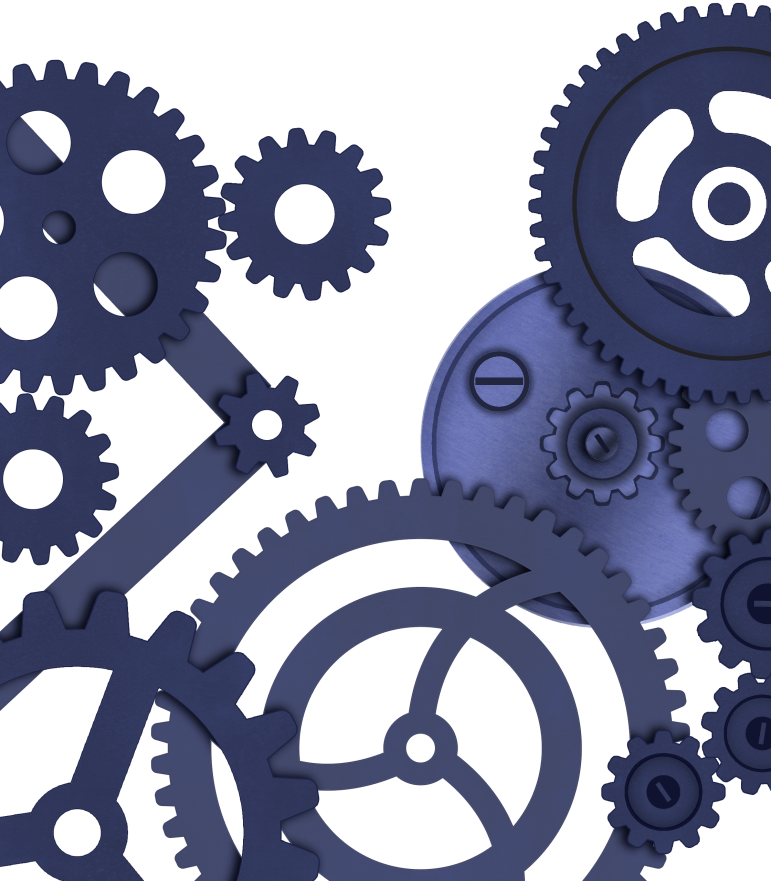
Sergey Konstantinov

The API



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

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

This book is written to share the expertise and describe the best practices in designing and developing APIs. In Section I, we'll discuss the API architecture as a concept: how to build the hierarchy properly, from high-level planning down to final interfaces. Section II is dedicated to expanding existing APIs in a backwards-compatible manner. Finally, in Section III we will talk about the API as a product.

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

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

The book you're holding in your hands comprises this Introduction and two sections: ‘The API Design’ and ‘The Backwards Compatibility’.

In Section I, we will discuss designing APIs as a concept: how to build the architecture properly, from high-level planning down to final interfaces.

Section II is dedicated to an API lifecycle: how interfaces evolve over time, and how to elaborate the product to match users' needs.

One more section is planned for the future: the ‘API as a Product’ section will be more about the un-engineering sides of the API, like API marketing, organizing customer support, and working with a community.

First, two sections are interesting to engineers mostly, while the third section is more relevant to both engineers and product managers. However, we insist that the third section is the most important for the API software developer. Since an API is a product for engineers, you cannot simply pronounce a non-engineering team responsible for product planning and support. Nobody but you understands more about your API's product features.

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. Encyclopedia tells us that ‘API’ is an acronym for ‘Application Program Interface’. This definition is fine, but useless. Much like ‘Man’ definition by Plato: Man stood 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, not ‘fine’ either. 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 API *means* 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 stuff must work correctly: parsing the URL according to the specification; DNS service; TLS handshake protocol; transmitting the data over HTTP protocol; HTML document parsing; CSS document parsing; correct HTML+CSS rendering.

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) works correctly. HTML document parsing is being performed according to hundreds of different specifications. Document rendering calls the underlying operating system API, or even directly graphical processor

API. And so on: down to modern CISC processor commands being implemented on top of the microcommands API.

In other words, hundreds or even thousands of different APIs must work correctly to make basic actions possible, like 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;
- backwards compatibility being broken not a single time in two thousand years.

What differs between a Roman aqueduct and a good API is that APIs presume a contract to be *programmable*. To connect two areas some *coding* is needed. The goal of this book is to help you in designing APIs which serve their purposes as solidly as a Roman aqueduct does.

An aqueduct also illustrates another problem of 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. From one side, you may provide access to the water to many more people through them, not spending your time plugging each individual house into your network. From the other side, you can't control the quality of suppliers' solutions, and you are to be blamed every time there is a water problem caused by their incompetence.

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

Chapter 3. API Quality Criteria

Before we start laying out the recommendations, we ought to specify what API we consider ‘fine’, and what's the profit of having a ‘fine’ API.

Let's discuss the second question first. Obviously, API ‘finesse’ is first of all defined through its capability to solve developers' problems. (One may reasonably say that solving developers' problems might not be the main purpose of offering the API of ours to developers. However, manipulating public opinion is out of this book's author's interest. Here we assume that APIs exist primarily to help developers in solving their problems, not for some other covertly declared purposes.)

So, how the API design might help the developers? Quite simple: a well-designed API must solve their problems in the most efficient and comprehensible manner. The distance from formulating the task to writing a working code must be as short as possible. Among other things, it means that:

- it must be totally obvious out of your API's structure how to solve a task; ideally, developers at first glance should be able to understand, what entities are meant to solve their problem;
- the API must be readable; ideally, developers write correct code after just looking at the method nomenclature, never bothering about details (especially API implementation details!); it is also

very important to mention, that not only problem solution (the ‘happy path’) should be obvious, but also possible errors and exceptions (the ‘unhappy path’);
- the API must be consistent; while developing new functionality (i.e. while using unknown API entities) developers may write new code similar to the code they already wrote using known API concepts, and this new code will work.

However static convenience and clarity of APIs are simple parts. After all, nobody seeks for making an API deliberately irrational and unreadable. When we are developing an API, we always start with clear basic concepts. Providing you've got 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 backwards compatibility 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 are the best practices) and in subjective ones too (what obviousness, readability, and consistency *really mean* to your API design).

The principles we are explaining below are specifically oriented to making APIs evolve smoothly over time, not being turned into a pile of mixed inconsistent interfaces. It is crucial to understand that this approach isn't free: a necessity to bear in mind all possible extension variants and to preserve essential growth points means interface redundancy and possibly excessing 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 an overengineering.**

Chapter 4. Backwards Compatibility

Backwards compatibility is a *temporal* characteristic of your API. An obligation to maintain backwards compatibility is the crucial point where API development differs from software development in general.

Of course, backwards compatibility isn't an absolute. In some subject areas shipping new backwards-incompatible API versions is a routine. Nevertheless, every time you deploy a new backwards-incompatible API version, the developers need to make some non-zero effort to adapt their code to the new API version. In this sense, releasing new API versions puts a sort of a ‘tax’ on customers. They must spend quite real money just to make sure their product continues working.

Large companies, which occupy firm market positions, could afford to imply such taxation. 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 imply hidden taxes on your customers. If you're able to avoid breaking backwards compatibility — never break it.

Of course, maintaining old API versions is a sort of a tax either. Technology changes, and you cannot foresee everything, regardless of how nice 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 5. On Versioning

Here and throughout we firmly stick to semver principles of versioning.

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

Sentences ‘major API version’ and ‘new API version, containing backwards-incompatible changes’ are therefore to be considered equivalent ones.

It is usually (though not necessary) agreed that the last stable API version 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 of defining the desired version (for example, `^1.2.3` reads like ‘get the last stable API version that is backwards-compatible to the `1.2.3` version’) or additional shortcuts (for example, `1.2-beta` to refer to the last beta-version 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 ‘Backwards Compatibility Problem Statement’ chapter.

Chapter 6. Terms and Notation Keys

Software development is being characterized, among other things, by the existence of many different engineering paradigms, whose adepts sometimes are quite aggressive towards other paradigms' adepts. 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 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 a 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_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 →) server returns a `404 Not Founds` 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 a single `error_message` field; field value absence means that field contains exactly what you expect it should contain — some error message in this case;
- if some token is too long to fit 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 to ‘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 be using 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 since types are omitted. We assume such imperative structures are readable enough to skip detailed grammar explanations.

SECTION I. THE API DESIGN

Chapter 7. 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 entity nomenclature. In fact, moving this way will eventually conclude with a ready-to-use API — 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 API design concepts using a hypothetical example of an API allowing for 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 8. Defining an Application Field

The key question you should ask yourself looks like that: what problem do we solve? It should be asked four times, each time putting an emphasis on another 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 a coffee? Why ordering a coffee via ‘human-to-human’ or ‘human-to-machine’ interfaces is 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-machines utilization in 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 a resource to solve a 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 we're going to quantify the impact our API makes?

In general, there are no simple answers to those questions. Ideally, you should give answers having all the relevant metrics measured: how much time is wasted exactly, and what numbers we're going to achieve providing we have such coffee machines density? Let us also stress that in a real-life 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, allowing for brewing coffee to 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 the operators which provide APIs, and there are the operators which 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 in more detail in Section III.

We should also note that you should try making an API when, and only when, your answer is "because that's our area of expertise" to question 3. Developing APIs is a sort of meta-engineering: you're writing some software to allow other companies 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 near 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 auditory. Something like the flights market looks like: there are air companies, which actually transport passengers; and there are trip planning services where users are choosing between trip variants the system generates for them. We're aggregating hardware access to allow app vendors for ordering freshly brewed coffee.

What and How

After finishing all these theoretical exercises, we should proceed right 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 their users may order a cup of coffee in the most efficient and convenient manner;
- abstracting access to coffee machines ‘hardware’ and delivering methods to select a beverage kind and some location to brew — and to make an order.

Chapter 9. Separating Abstraction Levels

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

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

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

Back 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 receive payments for this.
2. Each cup of coffee is prepared according to some `recipe`, which implies the presence of different ingredients and sequences of preparation steps.
3. Each beverage is being prepared on some physical `coffee machine`, occupying some position in space.

Every level presents a developer-facing ‘facet’ in our API. While elaborating on the hierarchy of abstractions, we are first of all trying to reduce the interconnectivity of different entities. That would help us to reach several goals.

1. Simplifying developers' work and the learning curve. At each moment of time, a developer is operating only those entities which are necessary for the task they're solving right now. And conversely, badly designed isolation leads to the situation when developers have to keep in mind lots of concepts mostly unrelated to the task being solved.

2. Preserving backwards 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 say we have the following interface:

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

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

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

Let's consider the question: how exactly developers should 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 beverage to the order state.

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 abovementioned 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 a choice to a customer, how many milliliters of lungo they desire exactly, then we have to perform one of the following tricks.

Variant 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 the mass production of recipes, only different in volume, or introduce some recipe ‘inheritance’ to be able to specify the ‘base’ recipe and just redefine the volume.

Variant II: we modify an interface, pronouncing volumes stated in recipes being 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 `GET /v1/recipes` context 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, we now have to live with that.

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 nomenclature must directly reflect API usage scenarios; low-level entities reflect the decomposition of scenarios into smaller parts.

2. From user subject field terms to ‘raw’ data subject field terms — in our case from high-level terms like ‘order’, ‘recipe’, ‘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 ‘"Chamomile" café chain’ to the raw byte data stream from ‘Good Morning’ coffee machine sensors.

The more is the distance between programmable contexts our API connects, the deeper is 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:

- from one side, an ‘order’ should not store the data regarding coffee machine sensors;
- on the other side, 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 to another. For example, introduce a `task` entity like that:

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

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 we really should determine beverage readiness? 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 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 prebuilt programs:

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

```
// Starts an execution
// of a specified program
// and returns execution status
POST /execute
{
  "program": 1,
  "volume": "200ml"
}
→
{
  // Unique identifier of the execution
  "execution_id": "01-01",
  // Identifier of the program
  "program": 1,
  // Beverage volume requested
  "volume": "200ml"
}
```

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

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

NB. Just in case: 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 would really get something like that from vendors, and this API is actually quite a sane one.

- Coffee machines with built-in functions:

```
// Returns a list of functions available
GET /functions
→
{
  "functions": [
    {
      // Operation type:
      // * set_cup
      // * grind_coffee
      // * pour_water
      // * discard_cup
      "type": "set_cup",
      // Arguments available
      // to each 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 sensors' state
GET /sensors
→
{
  "sensors": [
    {
      // Values allowed:
      // * cup_volume
      // * ground_coffee_volume
      // * cup_filled_volume
      "type": "cup_volume",
      "value": "200ml"
    },
    …
  ]
}
```

NB. The example is intentionally factitious to model a 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 second-kind machines will be more intricate.

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 to learn what problems they get 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 to expect it's 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; program identifier is of no use to developers since there is a ‘recipe’ concept;
- absence of explicit ‘ready’ status.

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

So we need to introduce two abstraction levels.

1. Execution control level, which provides the 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 error) 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 a correct `program` identifier, the handler runs a 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 interfaces more specific, 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. `run` method handler is capable of extracting all the program metadata and performing one of two actions:

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

Out of general concerns 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’ (e.g. 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:

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

NB: while 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 behalf, and forms a stateless request which contains all the necessary data (the 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, selecting one of 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.

Get back to our example. How retrieving order status would 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 (for example, checks user authorization), finds `program_run_id` identifier and performs a call to the `runs/{program_run_id}` endpoint;

- the `runs` endpoint in its turn completes operations corresponding to its level (for example, 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 a 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 a 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 ‘call chain’ wording shouldn't be treated literally. Each abstraction level might be organized differently in a technical sense:

- there might be explicit proxying of calls down the hierarchy;
- there might be a cache at each level, being updated upon receiving a callback call or an event. In particular, a low-level runtime execution cycle obviously must be independent of upper levels,

renew its state in the background, and not wait for an explicit call.

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

Let's now look at how the order cancel operation flows through our abstraction levels. In this case, the call chain will look like that:

- 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;
  - solves money issues, i.e. 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:
  - either calls the `POST /execution/cancel` method of a physical coffee machine API;
  - or invokes the `POST /v1/runtimes/{id}/terminate` method;

- in a 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` one requires more advanced abstraction levels juggling skills compared to non-modifying calls like the `GET /status` one. There are two important moments:

1. At each abstraction level the idea of ‘order canceling’ is reformulated:

    - at the `orders` level, this action in fact splits into several ‘cancels’ of other levels: you need to cancel money holding and to cancel an 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, the processing continues until the cup is discarded, and then the machine is to be unlocked (e.g. new

runtimes creation allowed). It's a task to the execution control level to couple those two states, outer (the order is canceled) and inner (the execution continues).

