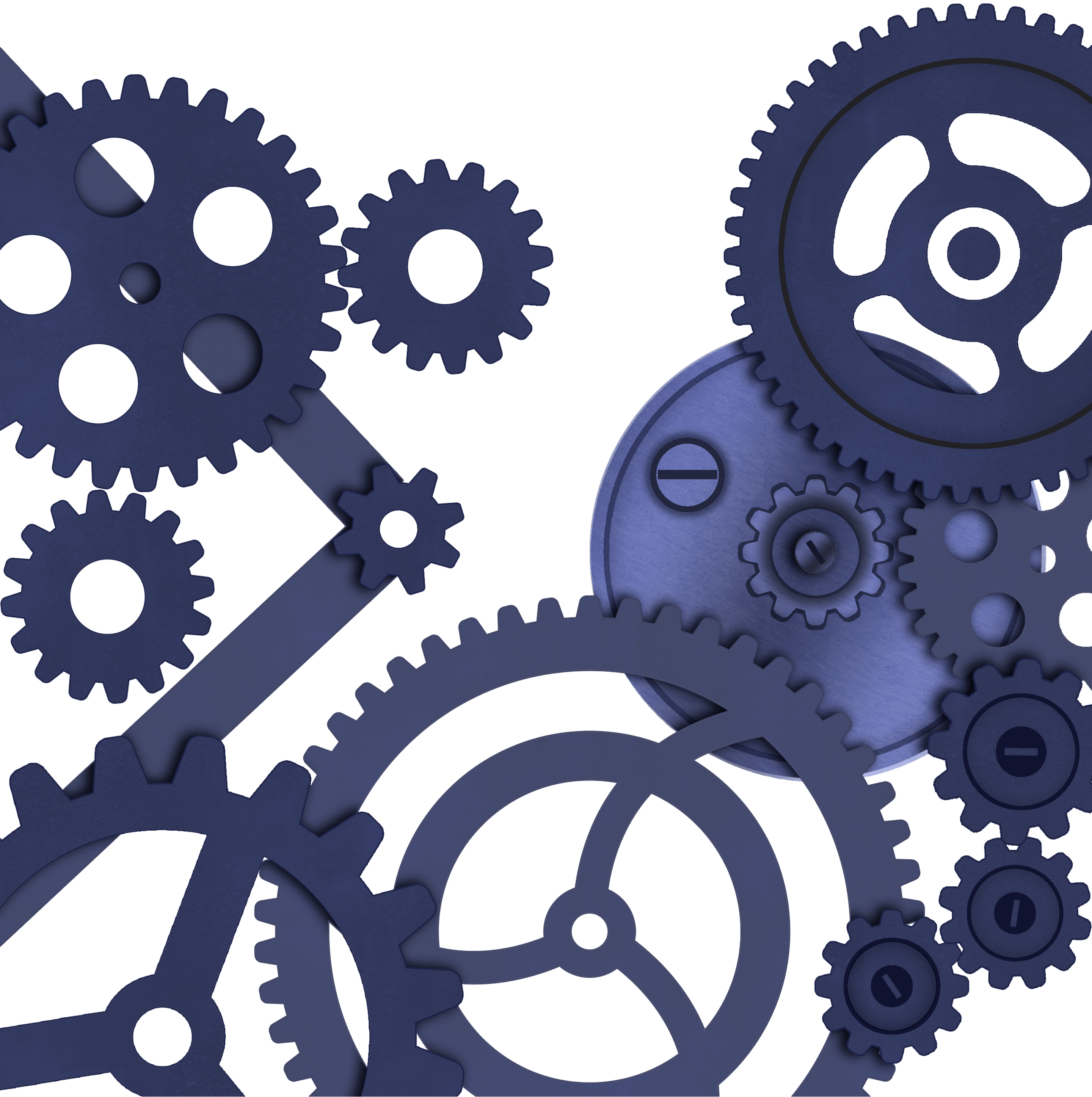
Sergey Konstantinov

The API



Sergey Konstantinov. The API.
twirl-team@yandex.ru · www.linkedin.com/in/twirl/ · www.patreon.com/yatwirl

The API-first development is one of the hottest technical topics in 2020, since many companies started to realize that API serves as a multiplicator to their opportunities—but it also amplifies the design mistakes as well.

