to partners, and conversely partners often require maximum flexibility to cover their specific needs, providing an API might be the optimal solution for both. Large companies have their own IT departments and more frequently need APIs to connect them to internal systems and integrate them into business processes. Also, the API provider company itself might play the role of such a B2B customer if its own products are built on top of the API.

NB: We rather disapprove of the practice of providing an external API merely as a byproduct of the internal one without making any changes to bring value to the market. The main problem with such APIs is that partners' interests are not taken into account, which leads to numerous problems:

- The API doesn't cover integration use cases well:
    - Internal customers employ a quite specific technological stack, and the API is poorly optimized to work with other programming languages / operating systems / frameworks.
    - For external customers, the learning curve will be pretty steep as they can't take a look at the source code or talk to the API developers directly, unlike internal customers who are much more familiar with the API concepts.
    - Documentation often covers only some subset of use cases needed by internal customers.

    - The API services ecosystem which we will describe in the "API Services Range" chapter usually doesn't exist.
- Any resources spent are directed towards covering internal customer needs first. It means the following:
    - API development plans are totally opaque to partners and sometimes look absurd with obvious problems being neglected for years.
    - Technical support of external customers is financed on leftovers.
- Often, developers of internal services break backward compatibility or issue new major versions whenever they need it and don't care about the consequences of these decisions for external API partners.

All those problems lead to having an external API that actually hurts the company's reputation, not improves it. You're providing a very bad service for a very critical and skeptical audience. If you don't have enough resources to develop the API as a product for external customers, it's better not to even start.

5. API = an Advertisement Site

In this case, we mostly talk about things like widgets and search engines as direct access to end users is a must to display commercials. The most typical examples of such APIs are advertisement networks APIs. However, mixed approaches do exist as well, meaning that some API, usually a searching one, goes with commercial insets. [In our coffee example, it means that the offer searching function will start promoting paid results on the search results page.]

6. API = Self-Advertisement and Self-PR

If an API has neither explicit nor implicit monetization, it might still generate some income by increasing the company's brand awareness through displaying logos and other recognizable elements in partners' apps, either native (if the API goes with UI elements) or agreed-upon ones (if partners are obliged to embed specific branding in those places

where the API functionality is used or the data acquired through API is displayed). The API provider company's goals in this case are either attracting users to the company's services or just increasing brand awareness in general. [In the case of our coffee example, let's imagine that our main business is something totally unrelated to coffee machine APIs, like selling tires, and by providing the API we hope to increase brand recognition and get a reputation as an IT company.]

The target audiences for such self-promotion might also differ:

- You might seek to increase brand awareness among end users (by embedding logos and links to your services on partner's websites and applications), and even build the brand exclusively through such integrations [for example if our coffee API provides coffee shop ratings and we're working hard to make consumers demand the coffee shops to publish the ratings].
- You might concentrate efforts on increasing awareness among business owners [for example, for partners integrating a coffee ordering widget on their websites, also pay attention to your tires catalog].
- Finally, you might provide APIs only to make developers know your company's name to increase their knowledge of your other products or just to improve your reputation as an employer (this activity is sometimes called “tech-PR”).

Additionally, we might talk about forming a community, i.e., a network of developers (or customers, or business owners) who are loyal to the product. The benefits of having such a community might be substantial: lowering the technical support costs, getting a convenient channel for publishing announcements regarding new services and new releases, and obtaining beta users for upcoming products.

7. API = a Feedback and UGC Tool

If a company possesses some big data, it might be useful to provide a public API for users to make corrections in the data or otherwise get involved in working with it. For example, cartographical API providers usually allow the audience to post feedback or correct mistakes right on

partners' websites and applications. [In the case of our coffee API, we might be collecting feedback to improve the service, both passively through building coffee shop ratings or actively through contacting business owners to convey users' requests or finding new coffee shops that are still not integrated with the platform.]

8. Terraforming

Finally, the most altruistic approach to API product development is providing it free of charge (or as an open-source or open-data project) just to change the landscape. If today nobody is willing to pay for the API, we might invest in popularizing the functionality hoping to find commercial niches later (in any of the aforementioned formats) or to increase the significance and usefulness of the API integrations for end users (and therefore the readiness of the partners to pay for the API). [In the case of our coffee example, imagine a coffee machine maker that starts providing APIs for free aiming to make having an API a “must” for every coffee machine vendor thus allowing for the development of commercial API-based services in the future.]

9. Gray Zones

One additional source of income for the API provider is the analysis of the requests that end users make. In other words, collecting and reselling some user data. You must be aware that the difference between acceptable data collection (such as aggregating search requests to understand trends or finding promising locations for opening a coffee shop) and unacceptable ones is quite vague and tends to vary in time and space (e.g., some actions might be totally legal on one side of the state border and totally illegal on the other side). Making a decision to monetize the API with it should be carried out with extreme caution.

In the last several years we have seen the trend of providing some functionality as an API (i.e., as a product for developers) instead of developing the service for end users. This approach, dubbed “API-first,” reflects the growing specialization in the IT world: developing APIs becomes a separate area of expertise that businesses are ready to outsource instead of spending resources to develop internal APIs for their applications by the in-house IT department. However, this approach is not universally accepted (yet), and you should keep in mind the factors that affect the decision of launching a service in the API-first paradigm:

1. The target market must be sufficiently heated up: there must be companies there that possess enough resources to develop services atop third-party APIs and pay for it (unless your aim is terraforming).

2. The quality of the service must not suffer if the service is provided only through the API.

3. You must really possess expertise in API development; otherwise, there are high chances of making too many design mistakes.

Sometimes providing APIs is a method to “probe the ground,” i.e., to evaluate the market and decide whether it's worth having a full-scale user service there. (We rather condemn this practice as it inevitably leads to discontinuing the API or limiting its functionality, either because the market turns out to be not as profitable as expected, or because the API eventually becomes a competitor to the main service.)

Chapter 52. Developing a Product Vision

The above-mentioned fragmentation of the API target audience, i.e., the "developers — business — end users" triad, makes API product management quite a non-trivial problem. Yes, the basics are the same: find your audience's needs and satisfy them. The problem is that your product has several different audiences, and their interests sometimes diverge. The end users' request for an affordable cup of coffee does not automatically imply business demand for a coffee machine API.

Generally speaking, the API product vision must include the same three elements:

- Grasping how *end users* would like to have their problems solved
- Projecting how *businesses* would solve those problems if appropriate tools existed
- Understanding what technical solutions for *developers* might exist to help them implement the functionality businesses would ask for, and where the boundaries of their applicability lie.

In different markets and different situations, the "weight" of each element differs. If you're creating an API-first product for developers with no UI components, you might skip the end users' problem analysis. On the other hand, if you're providing an API with extremely valuable functionality and you hold a close-to-monopolistic position in the market, you might actually not care about how developers love your software architecture or the convenience of your interfaces for them, as they simply have no other choice.

Still, in the majority of cases, we have to deal with two-step heuristics based on either technical capabilities or business demands:

- You might first form the vision of how you can help business owners given the technical capabilities you have (heuristics step one) then develop a general vision of how your API will be used to satisfy end users' needs (heuristics step two), or

- Given your understanding of business owners' problems, you can take one heuristic “step right” to outline future functionality for end users and one “step left” to evaluate possible technical solutions.

Since both approaches are still heuristic, the API product vision is inevitably fuzzy, and that's quite normal. If you had a full and clear understanding of what end-user products could be developed on top of your API, you might have developed them yourself, bypassing intermediary agents. It is also important to keep in mind that many APIs go through the “terraforming” stage (see the previous chapter), preparing the ground for new markets and new types of services. Therefore, your idealistic vision of a nearby future where delivering freshly brewed coffee by drones becomes the norm of life is to be refined and clarified as new companies providing new kinds of services enter the market. (This, in turn, will impact monetization models as detailing the countenance of the forthcoming will make your abstract KPIs and theoretical benefits of having an API more and more concrete.)

