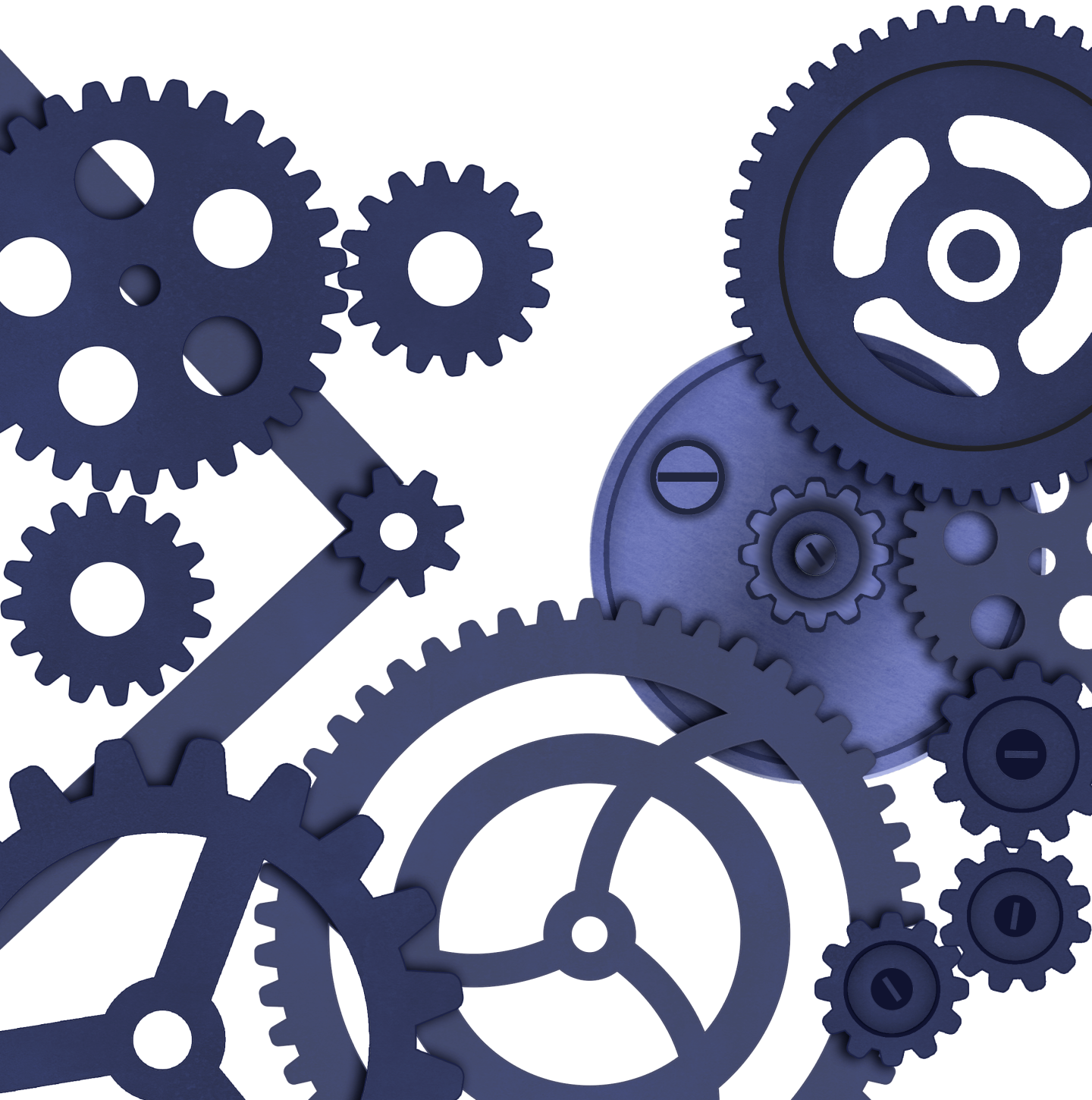
Sergey Konstantinov
The API



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

Illustrations by Maria Konstantinova


This work is licensed under a Creative Commons Attribution-NonCommercial 4.0 International License.

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 three large sections.

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

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

Finally, Section III is more about 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 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 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 a tip of an 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 of an example of a well-designed API, I usually show the picture of a Roman aqueduct:

Photo 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 you engineering structure, building their own structures upon it. From one side, you may provide an access to the water to much more people through them, not spending your time on plugging each individual house to your network. But from 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 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' problem 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 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 also 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 which 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 details: 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).

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* characteristics of your API. An obligation to maintain backwards compatibility is the crucial point where API development differs form 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 continue working.

Large companies, which occupy firm market positions, could afford implying such a 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 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 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 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 details. 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 an 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 right).

As for an entity's components, we regretfully failed to find a proper term, so we will use 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 with 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 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 of web one. Terms ‘agent’ and ‘user agent’ are synonymous to ‘client’.

Some request and response parts might be omitted if they are irrelevant to a 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-resources` 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 being 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 a real world, it would probably have very few in common with our fictional example.

Chapter 8. Defining an Application Field

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 an 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 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 an 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 a full knowledge what options they have right now and right here.
    - Possibly, we're optimizing waiting times? To save the time people waste waiting 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 for this entity is needed’, 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 inability to find nearby a toffee nut latte they long for? Do they really care about minutes they spend in lines?