It might look that forcing the abstraction levels isolation is redundant and makes interfaces more complicated. In fact, it is: it's very important 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 separate this gap into levels *explicitly*. The more implicitly abstraction levels are separated (or worse — blended into each other), the more complicated is your API's learning curve, and the worse is the code that uses it.

The Data Flow

One useful exercise allowing us to examine the entire abstraction hierarchy is excluding all the particulars and constructing (on a paper or just in your head) a data flow chart: 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 to 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, i.e. 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 shed’ or a ‘cup is being set’ notion. It's our software that treats incoming sensors 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 to be executed.
   It is important to note that we don't calculate new variables out from sensors data: we need to create a new dataset first, a context, an ‘execution program’ comprising a sequence of steps and conditions, and to 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 the low-level (and in fact useless to end users) context terms into the 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 statuses 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 into the ‘bad’ decision (forcing developers to determine actual order status on their own), being discussed at the beginning of this chapter, we could notice a data flow collision there:

- from one side, in the order context ‘leaked’ physical data (beverage volume prepared) is injected, therefore stirring abstraction levels irreversibly;
- from the other side, 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, from one side, helps us to separate abstraction levels properly, and, from the other side, to check if our theoretical structures work as intended.

Chapter 10. Isolating Responsibility Areas

Based on the previous chapter, we understand that the abstraction hierarchy in our hypothetical project would look like that:

- the user level (those entities users directly interact with and which are formulated in terms, understandable by users: 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 in 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 the first connected context terms (‘a lungo ordered by a user’) to the second connect context terms (‘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 some coffee beverage type, e.g. its customer properties. A `recipe` is an immutable entity for us, which means we could only read it.
    - 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 could only be read.
    - The `programs/matcher` entity is capable of coupling a `recipe` and a `program`, which in fact means ‘to retrieve a dataset needed to prepare a specific recipe on a specific coffee machine’.
    - A `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. `runtime` might 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` will operate high-level data (`recipe` and `coffee-machine`), enhancing them with its subject area terms (`program_run_id` for instance). This is totally fine: 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 into the developer's shoes and try writing code. Our task is to look at the entity nomenclature and make some estimates regarding their future usage.

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

Obviously, the first step is offering a choice to a 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. A developer has 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 a code that traverses all this data.

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

```
// 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 difficulties 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 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 tastes;
- 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 a 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. This decision, however, looks less semantically viable: coupling in one interface different modes of listing entities, by relevance and by order, is usually a bad idea because these two types of rankings imply different usage features and 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 to users, with specifying the coffee machine being just a part of an offer.

Coming back to the code developers are writing, 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 they exist is to generalize known API usage scenarios and facilitate implementing them. By ‘facilitating’ we mean not only reducing wordiness (getting rid of ‘boilerplates’) but also helping developers to avoid common problems and mistakes.

For instance, let's consider the order price question. Our search function returns some ‘offers’ with prices. But ‘price’ is volatile; coffee could cost less during ‘happy hours’, for example. Developers could make a mistake thrice while implementing this functionality:

- cache search results on a client device for too long (as a result, the price will always be nonactual);
- contrary to previous, call search method excessively just to actualize prices, thus overloading the network and the API servers;
- create an order with an invalid price (therefore deceiving a user, displaying one sum, and debiting another).

To solve the third problem we could demand including the displayed price in the order creation request, and return an error if it differs from the actual one. (In fact, any API working with money *shall* do so.) But it isn't helping with the first two problems and makes the user experience degrade. 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 to grasp the concept of getting the relevant price, but also solving a UX task of telling users about ‘happy hours’.

As an alternative, we could split endpoints: one for searching, another one for obtaining offers. This second endpoint would only be needed to actualize prices in the specified places.

Error Handling

And one more step towards making developers' life easier: how an ‘invalid price’ error would look like?

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

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

The main rule of error interfaces in the APIs is: an error response must help a client to understand *what to do with this error*. All other stuff is unimportant: if the error response was machine-readable, there would be no need for the user-readable message.

An error response content must address the following questions:

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

2. If the error is caused by a server, is there any sense to repeat the request? If yes, then when?
3. If the error is caused by a client, is it resolvable, or not?

   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 push 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's the kind of problem? Obviously, a client couldn't resolve a problem it's unaware of. For every resolvable problem, some *code* must be written (reobtaining the offer in our case), so a list of error descriptions must exist.
5. If the same kind of errors arise because of different parameters being invalid, then 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 goes wrong.↵
     Try restarting the app."
  "details": {
    // What's wrong exactly?
    // Which validity checks failed?
    "checks_failed": [
      "offer_lifetime"
    ]
  }
}
```

After getting this error, a client is to check the error's kind (‘some problem with offer’), check the specific error reason (‘order lifetime expired’), and send an offer retrieving request again. If the `checks_failed` field indicated another 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 were no error handlers for this specific reason, a client would 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 (since the client couldn't react usefully to unknown errors), but it doesn't mean that providing extended error data is excessive. A developer will

read it when fixing the error in the code. Also, check paragraphs 12 and 13 in the next chapter.

Decomposing Interfaces. The '7±2' Rule

Out of our own API development experience, we can tell without any doubt that the greatest final interface design mistake (and the greatest developers' pain accordingly) is 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 could 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 the '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 developers are never to operate more than 10 entities at a time.

Let's take a look at a simple example: what the coffee machine search function returns. To ensure an adequate UX of the app, quite bulky datasets are required.

```
{
  "results": [{
    "coffee_machine_id",
    "coffee_machine_type":
      "drip_coffee_maker",
    "coffee_machine_brand",
    "place_name": "The Chamomile",
    // Coordinates of a place
    "place_location_latitude",
    "place_location_longitude",
    "place_open_now",
    "working_hours",
    // Walking route parameters
    "walking_distance",
    "walking_time",
    // How to find the place
    "place_location_tip",
    "offers": [{
      "recipe": "lungo",
      "recipe_name":
        "Our brand new Lungo®™",
      "recipe_description",
      "volume": "800ml",
      "offer_id",
      "offer_valid_until",
      "localized_price":
        "Just $19 for a large coffee cup",
      "price": "19.00",
      "currency_code": "USD",
      "estimated_waiting_time": "20s"
    }, …]
  }, …]
}
```

This approach is quite normal, alas; could be found in almost every API. As we see, the number of entities' fields exceeds recommended 7, and even 9. Fields are being mixed into one single list, often with similar prefixes.

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

- data regarding a place where the coffee machine is located;
- properties of the coffee machine itself;
- route data;
- recipe data;
- recipe options specific to the particular place;
- offer data;
- pricing data.

Let's try to group it 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"
      },
      // Recipe specific options
      "options":
        { "volume" },
      // Offer metadata
      "offer":
        { "id", "valid_until" },
      // Pricing
      "pricing": {
        "currency_code",
        "price",
        "localized_price"
      },
      "estimated_waiting_time"
    }, …]
```

```
    }, …]
  }
```

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

It is important to say that readability is achieved not only by mere 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 but degrades.

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

Chapter 11. 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 are to describe the objects, fields, methods, and functions nomenclature in detail. In this chapter, we're giving purely practical advice on making APIs usable and understandable.

An important assertion at number 0:

0. Rules must not be applied unthinkingly

Rules are just simply formulated generalizations from one's experience. They are not to be applied unconditionally, and they don't make thinking redundant. Every rule has a rational reason to exist. If your situation doesn't justify following the rule — then you shouldn't do it.

For example, demanding a specification be consistent exists to help developers spare time on reading docs. If you *need* developers to read some entity's doc, it is totally rational to make its signature deliberately inconsistent.

This idea applies to every concept listed below. If you get an unusable, bulky, unobvious API because you follow the rules, it's a motive to revise the rules (or the API).

It is important to understand that you always can introduce concepts of your own. For example, some frameworks willfully reject paired `set_entity` / `get_entity` methods in a favor of a single `entity()` method, with an optional argument. The crucial part is being systematic in applying the concept. If it's rendered into life, you must apply it to every single API method, or at the very least elaborate a naming rule to discern such polymorphic methods from regular ones.

Ensuring readability and consistency

The most important task for the API vendor is to make code written by third-party developers atop of the API easily readable and maintainable. Remember that the law of large numbers works against you: if some concept or a signature might be treated wrong, they will be inevitably treated wrong by a number of partners, and this number will be increasing with the API popularity growth.

1. Explicit is always better than implicit

Entity name must explicitly tell what it does and what side effects to expect while using it.

Bad:

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

It's unobvious that a state field might be set, 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 got charged for computational resource usage. Even more so, default values must not be set in a manner leading 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 to be absolutely transparent about what the function does, what arguments it takes, and what's the result. While reading a code working with your API, it must be easy to understand what it does without reading docs.

Two important implications:

1.1. If the operation is modifying, it must be obvious from the signature. In particular, there might be no modifying operations using the `GET` 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.

2. Specify which standards are used

Regretfully, humanity is unable to agree on the most trivial things, like which day starts the week, to say nothing about more sophisticated standards.

So *always* specify exactly which standard is applied. Exceptions are possible if you're 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 tons of date formatting standards; you can't even tell which number means the day number and which number means 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 saying that in some areas the situation with standards is so spoiled that, whatever you do, someone got upset. A ‘classical’ example is geographical coordinates order (latitude-longitude vs longitude-latitude). Alas, the only working method of fighting frustration there is the ‘Serenity Notepad’ to be discussed in Section II.

3. Entities must have concrete names

Avoid single amoeba-like words, such as ‘get’, ‘apply’, ‘make’, etc.

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

Better: `user.get_id()`.

4. Don't spare the letters

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

Bad: `order.time()` — unclear, what time is actually returned: order creation time, order preparation time, order waiting time?...

Better: `order.get_estimated_delivery_time()`

Bad:

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

Possibly, an author of this API thought that the `pbrk` abbreviature would mean something to readers; clearly mistaken. Also, it's hard to tell from the signature which string (`str1` or `str2`) stands for a character set.

Better:

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

— though it's highly disputable whether this function should exist at all; a feature-rich search function would be much more convenient. Also, shortening a `string` to an `str` bears no practical sense, regretfully being a routine in many subject areas.

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

5. Naming implies typing

Field named `recipe` must be of a `Recipe` type. Field named `recipe_id` must contain a recipe identifier that we could find within the `Recipe` entity.

Same for primitive types. Arrays must be named in a plural form or as collective nouns, i.e. `objects`, `children`. If that's impossible, better add a prefix or a postfix to avoid doubt.

Bad: `GET /news` — 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 some qualitative state, i.e. `is_ready`, `open_now`.

Bad: `"task.status": true`
— statuses are not explicitly binary; also such API isn't extendable.

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

Specific platforms imply specific additions to this rule with regard to the first-class citizen types they provide. For example, JSON doesn't have a `Date` object type, so the dates are to be 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, better add an extra prefix or postfix to avoid misunderstanding.

Bad:

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

Word ‘function’ is many-valued. It could mean built-in functions, but also ‘a piece of code’, or a state (machine is functioning).

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

6. Matching entities must have matching names and behave alike

Bad: `begin_transition` / `stop_transition` — `begin` and `stop` terms don't match; developers will have to dig into the docs.

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:

- inconsistent underscore using;
- functionally close methods have different `needle`/`haystack` argument ordering;
- the first function finds the first occurrence while the second one finds them all, and there is no way to deduce that fact out of the function signatures.

We're leaving the exercise of making these signatures better to the reader.

7. Avoid double negations

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

Better: `"prohibit_calling": true` or `"avoid_calling": true`
— it's easier to read, though you shouldn't deceive yourself. Avoid semantical double negations, even if you've found a ‘negative’ word without a ‘negative’ prefix.

Also worth mentioning is that making mistakes in the de Morgan's laws usage is even simpler. For example, if you have two flags:

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

'Coffee might be prepared' condition 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 flag `!beans_absence && !cup_absence` which is equivalent to `!(beans_absence || cup_absence)` conditions, and in this transition, people tend to make mistakes. Avoiding double negations helps little, and regretfully only general advice could be given: avoid the situations when developers have to evaluate such flags.

8. Avoid implicit type conversion

This advice is opposite to 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 developers should discern explicit intention to abolish the option (`false`) from knowing not it exists (the field isn't set). They 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, which will effectively treat the field in an opposite manner. The same could happen if some special values (i.e. `null` or `-1`) to denote value absence are used.

NB: this observation is not valid if both the platform and the protocol unambiguously support special tokens to reset a field to its default value with zero abstraction overhead. However, full and consistent support of this functionality rarely sees implementation. Arguably, the only example of such an API among those being popular nowadays is SQL: the language has the `NULL` concept, and default field values functionality, and the support for operations like `UPDATE … SET field = DEFAULT` (in most dialects). Though working with the protocol is still complicated (for example, in many dialects there is no simple method of getting back those values reset by an `UPDATE … DEFAULT` query), SQL features working with defaults conveniently enough to use this functionality as is.

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 specially treated value absence 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 unobvious, and developers will have to read the docs. That's the way.

9. No results is a result

If a server processed a request correctly and no exceptional situation occurred — there must be no error. Regretfully, an antipattern is widespread — of throwing errors when zero results are found.

Bad

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

`4xx` statuses imply that a client made a mistake. But no mistakes were made by either a customer or a developer: a client cannot know whether the lungo is served in this location beforehand.

Better:

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

This rule might be reduced to: 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, if an empty array is acceptable semantically; an empty array of coordinates is a mistake for sure.)

10. Errors must be informative

While writing the code developers face problems, many of them quite trivial, like invalid parameter types or some boundary violations. The more convenient are the error responses your API return, the less is the amount of time developers waste struggling with it, and the more comfortable is working with the API.

Bad:

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

— of course, the mistakes (typo in the `"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 spare developers' time.

11. Maintain a proper error sequence

First, always return unresolvable errors before the resolvable ones:

```
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?

Second, maintain such a sequence of unresolvable errors which leads to a minimal amount of customers' and developers' irritation. In particular, this means returning the most significant errors first, solving which requires more effort.

Bad:

```
POST /v1/orders
{
  "items": [{
    "item_id": "123",
    "price": "0.10"
  }]
}
→
409 Conflict
{
  "reason": "price_changed",
  "details": [{
    "item_id": "123",
    "actual_price": "0.20"
  }]
}
// Request repeats
// with an actual price
POST /v1/orders
{
  "items": [{
    "item_id": "123",
    "price": "0.20"
  }]
}
→
409 Conflict
{
  "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.

Third, draw a chart: which error resolution might lead to the emergence of another one. Otherwise, you might eventually return the same error several times, or worse, make a cycle of errors.