The book is dedicated to designing APIs: how to build the architecture properly, from a high-level planning down to final interfaces.

Illustrations by Maria Konstantinova
www.instagram.com/art.mari.ka/

Source codes are available on GitHub

TABLE OF CONTENTS

INTRODUCTION

Chapter 1. On the Structure of This Book

The book you're holding in your hands comprises this Introduction and 'The API Design' Section. We will discuss designing APIs as a concept: how to build the architecture properly, from high-level planning down to final interfaces.

Two more sections are planned for the future revisions of this book: Section II will be dedicated to an API's lifecycle: how interfaces evolve over time, and how to elaborate the product to match users' needs. Finally, Section III will be more about the un-engineering sides of the API, like API marketing, organizing 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 HTTP protocol work you need the entire network stack (comprising 4-5 or even more different level protocols) work 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 API 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 being *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 to 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 on plugging each individual house to 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 is a simple part. 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 result 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 means* regarding your API).

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 developers' work harder. **Providing excess design complexities being reserved for future use makes sense only when 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 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. First number (major version) increases when backwards incompatible changes in the API are shipped.
3. Second Number (minor version) increases when new functionality is added to the API, keeping backwards compatibility intact.
4. Third number (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 as equivalent ones.

In Section II we will discuss versioning policies in more detail. In Section I, we will just use semver versions designation, specifically `v1`, `v2`, etc.

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 a 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 which, as we see it, helps to describe concepts in the most comprehensible manner. `GET /v1/orders` endpoint call could easily be replaced with `orders.get()` method call, local or remote; JSON could easily be replaced with any other data format. The meaning of assertions shouldn't change.

Let's take a look at the following example:

```
// Method description
POST /v1/bucket/{id}/some-resource
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"
}
```

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 arrow symbol →) server returns a `404 Not Founds` status code; the status might be omitted (treat it like `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.

Term 'client' here stands for an application being executed on a user's device, either native or web one. 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 we solve? It should be asked four times, each time putting an emphasis on another word.

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

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

4. What problem 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’ interface is inconvenient, why have ‘machine-to-machine’ interface?

    - Possibly, we're solving knowledge 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?

    ‘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 solving those problems specifically requires 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 you wrote ‘because that's our area of expertise’ in question 2. 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 regarding 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 being 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 abstractions hierarchy, 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 to request different cup volume when 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 `GET /v1/recipes`, but `GET /v1/orders`. 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 `GET /v1/recipes` context `volume` field means 'a volume to be prepared if no arbitrary volume is specified in `POST /v1/orders` request'; and it cannot be renamed to 'default volume' easily, we now have to live with that.

In third, the entire scheme becomes totally inoperable if different types of coffee machines produce different volumes of lungo. To introduce 'lungo volume depends on machine type' constraint we have to do quite a nasty thing: make recipes depend on coffee machine id. 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 bad. But how to make them *nice*?

Abstraction levels separation should go alongside 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;
- from 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 `POST /execute`
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, (b) in the real life, you really got something like that from vendors, and this API is quite sane, actually.

- 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. 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 is related to.

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/programs/{api_type}/{program_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 in 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 `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 next in chain handler 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:

- 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 a 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 corresponding to this level subject area terms. 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:

- 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 `rides/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, the `terminate` handler operates its internal state:
  - changes the `resolution` to `"terminated"`;
  - runs the `"discard_cup"` command.

Handling state-modifying operations like `cancel` requires more advanced abstraction levels juggling skills compared to non-modifying calls like `GET /status`. 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 a '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 provide access to the coffee machine's microcontrollers. However using such an API would be a disaster to 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, 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 user: 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 a 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 like that:

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, 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 (will be discussed below).
2. Program execution control level entities.