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

4. Finally, will we really solve a problem? How we're going to quantify an 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 this two roles in one entity is undesirable. We will talk about the motivation to specifically provide APIs in more details 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 sort of meta-engineering: you're writing some software to allow other companies to develop software to solve users' problems. You must possess an 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 coffee brewing market. Two distinct player groups took shape: some companies provide a ‘hardware’, i.e. coffee machines; other companies have an 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 a hardware access to allow app vendors for ordering the fresh 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 larger audience, so their users may order a cup of coffee in the most efficient and convenient manner;
- abstracting an 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 to 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 we see?

1. We're preparing an `order` via the API: one (or more) cup 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 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 abstraction 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 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 abovementioned principles.

In 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.

In 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 set different 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 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 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 builtin functions, like ‘grind specified coffee volume’, ‘shed specified amount of water’, etc.; such coffee machines lack ‘preparation programs’, but provide an 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 a real life you really got something like that from vendors, and this API is quite sane, actually.

- Coffee machines with builtin functions:

```
// Returns a list of functions available
GET /functions
→
{
  "functions": [
    {
      // Operation type:
      // * set_cup
      // * grind_coffee
      // * shed_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 ‘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 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, actually, 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 event 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 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 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 corresponding sum on user's credit card, forms a request to run, and calls the execution level. First, correct execution program needs to be fetched:

```
POST /v1/programs/match
{ "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/programs/{api_type}/match`
- `POST /v1/programs/{api_type}/{program_id}/run`

This approach has some benefits, like a 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 perform 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 which 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

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 case of the second API kind, 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 from upper levels and renew its state in background, not waiting 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 necessity to spawn statuses at each level independently.

Let's now look 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 need 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 an access to coffee machine's microcontrollers. However using such an API would be a disaster to a developer, not mentioning an inability to expand it.

Separating abstraction levels is first of all a logical procedure: how we explain to ourselves and to 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 which use it.

The Data Flow

One useful exercise allowing 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 you API entities, and how it's being altered at each step.

This exercise doesn't just helps, 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 entities 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 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 a 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 which 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 a 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 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 backwards.

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, being discussed in the beginning of this chapter (forcing developers to determine actual order status on their own), 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 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 don't provide the game rules.

We will discuss data contexts in more details in the 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 other side, to check if our theoretical structures work as intended.

Chapter 10. Isolating Responsibility Areas

Basing 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 support 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 developer's shoes and try writing code. Our task is to look at the entities 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 do something like that:

- 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 which traverse all this data.

If we try writing a 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, coffee-machines);
// 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.

A necessity of adding a new endpoint for searching becomes obvious. To design such an interface we must imagine ourselves being a UX designer, 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:

```
POST /v1/coffee-machines/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 request), or a some kind of marketing offering — 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 implies different usage features and scenarios.

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

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

Helpers

Methods similar to newly invented `coffee-machines/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 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 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"
}
```

Doing so we're not only helping developers to grasp a concept of getting 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 in 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 indicates 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: 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 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 the problem? Obviously, 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:

```
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 retrieve 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 an 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 a short-term memory. Manipulating a larger number of chunks complicates things for most of 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 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_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, a 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": { "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 a 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 degrades, not improves.

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

Chapter 11. Describing Final Interfaces

When all entities, their responsibilities, and relations to each other are defined, we proceed to developing the API itself. We are to describe the objects, fields, methods, and functions nomenclature in details. 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 the 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's 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 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 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, the 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 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 with 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 XXI 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 `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 first class citizen types they provide. For examples, 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 builtin 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 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 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 abovementioned reasons, it allows counting how many entities of each types there are in the system. You competitors will be able to calculate a precise number of orders you have each day, for example.

NB: this book 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. Clients must always know full system state

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 customers'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 non-empty default value. For example:

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

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. 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 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.

In 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 then 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. More viable approach comprise:

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

In second, shortening response sizes will backfire exactly with sploiling collaborative editing: one client won't see the changes the other client have 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 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's 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 a first place, repeating request in case of network failure isn't an exception, but a norm of life.

If 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
```

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 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 find 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 conflict resolving algorithms, allowing each client to implement it independently.

The server may compare request bodies, assuming that identical `updates` values means 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 possible to make an advanced optimization. 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 logics dramatically improves user experience, being fully backwards compatible and avoiding 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 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 details 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 a 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 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 in 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 bothers about record being added or removed in the processed part of the list: they continue to iterate over the records, either getting new ones (using `newer_than`) or older ones (using `older_than`). If there is no record removal operation, clients may easily cache responses — the URL will always return the same record set.

Another way to organize such lists is 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 full dataset, for example, moving it from a ‘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 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.

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 record will simply disappear from the listing because of moving to the first page. Clients will never get those records which 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 records order at the moment we've got initial request, e.g. our server produces the entire list and stores it in 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, an 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 less 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": -180,
          "max": 180
        },
        "message":
          "'position.latitude' value must fall within [-180, 180] interval"
      }
    ]
  }
}
```

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

18. Maintain a proper error sequence

In 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?

In 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 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, than emptiness of that array is not a mistake, but a correct response. (Of course, if empty array is acceptable semantically; empty coordinates array 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 user's language and user's jurisdiction are different things. Your API working cycle must always store 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), bit 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, first day of 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 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. Author of this books 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 #12 rule: `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 user's language and formatted according to 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 which suites 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.