```
// Create an order
// with a 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 a free delivery
POST /v1/orders
{
  "items": 3,
  "item_price": "3000.00"
  "currency_code": "MNT",
  "delivery_fee": "0.00",
  "total": "9000.00"
}
→ 409 Conflict
// Error: 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 over, 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.

Developing machine-readable interfaces

In pursuit of the API clarity for humans, we frequently forget that it's not developers themselves who interact with the endpoints, but the code they've written. Many concepts that work well with user interfaces, are badly suited for the program ones: specifically, developers can't make decisions based on textual information, and they can't ‘refresh’ the state in case of some confusing situation.

12. The system state must be observable by clients

Sometimes, program systems provide interfaces that do not expose to the clients all the data on what is now being executed on the user's behalf, specifically — which operations are running and what their statuses are.

Bad:

```
// Creates an order and returns its id
POST /v1/orders
{ … }
→
{ "order_id" }
```

```
// Returns an order by its id
GET /v1/orders/{id}
// The order isn't confirmed
// and awaits checking
→ 404 Not Found
```

— though the operation looks to be executed successfully, the client must store the order id and recurrently check the `GET /v1/orders/{id}` state. This pattern is bad per se, but gets even worse when we consider two cases:

- clients might lose the id, if system failure happened in between sending the request and getting the response, or if app data storage was damaged or cleansed;
- customers can't use another device; in fact, the knowledge of orders being created is bound to a specific user agent.

In both cases, customers might consider order creating failed, and make a duplicate order, with all the consequences to be blamed on you.

Better:

```
// Creates an order and returns it
POST /v1/orders
{ <order parameters> }
→
{
  "order_id",
  // The order is created in explicit
  // «checking» status
  "status": "checking",
  …
}
```

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

```
// Returns all customer's orders
// in all statuses
GET /v1/users/{id}/orders
```

This rule is applicable to errors as well, especially client ones. If the error might be corrected, the related data must be machine-readable.

Bad: `{ "error": "email malformed" }` — the only thing developers might do with this error is to show the message to the end user.

Better:

```
{
  // Machine-readable status
  "status": "validation_failed",
  // An array; if there are several
  // errors, the user might correct
  // them all at once
  "failed_checks": [
     {
       "field: "email",
       "error_type": "malformed",
       // Localized
       // human-readable message
       "message": "email malformed"
     }
  ]
}
```

13. Specify lifespans of resources and caching policies

In modern systems, clients usually have their own state and almost universally cache results of requests — no matter, session-wise or long-term, every entity has some period of autonomous existence. So it's highly desirable to make 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 stress that we understand ‘cache’ in the extended sense: which variation 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 price in cafes
// closest to the specified location
GET /price?recipe=lungo↵
  &longitude={longitude}↵
  &latitude={latitude}
→
{ "currency_code", "price" }
```

Two questions arise:

- until when the price is valid?
- in what vicinity of the location the price is valid?

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

```
// Returns an offer: for what money sum
// our service commits to make a lungo
GET /price?recipe=lungo↵
  &longitude={longitude}↵
  &latitude={latitude}
→
{
  "offer": {
    "id",
    "currency_code",
    "price",
    "conditions": {
      // Until when the price is valid
      "valid_until",
      // What vicinity
      // the price is valid within
      // * city
      // * geographical object
      // * …
      "valid_within"
    }
  }
}
```

14. Pagination, filtration, and cursors

Any endpoints returning data collections must be paginated. No exclusions exist.

Any paginated endpoint must provide an interface to iterate over all the data.

Bad:

```
// Returns a limited number of records
// sorted by creation date
// starting with a record with an index
// equals to `offset`
GET /v1/records?limit=10&offset=100
```

At the first glance, this is the most standard way of organizing the pagination in APIs. But let's ask ourselves some questions.

1. How clients could learn about new records being added at the beginning of the list? Obviously, a client could only retry the initial request (`offset=0`) and compare identifiers to those it already knows. But what if the number of new records exceeds the `limit`? Imagine the situation:
    - the client process records sequentially;
    - some problem occurred, and a batch of new records awaits processing;
    - the client requests new records (`offset=0`) but can't find any known records on the first page;
    - the client continues iterating over records page by page until it finds the last known identifier; all this time the order processing is idle;
    - the client might never start processing, being preoccupied with chaotic page requests to restore records sequence.

2. What happens if some record is deleted from the head of the list?
   Easy: the client will miss one record and will never learn this.
3. What cache parameters to set for this endpoint?
   None could be set: repeating the request with the same `limit` and `offset` parameters each time produces a new record set.

Better: in such unidirectional lists the pagination must use the key that implies the order. Like this:

```
// Returns a limited number of records
// sorted by creation date
// starting with a record with an identifier
// following the specified one
GET /v1/records↵
  ?older_than={record_id}&limit=10
// Returns a limited number of records
// sorted by creation date
// starting with a record with an identifier
// preceding the specified one
GET /v1/records↵
  ?newer_than={record_id}&limit=10
```

With the pagination organized like that, clients never bother about records being added or removed in the processed part of the list: they continue to iterate over the records, either getting new ones (using `newer_than`) or older ones (using `older_than`). If there is no record removal

operation, clients may easily cache responses — the URL will always return the same record set.

Another way to organize such lists is by returning a `cursor` to be used instead of the `record_id`, making interfaces more versatile.

```
// Initial data request
POST /v1/records/list
{
  // Some additional filtering options
  "filter": {
    "category": "some_category",
    "created_date": {
      "older_than": "2020-12-07"
    }
  }
}
→
{ "cursor" }
```

```
// Follow-up requests
GET /v1/records?cursor=<cursor value>
{ "records", "cursor" }
```

One advantage of this approach is the possibility to keep initial request parameters (i.e. the `filter` in our example) embedded into the cursor itself, thus not copying them in follow-up requests. It might be especially actual if the

initial request prepares the full dataset, for example, moving it from the ‘cold’ storage to a ‘hot’ one (then the `cursor` might simply contain the encoded dataset id and the offset).

There are several approaches to implementing cursors (for example, making a single endpoint for initial and follow-up requests and returning the first data portion in the first response). As usual, the crucial part is maintaining consistency across all such endpoints.

NB: some sources discourage this approach because in this case user can't see a list of all pages and can't choose an arbitrary one. We should note here that:

- such a case (pages list and page selection) exists if we deal with user interfaces; we could hardly imagine a *program* interface that needs to provide access to random data pages;
- if we still talk about an API to some application, which has a ‘paging’ user control, then a proper approach would be to prepare ‘paging’ data on the server side, including generating links to pages;
- cursor-based solutions don't prohibit using the `offset`/`limit` parameters; nothing could prevent us from creating a dual interface, which might serve both `GET /items?cursor=…` and `GET /items?offset=…&limit=…` requests;

- finally, if there is a necessity to provide access to arbitrary pages in the user interface, we should ask ourselves a question, which problem is being solved that way; probably, users use this functionality to find something: a specific element on the list, or the position they ended while working with the list last time; probably, we should provide more convenient controls to solve those tasks than accessing data pages by their indexes.

Bad:

```
// Returns a limited number of records
// sorted by a specified field
// in a specified order
// starting with a record with an index
// equals to `offset`
GET /records?sort_by=date_modified↵
  &sort_order=desc&limit=10&offset=100
```

Sorting by the date of modification usually means that data might be modified. In other words, some records might change after the first data chunk is returned, but before the next chunk is requested. Modified records will simply disappear from the listing because of moving to the first page. Clients will never get those records that were changed during the iteration process, even if the cursor-based scheme is implemented, and they never learn the sheer fact of such an omission. Also, this particular

interface isn't extendable as there is no way to add sorting by two or more fields.

Better: there is no general solution to this problem in this formulation. Listing records by modification time will always be unpredictably volatile, so we have to change the approach itself; we have two options.

Option one: fix the records ordering at the moment we've got the initial request, e.g. our server produces the entire list and stores it in the immutable form:

```
// Creates a view based on the parameters passed
POST /v1/record-views
{
  sort_by: [{
    "field": "date_modified",
    "order": "desc"
  }]
}
→
{ "id", "cursor" }
```

```
// Returns a portion of the view
GET /v1/record-views/{id}↵
  ?cursor={cursor}
```

Since the produced view is immutable, access to it might be organized in any form, including a limit-offset scheme, cursors, `Range` header, etc. However, there is a downside: records modified after the view was generated will be misplaced or outdated.

Option two: guarantee a strict records order, for example, by introducing a concept of record change events:

```
// `cursor` is optional
GET /v1/records/modified/list↵
  ?[cursor={cursor}]
→
{
  "modified": [
    { "date", "record_id" }
  ],
  "cursor"
}
```

This scheme's downsides are the necessity to create separate indexed event storage, and the multiplication of data items, since for a single record many events might exist.

Ensuring technical quality of APIs

Fine APIs must not only solve developers' and end users' problems but also ensure the quality of the solution, e.g. do not contain logical and technical mistakes (and do not provoke developers to make them), save computational resources, and in general implement the best practices applicable to the subject area.

15. Keep the precision of fractional numbers intact

If the protocol allows, fractional numbers with fixed precision (like 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 multiplicator) or strings.

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

16. 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're discussing client-server interaction in the first place, repeating requests in case of network failure isn't an exception, but a norm of life.

If the endpoint's idempotency can't be assured naturally, explicit idempotency parameters must be added, in a form of either 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>
```

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

An alternative:

```
// 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 since 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` being its idempotency key.

Also worth mentioning that adding idempotency tokens to naturally idempotent handlers isn't meaningless either, since it allows to distinguish two situations:

- a client didn't get the response because of some network issues, and is now repeating the request;
- a client made a mistake by posting conflicting requests.

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

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

The server retrieves the actual resource revision and finds it to be 124. How to respond correctly? `409 Conflict` might be returned, but then the client will be forced to understand the nature of the conflict and somehow resolve it, potentially confusing the user. It's also unwise to fragment the conflict-resolving algorithm, allowing each client to implement it independently.

The server may compare request bodies, assuming that identical `updates` values mean retrying, but this assumption might be dangerously wrong (for example if the resource is a counter of some kind, then repeating identical requests are routine).

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

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

— the server found out that the same token was used in creating revision 124, which means the client is retrying the request.

Or:

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

— the server found out that a different token was used in creating revision 124, which means an access conflict.

Furthermore, adding idempotency tokens not only resolves the issue but also makes advanced optimizations possible. If the server detects an access conflict, it could try to resolve it, ‘rebasing’ the update like modern version control

systems do, and return a `200 OK` instead of a `409 Conflict`. This logic dramatically improves user experience, being fully backwards compatible, and helps to avoid conflict-resolving code fragmentation.

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

- you can't really expect that clients generate truly random tokens — they may share the same seed or simply use weak algorithms or entropy sources; therefore you must put constraints on token checking: token must be unique to a specific user and resource, not globally;
- clients tend to misunderstand the concept and either generate new tokens each time they repeat the request (which deteriorates the UX, but otherwise healthy) or conversely use one token in several requests (not healthy at all and could lead to catastrophic disasters; another reason to implement the suggestion in the previous clause); writing detailed doc and/or client library is highly recommended.

17. Avoid non-atomic operations

There is a common problem with implementing the changes list approach: what to do if some changes were successfully applied, while others are not? The rule is simple: if you may ensure the atomicity (e.g. either apply all changes or none of them) — do it.

Bad:

```
// Returns a list of recipes
api.getRecipes();
→
{
  "recipes": [{
    "id": "lungo",
    "volume": "200ml"
  }, {
    "id": "latte",
    "volume": "300ml"
  }]
}
// Changes recipes' parameters
api.updateRecipes({
  "changes": [{
    "id": "lungo",
    "volume": "300ml"
  }, {
    "id": "latte",
    "volume": "-1ml"
  }]
});
→ Bad Request
// Re-reading the list
api.getRecipes();
→
{
  "recipes": [{
    "id": "lungo",
    // This value changed
    "volume": "300ml"
  }, {
    "id": "latte",
    // and this did not
```

```
      "volume": "300ml"
    }]
  }
```

— there is no way how the client might learn that failed operation was actually partially applied. Even if there is an indication of this fact in the response, the client still cannot tell, whether the lungo volume changed because of the request, or if some other client changed it.

If you can't guarantee the atomicity of an operation, you should elaborate in detail on how to deal with it. There must be a separate status for each individual change.

Better:

```
api.updateRecipes({
  "changes": [{
    "recipe_id": "lungo",
    "volume": "300ml"
  }, {
    "recipe_id": "latte",
    "volume": "-1ml"
  }]
});
// You may actually return
// a ‘partial success’ status
// if the protocol allows it
→
{
  "changes": [{
    "change_id",
    "occurred_at",
    "recipe_id": "lungo",
    "status": "success"
  }, {
    "change_id",
    "occurred_at",
    "recipe_id": "latte",
    "status": "fail",
    "error"
  }]
}
```

Here:

- the `change_id` field is a unique identifier of each atomic change;

- the `occurred_at` field is a moment of time when the change was actually applied;
- the `error` field contains the error data related to the specific change.

Might be of use:

- introducing `sequence_id` parameters in the request to guarantee execution order and to align item order in response with the requested one;
- expose a separate `/changes-history` endpoint for clients to get the history of applied changes even if the app crashed while getting a partial success response or there was a network timeout.

Non-atomic changes are undesirable because they erode the idempotency concept. Let's take a look at the example:

```
api.updateRecipes({
  "idempotency_token",
  "changes": [{
    "recipe_id": "lungo",
    "volume": "300ml"
  }, {
    "recipe_id": "latte",
    "volume": "400ml"
  }]
});
→
{
  "changes": [{
    …
    "status": "success"
  }, {
    …
    "status": "fail",
    "error": {
      "reason":
        "too_many_requests"
    }
  }]
}
```

Imagine the client failed to get a response because of a network error, and it repeats the request:

```
api.updateRecipes({
  "idempotency_token",
  "changes": [{
    "recipe_id": "lungo",
    "volume": "300ml"
  }, {
    "recipe_id": "latte",
    "volume": "400ml"
  }]
});
→
{
  "changes": [{
    …
    "status": "success"
  }, {
    …
    "status": "success",
  }]
}
```

To the client, everything looks normal: changes were applied, and the last response got is always actual. But the resource state after the first request was inherently different from the resource state after the second one, which contradicts the very definition of ‘idempotency’.

It would be more correct if the server did nothing upon getting the second request with the same idempotency token, and returned the same status list breakdown. But it implies that storing these breakdowns must be

implemented.

Just in case: nested operations must be idempotent themselves. If they are not, separate idempotency tokens must be generated for each nested operation.

18. Don't invent security

If the author of this book was given a dollar each time he had to implement the additional security protocol invented by someone, he would already retire. The API developers' passion for signing request parameters or introducing complex schemes of exchanging passwords for tokens is as obvious as meaningless.

First, almost all security-enhancing procedures for every kind of operation *are already invented*. There is no need to re-think them anew; just take the existing approach and implement it. No self-invented algorithm for request signature checking provides the same level of preventing Man-in-the-Middle attack as a TLS connection with mutual certificate pinning.

Second, it's quite presumptuous (and dangerous) to assume you're an expert in security. New attack vectors come every day, and being aware of all the actual threats is a full-day job. If you do something different during workdays, the security system designed by you will contain vulnerabilities that you have never heard about — for example, your password-checking algorithm might be

susceptible to the timing attack, and your web-server, to the request splitting attack.

19. Explicitly declare technical restrictions

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 what are the limits automatically implies that partners' code might stop working at any moment because of the 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 which exact boundary 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.

20. Count the amount of traffic

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

There are three obvious reasons for inflating network traffic:

- no data pagination provided;
- no limits on the data fields set, or too large binary data (graphics, audio, video, etc.) is being transmitted;
- clients query for the data too frequently or cache them too little.

If the first two problems are solved by applying pure technical measures (see the corresponding paragraphs), the third one is more of a logical kind: how to organize the client updates stream to find a balance between the responsiveness of the system and the resources spent to ensure it. Here are several recommendations:

- do not rely too heavily on asynchronous interfaces;
    - on one side, they allow tackling many technical problems related to the API performance, which, in turn, allows for maintaining backwards compatibility: if some method is asynchronous from the very beginning, the

latencies and the data consistency models might be easily tuned if needed;
    - from the other side, the number of requests clients generate becomes hardly predicable, as a client in order to retrieve a result needs to make some unpredictable number of attempts;

- declare an explicit retry policy (for example, with the `Retry-After` header);

    - yes, some partners will ignore it as developers will get too lazy to implement it, but some will not (especially if you provide the SDKs as well);

- if you expect a significant number of asynchronous operations in the API, allow developers to choose between the poll model (clients make repeated requests to an endpoint to check the asynchronous procedure status) and the push model (the server notifies clients of status changes, for example, via webhooks or server-push mechanics);

- if some entity comprises both ‘lightweight’ data (let's say, the name and the description of the recipe) and ‘heavy’ data (let's say, the promo picture of the 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 (a link to the image, in our case) — this will allow at least 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 (for example, making a single order) to count the number of requests and the amount of traffic that it takes.

21. Avoid implicit partial updates

One of the most common API design antipatterns is an attempt to spare something on detailed state change descriptions.

Bad:

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

```
// Partially rewrites the order,
// updates the volume
// of the second item
PATCH /v1/orders/{id}
{
  "items": [null, {
    "volume": "800ml"
  }]
}
→
{ /* updates accepted */ }
```

This signature is bad per se as it's unreadable. What does `null` as the first array element mean — is it a deletion of an element or an indication that no actions are needed towards it? What happens with the fields that are not stated in the update operation body (`delivery_address`, `milk_type`) — will they be reset to defaults, or stay unchanged?

The nastiest part is that whatever option you choose, the number of problems will only multiply further. Let's say we agreed that the `{"items":[null, {…}]}` statement means that the first element of the array is left untouched, e.g. no changes are needed. Then, how shall we encode its deletion? Invent one more ‘magical’ value meaning ‘remove it’? Similarly, if the fields that are not explicitly mentioned retain their value — how to reset them to defaults?

The simple solution is always rewriting the data entirely, e.g. to require passing the entire object, to replace the current state with it, and to return the full state as a result of the operation. This obvious solution is frequently rejected with the following reasoning:

- increased requests sizes and therefore, the amount of traffic;
- the necessity to detect which fields are changed (for instance, to generate proper state change events for subscribers);
- the inability of organizing cooperative editing when two clients are editing different object properties simultaneously.

However, if we take a deeper look, all these disadvantages are actually imaginative:

- the reasons for increasing the amount of traffic were described in the previous paragraphs, and serving extra fields is not one of them (and if it is, it's rather a rationale to decompose the endpoint);
- the concept of sending only those fields that changed is in fact about shifting the responsibility of change detection to clients;
    - it doesn't make the task any easier, and also introduces the problem of client code fragmentation as several independent implementations of the change detection algorithm will occur;

    - furthermore, the existence of the client algorithm for finding the fields that changed doesn't mean that the server might skip implementing it as client developers might make mistakes or simply spare the effort and always send all the fields;
- finally, this naïve approach to organizing collaborative editing works only with transitive changes (e.g. if the final result does not depend on the order in which the operations were executed), and in our case, it's already not true: deletion of the first element and editing the second element are non-transitive;
    - often, in addition to sparing traffic on requests, the same concept is applied to responses as well, e.g. no data is returned for modifying operations; thus two clients making simultaneous edits do not see one another's changes.

Better: split the functionality. This also correlates well with the decomposition principle we've discussed in the previous chapter.

```
// Creates an order comprising
// two items
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 order parameters
// that affect all items
PUT /v1/orders/{id}/parameters
{ "delivery_address" }
→
{ "delivery_address" }
```

```
// Partially updates one item,
// sets the volume of one of
// the beverages
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 order item
DELETE /v1/orders/{id}/items/{item_id}
```

Now to reset `volume` to its default value it's enough to omit it in the `PUT /items/{item_id}` request body. Also, the operations of deleting one item while simultaneously modifying another one are now transitive.

This approach also allows for separating non-mutable and calculated fields (in our case, `created_at` and `status`) from editable ones without creating ambiguous situations (what should happen if a client tries to change the `created_at` field?)

It is also possible to return full order objects from `PUT` endpoints instead of just the sub-resource that was overwritten (though it requires some naming convention).

NB: while decomposing endpoints, the idea of splitting them into mutable and non-mutable data often looks tempting. It makes possible to mark the latter as infinitely cacheable and never bother about pagination ordering and update format consistency. The plan looks solid on paper, but with the API expansion, it frequently happens that immutable fields eventually cease being immutable, and the entire concept not only stops working properly but even starts looking like a design flaw. We would rather recommend designating data as immutable in one of the two cases: (1) making them editable will really mean breaking backwards compatibility, or (2) the link to the resource (for example, an image) is served via the API as well, and you do possess the capability of making those links persistent (e.g. you might generate a new link to the image instead of rewriting the contents of the old one).

Even better: design a format for atomic changes.

```
POST /v1/order/changes
X-Idempotency-Token: <idempotency token>
{
  "changes": [{
    "type": "set",
    "field": "delivery_address",
    "value": <new value>
  }, {
    "type": "unset_item_field",
    "item_id",
    "field": "volume"
  }],
  …
}
```

This approach is much harder to implement, but it's the only viable method to implement collaborative editing since it explicitly reflects what a user was actually doing with entity representation. With data exposed in such a format, you might actually implement offline editing, when user changes are accumulated and then sent at once, while the server automatically resolves conflicts by ‘rebasing’ the changes.

Ensuring API product quality

Apart from the technological limitations, any real API will soon face the imperfection of the surrounding reality. Of course, any one of us would prefer living in the world of pink unicorns, free of piles of legacy code, evil-doers,

national conflicts, and competitors' scheming. Fortunately or not, we live in the real world, and API vendors have to mind all those while developing the API.

22. Use globally unique identifiers

It's considered a good form to use globally unique strings as entity identifiers, either semantic (i.e. "lungo" for beverage types) or random ones (i.e. 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 understand quickly 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 a precise number of orders you have each day, for example.

NB: in this book, we often use short identifiers like "123" in code examples; that's for the convenience of reading the book on small screens. Do not replicate this practice in a real-world API.

23. Stipulate future restrictions

With the API popularity growth, it will inevitably become necessary to introduce technical means of preventing illicit API usage, such as displaying captchas, setting honeypots, raising the ‘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 terms of service. 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-factored 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.

24. Don't provide endpoints for mass downloading of sensitive data

If it's possible to get through the API users' personal data, bank card numbers, private messages, or any other kind of information, exposing of which might seriously harm users, partners, and/or you — there must be *no* methods of bulk getting the data, or at least there must be rate limiters, page size restrictions, and, ideally, multi-factored authentication in front of them.

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

25. Localization and internationalization

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

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 absence of location data. In most cases reducing the location to just a country code is enough.

The thing is that lots of parameters potentially affecting data formats depend not on language, but on a 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: user residence location and user ‘viewport’. For example, if a US citizen is planning a European trip, it's convenient to show prices in 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. Take a look at the example in rule #12: the `localized_message` is meant for the user; the app should show it if there is no specific handler for this error exists in the 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. In the software development world, it usually means ‘in English’.

Worth mentioning is that the `localized_` prefix in the example 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 12. 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 BACKWARDS COMPATIBILITY

Chapter 13. The Backwards Compatibility Problem Statement

As usual, let's conceptually define ‘backwards compatibility’ before we start.

Backwards 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 working functionally correctly for a long period of time**. There are two important questions to this definition and two explanations:

1. What does ‘functionally correctly’ mean?

   It means that the code continues to serve its function, e.g. solve some users' problems. It doesn't mean it continues working indistinguishably: for example, if you're maintaining a UI library, changing functionally insignificant design details like shadow depth or border stoke type is backwards compatible, whereas changing visual components size is not.

2. What does ‘a long period of time’ mean?

   From our point of view, the backwards compatibility maintenance period should be reconciled with the subject area application lifetime. Platform LTS periods are decent guidance in most cases. Since apps will be rewritten anyway when the platform

> maintenance period ends, it is reasonable to expect developers to move to the new API version also. In mainstream subject areas (e.g. desktop and mobile operating systems) this period lasts several years.

From the definition becomes obvious why backwards compatibility needs to be maintained (including taking necessary measures at the API design stage). An outage, full or partial, caused by the API vendor, is an extremely uncomfortable situation for every developer, if not a disaster — especially if they pay money for the API usage.

But let's take a look at the problem from another angle: why the maintaining backwards compatibility problem exists at all? Why would anyone *want* to break it? This question, though it looks quite trivial, is much more complicated than the previous one.

We could say the *we break backwards compatibility to introduce new features to the API*. But that would be deceiving: new features are called *'new'* just because they cannot affect existing implementations which are not using them. We must admit there are several associated problems, which lead to the aspiration to rewrite *our* code, the code of the API itself, and ship a new major version:

- the code eventually becomes outdated; making changes, even introducing totally new functionality, is impractical;

- the old interfaces aren't suited to encompass new features; we would love to extend existing entities with new properties, but simply couldn't;

- finally, with years passing since the initial release, we understood more about the subject area and API usage best practices, and we would implement many things differently.

These arguments could be summarized frankly as ‘the API developers don't want to support the old code’. But this explanation is still incomplete: even if you're not going to rewrite the API code to add new functionality, or you're not going to add it at all, you still have to ship new API versions, minor and major alike.

NB: in this chapter, we don't make any difference between minor versions and patches: ‘minor version’ means any backwards-compatible API release.

Let us remind that an API is a bridge, a meaning of connecting different programmable contexts. No matter how strong our desire to keep the bridge intact is, our capabilities are limited: we could lock the bridge, but we cannot command the rifts and the canyon itself. That's the source of the problems: we can't guarantee that *our own* code won't change, so at some point, we will have to ask the clients to change *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 underlying platforms' erosion.

Consumer applications fragmentation

When you shipped the very first API version, and the first clients started to use it, the situation was perfect. There was only one version, and all clients were using just it. When this perfection ends, two scenarios are possible.

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

   It doesn't mean that you can't break backwards compatibility. You still can make a mess with cache-control headers or just overlook a bug in the code. Besides, even code-on-demand systems don't get updated instantly. The author of this book faced the situation when users were deliberately keeping a browser tab open *for weeks* to get rid of updates. But still, you usually don't have to support more than two API versions — the last one and the penultimate one.

Furthermore, you may try to rewrite 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 in modern mobile operating systems, then the situation becomes more severe. Each client effectively borrows a snapshot of the code, working with your API, frozen at the moment of compilation. Client application updates are scattered over time at much more extent than Web application updates. The most painful thing is that *some clients will never be up to date*, because of one of the three reasons:

    - developers simply don't want to update the app, e.g. its development stopped;
    - users don't want to get updates (sometimes because users think that developers ‘spoiled’ the app in new versions);
    - users can't get updates because their devices are no longer supported.

    In modern times these three categories combined could easily constitute tens of per cent of auditory. It implies that cutting the support of any API version might be remarkable — especially if developers' apps continue supporting a more broad spectrum of platforms than the API does.

> You could have never issued any SDK, providing just the server-side API, for example in a form of HTTP endpoints. You might think, given your API is less competitive on the market because of a lack of SDKs, that the backwards compatibility problem is mitigated. That's not true: if you don't provide an SDK, then developers will either adopt an unofficial one (if someone bothers to make it) or just write a framework themselves — independently. 'Your framework — your problems' strategy, fortunately or not, works badly: if developers write poor quality code upon your API, then your API is of poor quality itself. Definitely in the view of developers, possibly in the view of end-users, if the API performance within the app is visible to them.

Certainly, if you provide a stateless API that doesn't require client SDKs (or they might be auto-generated from the spec), those problems will be much less noticeable, but not fully avoidable, unless you never issue any new API version. If you do, you will still have to deal with some fragmentation of users by API and SDK versions.

Subject area evolution

The other side of the canyon is the underlying functionality you're exposing via the API. It's, of course, not static and somehow evolves:

- 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 models to pop up, which are to be supported by the platform. New models tend to provide new APIs, and it's hard to guarantee they might be adopted while preserving the same high-level API. And anyway, the code needs to be altered, which might lead to incompatibility, albeit unintentional.

Let us also stress that low-level API vendors are not always as resolute regarding maintaining backwards compatibility for their APIs (actually, any software they provide) as (we hope so) you are. You should be warned that keeping your API in an operational state, e.g. writing and supporting facades to the shifting subject area landscape, will be your problem, and rather a sudden one.

Platform drift

Finally, there is a third side to a story — the ‘canyon’ you're crossing over with a bridge of your API. Developers write code that is executed in some environment you can't control, and it's evolving. New versions of operating systems, browsers, protocols, and programming language SDKs emerge. New standards are being developed, new arrangements made, some of them being backwards-incompatible, and nothing could be done about that.

Older platform versions lead to fragmentation just like older app versions do, because developers (including the API developers) are struggling with supporting older platforms, and users are struggling with platform updates — and often can't update at all, since newer platform versions require newer devices.

The nastiest thing here is that not only does incremental progress in a form of new platforms and protocols demand changing the API, but also does vulgar fashion. Several years ago realistic 3d icons were popular, but since then the public taste changed in a favor of flat and abstract ones. UI components developers had to follow the fashion, rebuilding their libraries, either shipping new icons or replacing old ones. Similarly, right now ‘night mode’ support is introduced everywhere, demanding changes in a broad range of APIs.

Backwards compatibility policy

To summarize the above:

- you will have to deploy new API versions because of apps, platforms, and subject area evolution; different areas are evolving at a different pace, but never stop doing so;
- that will lead to fragmenting the API versions usage over different platforms and apps;
- you have to make decisions critically important to your API's sustainability in the customers' view.

Let's briefly describe these decisions and the key factors for making them.

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

   That's primarily a *product* question. A new major API version is to be released when the critical mass of functionality is reached — a critical mass of features that couldn't be introduced in the previous API versions, or introducing them is too expensive. In stable markets, such a situation occurs once in several years, usually. In emerging markets, new API major versions might be shipped more frequently, only depending on your capabilities of supporting the zoo of previous versions. However, we should note that deploying a new version before the previous one was stabilized (which commonly takes from several months up to a year) is always a troubling sign to developers, meaning they're risking dealing with the unfinished platform glitches permanently.

2. How many *major* versions should be supported at a time?

   As for major versions, we gave *theoretical* advice earlier: ideally, the major API version lifecycle should be a bit longer than the platform's one. In stable niches like desktop operating systems, it constitutes 5 to 10 years. In new and emerging ones, it is less but still measured in years. *Practically* speaking you

should look at the size of the auditory which continues using older versions.

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

   As for minor versions, there are two options:

   - if you provide server-side APIs and compiled SDKs only, you may basically do not expose minor versions at all, just the actual one: the server-side API is totally within your control, and you may fix any problem efficiently;
   - if you provide code-on-demand SDKs, it is considered a good form to provide an access to previous minor versions of SDK for a period of time sufficient enough for developers to test their application and fix some issues if necessary. Since full rewriting isn't necessary, it's fine to align with apps release cycle duration in your industry, which is usually several months in worst cases.

Simultaneous access to several API versions

In modern professional software development, especially if we talk about internal APIs, a new API version usually fully replaces the previous one. If some problems are found, it might be rolled back (by releasing the previous version), but the two builds never co-exist. However, in the case of public APIs, the more the number of partner integrations is, the more dangerous this approach becomes.

Indeed, with the growth of the number of users, the ‘rollback the API version in case of problems’ paradigm becomes increasingly destructive. To a partner, the optimal solution is rigidly referencing the specific API version — the one that had been tested (ideally, at the same time having the API vendor somehow seamlessly fixing security issues and making their software compliant with newly introduced legislation).

NB. From the same considerations, providing beta (or maybe even alpha) versions of the popular APIs becomes more and more desirable as well, to make partners test the upcoming version and address the possible issues in advance.

The important (and undeniable) advantage of the *semver* system is that it provides the proper version granularity:

- stating the first digit (major version) allows for getting a backwards-compatible version of the API;
- stating two digits (major and minor versions) allows to guarantee that some functionality that was added after the initial release will be available;
- finally, stating all three numbers (major version, minor version, and patch) allows for fixing a concrete API release with all its specificities (and errors), which — theoretically — means that the integration will remain operable till this version is physically available.

Of course, preserving minor versions infinitely isn't possible (partly because of security and compliance issues that tend to pile up). However, providing such access for a reasonable period of time is rather a hygienic norm for popular APIs.

NB. Sometimes to defend the single accessible API version concept, the following argument is put forward: preserving the SDK or API application server code is not enough to maintain strict backwards compatibility, as it might be relying on some un-versioned services (for example, some data in the DB that are shared between all the API versions). We, however, consider this an additional reason to isolate such dependencies (see ‘The Serenity Notepad’ chapter) as it means that changes to these subsystems might lead to the inoperability of the API.

Chapter 14. On the Waterline of the Iceberg

Before we start talking about the extensible API design, we should discuss the hygienic minimum. A huge number of problems would have never happened if API vendors had paid more attention to marking their area of responsibility.

1. Provide a minimal amount of functionality

At any moment in its lifetime, your API is like an iceberg: it comprises an observable (e.g. documented) part and a hidden one, undocumented. If the API is designed properly, 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 vice versa. The code developers write upon your API must 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. Probably, this API simply isn't needed at all.

- Revoking the API functionality causes losses. If you've promised to provide some functionality, you will have to do so ‘forever’ (until this API version's maintenance period is over). Pronouncing some functionality deprecated is a tricky thing, potentially alienating your customers.

Rule #1 is the simplest: if some functionality might be withheld — then never expose it. It might be reformulated like: every entity, every field, and every public API method is a *product solution*. There must be solid *product* reasons why some functionality is exposed.

2. Avoid gray zones and ambiguities

Your obligations to maintain some functionality must be stated as clearly as possible. Especially regarding those environments and platforms where no native capability to restrict access to undocumented functionality exists. Unfortunately, developers tend to consider some private features they found to be eligible for use, thus presuming the API vendor shall maintain them intact. Policy on such 'findings' must be articulated explicitly. At the very least, in case of such non-authorized usage of undocumented functionality, you might refer to the docs, and be in your own rights in the eyes of the community.

However, API developers often legitimize such gray zones themselves, for example, by:

- returning undocumented fields in endpoints' responses;
- using private functionality in code examples — in the docs, responding to support messages, in conference talks, etc.

One cannot make a partial commitment. Either you guarantee this code will always work or do not slip the slightest note such functionality exists.

3. Codify implicit agreements

The third principle is much less obvious. Pay close attention to the code which you're suggesting developers to develop: are there any conventions that you consider evident, but never wrote them 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 moved to the asynchronous replication of the database. This would lead to the situation when querying for the order status right after order creating might return `404` if an asynchronous replica hasn't got the update yet. In fact, thus we abandon a strict consistency policy in a favor of an eventual one.

What would be the result? The code above will stop working. A developer creates an order, then tries to get its status — but gets the error. It's very hard to predict what approach developers would implement to tackle this error. Probably, none at all.

You may say something like, ‘But we've never promised the 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:

```
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 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 can't make these changes in the API. You will effectively break backwards compatibility, which will lead to huge problems with your customers' apps, intensified by the fact they can't be simply reproduced.

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 is called. But this assumption is based on nothing: there are no clues indicating 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, good advice is to avoid such conventions, since they are simply unobvious while reading the code. And of course, under no circumstances, you can actually change this behavior to an asynchronous one.

Example #3. Imagine you're providing animations API, which includes two independent functions:

```
// Animates object's width,
// beginning with first value,
// ending with second
// in a specified time period
object.animateWidth(
  '100px', '500px', '1s'
);
// Observes object's width changes
object.observe(
  'widthchange', observerFunction
);
```

A question arises: how frequently and at what time fractions the `observerFunction` will 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 switched to implementing both functions atop of a system's native functionality — and so you simply don't

know, when and how frequently the `observerFunction` will be called.

Just changing call frequency might result in making some code dysfunctional — for example, if the callback function makes some complex calculations, and no throttling is implemented since the developer just relied on your SDK's built-in throttling. And if the `observerFunction` ceases to be called when exactly '500px' is reached because of some system algorithms 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 event "ready", without a "payment_approved" event being emitted. It might appear to you that this modification *is* backwards-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 some valuable business procedure, for example, gathering income and expenses analytics. It's quite logical to expect this code operates a state machine, which switches from one state to another depending on getting (or getting not) 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 have to cope with the error's cause; the worst-case, partners will operate wrong statistics for an indefinite period of time until they find a mistake.

A proper decision would be, first, documenting the event order and the allowed states; second, continuing generating 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 backwards-compatible as well

State transition graph, event order, possible causes of status changes — such critical things must be documented. Not every piece of business logic might be defined in a form of a programmatical contract; some cannot be represented at all.

Imagine that one day you start to take phone calls. A client may contact the call center to cancel an order. You might even make this functionality *technically* backwards-compatible, 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. Partner's business analytical code might be broken likewise, or start displaying weather on Mars since it was written knowing nothing about the possibility of canceling orders somehow 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 may only cancel those orders which were created with this flag set. But that would be a bad decision from a *product* point of view. 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’ to be discussed in the last chapter of this Section.

Chapter 15. Extending through Abstracting

In previous chapters, we have tried to outline theoretical rules and illustrate them with practical examples. However, understanding the principles of change-proof API design requires practice above all things. An ability to anticipate future growth problems comes from a handful of grave mistakes once made. One cannot foresee everything but can develop certain technical intuition.

So in the following chapters, we will try to probe our study API from the previous Section, testing its robustness from every possible viewpoint, thus carrying out some '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 an 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 of two methods: `POST /offers/search` and `POST /orders`.

Let us make the next logical step there 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 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 is like that:

- 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 we now 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 look like there are no such things in our API since it's quite simple and basically just describes making some HTTP call — 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 some 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 will we make these implicit arrangements explicit if we need that. 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 like 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 allowing for tackling those restrictions set in the previous paragraph.

3. Pronounce the existing interfaces (those defined in #1) being ‘helpers’ to new ones (those defined in #2) which sets some options to 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 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 `PUT /coffee-machines` endpoint as it now stands in the protocol being 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 talking about *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 now are defining these in a course of expanding the public API, it's a very strong claim to break backwards compatibility, and we should never do that (see Chapter 14).

Limits of Applicability

Though this exercise looks very simple and universal, its consistent usage is possible only if the hierarchy of entities is well designed from the very beginning and, which is more important, the vector of the further API expansion is clear. Imagine that after some time passed, the options list got new items; let's say, adding syrup or a second espresso shot. We are totally capable of expanding the list — but not the defaults. So the 'default' `PUT /coffee-machines` interface will eventually become totally useless because the default set of three options will not only be any longer of use but will also look ridiculously: why these three options, what are the selection criteria? In fact, the defaults and the method list will be reflecting the historical stages of our

API development, and that's totally not what you'd expect from the helpers and defaults nomenclature.

Alas, this dilemma can't be easily resolved. From one side, we want developers to write neat and laconic code, so we must provide useful helpers and defaults. On the other side, we can't know in advance which sets of options will be the most frequent after several years of the API expansion.

NB. We might mask this problem in the following manner: one day gather all these oddities and re-define all the defaults with one single parameter. For example, introduce a special method like `POST /use-defaults {"version": "v2"}` which would overwrite all the defaults with more suitable values. That will ease the learning curve, but your documentation will become even worse after that.

In the real world, the only viable approach to somehow tackle the problem is the weak entity coupling, which we will discuss in the next chapter.

Chapter 16. Strong Coupling and Related Problems

To demonstrate the strong coupling problematics let us move to *really interesting* things. Let's continue our ‘variation analysis’: what if the partners wish to offer not only the standard beverages but their own unique coffee recipes to end-users? There is a catch in this question: the partner API as we described it in the previous chapter, does not expose the very existence of the partner network to the end-user, and thus describes a simple case. Once we start providing methods to alter the core functionality, not just API extensions, we will soon face next-level problems.

So, let us add one more endpoint to register the partner's own recipe:

```
// Adds new recipe
POST /v1/recipes
{
  "id",
  "product_properties": {
    "name",
    "description",
    "default_value"
    // Other properties, describing
    // a beverage to end-user
    …
  }
}
```

At first glance, again, it looks like a reasonably simple interface, explicitly decomposed into abstraction levels. But let us imagine the future — what would happen with this interface when our system evolves further?

The first problem is obvious to those who read chapter 11 thoroughly: product properties must be localized. That will lead us to the first change:

```
"product_properties": {
  // "l10n" is a standard abbreviation
  // for "localization"
  "l10n" : [{
    "language_code": "en",
    "country_code": "US",
    "name",
    "description"
  }, /* other languages and countries */ … ]
]
```

And here the first big question arises: what should we do with the `default_volume` field? From one side, that's an objective quality measured in standardized units, and it's being passed to the program execution engine. On the other side, in countries like the United States, we had to specify beverage volume not like ‘300 ml’, but ‘10 fl oz’. We may propose two solutions:

- either the partner provides the corresponding number only, and we will make readable descriptions on our own behalf,

- or the partner provides both the number and all of its localized representations.

The flaw in the first option is that a partner might be willing to use the service in some new country or language — and will be unable to do so until the API supports them. The flaw in the second option is that it works with predefined volumes only, so you can't order an arbitrary beverage volume. So the very first step we've made effectively has 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`? They are simply non-machine-readable strings with no specific semantics. At first glance, we need them to return them back in the `/v1/search` method response, but that's not a proper answer: why do we really return these strings from `search`?

The correct answer lies a way beyond this specific interface. We need them *because some representation exists*. There is a UI for choosing beverage type. Probably the `name` and `description` fields are simply two designations of the beverage for a 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). It actually means that we are setting the requirements to the API based on some very specific design. But *what if* a partner is making their own UI for their own app? Not only they might not actually need two descriptions, but we are also *deceiving* them. The `name` is not ‘just a name’ actually, it implies some

restrictions: it has recommended length which is optimal to some specific UI, and it must look consistently on the search results page. Indeed, ‘our best quality™ coffee’ or ‘Invigorating Morning Freshness®’ designation would look very weird in between ‘Cappuccino’, ‘Lungo’, and ‘Latte’.

There is also another side to this story. As UIs (both ours and partners) tend to evolve, new visual elements will be eventually introduced. For example, a picture of a beverage, its energy value, allergen information, etc. `product_properties` will become a scrapyard for tons of optional fields, and learning how setting what field results in what effects in the UI will be an interesting quest, full of probes and mistakes.

Problems we're facing are the problems of *strong coupling*. Each time we offer an interface like described above, we in fact prescript implementing one entity (recipe) based on implementations of other entities (UI layout, localization rules). This approach disrespects the very basic principle of the ‘top to bottom’ API design because **low-level entities must not define high-level ones**.

The rule of contexts

To make things worse, let us state that the inverse principle is actually correct either: high-level entities must not define low-level ones, since that simply isn't their responsibility. The exit from this logical labyrinth is that high-level entities must *define a context*, which other objects are to interpret. To properly design adding a new

recipe interface we shouldn't try to find a better data format; we need to understand what contexts, both explicit and implicit, exist in our subject area.