    - A `program` describes some general execution plan for a coffee machine. Programs could only be read.
    - The program matcher `programs/matcher` 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 program execution `programs/run` describes a single fact of running a program on a coffee machine. `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 upon 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 user
app.display(coffeeMachines);
```

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 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 been 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 `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 newly invented `offers/search` 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 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 a 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 `4xx` status codes to indicate client problems, `5xx` to indicate server problems (with the exception of a `404`, 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 `409 Conflict` code, and unresolvable ones with `400 Bad Request`.
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 `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 handler for this specific reason, a client would show `localized_message` to the user, and invoke 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&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 interfaces 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 '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, `place` and `route` could be joined in a single `location` structure, or `offer` and `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 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.

Important assertion at number 0:

0. Rules are just generalizations

Rules are not to be applied unconditionally. They are not making 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 being 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.

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
GET /orders/cancellation
```

It's quite a surprise that accessing the `cancellation` resource (what is it?) with the non-modifying `GET` method actually cancels an order.

Better:

```
// Cancels an order
POST /orders/cancel
```

Bad:

```
// Returns aggregated statistics
// since the beginning of time
GET /orders/statistics
```

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:

```
// Returns aggregated statistics
// for a specified period of time
POST /v1/orders/statistics/aggregate
{ "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 `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
`"duration": {"unit": "ms", "value": 5000}`.

One particular implication from 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 with frustration there is the ‘Serenity Notepad’ to be discussed in Section II.

3. Keep fractional numbers precision 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.

4. Entities must have concrete names

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

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

Better: `user.get_id()`.

5. Don't spare the letters

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

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

6. Naming implies typing

Field named `recipe` must be of `Recipe` type. Field named `recipe_id` must contain a recipe identifier which 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, entities of `Date` type (if such type is present) would benefit from being indicated with `_at` or `_date` postfix, i.e. `created_at`, `occurred_at`.

If 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`

7. Matching entities must have matching names and behave alike

Bad: `begin_transition` / `stop_transition`
— `begin` and `stop` doesn'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 order;
- 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.

8. Use globally unique identifiers

It's considered 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 advice 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 is there 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 reading the book on small screens convenience. Do not replicate this practice in a real-world API.

9. System state must be observable by clients

This rule could be reformulated as ‘don't make clients guess’.

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 order id and recurrently check `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
```

10. 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 'negative' prefix.

Also worth mentioning, that making mistakes in 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 one of `!beans_absence && !cup_absence` ⇔ `!(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.

11. 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:

```
POST /v1/orders
{}
→
{
  "contactless_delivery": true
}
```

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 (field isn't set). They have to write something like:

```
if (Type(order.contactless_delivery) == 'Boolean' &&
    order.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 a quite opposite manner. The same could happen if some special values (i.e. `null` or `-1`) to denote value absence are used.

The universal rule to deal with such situations is to make all new Boolean flags being false by default.

Better

```
POST /v1/orders
{}
→
{
  "force_contact_delivery": false
}
```

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

Bad:

```
// Creates a user
POST /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
POST /users
{
  …
  "spending_monthly_limit_usd": null
}
```

Better

```
POST /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.

12. Avoid implicit partial updates

Bad:

```
// Return the order state
// by its id
GET /v1/orders/123
→
{
  "order_id",
  "delivery_address",
  "client_phone_number",
  "client_phone_number_ext",
  "updated_at"
}
// Partially rewrites the order
PATCH /v1/orders/123
{ "delivery_address" }
→
{ "delivery_address" }
```

— this approach is usually chosen to lessen request and response body sizes, plus it allows to implement collaborative editing cheaply. Both these advantages are imaginary.

First, sparing bytes on semantic data is seldom needed in modern apps. Network packets sizes (MTU, Maximum Transmission Unit) are more than a kilobyte right now; shortening responses is useless while they're less than a kilobyte.

Excessive network traffic usually occurs if:

- no data pagination is provided;
- no limits on field values are set;
- binary data is transmitted (graphics, audio, video, etc.)

Transferring only a subset of fields solves none of these problems, in the best case just masks them. A more viable approach comprises:

- making separate endpoints for ‘heavy’ data;
- introducing pagination and field value length limits;
- stopping saving bytes in all other cases.

Second, shortening response sizes will backfire exactly with spoiling collaborative editing: one client won't see the changes the other client has made. Generally speaking, in 9 cases out of 10, it is better to return a full entity state from any modifying operation, sharing the format with the read-access endpoint. Actually, you should always do this unless response size affects performance.

In third, this approach might work if you need to rewrite a field's value. But how to unset the field, return its value to the default state? For example, how to *remove* `client_phone_number_ext`?