The same fuzziness should be kept in mind when conducting interviews and gathering feedback. Software engineers will mainly report problems they encountered during technical integrations and rarely discuss business-related issues. Meanwhile, business owners care little about the inconvenience of writing code. Both groups will have some knowledge regarding end users' problems, but it's usually limited to the market segment in which the partner operates.

If you do have access to end users' action monitoring (see “The API Key Performance Indicators” chapter), you can try to analyze typical user behavior through these logs and understand how users interact with the partners' applications. However, you will need to conduct this analysis on a per-application basis and attempt to clusterize the most common scenarios.

Checking Product Hypotheses

In addition to the general complexity of formulating the product vision, there are also tactical issues with checking product hypotheses. “The Holy Grail” of product management — creating a cheap (in terms of resource expenditure) minimal viable product (MVP) — is typically unavailable for an API product manager. The reason is that you can't easily *test* the solution even if you manage to develop an API MVP. To do so, partners would need to *develop an application*, i.e., invest their money. If the outcome of the experiment is negative, meaning that further development appears unpromising, this money will be wasted. Of course, partners will be a little bit skeptical towards such proposals. Therefore, a “cheap” MVP should include either compensation for partners' expenses or a budget to develop a reference implementation (i.e., a complementary application specifically developed to support the API MVP).

You can partially address the problem by having a third-party company release the MVP, for example, in the form of an open-source module published in a developer's personal repository. However, you will struggle with hypothesis validation issues as such modules might easily go unnoticed.

Another option for checking conjectures is recruiting other developers within the API provider company to try the API in their services. Internal customers are usually much more loyal towards spending some effort to check a hypothesis, and it's much easier to negotiate MVP curtailing or freezing with them. The problem is that you can only validate those ideas that are relevant to the company's internal needs.

Also, applying the “eat your own dog food” concept to APIs means that the API product team should have their own test applications (so-called “pet projects”) on top of the API. Given the complexity of developing such applications, it makes sense to encourage having them, for example, by giving free API quotas to team members and providing sufficient free computational resources.

Such pet projects are also valuable because of the unique experience they allow to gain: everyone might try a new role. Developers will learn about product managers' typical problems: it's not enough to write good code, you also need to know your customer, understand their demands,

formulate an attractive concept, and effectively communicate it. On the other hand, product managers will gain understanding of how exactly easy or hard it is to bring their product vision to life and what challenges they may face. Finally, both groups will benefit from taking a fresh look at the API documentation and putting themselves in the shoes of a developer who is hearing about the API product for the first time and struggling to grasp the basics.

Chapter 53. Communicating with Developers

As we have described in the previous chapters, managing an API product requires building relationships with both business partners and developers. (Ideally, with end users as well, although this option is seldom available to API providers.)

Let's start with developers. The specifics of software engineers as an audience are as follows:

- Developers are highly educated individuals with practical thinking. As a rule, they choose technical products with extreme rationality (unless you're giving them cool backpacks with fancy prints for free).
    - This doesn't prevent them from having a certain aptitude towards, let's say, specific programming languages or frameworks; however, *influencing* those aptitudes is extremely hard and is normally not within the API vendor's power.
- Developers are quite skeptical towards promotional materials and overstatements. They are ready to actually check whether your claims are true.
- It is very hard to communicate with developers via regular marketing channels. They get information from highly specialized communities and they stick to opinions proved by concrete numbers and examples (ideally, code samples).
    - The words of “influencers” are not very valuable to them, as no opinions are trusted if unsubstantiated.
- The ideas of open source and free software are widespread among developers If you try to make money out of things that they believe should be free and/or open (for example, by proclaiming interfaces as intellectual property), you will face resistance (and views on these “shoulds” differ).

Because of the aforementioned specifics (especially the relative insignificance of influencers and the critical attitude towards promotions), you will have to communicate to developers via very specific media:

- Collective blogs (like the “r/programming” subreddit or dev.to)
- Q&A sites (Stack Overflow, Experts Exchange)
- Educational services (Codecademy, Udemy)
- Technical conferences and webinars.

In all these channels, direct advertising of your API is either problematic or impossible. (Well, strictly speaking, you may buy a banner on one of the sites advertising the advantages of your API, but we highly doubt it will improve your relations with developers.) You need to generate valuable and/or interesting content for them, which will improve their knowledge of your API. And this is the job for your developers: writing articles, answering questions, recording webinars, and giving pitches.

Developers enjoy sharing their experiences and will probably be eager to do so — during their work hours. A proper conference talk, let alone an educational course, requires a lot of preparation time. From this book's author's experience, two things are crucial for tech PR:

- Incentives, even nominal ones — the job of promoting a product should be rewarded
- Methodicalness and quality standards — you might actually do content review just like you do code review.

Nothing could be worse counter-advertising for your product than a poorly prepared pitch (as we said, mistakes will inevitably be found and pointed out) or a badly camouflaged commercial in the form of a pitch (the reason is actually the same). Texts need to be worked upon: pay attention to the structure, logic, and tempo of the narration. Even a technical story must be finely constructed; after it's ended, the listeners must have a clear understanding of the idea you wanted to communicate (and it should rather be linked with your API's fitness for their needs).

A word on “evangelists” (those are people who have credibility in the IT community and work on promoting a technology or a tech company, either as a company's contractor or even a staff member, effectively carrying out all those aforementioned activities, such as blog posting, course preparing, conference speaking, etc.) Having an evangelist makes the API development team exempt from the necessity of performing

tech PR. However, we would rather advise having this expertise inside the team, as direct interaction with developers helps in forming the product vision. (That doesn't mean evangelists are not needed at all - you might well combine these two strategies.)

Open Source

The important question that will sooner or later arise for any API vendor is whether to make the source code open. This decision has both advantages and disadvantages:

- You will improve the knowledge of the brand, and some respect will be paid to you by the IT community.
    - Given that your code is finely written and commented.
- You will receive some additional feedback, ideally in the form of pull requests from third-party developers.
    - You will also receive a number of inquiries and comments ranging from useless to obviously provocative ones, to which you will have to respond politely.
- Donating code to open source makes developers trust the company more, and affects their readiness to rely on the platform.
    - However, it also increases risks, both from an information security point of view and in terms of the product, as a dissatisfied community might fork your repository and create a competing product.

Finally, just the preparations to make the code open might be very expensive. You need to clean the code, switch to open building and testing tools, and remove all references to proprietary resources. This decision is to be made very cautiously after considering all pros and cons. We might add that many companies try to reduce the risks by splitting the API code into two parts: the open one and the proprietary one. Additionally, the risks could be mitigated by selecting a license that disallows harming the company's interests by using the open-sourced code (for example, by prohibiting selling hosted solutions or by requiring the derivative works to be open-sourced as well).

The Audience Fragmentation

There is one very important addition to the discourse: as information technologies are universally in great demand, a significant percentage of your customers will not be professional software engineers. A huge number of people are somewhere on the path to mastering the occupation. Some are trying to write code in addition to the basic duties, others are undergoing retraining, and some are studying the basics of computer science on their own. Many of these non-professional developers have a direct impact on the process of selecting an API vendor, such as small business owners who seek to automate routine tasks programmatically.

It would be more accurate to say that API providers are actually working for two main types of audiences:

- Beginners and amateurs, for whom each integration task would be completely new and unexplored territory
- Professional developers who possess vast experience in integrating different third-party systems.

This fact greatly affects everything we discussed previously (except for, perhaps, open-sourcing, as amateur developers pay little attention to it):

- Your pitches, webinars, lectures, etc., must somehow cater to both professional and semi-professional audiences.
- A significant share of inquiries to your customer support service will be generated by the first category of developers. It is much harder for amateurs or beginners to find answers to their questions by themselves, and they will reach out to you for assistance.
- At the same time, the second category is much more sensitive to the quality of both the product and customer support, and fulfilling their requests might be non-trivial.