We have already found a localization context. There is some set of languages and regions we support in our API, and there are requirements — what exactly the partner must provide to make our API work in a new region. More specifically, there must be some formatting function to represent beverage volume somewhere in our API code:

```
l10n.volume.format(
  value, language_code, country_code
)
// l10n.formatVolume(
//   '300ml', 'en', 'UK'
// ) → '300 ml'
// l10n.formatVolume(
//   '300ml', 'en', 'US'
// ) → '10 fl oz'
```

To make our API work correctly with a new language or region, the partner must either define this function or point which pre-existing implementation to use. Like this:

```
// Add a general formatting rule
// for Russian language
PUT /formatters/volume/ru
{
  "template": "{volume} мл"
}
// Add a specific formatting rule
// for Russian language in the ‘US’ region
PUT /formatters/volume/ru/US
{
  // in US we need to recalculate
  // the number, then add a postfix
  "value_preparation": {
    "action": "divide",
    "divisor": 30
  },
  "template": "{volume} ун."
}
```

NB: we are more than aware that such a simple format isn't enough to cover real-world localization use-cases, and one either relies on existing libraries or designs a sophisticated format for such templating, which takes into account such things as grammatical cases and rules of rounding numbers up or allow defining formatting rules in a form of function code. The example above is simplified for purely educational purposes.

Let us deal with the `name` and `description` problem then. To lower the coupling level there we need to formalize (probably just to ourselves) a ‘layout’ concept. We are asking for providing `name` and `description` not because we just need them, but for representing them in some specific user interface. This specific UI might have an identifier or a semantic name.

```
GET /v1/layouts/{layout_id}
{
  "id",
  // We would probably have lots of layouts,
  // so 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 explicit
    // `search_title` instead
    "field": "search_title",
    "view": {
      // 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"
  ]
}
```

So the partner may decide, which option better suits them. They can provide mandatory fields for the standard layout:

```
PUT /v1/recipes/{id}/properties/l10n/{lang}
{
  "search_title", "search_description"
}
```

or create a layout of their own and provide data fields it requires:

```
POST /v1/layouts
{
  "properties"
}
→
{ "id", "properties" }
```

or they may ultimately design their own UI and don't use this functionality at all, defining neither layouts nor data fields.

Then our interface would ultimately look like this:

```
POST /v1/recipes
{ "id" }
→
{ "id" }
```

This conclusion might look highly counter-intuitive, but lacking any fields in a ‘Recipe’ simply tells us that this entity possesses no specific semantics of its own, and is simply an identifier of a context; a method to point out where to look for 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 desirable layouts
  "layouts": [{
    "id", "kind", "properties"
  }],
  // Add all the formatters needed
  "formatters": {
    "volume": [
      {
        "language_code",
        "template"
      }, {
        "language_code",
        "country_code",
        "template"
      }
    ]
  },
  // Other actions needed to be done
  // to register new recipe in the system
  …
}
```

We should also note that providing a newly created entity identifier by the requesting side isn't exactly the best pattern. However, since we decided from the very beginning to keep recipe identifiers semantically meaningful, we have to live with this convention. Obviously, we're risking getting lots of collisions on recipe names used by different partners, so we actually need to modify this operation: either the partner must always use a pair of identifiers (i.e. recipe's one plus partner's own id), or we need to introduce composite identifiers, as we recommended earlier in Chapter 11.

```
POST /v1/recipes/custom
{
  // First part of the composite
  // identifier, for example,
  // the partner's own id
  "namespace": "my-coffee-company",
  // 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 totally isolated namespaces, or conversely share them. Furthermore, we might introduce special namespaces (like

‘common’, for example) to allow for publishing new recipes for everyone (and that, by the way, would allow us to organize our own backoffice to edit recipes).

Chapter 17. Weak Coupling

In the previous chapter we've demonstrated how breaking the strong coupling of components leads to decomposing entities and collapsing their public interfaces down to a reasonable minimum. A mindful reader might have noted that this technique was already used in our API study much earlier in Chapter 9 with regards to the ‘program’ and ‘program run’ entities. Indeed, we might do it without the `program-matcher` endpoint and make it this way:

```
GET /v1/recipes/{id}/run-data/{api_type}
→
{ /* A description, how to
     execute a specific recipe
     using a specified API type */ }
```

Then developers would have to make this trick to get coffee prepared:

- learn the API type of the specific coffee machine;
- get the execution description, as stated above;
- depending on the API type, run some specific commands.

Obviously, such an interface is absolutely unacceptable, simply because in the majority of use cases developers don't care at all, which API type the specific coffee machine runs. To avoid the necessity of introducing such bad

interfaces we created a new ‘program’ entity, which constitutes merely a context identifier, just like a ‘recipe’ entity does. A `program_run_id` entity is also organized in this manner, it also possesses no specific properties, being *just* a program run identifier.

But let us return to the question we have previously mentioned in Chapter 15: how should we parametrize the order preparation process implemented via third-party API. In other words, what's this `program_execution_endpoint` that we ask upon the API type registration?

```
PUT /v1/api-types/{api_type}
{
    "order_execution_endpoint": {
        // ???
    }
}
```

Out of general considerations, we may assume that every such API would be capable of executing three functions: run a program with specified parameters, return the current execution status, and finish (cancel) 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:

```
// This is an endpoint for partners
// to register their coffee machines
// in the system
PUT /partners/{id}/coffee-machines
{
  "coffee-machines": [{
    "id",
    …
    "order_execution_endpoint": {
      "program_run_endpoint": {
        /* Some description of
            the remote function call */
        "type": "rpc",
        "endpoint": <URL>,
        "format"
      },
      "program_state_endpoint",
      "program_cancel_endpoint"
    }
  }, …]
}
```

NB: doing so we're transferring the complexity of developing the API onto a plane of developing appropriate data formats, e.g. how exactly would we send order parameters to the `program_run_endpoint`, and what format the `program_state_endpoint` shall return, etc., but in this chapter, we're focusing on different questions.

Though this API looks absolutely universal, it's quite easy to demonstrate how once simple and clear API ends up being confusing and convoluted. This design presents two main problems.

1. It describes nicely the integrations we've already implemented (it costs almost nothing to support the API types we already know), but brings no flexibility in the approach. In fact, we simply described what we'd already learned, not even trying to look at a larger picture.
2. This design is ultimately based on a single principle: every order preparation might be codified with these three imperative commands.

We may easily disprove the #2 principle, and that will uncover the implications of the #1. For the beginning, let us imagine that on a course of further service growth we decided to allow end-users to change the order after the execution started. For example, ask for a cinnamon sprinkling or contactless takeout. That would lead us to creating a new endpoint, let's say, `program_modify_endpoint`, and new difficulties in data format development (we need to understand in the real-time, could we actually sprinkle cinnamon on this specific cup of coffee or not). What *is* important is that both endpoint and new data fields would be optional because of backwards 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 via our API not a coffee house, but a vending machine? From one side, it means that the `modify` endpoint and all related stuff are simply meaningless: a vending machine couldn't sprinkle cinnamon over a coffee cup, and the contactless takeout requirement means nothing to it. On the other side, the machine, unlike the people-operated café, requires *takeout approval*: the end-user places an order being somewhere in some other place then walks to the machine and pushes the ‘get the order’ button in the app. We might, of course, require the user to stand 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 three endpoints:

- to have vending machines integrated a partner must implement the `program_takeout_endpoint`, but doesn't actually need the `program_modify_endpoint`;
- to have regular coffee houses integrated a partner must implement the `program_modify_endpoint`, but doesn't actually need the `program_takeout_endpoint`.

Furthermore, we have to describe both endpoints in the docs. It's quite natural that the `takeout` endpoint is very specific; unlike cinnamon sprinkling, 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 — but developers would need to study the docs nonetheless to understand, which methods are needed in your specific situation, and which are not.

We actually don't know, whether in the real world of coffee machine APIs this problem will really occur or not. But we can say with all confidence regarding ‘bare metal’ integrations that 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 borrows 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 become automated, we will finally end up with the situation with half of the methods *isn't actually needed at all*, like the requesting a contactless takeout one.

It is also worth mentioning that we unwittingly violated the abstraction levels isolation principle. At the vending machine API level, there is no such thing as a ‘contactless takeout’, that's actually a product concept.

So, how would we tackle this issue? Using 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 from both sides? 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, sprinkle with cinnamon, allow this user to take it;
- the underlying program execution API level doesn't care what other same-level implementations exist; it just interprets those parts of the task which 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 Chapter 9.

In our case we need to implement the following mechanisms:

- running a program creates a corresponding context comprising all the essential parameters;

- there is a method to stream the information regarding the state modifications: the execution level may read the context, learn about all the changes and report back the changes of its own.

There are different techniques to organize this data flow, but, basically, we always have two context descriptions and a two-way event stream in-between. If we were developing an SDK we would express the idea like this:

```
/* Partner's implementation of the program
   run procedure for a custom API type */
registerProgramRunHandler(
  apiType,
  (program) => {
    // Initiating an execution
    // on partner's side
    let execution = initExecution(…);
    // Listen to parent context's changes
    program.context.on(
      'takeout_requested',
      () => {
        // If takeout is requested, initiate
        // corresponding procedures
        execution.prepareTakeout(() => {
          // When the cup is ready for takeout,
          // emit corresponding event for
          // a higher-level entity to catch it
          execution.context
            .emit('takeout_ready');
        }
      );
    }
  );

  return execution.context;
});
```

NB: In the case of HTTP API corresponding example would look rather bulky as it involves implementing several additional endpoints for message queues like `GET /program-run/events` and `GET`

`/partner/{id}/execution/events`. We would leave this exercise to the reader. Also worth mentioning that in real-world systems such event queues are usually organized using external event message systems like Apache Kafka or Amazon SNS/SQS.

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` objects fields 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 solves another problem: how to make the code written by developers stay clean and maintainable. Why would strong-coupled integration (i.e. coupling entities via methods) render the code unreadable? Because both sides *are obliged* to implement the functionality which is meaningless in their corresponding subject areas. And

these implementations would actually comprise a handful of methods to say that this functionality is either not supported at all, or supported always and unconditionally.

The difference between strong coupling and weak coupling is that the field-event mechanism *isn't obligatory to both sides*. Let us remember what we sought to achieve:

- a higher-level context doesn't actually know how 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 actually happen under its specific conditions.

It's ultimately possible that both sides would know nothing about each other and wouldn't interact at all. This might actually happen at some point in the future with the evolution of underlying technologies.

Worth mentioning that the number of entities (fields, events), though effectively doubled compared to strong-coupled API design, increased qualitatively, not quantitatively. The `program` context describes fields and events in its own terms (type of beverage, volume, cinnamon sprinkling), while the `execution` context must reformulate those terms according to its own subject area

(omitting redundant ones, by the way). It is also important that the `execution` context might concretize these properties for underlying objects according to their own specifics, while the `program` context must keep its properties general enough to be applicable to any possible underlying technology.

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 upcoming 'SDK' 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 program
   run procedure for a custom API type */
registerProgramRunHandler(
  apiType,
  (program) => {
    // Initiating an execution
    // on partner's side
    let execution = initExecution(…);
    // Listen to parent context's changes
    program.context.on(
      'takeout_requested',
      () => {
        // If takeout is requested, initiate
        // corresponding procedures
        execution.prepareTakeout(() => {
          /* When the order is ready
             for takeout, signalize about that
             by calling a 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 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 making all this stuff up because we expect alternative implementations of the *lower* abstraction level. Situations with different realizations of *higher* abstraction levels emerging are, of course, possible, but quite rare. The tree of alternative implementations usually grows from top to bottom.

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, i.e. we start selling flight tickets instead of preparing coffee, there is literally no sense to preserve backwards compatibility at technical abstraction levels. Ironically, we may actually make our program run API sell tickets instead of brewing coffee without breaking backwards compatibility, but the partners' code will still become obsolete.

In conclusion, because of the abovementioned reasons, higher-level APIs are evolving more slowly and much more consistently than low-level APIs, which means that 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:

```
execution.prepareTakeout(() => {
  // Instead of generating events
  // or calling higher-level methods,
  // an `execution` entity calls
  // a global or quasi-global
  // 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 — of abstraction levels isolation, and therefore isn't suitable 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 interacting with its closest higher-level neighbors only, delegating the responsibility of calling high-level or global methods to them.

```
execution.prepareTakeout(() => {
  // Instead of initiating global actions
  // 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)
  );
}
```

Test Yourself

So, we have designed the interaction with third-party APIs as described in the previous paragraph. And now we should (actually, must) check whether these interfaces are compatible with our own abstraction we had developed in the Chapter 9. In other words, could we start an execution of an order if we operate the low-level API instead of the high-level one?

Let us recall that we had proposed the following abstract interfaces to work with arbitrary coffee machine API types:

- `POST /v1/program-matcher` returns the id of the program based on the coffee machine and recipe ids;
- `POST /v1/programs/{id}/run` executes the program.

As we can easily prove, it's quite simple to make these interfaces compatible: we only need to assign a `program_id` identifier to the (API type, recipe) pair, for example, through returning it in the `PUT /coffee-machines` method response:

```
PUT /v1/partners/{partnerId}/coffee-machines
{
  "coffee_machines": [{
    "id",
    "api_type",
    "location",
    "supported_recipes"
  }, …]
}
→
{
  "coffee_machines": [{
    "id",
    "recipes_programs": [
      {"recipe_id", "program_id"},
      …
    ]
  }, …]
}
```

So the method we'd developed:

```
POST /v1/programs/{id}/run
```

will work with the partner's coffee machines (like it's a third API type).

Delegate!

From what was said, one more important conclusion follows: doing a real job, e.g. implementing some 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 some specific implementation algorithms, then (as we have demonstrated on the `order_execution_endpoint` example) we will soon face a situation of inconsistent methods and interaction protocols nomenclature, most of which have no specific meaning when we talk about some specific hardware context.

Contrariwise, applying the paradigm of concretizing the contexts at each new abstraction level, we will eventually fall into the bunny hole deep enough to have nothing to concretize: the context itself unambiguously matches the functionality we can programmatically control. And at that level, we must stop detailing contexts further, and just realize the algorithms needed. Worth mentioning that the abstraction deepness for different underlying platforms might vary.

NB. In the Chapter 9 we have illustrated exactly this: when we speak about the first coffee machine API type, there is no need to extend the tree of abstractions further than running programs, but with the second API type, we need one more intermediary abstraction level, namely the runtimes API.

Chapter 18. Interfaces as a Universal Pattern

Let us summarize what we have written in the three previous chapters.

1. Extending API functionality is realized through abstracting: the entity nomenclature is to be reinterpreted so that existing methods become partial (ideally — the most frequent) simplified cases to more general functionality.
2. Higher-level entities are to be the informational contexts for low-level ones, e.g. don't prescribe any specific behavior but translate their state and expose functionality to modify it (directly through calling some methods or indirectly through firing events).
3. Concrete functionality, e.g. 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 being the SOLID architecture principles. There is no surprise in that either, because SOLID concentrates on contract-oriented development, and APIs are contracts by definition. We've just added ‘abstraction levels’ and ‘informational contexts’ concepts there.

However, there is an unanswered question: how should we design the entity nomenclature from the beginning so that extending the API won't make it a mess of different inconsistent methods of different ages. The answer is

pretty obvious: to avoid clumsy situations while abstracting (as with the coffee machine's supported options), all the entities must be originally considered being a specific implementation of a more general interface, even if there are no planned alternative implementations for them.

For example, we should have asked ourselves a question while designing the `POST /search` API: what is a ‘search result’? What abstract interface does it implement? To answer this question we must neatly decompose this entity to find 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 we create an order, we need from the search result to provide those fields which describe the order itself; it might be a structure like:

  `{coffee_machine_id, recipe_id, volume, currency_code, price}`,

  or we can encode this data in the single `offer_id`;

- to have this search result displayed in the app, we need a different data set: `name`, `description`, formatted and localized price.

So our interface (let us call it `ISearchResult`) is actually a composition of two other interfaces: `IOrderParameters` (an entity that allows for creating an order) and `ISearchItemViewParameters` (some abstract representation of the search result in the UI). This interface split should automatically 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 Chapter 16.

2. And what is the ‘abstract representation of the search result in the UI’? Do we have other kinds of search, should the `ISearchItemViewParameters` interface be a subtype of some even more general interface, or maybe a composition of several such ones?

Replacing specific implementations with interfaces not only allows us to answer more clearly many questions which should have popped out in the API design phase but also helps us to outline many possible API evolution vectors, which should help in avoiding API inconsistency problems in the future.

Chapter 19. The Serenity Notepad

Apart from the abovementioned abstract principles, let us give a list of concrete recommendations: how to make changes in the existing API to maintain the backwards compatibility.

1. Remember the iceberg's waterline

If you haven't given any formal guarantee, it doesn't mean that you can violate informal once. Often, even just fixing bugs in APIs might make some developers' code inoperable. We might illustrate it with a real-life example that the author of this book has 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 was fixed, we got a bunch of complaints: clients did really use this flaw to stack the buttons in the container's corner.

If fixing the 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 if you develop a large API with a huge

audience. For example, operating systems API developers literally have to transfer old bugs to new OS versions.

2. Test the formal interface

Any software must be tested, and APIs ain't an exclusion. However, there are some subtleties there: as APIs provide formal interfaces, it's the formal interfaces that are needed to be tested. That leads to several kinds of mistakes:

1. Often the requirements like ‘the `getEntity` function returns the value previously being set by the `setEntity` function’ appear to be too trivial to 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 be tested either. 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.

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 a life much simpler by sparing an effort needed to implement the weak-coupled interaction between services. For example, while developing program execution interfaces as described in the Chapter 9 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 backwards 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, e.g. develop an abstraction level that will allow for:

- keeping backwards compatibility intact because of extension capabilities incorporated in the API design;
- negating partner's problems by the technical means:
    - limiting the partner's API usage in case of an unpredicted surge in your API usage;

    - implementing the 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 automatical fallback to another partner or alternative API.

4. Implement your API functionality atop of public interfaces

There is an antipattern that occurs frequently: API developers use some internal closed implementations of some methods which 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;
- on a 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 Chapter 16 developers have created a new, more general version of the volume formatter but hasn't changed the

implementation of the existing one, so it continues working in case of pre-existing languages.

There are obvious local problems with this approach (like the inconsistency in functions' behavior, or the bugs which 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.

NB. The perfect example of avoiding this anti-pattern is compiler development; usually, the next compiler's version is compiled with the previous compiler's version.

5. Keep a notepad

Whatever tips and tricks that were described in the previous chapters you use, it's often quite probable that you can't do *anything* to prevent the API inconsistencies start 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 backwards-incompatible minor version to fix some design flaws.

We highly recommend never doing that. Remember that the API is a multiplier of your mistakes either. What we recommend is to keep a serenity notepad — to fix the lessons learned, and not to forget to apply this knowledge when the major API version is released.

SECTION III. THE API PRODUCT

Chapter 20. 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're ‘selling’ them in the same manner, and all the principles of product management are fully applicable to them. It's quite doubtful you would be able to develop APIs well unless you conducted proper market research, learned customers' needs, and studied competitors, supply, and demand.

2. Still, APIs are *quite special products*. You're selling a possibility to make 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 exactly this question: why your customers would prefer making some actions programmatically. It's not an idle question: out of this book's author's experience, the product owners' lack of expertise in working with APIs exactly *is* the largest problem of API product development.

End users don't interact with your API directly but through the application 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 point of view, the API's target audience resembles a Maslow-like pyramid:

- users constitute the pyramid's base; they look for the fulfillment of their needs and don't think about technicalities;
- business owners form a middle level; they match users' needs against technical capabilities declared by developers and build products;
- developers make up the pyramid's apex; 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 market competitiveness in this dimension only and mainly channel the product promotion efforts to the developers' auditory.

'Stop!', the mindful reader must yell at this moment. The actual order of things is exactly the opposite:

- developers are normally a hired workforce implementing the tasks set by business owners (and even if a developer implements a project of his own, they still choose an API which fits the project best, thus being an employer of themselves);

- business leaders don't set tasks out of their sense of style or code elegance; they need some functionality being implemented — one that is needed to solve their customers' problems;
- finally, customers don't care about the technical aspects of the solution; they choose the product they need and ask for some specific functionality implemented.

So it turns out that customers are at the apex of the pyramid: it is customers you need to convince they need not *any* cup of coffee, but a cup of coffee brewed using our API (interesting question: how will we convey the knowledge which API works under the hood, and why customers should pay their money 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 into API's readability and consistency is not *that* important).

The truth, of course, lies somewhere in between. In some markets and subject areas, it is developers who make decisions (i.e. 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 meet any resistance while developing, let's say, a new mobile operating system requires million-dollar investments into promotions and strategic partnerships.

Here and after, we will describe some ‘averaged’ situations, meaning that all three pyramid levels are important: customers choosing the product which fits their needs best, business owners seeking quality guarantees and lower development costs, as well as software engineers caring about the API capabilities and the convenience of working with it.

Chapter 21. The API Business Models

Before we proceed to the API product management principles description, let us draw your attention to the issue of profits that companies providing APIs might extract from it. As we will demonstrate in the next chapters, this issue directly affects making product decisions and setting KPIs for the API team. [In brackets, we will provide examples of such models applicable to our coffee-machine API study.]

Developers = end users

The easiest and the most understandable case is that of providing a service for developers, with no end users involved. First of all, we talk about different software engineering tools: APIs of programming languages, frameworks, operating systems, UI libraries, game engines, etc; in other words, general-purpose interfaces. [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 whoever 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, an open license might be ‘contagious’, e.g. require publishing the derived code under the same license, or using the API for some purposes might be prohibited).

3. The API itself might be free, but the developer company 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, the API developer company might be attracting attention to other related programming tools and hoping to increase sales by publishing the API under a free license.

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.

1. API = the main and/or the only mean of accessing 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 platforms 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 mean 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.

2. API = a partner program

Many commercial services provide the 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 coffeeshops, 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 extended auditory and additional advertisement 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.

3. API = additional access to the service

If a company possesses some unique expertise, usually in a form of some 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 maintaining its freshness is extremely expensive, and 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 machines locations, beverages 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: the expertise owner could not physically develop all imaginable services utilizing the expertise. 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 till some conditions are met, 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 are more frequently need APIs to connect to their internal systems and integrate 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 absolutely condemn the practice of providing an external API merely as a byproduct of the internal one without any additional product and technical efforts. The main problem of 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 usually employ quite a specific technological stack, and the API is poorly optimized to work with other programming languages / operating systems / frameworks;
    - internal customers are much more familiar with the API concepts; they might take a look at the source code or talk to the API developers directly, so the learning curve is pretty flat for them;
    - documentation only covers some subset of use-cases needed by internal customers;

    - the API services ecosystem which we will describe in the corresponding chapter later usually doesn't exist.
- Any resources spent are directed to covering internal customer needs first. It means the following:
    - API development plans are totally opaque to partners, and sometimes look just absurdly with obvious problems being neglected for years;
    - technical support of external customers is financed on leftovers.

All those problems lead to having an external API that actually hurts the company's reputation, not improves it. In fact, you're providing a very bad service for a very critical and skeptical auditory. If you don't have a resource to develop the API as a product for external customers, better don't even start.

4. API = an advertisment site

In this case, we talk mostly about widgets and search engines; to display commercials direct access to end users is a must. The most typical examples of such APIs are advertisement networks APIs. However, mixed approaches do exist either — meaning that some API, usually a searching one, goes with commercial incuts. [In our coffee example, it means that the offer searching function will start promoting paid results on the search results page.]

5. API = self-advertisement and self-PR

If an API has neither explicit nor implicit monetization, it might still generate some income, increasing the company's brand awareness through displaying logos and other recognizable elements while working with the API, either native (if the API goes with UI elements) or agreed-upon ones (if partners are obliged to embed specifical 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 API, let's imagine that we're providing some totally unrelated service, 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 of such self-promotion might also differ:

- you might seek to increase the 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 coffeeshop ratings, and we're working hard on making consumers demand the coffeeshops to publish the ratings];

- you might concentrate efforts on increasing awareness among business owners [for example, for partners integrating coffee ordering widget on their websites to also pay attention to your tires catalogue];
- 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, e.g. the 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.

6. 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 to post feedback or correct a mistake 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 coffeeshops ratings or actively

through contacting business owners to convey users' requests or through finding new coffee shops that are still not integrated with the platform.]

7. Terraforming

Finally, the most altruistic approach to API product development is providing it free of charge (or as an open source and open data project) just to change the landscape. If today nobody's willing to pay for the API, we might invest in popularizing the functionality hoping to find commercial niches later (in any of the abovementioned 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.]

8. 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 re-selling some user data. You must be aware that the difference between acceptable data collecting (such as aggregating search requests to understand trends or finding promising locations for opening a coffee shop) and unacceptable ones are quite

vague, and tends to vary in time and space (e.g. some actions might be totally legal at the one side of the state border, and totally illegal at the other side), so making a decision of monetizing the API with it should be carried out with extreme caution.

The API-first approach

Last several years we see the trend of providing some functionality as an API (e.g. 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 business is ready to out-source instead of spending resources to develop internal APIs for the applications by 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 the expertise in API development; otherwise, there are high chances to make too many design mistakes.

Sometimes providing APIs is a method to ‘probe the ground’, e.g. 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 so profitable as expected, or because the API eventually becomes a competitor to the main service.)

Chapter 22. Developing a Product Vision

The above-mentioned fragmentation of the API target audience, e.g. the ‘developers — business — end users’ triad, makes API product management quite a non-trivial problem. Yes, the basics are the same: find your auditory's needs and satisfy them; the problem is that your product has several different auditories, 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 the business would solve those problems if appropriate tools existed;
- understanding what technical solutions might exist and where are the boundaries of their applicability.

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 the skip end users' problems analysis; and, by contrast, if you're providing an API to extremely valuable functionality and you're holding a close-to-monopolistic position on the market, you might actually never care about how developers love your software architecture or how

convenient your interfaces are for them, as they simply have no other choice.

In the majority of cases, we still have to deal with two-step heuristics based on either technical capabilities or business demands:

- you might first form the vision of how you might help business owners given the technical capabilities you have (heuristics step one); then, the 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 might make one heuristical ‘step right’ to outline future functionality for end users and one ‘step left’ to evaluate possible technical solutions.

As both approaches are still heuristic, the API product vision is inevitably fuzzy, and it's rather normal: if you could have got a full and clear understanding of what end-user products might be developed on top of your API, you might have developed them on your own behalf, skipping intermediary agents. It is also important to keep in mind that many APIs pass the ‘terraforming’ stage (see the previous chapter) thus preparing the ground for new markets and new types of services — so your idealistic vision of a nearby future where delivering freshly brewed coffee by drones will be a norm of life is to be refined and clarified while new companies providing new kinds of services are coming to the market. (Which in its turn will make an impact on the monetization model: detalizing 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 while making interviews and getting feedback. Software engineers will mainly report the problems they've got with the technical integrations, and rarely speak of business-related issues; meanwhile, business owners care little about code-writing inconvenience. Both will have some knowledge regarding the end users problems, but it's usually limited to the market segment the partner operates on.

If you do have an access to end users actions monitoring (see the 'API KPIs' chapter), then you might try to analyze the typical user behavior through these logs, and understand how users interact with the partners' applications. But you will need to make this analysis on a per-application basis, and try to clusterize the most common scenarios.

Checking product hypotheses

Apart from the general complexity of formulating the product vision, there are also tactical issues with checking product hypotheses. 'The Holy Grail' of product management — that is, creating a cheap (in terms of resource spent) minimal viable product (MVP) — is normally unavailable for an API product manager. The thing is that you can't easily *test* the solution even if you managed to develop an API MVP: to do so, partners are to

develop some code, e.g. invest their money; and if the outcome of the experiment is negative (e.g. the further development looks unpromising), this money will be wasted. Of course, partners will be a little bit skeptical towards such proposals. Thus a ‘cheap’ MVP should include either the compensation for partners' expenses or the budget to develop a reference implementation (e.g. a complementary application must be developed alongside the API MVP).

You might partially solve the problem by making some third-party company release the MVP (for example, in a form of an open source module published in some developer's personal repository) but then you will struggle with hypothesis validation issues as such modules might easily go unnoticed.

Another option for checking conjectures is recruiting an API provider company's services developers if they exist. 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 check only those ideas that are relevant to internal services needs.

Generally speaking, the ‘eat your own dogfood’ concept applied to APIs means that the product team should have their own test applications (i.e. ‘pet project’) on top of the APIs. Given the complexity of developing such applications, it makes sense to encourage having them, e.g. 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 product managers' typical problems: it's not enough to write a fine code, you also need to know your customer, understand their demands, formulate an attractive concept, and communicate it. In their turn, product managers will get some understanding of how exactly easy or hard is to render their product vision into life, and what problems the implementation will bring. Finally, both will benefit from taking a fresh look at the API documentation and putting themselves in the shoes of a developer who had heard about the API product for the first time and is now struggling with grasping the basics.

Chapter 23. Communicating with Developers

As we have described in the previous chapters, managing an API product requires building relations with both business partners and developers. (Ideally, with end users as well; though this option is seldom available to API providers.)

Let's start with developers. The specifics of software engineers as an auditory are the following:

- 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 certain aptitude towards, let's say, specific programming languages or mobile OS vendors; however, *affecting* those aptitudes is extremely hard and is normally not in the API vendor's power;