In such cases, special values are often being used, like `null`. But as we discussed above, this is a defective practice. Another variant is prohibiting non-required fields, but that would pose considerable obstacles in a way of expanding the API.

Better: one of the following two strategies might be used.

Option #1: splitting the endpoints. Editable fields are grouped and taken out as separate endpoints. This approach also matches well against the decomposition principle we discussed in the previous chapter.

```
// Return the order state
// by its id
GET /v1/orders/123
→
{
  "order_id",
  "delivery_details": {
    "address"
  },
  "client_details": {
    "phone_number",
    "phone_number_ext"
  },
  "updated_at"
}
// Fully rewrite order delivery options
PUT /v1/orders/123/delivery-details
{ "address" }
// Fully rewrite order customer data
PUT /v1/orders/123/client-details
{ "phone_number" }
```

Omitting `client_phone_number_ext` in `PUT client-details` request would be sufficient to remove it. This approach also helps to separate constant and calculated fields (`order_id` and `updated_at`) from editable ones, thus getting rid of ambiguous situations (what happens if a client tries to rewrite the `updated_at` field?). You may also return the entire `order` entity from `PUT` endpoints (however, there should be some naming convention for that).

Option 2: design a format for atomic changes.

```
POST /v1/order/changes
X-Idempotency-Token: <see next paragraph>
{
  "changes": [{
    "type": "set",
    "field": "delivery_address",
    "value": <new value>
  }, {
    "type": "unset",
    "field": "client_phone_number_ext"
  }]
}
```

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.

13. All API operations must be idempotent

Let us recall that idempotency is the following property: repeated calls to the same function with the same parameters don'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
```

The 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 `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}
{ "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 `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's mistaken, trying to make conflicting changes.

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 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 `200 OK` instead of `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 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.

14. 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
GET /v1/recipes
→
{
  "recipes": [{
    "id": "lungo",
    "volume": "200ml"
  }, {
    "id": "latte",
    "volume": "300ml"
  }]
}
// Changes recipes' parameters
PATCH /v1/recipes
{
  "changes": [{
    "id": "lungo",
    "volume": "300ml"
  }, {
    "id": "latte",
    "volume": "-1ml"
  }]
}
→ 400 Bad Request
// Re-reading the list
GET /v1/recipes
→
{
  "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 lungo volume changed because of the request, or 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:

```
PATCH /v1/recipes
{
  "changes": [{
    "recipe_id": "lungo",
    "volume": "300ml"
  }, {
    "recipe_id": "latte",
    "volume": "-1ml"
  }]
}
// You may actually return
// a ‘partial success’ status
// if the protocol allows it
→ 200 OK
{
  "changes": [{
    "change_id",
    "occurred_at",
    "recipe_id": "lungo",
    "status": "success"
  }, {
    "change_id",
    "occurred_at",
    "recipe_id": "latte",
    "status": "fail",
    "error"
  }]
}
```

Here:

- `change_id` is a unique identifier of each atomic change;
- `occurred_at` is a moment of time when the change was actually applied;
- `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 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:

```
PATCH /v1/recipes
{
  "idempotency_token",
  "changes": [{
    "recipe_id": "lungo",
    "volume": "300ml"
  }, {
    "recipe_id": "latte",
    "volume": "400ml"
  }]
}
→ 200 OK
{
  "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:

```
PATCH /v1/recipes
{
  "idempotency_token",
  "changes": [{
    "recipe_id": "lungo",
    "volume": "300ml"
  }, {
    "recipe_id": "latte",
    "volume": "400ml"
  }]
}
→ 200 OK
{
  "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.

15. Specify caching policies

Client-server interaction usually implies that network and server resources are limited, therefore caching operation results on client devices is standard practice.

So it's highly desirable to make caching options clear, if not from functions' signatures then at least from docs.

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 `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"
    }
  }
}
```

16. 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 some questions to ourselves.

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` each time produces new records set.

Better: in such unidirectional lists the pagination must use that key which 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 recordset.