Finally, it is almost impossible to create an API that will suit both amateur and professional developers well within a single product. The former need maximum simplicity in implementing basic use cases, while the latter seek the ability to adapt the API to match their technological

stack and development paradigms and the problems they solve usually require deep customization. We will discuss this matter in the “API Services Range” chapter.

Chapter 54. Communicating with Business Owners

The basics of interacting with business partners are to some extent paradoxically contrary to the basics of communicating with developers:

- On one hand, partners are much more loyal and sometimes even enthusiastic regarding the opportunities you offer (especially free ones).
- On the other hand, conveying the meaning of your offer to business owners is much more complicated than explaining it to developers, as it is generally hard to explain the advantages of integrating via APIs as a concept to a non-technical person.

After all, working with a business audience essentially means elucidating the characteristics and advantages of the product. In that sense, an API “sells” just like any other kind of software.

As a rule, the farther an industry sector is from information technologies, the more enthusiastic its representatives are about your API features and the less likely this enthusiasm will be converted into a real integration. One thing that could help the case is extensive work with the developer community (see the previous chapter), which will result in establishing a circle of freelancers and outsourcers eager to help non-IT businesses with integrations. You can contribute to developing this market by creating educational courses and issuing certificates that prove the bearer's skills in working with your API (or a broader layer of technology).

Market research and gathering feedback from business owners work similarly. Businesses that are far from IT usually struggle to articulate their demands, so you should be rather creative (and critical-minded) when analyzing the gathered data.

Chapter 55. An API Services Range

The important rule of API product management that any major API provider will soon learn is this: don't just ship one specific API; there is always room for a range of products, and this range is two-dimensional.

Horizontal Scaling of API Services

Usually, any functionality available through an API can be split into independent units. For example, in our coffee API, there are offer search endpoints and order processing endpoints. Nothing could prevent us from either pronouncing those functional clusters as different APIs or, vice versa, considering them as parts of one API.

Different companies employ different approaches to determining the granularity of API services, i.e., what is counted as a separate product and what is not. To some extent, this is a matter of convenience and subjective judgment. Consider splitting an API into parts if:

- It makes sense for partners to integrate only one API part, i.e., there are isolated subsets of the API that alone provide enough means to solve users' problems.
- API parts can be versioned separately and independently, and it is meaningful from the partners' point of view. This usually means that those “isolated” APIs are either fully independent or maintain strict backward compatibility and introduce new major versions only when absolutely necessary. Otherwise, maintaining a matrix which API #1 version is compatible with which API #2 version will soon become a catastrophe.
- It makes sense to set tariffs and limits for each API service independently.
- The audiences of the API segments (developers, business owners, or end users) do not overlap, and “selling” a granular API to customers is much easier than an aggregated one.

NB: These split APIs might still be part of a united SDK to make programmers' lives easier.

Vertical Scaling of API Services

However, it often makes sense to provide multiple API services that feature the same functionality. Let us remind you that there are two types of developers: professional ones who seek extensive customization capabilities (as they usually work in big IT companies with a specific mindset towards integrations) and semi-professionals who just need the gentlest possible learning curve. The only way to cover the needs of both categories is to develop a range of products with different entry thresholds and requirements for developers' professional level. We can identify several sub-types of APIs, ordered from the most technically demanding to the less complex ones.

1. The most advanced level is that of physical APIs and abstractions built on top of them. [In our coffee example, it refers to a collection of entities that describe working with APIs of physical coffee machines as discussed in the "Separating Abstraction Levels" and "Weak Coupling" chapters.]

2. Next is the basic level, which involves working with product entities through formal interfaces. [In our study example, this level corresponds to the HTTP API for placing orders.]

3. Working with product entities can be simplified by providing SDKs for popular platforms that tailor API concepts according to the paradigms of those platforms. This benefits developers who are proficient with specific platforms and saves them effort in dealing with formal protocols and interfaces.

4. The next simplification step is providing services for code generation. In this service, developers can choose from pre-built integration templates, customize option values, and obtain a ready-to-use piece of code that can be easily copied and pasted into their application code (and can be further customized by adding some level 1-3 code). This approach is sometimes called "point-and-click programming." [In the case of our coffee API, an example of such a service might have a form or screen editor for developers to place UI elements and generate the working

application code or a console script to automatically produce application boilerplate.]

5. Finally, this approach can be simplified even further if the service generates not just code but a ready-to-use component / widget / frame and a one-liner to integrate it. [For example, if we allow embedding an iframe that handles the entire coffee ordering process directly on the partner's website, or describe the rules of forming a “deep link” to our service.]

   Ultimately, we will end up with the concept of a meta-API where these high-level components have their own API built on top of the basic API.

The important advantage of having a range of APIs is not only about adapting it to the developer's capabilities but also about increasing the level of control you have over the code that partners embed into their applications:

1. Applications that use physical interfaces are out of your control. For example, you can't force switching to newer versions of the platform or, let's say, add commercial inlets to them.
2. Applications that operate base APIs will allow you to manipulate underlying abstraction levels — move to newer technologies or alter the way search results are presented to an end user.
3. SDKs, especially those providing UI components, provide a higher degree of control over the look and feel of partners' applications, which allows you to evolve the UI by adding new interactive elements and enriching the functionality of existing ones. [For example, if our coffee SDK contains a map of coffee shops, nothing could stop us from making map objects clickable in the next API version or highlighting paid offerings.]
4. Code generation makes it possible to manipulate the desired form of integrations. For example, if our KPI is the number of searches performed through the API, we can alter the generated code so that the search panel will be displayed in the most convenient position in the application. As partners who use code-generation

services rarely make any changes to the resulting code, this will help us reach our goals.

5. Finally, ready-to-use components and widgets are under your full control, and you can experiment with the functionality exposed through them in partners' applications as if it were your own services. (However, this doesn't automatically mean that you can draw profits from having this control. For example, if you allow hotlinking pictures by their direct URL, your control over this integration is rather negligible, so it's generally better to provide integration options that allow for more control over the functionality in partners' applications.)

NB: While developing a "vertical" range of APIs, it is crucial to follow the principles discussed in the "On the Waterline of the Iceberg" chapter. You will only be able to manipulate the content and behavior of a widget if developers cannot "escape the sandbox," meaning they do not have direct access to low-level objects encapsulated within the widget.

In general, your aim should be to have each partner use the API services in a manner that maximizes your profit as an API vendor. When a partner only needs a typical solution, you would benefit from making them use widgets as they are under your direct control. This will help ease the API version fragmentation problem and allow for experimentation to reach your KPIs. When the partner possesses expertise in the subject area and develops a unique service on top of your API, you would benefit from allowing full freedom in customizing the integration. This way, they can cover specific market niches and enjoy the advantage of offering more flexibility compared to services using competing APIs.

Chapter 56. API Key Performance Indicators

As we described in the previous chapters, there are many API monetization models, both direct and indirect. Importantly, most of them are fully or conditionally free for partners, and the direct-to-indirect benefits ratio tends to change during the API lifecycle. That naturally leads us to the question of how exactly shall we measure the API success and what goals are to be set for the product team.

Of course, the most explicit metric is money: if your API is monetized directly or attracts visitors to a monetized service, the rest of the chapter will be of little interest to you, maybe just as a case study. If, however, the contribution of the API to the company's income cannot be simply measured, you have to stick to other, synthetic, indicators.

The obvious key performance indicator (KPI) #1 is the number of end users and the number of integrations (i.e., partners using the API). Normally, they are in some sense a business health barometer: if there is a normal competitive situation among the API suppliers, and all of them are more or less in the same position, then the figure of how many developers (and consequently, how many end users) are using the API is the main metric of success of the API product.