- in particular, developers are quite skeptical towards promotional materials and overstatements and are ready to actually check whether your claims are true;

- it is very hard to communicate to them 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 ‘influencers’ words are not very valuable to them, as no opinions are trusted if unsubstantiated;
- the Open Source and free software ideas are widespread among developers; if you try to make money out of things that must be free and/or open (for example, by proclaiming interfaces an intellectual property), you will face resistance (and views on this ‘musts’… differ).

Because of the above-mentioned specifics (first of all, 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 (StackOverflow, Experts Exchange);
- educational services (CodeAcademy, Udemy);
- technical conferences and webinars.

In all these channels, the direct advertising of your API is either problematic or impossible. (Well, strictly speaking, you may buy the banner on one of the sites advertising the advantages of your API, but we hardly doubt it will improve your relations with developers.) You need to generate some valuable and/or interesting content for them, which will improve the knowledge of your API. And this is the job for your developers: writing articles, answering questions, recording webinars, and giving pitches.

Developers do like sharing the experience, and will probably be eager to do it — during their work hours. A proper conference talk, to say nothing about an educational course, requires a lot of preparation time. Out of 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 the content review just like you do the code review.

Nothing could make the worse counter-advertising for your product than a poorly prepared pitch (as we said, the mistakes will be inevitably found and pointed to) or a badly camouflaged commercial in a form of a pitch (the reason is actually the same). Texts are 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 what idea you wanted to communicate

(and it'd rather be somehow coupled with your API's fitness for their needs).

A word on ‘evangelists’ (those are people who have some credibility in the IT community and work on promoting a technology or a tech company, being a company's contractor or even a staff member, effectively carrying out all those above-mentioned activities like blog-posting, course-preparing, conference-speaking, etc.) An evangelist is thus making the API development team exempt from the necessity of performing the tech-PR. However, we would rather advise having this expertise inside the team, as direct interaction with developers helps with forming the product vision. (That doesn't mean the evangelists are not needed at all - you might well combine these two strategies.)

Open Source

The important question which sooner or later will stand in any API vendor's way is making 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;
    - that's given your code is finely written and commented;
- you will get some additional feedback — ideally, pull requests from third-party developers;

    - and you will also get 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;
    - but it also increases risks, both from the information security point of view and from the product one, as a dissatisfied community might fork your repo 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 having all pros and cons elaborated over. 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, and also 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 auditory fragmentation

There is one very important addition to the discourse: as informational 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 a track of mastering the occupation: someone is trying to write code in addition to the basic duties, another one is being retrained now, and the third one is studying the basics of computer science on their own. Many of those non-professional developers make a direct impact on the process of selecting an API vendor — for example, small business owners who additionally seek to automate some routine tasks programmatically.

It will be more correct if we say that you're actually working for two main auditories:

- professional developers who possess a vast experience in integrating different third-party systems;
- beginners and amateurs, for whom each of those integration tasks would be completely new and unexplored territory.

This fact greatly affects everything we had discussed previously (except for, maybe, open-sourcing, as amateur developers pay little attention to it). Your pitches, webinars, lectures, etc., must somehow fit both professional and semi-professional auditories. There is a popular opinion that you actually can't satisfy both: the former seeks extensive customization capabilities (as they

usually work in big IT companies that have a specific mindset regarding those integrations) while the latter just needs the gentlest possible learning curve. We would rather disagree with that, the reasons to be discussed in the 'API Services Range' chapter.

Chapter 24. 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 side, partners are much more loyal and sometimes even enthusiastic regarding opportunities you offer (especially free ones);
- on another side, communicating the meaning of your offer to the business owners is much more complicated than conveying it to developers, as it's generally hard to explain what are the advantages of having an API (as a concept) compared to having a service for end users.

After all, working with business auditory essentially means lucidly explaining the characteristics and the advantages of the product. In that sense, API ‘sells’ just like any other kind of software.

As a rule, the farther some industry sector is from information technologies, the more enthusiastic its representatives are about your API features, and the less the chance is that this enthusiasm will be converted into a real integration. The one thing that should help the case is extensive work with the community (see the corresponding chapter) that will result in establishing a circle of freelances and outsourcers eager to help non-IT businesses with integrations. You might help develop this market by the creation of educational courses and issuing certificates

proving the bearer's skills of working with your API (or some broader layer of technology).

Market research and getting feedback from business owners work similarly. Those businesses that are far from IT usually can't formulate their demands, so you should be rather creative (and critical-minded) while analyzing the gathered data.

Chapter 25. The API Services Range

The important rule of API product management that any major API provider will soon learn formulates like that: there is no sense to ship one specific API; there is always a room for a range of products, and this range is two-dimensional.

Horizontal scaling of API services

Usually, any functionality available through an API might 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 different APIs or, vice versa, considering them as parts of one API.

Different companies employ different approaches to determining the granularity of API services, what is counted as a separate product and what is not. To some extent, this is a matter of convenience and taste judgment. Consider splitting an API into parts if:

- it makes sense for partners to integrate only one API part, e.g. some isolated subset of the API provides enough means to solve users' problems;
- API parts might 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 backwards compatibility and

introduce new major versions only when it's absolutely necessary; otherwise, maintaining a matrix which API No. 1 version is compatible with which API No. 2 version will soon become a catastrophe);
- it makes sense to set tariffs and limits for each API service independently;
- the auditory of the API segments (either developers, business owners, or end users) is not overlapping, and ‘selling’ granular API to customers is much easier than aggregated.

NB: still, those split APIs might still be a part of a united SDK, to make programmer's life easier.

Vertical scaling of API services

However, frequently it makes sense to provide several API services manipulating the same functionality. The thing is that the fragmentation of API customers across their professional skill levels results in several important implications:

- it's almost impossible in a course of a single product to create an API that will fit well both amateur developers and professional ones: the former need the maximum simplicity of implementing basic use cases, while the latter seek the ability to adapt the API to match technological stack and development paradigms, and the problems they solve usually require deep customization;

- the huge share of customers' inquiries to your customer support service will be generated by the first category of developers: it's much harder for amateurs or beginners to find answers to their questions by themselves, and they will address them to you;
- 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.

The most important conclusion here is that the only way to cover the needs of all categories of customers is to develop a range of products with different entry thresholds and requirements for developers' professional level. We might name several API sub-types, ordered from the most technically demanding to less complex ones.

1. The most advanced level is that of physical APIs and the abstractions on top of them. [In our coffee example, the collection of entities describing working with APIs of physical coffee machines, see Chapter 9 and Chapter 17.]
2. The basic level of working with product entities via formal interfaces. [In our study example, that will be HTTP API for making orders.]
3. Working with product entities might be simplified if SDKs are provided for some popular platforms that tailor API concepts according to the paradigms of those platforms (for those developers who are proficient with specific platforms only that will save a

lot of effort on dealing with formal protocols and interfaces).

4. The next simplification step is providing services for code generation. In this service, developers choose one of the pre-built integration templates, customize some options, and got a ready-to-use piece of code that might be simply copy-pasted into the application code (and might be additionally 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 a developer to place UI elements and get the working application code.]
5. Finally, this approach might be simplified even further if the service generates not 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 right on the partner's website, or describes the rules of forming the image URL that will show the most relevant offer to an end user if embedded in the partner's app.]

Ultimately, we will end up with a concept of meta-API, e.g. those high-level components will have an API of their own built on top of the basic API.

The most 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 apps.

1. The apps 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. The apps that operate base APIs will let you manipulate underlying abstraction levels — move to newer technologies or alter the way search results are presented to an end user.
3. SDKs, especially those proving UI components, provide a higher degree of control over the look and feel of partners' applications, which allows you to evolve the UI, adding new interactive elements and enriching the functionality of existing ones. [For example, if our coffee SDK contains the 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 a number of searches performed through the API, we might alter the generated code so it will show the search panel in the most convenient position in the app; as partners using code-generation services rarely make any changes in the resulting code, this will help you in reaching the goal.

5. Finally, ready-to-use components and widgets are under your full control, and you might experiment with functionality exposed through them in partners' applications just like if it was your own service. (However, it doesn't automatically mean that you might draw some profits from having this control; for example, if you're allowing inserting pictures by their direct URL, your control over this integration is rather negligible, so it's generally better to provide those kinds of integration that allow having more control over the functionality in partners' apps.)

NB. While developing a ‘vertical’ range of APIs, following the principles stated in the Chapter 14 ‘On the Iceberg's Waterline’ is crucial. You might manipulate widget content and behavior if, and only if, developers can't ‘escape the sandbox’, e.g. have direct access to low-level objects encapsulated within the widget.

In general, you should aim to have each partner service using the API service that maximizes your profit as an API vendor. Where the partner doesn't try to make some unique experience and needs just a typical solution, you would benefit from making them use widgets, which are under your full control and thus ease the API version fragmentation problem and allow for experimenting in order to reach your KPIs. Where the partner possesses some unique 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, so they might cover specific market niches and enjoy the

advantage of offering more flexibility compared to using other APIs.

Chapter 26. The 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) No. 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 the 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 twist the statistics:

- the high-level API services that are meant for straightforward integration (see the previous chapter) are significantly distorting the statistics 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 site 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 auditory 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, e.g. 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 auditory 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 the 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 auditory).

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 point out 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 data access or a 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, e.g. 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 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: imagine the partner accidentally 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 messages 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 the API services. Indeed, how could you learn how many integrations have 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 market research, you will have to either commission such a study or make your own estimations — conversely, through interviewing potential customers.

Chapter 27. 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, e.g. 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 visitor 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 amount 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, e.g. 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, e.g. some technical entity. Meanwhile, an owner is a legal body that you have the API usage agreement signed with, e.g. a legal body. if API tariffs imply some limits and/or tariffs depend 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 the 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 a 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 client 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 the API access to 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 shares 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 the 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 the 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]` Referers, and also for your own sandbox if it exists. Yes, abusers will sooner or later learn this fact and will start using 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 the 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 (e.g. 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-Forwarder-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 a fact that you can't just count unique addresses — the aggregates are to be tracked: * the cumulative number of requests by networks, e.g. 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 in this region NATs are widespread).

2. Additional means of tracking are users' unique identifiers, most notably cookies. However, most recently this method of gathering data is under attack from several sides: 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 the statistics and analyzing user behavior.

> And that impacts not only fighting all kinds of fraud but analyzing use cases as well. That's 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 28. 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 generally comprises three steps:

- identifying suspicious users;
- optionally, asking for an additional authentication factor;
- making decisions and applying the 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 Referers (actually, *any* piece of information suits if it divides users into more or less independent groups: for example, OS version or system language 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 ‘user’, we will have to make a second analytical contour to work with IP addresses, as malefactors aren't obliged to preserve cookies or other identification tokens, or will keep a pool of such tokens 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, e.g. 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 the API it might be problematic, especially if you initially neglected the ‘Stipulate restrictions’ advice. In many cases, you will have to impose this responsibility on partners (e.g. 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 any 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 backwards-compatible manner.

Other popular mechanics of identifying robots include offering a bait (‘honeypot’) or employing the execution environment checks (starting from rather trivial like executing JavaScript on the webpage and ending with sophisticated techniques of checking application integrity).

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 layered: 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 option number one 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 legal applications.

An opinion exists, which the author of this book shares, that engaging into 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 whom will be way harder than those who just naively call API endpoints with `curl`. What is even more important, in the final phase — e.g. 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 (e.g. 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, e.g. 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 Referers, 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 Referers 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 prohibiting 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 during a short period of time (usually, one minute) only, which makes working with stealing keys much more sophisticated.

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 29. Supporting customers

First of all, an important remark: when we talk about supporting API customers, we mean supporting 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 regards 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 the relevant external resource (let's say, the support service of the platform or the community forum of the programming language), or 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 inexperienced developers who use the API and ask questions; 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 hiring engineers for the first line of support.

3. Partly (or, sometimes, fully) the developer community might help with solving the amateur problems (see the corresponding 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 consider the time needed to investigate inquiries while planning the API development team work. 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 with a perspective of sorting the 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 deal with support; 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 sites that have convenient tools for that (like subscribing to specific tags).

Chapter 30. The Documentation

Regretfully, many API providers pay miserable attention to the documentation quality. 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 (e.g. 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 (e.g. 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 being unreasonably inflated. So instead of a two-word answer to the 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. This requirement somehow contradicts the simple-and-laconic principle; that's 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, e.g. comply with some standard, for example, OpenAPI.

A 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 most 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 the crucial tool to acquire and retain new API users. It's extremely important to choose the 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 the 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, i.e. the reference should contain code samples relevant to the entity being described.

3. Sandboxes

Code samples will be much more useful to developers if they are ‘live’, e.g. provided as live 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 a ‘book’ that you write to explain to the reader how to work with your API. So, a proper tutorial must follow book-writing patterns, e.g. 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 some cartographical API must explain trivia regarding geographical coordinates and working with them;
- proper API usage scenarios, e.g. the ‘happy paths’;
- proper reactions to program errors that could happen;
- detailed studies on advanced API functionality (with detailed examples).

As usual, 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.

Usually, tutorials contain a ‘Quick Start’ (‘Hello, world!’) section: the smallest possible code sample that would allow developers to build a small app atop of 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 a knowledge base

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 generally 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, e.g. 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 valid for a specific date.

Content Duplication Problems

A significant problem that harms documentation clarity is the 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 backwards compatibility, it is possible to create the 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, e.g. 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, e.g. 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 31. The Testing Environment

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 a full set of testing APIs and administrative interfaces. It means that developers will need 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:

- end user application developers 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 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’ 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 ‘manual’ way of testing usage scenarios must also 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 32. 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 the 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. Any program will once become obsolete and will be rewritten; and if during this re-developing partners need to also switch to a newer API version, it will be met with some degree of understanding. Of course, in different subject areas, this timespan differs, depending on the evolution rate of the underlying platform.

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: from one side, the former are much more likely to find undocumented features and unfixed bugs in your code; on the other side, 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 platform

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 supporting a platform *is* a backwards-incompatible change, and might lead to breaking some integration for some end users. So it's highly important to have clearly formulated policies on 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 backwards-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 through 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 developer', business', and end user's needs, and in expanding them (while maintaining the backwards compatibility) for the next two thousand years (or maybe more).