Another way to organize such lists is returning a `cursor` to be used instead of `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. `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 `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, 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 which 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 solution doesn't prohibit using `offset/limit`; nothing could stop 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` 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 record order 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:

```
POST /v1/records/modified/list
{
  // Optional
  "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.

17. Errors must be informative

While writing the code developers face problems, many of them quite trivial, like invalid parameter type or some boundary violation. The more convenient are error responses your API return, the fewer is the time developers waste in 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 `"lngo"` and wrong coordinates) are obvious. But the handler checks them anyway, 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 [-90, 90] interval"
      }
    ]
  }
}
```

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

18. 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.

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 finishes, and the user is able to commit another one, prices, items availability, and other order parameters will likely need another correction.

In 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 payed 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: munimal 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.

19. 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 /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 /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.)

20. 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 user 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 the 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 a 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 #19: `localized_message` is meant for the user; the app should show it if there is no specific handler for this error exists in code. This message must be written in the user's language and formatted according to the user's location. But `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 a software development world, it usually means 'in English'.

Worth mentioning is that `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 specefied 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. 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 which 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' problem. 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, backwards compatibility maintenance period should be reconciled with subject area applications lifetime. Platform LTS periods are a decent guidance in the 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 at? 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 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, for 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 *our own* code wouldn't change, so at some point we will have to ask the clients to change *their* code.

Apart from our own 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 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, like the good old Web does, and you weren't too lazy to implement that code-on-demand (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 previous major version of the library, implementing it upon the actual version API.

2. If code-on-demand 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 three reasons:

   - developers simply don't want to update the app, its development stopped;
   - users don't wont 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 percent 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 of 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 a 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 which doesn't need 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. Otherwise you still had to deal with some distribution 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 much more granular subject area. In 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 included 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 (actually, any software they provide) as (we hope so) you are. You should be prepared 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 which is executed in some environment you can't control, and it's evolving. New versions of operating system, browsers, protocols, 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 the fragmentation, just like older apps 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 incremental progress in a form of new platforms and protocols demands changing the API, but also a vulgar fashion does. 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 with different rate, but never a zero one;
- 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 key factors of making them.

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

   That's primarily a *product* question. New major API version is to be released when the critical mass of functionality is achieved — a critical mass of features which couldn't be introduced in the previous API versions, or introducing them is too expensive. On stable markets such a situation occurs once in several years, usually. On 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 measures in years. *Practically* speaking you should look at the size of 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.

We will address these questions in more details in the next chapters. Additionally, in the Section III we will also discuss, how to communicate to customers about new versions releases and older versions support discontinuance, and how to stimulate them to adopt new API versions.

Chapter 14. On the Iceberg's Waterline

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, 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 a 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 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, 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 `createOrder` operation must be asynchronous and return the result when all replicas are synchronized, or the retry policy must be hidden inside `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 `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 that `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 `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; worst-case, partners will operate wrong statistics for an indefinite period of time until they find a mistake.

A proper decision would be, in first, documenting the event order and allowed states; in second, continuing generating "payment_approved" event before "preparing_started" (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.

Technically correct decision would be adding '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 cancels 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 haven 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 it. For example, if different coffee machines provide different functionality — for example, some of them provide 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/{partnerId}/coffee-machines-with-options
   {
     "coffee_machines": [{
       "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. From the other side, we can't know in advance which options sets 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 of 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: high-level entities must *define a context*, which other objects are to interpret. To properly design adding 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 rely on existing libraries, or design 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:

```
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 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 ‘program’ and ‘program run’ entities. Indeed, we might do it without `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 current execution status, and finish (cancel) the order. An obvious way to provide the common interface is to require these three functions being executed via a remote call, for example, 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_stop_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 either: what if we're plugging via our API not a coffee house, but a vending machine? From one side, it means that `modify` endpoint and all related stuff are simply meaningless: vending machine couldn't sprinkle cinnamon over a coffee cup, and contactless takeout requirement means nothing to it. From 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 `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 when half of the methods *isn't actually needed at all*, like requesting contactless takeout method.

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 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 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 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 field-event mechanism *isn't obligatory to both sides*. Let us remember what we sought to achieve:

- higher-level context doesn't actually know how low-level API works — and it really doesn't; it describes the changes which occurs within the context itself, and reacts only to those events which mean something to it;
- 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 which 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, raised 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 its 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 of 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, but 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 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 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 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 has 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, 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 been appeared 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 which 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;
- when the error was fixed, we've 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. 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.

4. 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, 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. to release 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.