However, sheer numbers might be deceiving, especially if we talk about free-to-use integrations. There are several factors that make them less reliable:

- The high-level API services that are meant for point-and-click integration (see the previous chapter) are significantly distorting the statistics, especially if the competitors don't provide such services; typically, for one full-scale integration there will be tens, maybe hundreds, of those lightweight embedded widgets.
    - Thereby, it's crucial to have partners counted for each kind of the integration independently.
- Partners tend to use the API in suboptimal ways:
    - Embed it at every website page / application screen instead of only those where end users can really interact with the API

    - Put widgets somewhere deep in the page / screen footer, or hide it behind spoilers
    - Initialize a broad range of API modules, but use only a limited subset of them.
- The greater the API audience is, the less the number of unique visitors means as at some moment the penetration will be close to 100%; for example, a regular Internet user interacts with Google or Facebook counters, well, every minute, so the daily audience of those API fundamentally cannot be increased further.

All the abovementioned problems naturally lead us to a very simple conclusion: not only the raw numbers of users and partners are to be gauged, but their engagement as well, i.e., the target actions (such as searching, observing some data, interacting with widgets) shall be determined and counted. Ideally, these target actions must correlate with the API monetization model:

- If the API is monetized through displaying ads, then the user's activity towards those ads (e.g., clicks, interactions) is to be measured.
- If the API attracts customers to the core service, then count the transitions.
- If the API is needed for collecting feedback and gathering UGC, then calculate the number of reviews left and entities edited.

Additionally, the functional KPIs are often employed: how frequently some API features are used. (Also, it helps with prioritizing further API improvements.) In fact, that's still measuring target actions, but those that are made by developers, not end users. It's rather complicated to gather the usage data for software libraries and frameworks, though still doable (however, you must be extremely cautious with that, as any audience rather nervously reacts to finding that some statistic is gathered automatically).

The most complicated case is that of API being a tool for (tech)PR and (tech)marketing. In this case, there is a cumulative effect: increasing the API audience doesn't momentarily bring any profit to the company. *First*, you got a loyal developer community, *then* this reputation helps

you to hire people. *First*, your company's logo flashes on third-party webpages and applications, *then* the top-of-mind brand knowledge increases. There is no direct method of evaluating how some action (let's say, a new release or an event for developers) affects the target metrics. In this case, you have to operate indirect metrics, such as the audience of the documentation site, the number of mentions in the relevant communication channels, the popularity of your blogs and seminars, etc.

Let us summarize the paragraph:

- Counting direct metrics such as the total number of users and partners is a must and is totally necessary for moving further, but that's not a proper KPI.
- The proper KPI should be formulated based on the number of target actions that are made through the platform.
- The definition of target action depends on the monetization model and might be quite straightforward (like the number of paying partners, or the number of paid ad clicks) or, to the contrary, pretty implicit (like the growth of the company's developer blog audience).

SLA

This chapter would be incomplete if we didn't mention the “hygienic” KPI — the service level and the service availability. We won't be describing the concept in detail, as the API SLA isn't any different from any other digital services SLAs. Let us just state that this metric must be tracked, especially if we talk about pay-to-use APIs. However, in many cases, API vendors prefer to offer rather loose SLAs, treating the provided functionality as a data access or content licensing service.

Still, let us re-iterate once more: any problems with your API are automatically multiplied by the number of partners you have, especially if the API is vital for them, i.e., the API outage makes the main functionality of their services unavailable. (And actually, because of the above-mentioned reasons, the average quality of integrations implies that partners' services will suffer even if the availability of the API is not

formally speaking critical for them, but because developers use it excessively and do not bother with proper error handling.)

It is important to mention that predicting the workload for the API service is rather complicated. Sub-optimal API usage, e.g., initializing the API in those application and website parts where it's not actually needed, might lead to a colossal increase in the number of requests after changing a single line of partner's code. The safety margin for an API service must be much higher than for a regular service for end users — it must survive the situation of the largest partner suddenly starting querying the API on every page and every application screen. (If the partner is already doing that, then the API must survive doubling the load if the partner by accident starts initializing the API twice on each page / screen.)

Another extremely important hygienic minimum is the informational security of the API service. In the worst-case scenario, namely, if an API service vulnerability allows for exploiting partner applications, one security loophole will in fact be exposed *in every partner application*. Needless to say that the cost of such a mistake might be overwhelmingly colossal, even if the API itself is rather trivial and has no access to sensitive data (especially if we talk about webpages where no “sandbox” for third-party scripts exists, and any piece of code might let's say track the data entered in forms). API services must provide the maximum protection level (for example, choose cryptographical protocols with a certain overhead) and promptly react to any reports regarding possible vulnerabilities.

Comparing to Competitors

While measuring KPIs of any service, it's important not only to evaluate your own numbers but also to match them against the state of the market:

- What is your market share, and how is it evolving over time?
- Is your service growing faster than the market itself or is the rate the same, or is it even less?

- What proportion of the growth is caused by the growth of the market, and what is related to your efforts?

Getting answers to those questions might be quite non-trivial in the case of API services. Indeed, how could you learn how many integrations has your competitor had during the same period of time, and what number of target actions had happened on their platform? Sometimes, the providers of popular analytical tools might help you with this, but usually, you have to monitor the potential partners' apps and websites and gather the statistics regarding APIs they're using. The same applies to market research: unless your niche is significant enough for some analytical company to conduct a study, you will have to either commission such work or make your own estimations — conversely, through interviewing potential customers.

Chapter 57. Identifying Users and Preventing Fraud

In the context of working with an API, we talk about two kinds of users of the system:

- Users-developers, i.e., your partners writing code atop of the API
- End users interacting with applications implemented by the users-developers.

In most cases, you need to have both of them identified (in a technical sense: discern one unique customer from another) to have answers to the following questions:

- How many users are interacting with the system (simultaneously, daily, monthly, and yearly)?
- How many actions does each user make?

NB: Sometimes, when an API is very large and/or abstract, the chain linking the API vendor to end users might comprise more than one developer as large partners provide services implemented atop of the API to the smaller ones. You need to count both direct and "derivative" partners.

Gathering this data is crucial because of two reasons:

- To understand the system's limits and to be capable of planning its growth
- To understand the number of resources (ultimately, money) that are spent (and gained) on each user.

In the case of commercial APIs, the quality and timeliness of gathering this data are twice that important, as the tariff plans (and therefore the entire business model) depend on it. Therefore, the question of *how exactly* we're identifying users is crucial.

Identifying Applications and Their Owners

Let's start with the first user category, i.e., API business partners-developers. The important remark: there are two different entities we must learn to identify, namely applications and their owners.

An application is roughly speaking a logically separate case of API usage, usually — literally an application (mobile or desktop one) or a website, i.e., some technical entity. Meanwhile, an owner is a legal body that you have the API usage agreement signed. If API Terms of Service (ToS) imply different limits and/or tariffs depending on the type of the service or the way it uses the API, this automatically means the necessity to track one owner's applications separately.

In the modern world, the factual standard for identifying both entities is using API keys: a developer who wants to start using an API must obtain an API key bound to their contact info. Thus the key identifies the application while the contact data identifies the owner.

Though this practice is universally widespread we can't but notice that in most cases it's useless, and sometimes just destructive.

Its general advantage is the necessity to supply actual contact info to get a key, which theoretically allows for contacting the application owner if needed. (In the real world, it doesn't work: key owners often don't read mailboxes they provided upon registration; and if the owner is a company, it easily might be a no-one's mailbox or a personal email of some employee that left the company a couple of years ago.)

The main disadvantage of using API keys is that they *don't* allow for reliably identifying both applications and their owners.

If there are free limits to API usage, there is a temptation to obtain many API keys bound to different owners to fit those free limits. You may raise the bar of having such multi-accounts by requiring, let's say, providing a phone number or bank card data, but there are popular services for automatically issuing both. Paying for a virtual SIM or credit card (to say nothing about buying the stolen ones) will always be cheaper than paying the proper API tariff — unless it's the API for creating those cards. Therefore, API key-based user identification (if you're not requiring the physical contract to be signed) does not mean

you don't need to double-check whether users comply with the terms of service and do not issue several keys for one app.

Another problem is that an API key might be simply stolen from a lawful partner; in the case of mobile or web applications, that's quite trivial.

It might look like the problem is not that important in the case of server-to-server integrations, but it actually is. Imagine that a partner provides a public service of their own that uses your API under the hood. That usually means there is an endpoint in the partner's backend that performs a request to the API and returns the result, and this endpoint perfectly suits as a free replacement of direct access to the API for a cybercriminal. Of course, you might say this fraud is a problem of partners', but, first, it would be naïve to expect each partner develops their own anti-fraud system, and, second, it's just sub-optimal: obviously, a centralized anti-fraud system would be way more effective than a bunch of amateur implementations. Also, server keys might also be stolen: it's much harder than stealing client keys but doable. With any popular API, sooner or later you will face the situation of stolen keys made available to the public (or a key owner just shared it with acquaintances out of the kindness of their heart).

One way or another, a problem of independent validation arises: how can we control whether the API endpoint is requested by a user in compliance with the terms of service?

Mobile applications might be conveniently tracked through their identifiers in the corresponding store (Google Play, App Store, etc.), so it makes sense to require this identifier to be passed by partners as an API initialization parameter. Websites with some degree of confidence might be identified by the `Referer` and `Origin` HTTP headers.

This data is not itself reliable, but it allows for making cross-checks:

- If a key was issued for one specific domain but requests are coming with a different `Referer`, it makes sense to investigate the situation and maybe ban the possibility to access the API with this `Referer` or this key.

- If an application initializes API by providing a key registered to another application, it makes sense to contact the store administration and ask for removing one of the apps.

NB: Don't forget to set infinite limits for using the API with the `localhost`, `127.0.0.1` / `[::1]` `Referer`s, and also for your own sandbox if it exists. Yes, abusers will sooner or later learn this fact and will start exploiting it, but otherwise, you will ban local development and your own website much sooner than that.

The general conclusion is:

- It is highly desirable to have partners formally identified (either through obtaining API keys or by providing contact data such as website domain or application identifier in a store while initializing the API).
- This information shall not be trusted unconditionally; there must be double-checking mechanisms that identify suspicious requests.

Identifying End Users

Usually, you can put forward some requirements for self-identifying of partners, but asking end users to reveal contact information is impossible in most cases. All the methods of measuring the audience described below are imprecise and often heuristic. (Even if partner application functionality is only available after registration and you do have access to that profile data, it's still a game of assumptions, as an individual account is not the same as an individual user: several different persons might use a single account, or, vice versa, one person might register many accounts.) Also, note that gathering this sort of data might be legally regulated (though we will be mostly speaking about anonymized data, there might still be some applicable law).

1. The most simple and obvious indicator is an IP address. It's very hard to counterfeit them (i.e., the API server always knows the remote address), and the IP address statistics are reasonably demonstrative.

If the API is provided as a server-to-server one, there will be no access to the end user's IP address. However, it makes sense to require partners to propagate the IP address (for example, in a form of the `X-Forwarded-For` header) — among other things, to help partners fight fraud and unintended usage of the API.

Until recently, IP addresses were also a convenient statistics indicator because it was quite expensive to get a large pool of unique addresses. However, with ipv6 advancement this restriction is no longer actual; ipv6 rather put the light on the fact that you can't just count unique addresses — the aggregates are to be tracked:

- The cumulative number of requests by networks, i.e., the hierarchical calculations (the number of /8, /16, /24, etc. networks)
- The cumulative statistics by autonomous networks (AS)
- The API requests through known public proxies and TOR network.

An abnormal number of requests in one network might be evidence of the API being actively used inside some corporative environment (or NATs being widespread in the region).

2. Additional means of tracking are users' unique identifiers, most notably cookies. However, most recently this method of gathering data got attacked from several directions: browser makers restrict third-party cookies, users are employing anti-tracker software, and lawmakers started to roll out legal requirements against data collection. In the current situation, it's much easier to drop cookie usage than to be compliant with all the regulations.

   All this leads to a situation when public APIs (especially those installed on free-to-use sites and applications) are very limited in the means of collecting statistics and analyzing user behavior. And that impacts not only fighting all kinds of fraud but analyzing use cases as well. This is the way.

NB: In some jurisdictions, IP addresses are considered personal data, and collecting them is prohibited as well. We don't dare to advise on how an API vendor might at the same time be able to fight prohibited content on the platform and don't have access to users' IP addresses. We presume that complying with such legislation implies storing statistics by IP address hashes. (And just in case we won't mention that building a rainbow table for SHA-256 hashes covering the entire 4-billion range of IPv4 addresses would take several hours on a regular office-grade computer.)

Chapter 58. The Technical Means of Preventing ToS Violations

Implementing the paradigm of a centralized system of preventing partner endpoints-bound fraud, which we described in the previous chapter, in practice faces non-trivial difficulties.

The task of filtering out illicit API requests comprises three steps:

- Identifying suspicious users
- Optionally, asking for an additional authentication factor
- Making decisions and applying access restrictions.

1. Identifying Suspicious Users

Generally speaking, there are two approaches we might take, the static one and the dynamic (behavioral) one.

Statically we monitor suspicions activity surges, as described in the previous chapter, marking an unusually high density of requests coming from specific networks or `Referer`s (actually, *any* piece of information suits if it splits users into more or less independent groups: for example, OS version or system language would suffice if you can gather those).

Behavioral analysis means we're examining the history of requests made by a specific user, searching for non-typical patterns, such as "unhuman" order of traversing endpoints or too small pauses between requests.

Importantly, when we talk about "users," we will have to make duplicate systems to observe them both using tokens (cookies, logins, phone numbers) and IP addresses, as malefactors aren't obliged to preserve the tokens between requests, or might keep a pool of them to impede their exposure.

2. Requesting an Additional Authentication Factor

As both static and behavioral analyses are heuristic, it's highly desirable to not make decisions based solely on their outcome but rather ask the suspicious users to additionally prove they're making legitimate requests. If such a mechanism is in place, the quality of an anti-fraud system will be dramatically improved, as it allows for increasing system sensitivity and enabling pro-active defense, i.e., asking users to pass the tests in advance.

In the case of services for end users, the main method of acquiring the second factor is redirecting to a captcha page. In the case of APIs it might be problematic, especially if you initially neglected the "Stipulate Restrictions" rule we've given in the "Describing Final Interfaces" chapter. In many cases, you will have to impose this responsibility on partners (i.e., it will be partners who show captchas and identify users based on the signals received from the API endpoints). This will, of course, significantly impair the convenience of working with the API.

NB: Instead of captcha, there might be other actions introducing additional authentication factors. It might be the phone number confirmation or the second step of the 3D-Secure protocol. The important part is that requesting an additional authentication step must be stipulated in the program interface, as it can't be added later in a backward-compatible manner.

Other popular mechanics of identifying robots include offering a bait ("honeypot") or employing the execution environment checks (starting from rather trivial ones like executing JavaScript on the webpage and ending with sophisticated techniques of checking application integrity checksums).

3. Restricting Access

The illusion of having a broad choice of technical means of identifying fraud users should not deceive you as you will soon discover the lack of effective methods of restricting those users. Banning them by cookie / `Referer` / `User-Agent` makes little to no impact as this data is supplied by clients, and might be easily forged. In the end, you have four mechanisms for suppressing illegal activities:

- Banning users by IP (networks, autonomous systems)
- Requiring mandatory user identification (maybe tiered: login / login with confirmed phone number / login with confirmed identity / login with confirmed identity and biometrics / etc.)
- Returning fake responses
- Filing administrative abuse reports.

The problem with the first option is the collateral damage you will inflict, especially if you have to ban subnets.

The second option, though quite rational, is usually inapplicable to real APIs, as not every partner will agree with the approach, and definitely not every end user. This will also require being compliant with the existing personal data laws.

The third option is the most effective one in technical terms as it allows to put the ball in the malefactor's court: it is now them who need to invent how to learn if the robot was detected. But from the moral point of view (and from the legal perspective as well) this method is rather questionable, especially if we take into account the probability of false-positive signals, meaning that some real users will get the fake data.

Thereby, you have only one method that really works: filing complaints to hosting providers, ISPs, or law enforcement authorities. Needless to say, this brings certain reputational risks, and the reaction time is rather not lightning fast.

In most cases, you're not fighting fraud — you're actually increasing the cost of the attack, simultaneously buying yourself enough time to make administrative moves against the perpetrator. Preventing API misusage completely is impossible as malefactors might ultimately employ the expensive but bulletproof solution — to hire real people to make the requests to the API on real devices through legitimate applications.

An opinion exists, which the author of this book shares, that engaging in this sword-against-shield confrontation must be carefully thought out, and advanced technical solutions are to be enabled only if you are one hundred percent sure it is worth it (e.g., if they steal real money or data). By introducing elaborate algorithms, you rather conduct an evolutional

selection of the smartest and most cunning cybercriminals, counteracting to whom will be way harder than to those who just naïvely call API endpoints with `curl`. What is even more important, in the final phase — i.e., when filing the complaint to authorities — you will have to prove the alleged ToS violation, and doing so against an advanced fraudster will be problematic. So it's rather better to have all the malefactors monitored (and regularly complained against), and escalate the situation (i.e., enable the technical protection and start legal actions) only if the threat passes a certain threshold. That also implies that you must have all the tools ready, and just keep them below fraudsters' radars.

Out of the author of this book's experience, the mind games with malefactors, when you respond to any improvement of their script with the smallest possible effort that is enough to break it, might continue indefinitely. This strategy, i.e., making fraudsters guess which traits were used to ban them this time (instead of unleashing the whole heavy artillery potential), annoys amateur "hackers" greatly as they lack hard engineering skills and just give up eventually.

Dealing with Stolen Keys

Let's now move to the second type of unlawful API usage, namely using in the malefactor's applications keys stolen from conscientious partners. As the requests are generated by real users, captcha won't help, though other techniques will.

1. Maintaining metrics collection by IP addresses and subnets might be of use in this case as well. If the malefactor's app isn't a public one but rather targeted to some closed audience, this fact will be visible in the dashboards (and if you're lucky enough, you might also find suspicious `Referer`s, public access to which is restricted).

2. Allowing partners to restrict the functionality available under specific API keys:

    - Setting the allowed IP address range for server-to-server APIs, allowed `Referer`s and application ids for client APIs
    - White-listing only allowed API functions for a specific key

    - Other restrictions that make sense in your case (in our coffee API example, it's convenient to allow partners to prohibit API calls outside of countries and cities they work in).

3. Introducing additional request signing:

    - For example, if on the partner's website, there is a form displaying the best lungo offers, for which the partners call the API endpoint like `/v1/search?recipe=lungo&api_key={apiKey}`, then the API key might be replaced with a signature like `sign = HMAC("recipe=lungo", apiKey)`. The signature might be stolen as well, but it will be useless for malefactors as they will be able to find only lungo with it.
    - Instead of API keys, time-based one-time passwords (TOTP) might be used. These tokens are valid for a short period of time only (typically, one minute), which makes using stolen keys much more complicated.

4. Filing complaints to the administration (hosting providers, app store owners) in case the malefactor distributes their application through stores or uses a diligent hosting service that investigates abuse filings. Legal actions are also an option, and even much so compared to countering user fraud, as illegal access to the system using stolen credentials is unambiguously outlawed in most jurisdictions.

5. Banning compromised API keys; the partners' reaction will be, of course, negative, but ultimately every business will prefer temporary disabling of some functionality over getting a multi-million bill.

Chapter 59. Supporting customers

From banning users, let's change the topic to supporting them. First of all, an important remark: when we talk about supporting API customers, we mean helping developers and to some extent business partners. End users seldom interact with APIs directly, with an exception of several non-standard cases:

1. If you can't reach partners that are using the API incorrectly, you might have to display errors that end users can see. This might happen if the API was provided for free and with minimum partner identification requirements while in the growth phase, and then the conditions changed (a popular API version is no longer supported or became paid).

2. If the API vendor cannot reproduce some problem and has to reach out end users to get additional diagnostics.

3. If the API is used to gather UGC content.

The first two cases are actually consequences of product-wise or technical flaws in the API development, and they should be avoided. The third case differs little from supporting end users of the UGC service itself.

If we talk about supporting partners, it's revolving around two major topics:

- Legal and administrative support with regard to the terms of service and the SLA (and that's usually about responding to business owners' inquiries)
- Helping developers with technical issues.

The former is of course extremely important for any healthy service (including APIs) but again bears little API-related specifics. In the context of this book, we are much more interested in the latter.

As an API is a program product, developers will be in fact asking how this specific piece of code that they have written works. This fact raises the level of required customer support staff members' expertise quite high as you need a software engineer to read the code and understand the problem. But this is but half of the problem; another half is, as we have mentioned in the previous chapters, that most of these questions will be asked by inexperienced or amateur developers. In a case of a popular API, it means that 9 out of 10 inquiries *will not be about the API*. Less skilled developers lack language knowledge, their experience with the platform is fragmented, and they can't properly formulate their problem (and therefore search for an answer on the Internet before contacting support; though, let us be honest, they usually don't even try).

There are several options for tackling these issues:

1. The most user-friendly scenario is hiring people with basic technical skills as the first line of support. These employees must possess enough expertise in understanding how the API works to be able to identify those unrelated questions and respond to them according to some FAQ, point out to a relevant external resource (let's say, the support service of the OS or the community forum of the programming language) if the problem is not related to the API itself, and redirect relevant issues to the API developers.

2. The inverse scenario: partners must pay for technical support, and it's the API developers who answer the questions. It doesn't actually make a significant difference in terms of the quality of the issues (it's still mostly inexperienced developers who can't solve the problem on their own; you will just cut off those who can't afford paid support) but at least you won't have a hiring problem as you might allow yourself the luxury of having engineers for the first line of support.

3. Partly (or, sometimes, fully) the developer community might help with solving the amateur problems (see the “Communicating with Developers” chapter). Usually, community members are pretty capable of answering those questions, especially if moderators help them.

Importantly, whatever options you choose, it's still the API developers in the second line of support simply because only they can fully understand the problem and the partners' code. That implies two important consequences:

1. You must take into account working with inquiries while planning the API development team time. Reading unfamiliar code and remote debugging are very hard and exhausting tasks. The more functionality you expose and the more platforms you support, the more load is put on the team in terms of dealing with support tickets.

2. As a rule, developers are totally not happy about the perspective of coping with incoming requests and answering them. The first line of support will still let through a lot of dilettante or badly formulated questions, and that will annoy on-duty API developers. There are several approaches to mitigate the problem:

    - Try to find people with a customer-oriented mindset, who like this activity, and encourage them (including financial stimulus) to perform support functions; it might be someone on the team (and not necessarily a developer) or some active community member.
    - The remaining load must be distributed among the developers equally and fairly, up to introducing the duty calendar.

And of course, analyzing the questions is a useful exercise to populate FAQs and improve the documentation and the first-line support scripts.

External Platforms

Sooner or later, you will find that customers ask their questions not only through the official channels, but also on numerous Internet-based forums, starting from those specifically created for this, like StackOverflow, and ending with social networks and personal blogs. It's up to you whether to spend time searching for such inquiries. We would

rather recommend providing support through those platforms that have convenient tools for that (like subscribing to specific tags).

Chapter 60. The Documentation

Regretfully, many API providers pay miserable attention to the quality of documentation. Meanwhile, the documentation is the face of the product and the entry point to it. The problem becomes even worse if we acknowledge that it's almost impossible to write the help docs the developers will consider at least satisfactory.

Before we start describing documentation types and formats, we should stress one important statement: developers interact with your help articles totally unlike you expect them to. Remember yourself working on the project: you make quite specific actions.

1. First, you need to determine whether this service covers your needs in general (as quickly as possible);
2. If it does, you look for specific functionality to resolve your specific case.

In fact, newcomers (i.e., those developers who are not familiar with the API) usually want just one thing: to assemble the code that solves their problem out of existing code samples and never return to this issue again. Sounds not exactly reassuringly, given the amount of work invested into the API and its documentation development, but that's what the reality looks like. Also, that's the root cause of developers' dissatisfaction with the docs: it's literally impossible to have articles covering exactly that problem the developer comes with being detailed exactly to the extent the developer knows the API concepts. In addition, non-newcomers (i.e., those developers who have already learned the basics concepts and are now trying to solve some advanced problems) do not need these “mixed examples” articles as they look for some deeper understanding.

Introductory Notes

Documentation frequently suffers from being excessively clerical; it's being written using formal terminology (which often requires reading the glossary before the actual docs) and is frequently unreasonably inflated. So instead of a two-word answer to a user's question, a couple

of paragraphs is conceived — a practice we strongly disapprove of. The perfect documentation must be simple and laconic, and all the terms must be either explained in the text or given a reference to such an explanation. However, “simple” doesn't mean “illiterate”: remember, the documentation is the face of your product, so grammar errors and improper usage of terms are unacceptable.

Also, keep in mind that documentation will be used for searching as well, so every page should contain all the keywords required to be properly ranked by search engines. Unfortunately, this requirement contradicts the simple-and-laconic principle; this is the way.

Documentation Content Types

1. Specification / Reference

Any documentation starts with a formal functional description. This content type is the most inconvenient to use, but you must provide it. A reference is the hygienic minimum of the API documentation. If you don't have a doc that describes all methods, parameters, options, variable types, and their allowed values, then it's not an API but amateur dramatics.

Today, a reference must be also a machine-readable specification, i.e., comply with some standard, for example, OpenAPI.

The specification must comprise not only formal descriptions but implicit agreements as well, such as the event generation order or unobvious side-effects of the API methods. Its important applied value is advisory consulting: developers will refer to it to clarify unobvious situations.

Importantly, formal specification *is not documentation* per se. The documentation is *the words you write* in the descriptions of each field and method. Without them, the specification might be used just for checking whether your namings are fine enough for developers to guess their meaning.

Today, the method nomenclature descriptions are frequently additionally exposed as ready-to-use request collections or code fragments for Postman or analogous tools.

2. Code Samples

From the above-mentioned, it's obvious that code samples are a crucial tool to acquire and retain new API users. It's extremely important to choose examples that help newcomers to start working with the API. Improper example selection will greatly reduce the quality of your documentation. While assembling the set of code samples, it is important to follow the rules:

- Examples must cover actual API use cases: the better you guess the most frequent developers' needs, the more friendly and straightforward your API will look to them.
- Examples must be laconic and atomic: mixing a bunch of tricks in one code sample dramatically reduces its readability and applicability.
- Examples must be close to real-world app code. The author of this book once faced a situation when a synthetic code sample, totally meaningless in the real world, was mindlessly replicated by developers in abundance.

Ideally, examples should be linked to all other kinds of documentation, e.g., the reference might contain code samples relevant to the entity being described.

3. Sandboxes

Code samples will be much more useful to developers if they are “live,” i.e., provided as editable pieces of code that might be modified and executed. In the case of library APIs, the online sandbox featuring a selection of code samples will suffice, and existing online services like JSFiddle might be used. With other types of APIs, developing sandboxes might be much more complicated:

- If the API provides access to some data, then the sandbox must allow working with a real dataset, either a developer's own one (e.g., bound to their user profile) or some test data.
- If the API provides an interface, visual or programmatic, to some non-online environment, like UI libs for mobile devices do, then the sandbox itself must be an emulator or a simulator of that environment, in a form of an online service or a standalone app.

4. Tutorial

A tutorial is a specifically written human-readable text describing some concepts of working with the API. A tutorial is something in-between a reference and examples. It implies some learning, more thorough than copy-pasting code samples, but requires less time investment than reading the whole reference.

A tutorial is a sort of “book” that you write to explain to the reader how to work with your API. So, a proper tutorial must follow book-writing patterns, i.e., explain the concepts coherently and consecutively chapter after chapter. Also, a tutorial must provide:

- General knowledge of the subject area; for example, a tutorial for cartographical APIs must explain trivia regarding geographical coordinates and working with them
- Proper API usage scenarios, i.e., the “happy paths”
- Proper reactions to program errors that could happen
- Detailed studies on advanced API functionality (with detailed examples).

Usually, a tutorial comprises a common section (basic terms and concepts, notation keys) and a set of sections regarding each functional domain exposed via the API. Frequently, tutorials contain a “Quick Start” (“Hello, world!”) section: the smallest possible code sample that would allow developers to build a small app atop the API. “Quick Starts” aim to cover two needs:

- To provide a default entry-point, the easiest to understand and the most useful text for those who heard about your API for the first time
- To engage developers, to make them touch the service by a mean of a real-world example.

Also, “Quick starts” are a good indicator of how exactly well did you do your homework of identifying the most important use cases and providing helper methods. If your Quick Start comprises more than ten lines of code, you have definitely done something wrong.

5. Frequently Asked Questions and Knowledge Bases

After you publish the API and start supporting users (see the previous chapter) you will also accumulate some knowledge of what questions are asked most frequently. If you can't easily integrate answers into the documentation, it's useful to compile a specific “Frequently Asked Questions” (aka FAQ) article. A FAQ article must meet the following criteria:

- Address the real questions (you might frequently find FAQs that were reflecting not users' needs, but the API owner's desire to repeat some important information once more; it's useless, or worse — annoying; perfect examples of this anti-pattern realization might be found on any bank or air company website)
- Both questions and answers must be formulated clearly and succinctly. It's acceptable (and even desirable) to provide links to corresponding reference and tutorial articles, but the answer itself can't be longer than a couple of paragraphs.

Also, FAQs are a convenient place to explicitly highlight the advantages of the API. In a question-answer form, you might demonstrably show how your API solves complex problems easily and handsomely. (Or at least, *solves them*, unlike the competitors' products.)

If technical support conversations are public, it makes sense to store all the questions and answers as a separate service to form a knowledge base, i.e., a set of “real-life” questions and answers.

6. Offline Documentation

Though we live in the online world, an offline version of the documentation (in a form of a generated doc file) still might be useful — first of all, as a snapshot of the API specification valid for a specific date.

Content Duplication Problems

A significant problem that harms documentation clarity is API versioning: articles describing the same entity across different API versions are usually quite similar. Organizing convenient searching capability over such datasets is a problem for internal and external search engines as well. To tackle this problem ensure that:

- The API version is highlighted on the documentation pages
- If a version of the current page exists for newer API versions, there is an explicit link to the actual version
- Docs for deprecated API versions are pessimized or even excluded from indexing.

If you're strictly maintaining backward compatibility, it is possible to create a single documentation for all API versions. To do so, each entity is to be marked with the API version it is supported from. However, there is an apparent problem with this approach: it's not that simple to get docs for a specific (outdated) API version (and, generally speaking, to understand which capabilities this API version provides). (Though the offline documentation we mentioned earlier will help.)

The problem becomes worse if you're supporting not only different API versions but also different environments / platforms / programming languages; for example, if your UI lib supports both iOS and Android. Then both documentation versions are equal, and it's impossible to pessimize one of them.

In this case, you need to choose one of the following strategies:

- If the documentation topic content is totally identical for every platform, i.e., only the code syntax differs, you will need to develop generalized documentation: each article provides code samples (and maybe some additional notes) for every supported platform on a single page.
- On the contrary, if the content differs significantly, as is in the iOS/Android case, we might suggest splitting the documentation sites (up to having separate domains for each platform): the good news is that developers almost always need one specific version, and they don't care about other platforms.

The Documentation Quality

The best documentation happens when you start viewing it as a product in the API product range, i.e., begin analyzing customer experience (with specialized tools), collect and process feedback, set KPIs and work on improving them.

Was This Article Helpful to You?

Yes / No

Chapter 61. Testing Environments

If the operations executed via the API imply consequences for end users or partners (cost money, in particular) you must provide a test version of the API. In this testing API, real-world actions either don't happen at all (for instance, orders are created but nobody serves them) or are simulated by cheaper means (let's say, instead of sending an SMS to a user, an email is sent to the developer's mailbox).

However, in many cases having a test version is not enough — like in our coffee-machine API example. If an order is created but not served, partners are not able to test the functionality of delivering the order or requesting a refund. To run the full cycle of testing, developers need the capability of pushing the order through stages, as this would happen in reality.

A direct solution to this problem is providing test versions for a full set of APIs and administrative interfaces. It means that developers will be able to run a second application in parallel — the one you're giving to coffee shops so they might get and serve orders (and if there is a delivery functionality, the third app as well: the courier's one) — and make all these actions that coffee shop staff normally does. Obviously, that's not an ideal solution, because of several reasons:

- Developers of end user applications will need to additionally learn how coffee shop and courier apps work, which has nothing to do with the task they're solving.
- You will need to invent and implement some matching algorithm: an order made through a test application must be assigned to a specific virtual courier; this actually means creating an isolated virtual “sandbox” (meaning — a full set of services) for each specific partner.
- Executing a full “happy path” of an order will take minutes, maybe tens of minutes, and will require making a multitude of actions in several different interfaces.

There are two main approaches to tackling these problems.

1. The Testing Environment API

The first option is providing a meta-API to the testing environment itself. Instead of running the coffee-shop app in a separate simulator, developers are provided with helper methods (like `simulateOrderPreparation`) or some visual interface that allows controlling the order execution pipeline with minimum effort.

Ideally, you should provide helper methods for any actions that are conducted by people in the production environment. It makes sense to ship this meta-API complete with ready-to-use scripts or request collections that show the correct API call orders for standard scenarios.

The disadvantage of this approach is that client developers still need to know how the "flip side" of the system works, though in simplified terms.

2. The Simulator of Pre-Defined Scenarios

The alternative to providing the testing environment API is simulating the working scenarios. In this case, the testing environment takes control over "underwater" parts of the system and "plays out" all external agents' actions. In our coffee example, that means that, after the order is submitted, the system will simulate all the preparation steps and then the delivery of the beverage to the customer.

The advantage of this approach is that it demonstrates vividly how the system works according to the API vendor design plans, e.g., in which sequence the events are generated, and which stages the order passes through. It also reduces the chance of making mistakes in testing scripts, as the API vendor guarantees the actions will be executed in the correct order with the right parameters.

The main disadvantage is the necessity to create a separate scenario for each unhappy path (effectively, for every possible error), and give developers the capability of denoting which scenario they want to run. (For example, like that: if there is a pre-agreed comment to the order,

the system will simulate a specific error, and developers will be able to write and debug the code that deals with the error.)

The Automation of Testing

Your final goal in implementing testing APIs, regardless of which option you choose, is allowing partners to automate the QA process for their products. The testing environment should be developed with this purpose in mind; for example, if an end user might be brought to a 3-D Secure page to pay for the order, the testing environment API must provide some way of simulating the successful (or not) passing of this step. Also, in both variants, it's possible (and desirable) to allow running the scenarios in a fast-forward manner that will allow making auto-testing much faster than manual testing.

Of course, not every partner will be able to employ this possibility (which also means that a “manual” way of testing usage scenarios must always be supported alongside the programmatical one) simply because not every business might afford to hire a QA automation engineer. Nevertheless, the ability to write such auto-tests is your API's huge competitive advantage from a technically advanced partner's point of view.

Chapter 62. Managing Expectations

Finally, the last aspect we would like to shed the light on is managing partners' expectations regarding the further development of the API. If we talk about consumer qualities, APIs differ little from other B2B software products: in both cases, you need to form some understanding of SLA conditions, available features, interface responsiveness and other characteristics that are important for clients. Still, APIs have their specificities

Versioning and Application Lifecycle

Ideally, the API once published should live eternally; but as we all are reasonable people, we do understand it's impossible in the real life. Even if we continue supporting older versions, they will still become outdated eventually, and partners will need to rewrite the code to use newer functionality.

The author of this book formulates the rule of issuing new major API versions like this: the period of time after which partners will need to rewrite the code should coincide with the application lifespan in the subject area (see "The Backward Compatibility Problem Statement" chapter). Apart from updating *major* versions, sooner or later you will face issues with accessing some outdated *minor* versions as well. As we mentioned in the "On the Waterline of the Iceberg" chapter, even fixing bugs might eventually lead to breaking some integrations, and that naturally leads us to the necessity of keeping older *minor* versions of the API until the partner resolves the problem.

In this aspect, integrating with large companies that have a dedicated software engineering department differs dramatically from providing a solution to individual amateur programmers: on one hand, the former are much more likely to find undocumented features and unfixed bugs in your code; on the other hand, because of the internal bureaucracy, fixing the related issues might easily take months, save not years. The common recommendation there is to maintain old minor API versions for a period of time long enough for the most dilatory partner to switch no the newest version.

Supporting Platforms

Another aspect crucial to interacting with large integrators is supporting a zoo of platforms (browsers, programming languages, protocols, operating systems) and their versions. As usual, big companies have their own policies on which platforms they support, and these policies might sometimes contradict common sense. (Let's say, it's rather a time to abandon TLS 1.2, but many integrators continue working through this protocol, or even the earlier ones.)

Formally speaking, ceasing support of a platform *is* a backward-incompatible change, and might lead to breaking some integration for some end users. So it's highly important to have clearly formulated policies regarding which platforms are supported based on which criteria. In the case of mass public APIs, that's usually simple (like, API vendor promises to support platforms that have more than N% penetration, or, even easier, just last M versions of a platform); in the case of commercial APIs, it's always a bargain based on the estimations, how much will non-supporting a specific platform would cost to a company. And of course, the outcome of the bargain must be stated in the contracts — what exactly you're promising to support during which period of time.

Moving Forward

Finally, apart from those specific issues, your customers must be caring about more general questions: could they trust you? Could they rely on your API evolving, absorbing modern trends, or will they eventually find the integration with your API on the scrapyard of history? Let's be honest: given all the uncertainties of the API product vision, we are very much interested in the answers as well. Even the Roman viaduct, though remaining backward-compatible for two thousand years, has been a very archaic and non-reliable way of solving customers' problems for quite a long time.

You might work with these customer expectations by publishing roadmaps. It's quite common that many companies avoid publicly announcing their concrete plans (for a reason, of course). Nevertheless, in the case of APIs, we strongly recommend providing the roadmaps, even if they are tentative and lack precise dates — *especially* if we talk about deprecating some functionality. Announcing these promises (given the company keeps them, of course) is a very important competitive advantage to every kind of consumer.

With this, we would like to conclude this book. We hope that the principles and the concepts we have outlined will help you in creating APIs that fit all the developers, businesses, and end users' needs, and in expanding them (while maintaining the backward compatibility) for the next two thousand years or so.