Sergey Konstantinov
The API

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 the API as a concept: how to build the architecture properly, from a high-level planning down to final interfaces.

Section II is dedicated to 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 the most interesting to engineers, while third section is being more relevant to both engineers and product managers. But we insist that this section is the most important for the API software developer. Since API is the product for engineers, you cannot simply pronounce non-engineering team responsible for its product planning and support. Nobody but you understands more what product features your API is capable of.

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 contemporary CISC processor commands implemented on top of microcommands API.

In other words, hundreds or even thousands of different APIs must work correctly to make possible basic actions like viewing a webpage. Contemporary 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 viaduct:

- it interconnects two areas;
- backwards compatibility being broken not a single time in two thousand years.

What differs between a Roman viaduct 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 viaduct does.

A viaduct 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 water access 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 responsibilities. **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 API design might help the developers? Quite simple: well-designed API must solve their problems in the most efficient and comprehensible manner. Distance from formulating the task to writing 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 method nomenclature, never bothering about details (especially API implementation details!); it also also very important to mention, that not only problem solution should be obvious, but also possible errors and exceptions;
- the API must be consistent; while developing new functionality (i.e. while using unknown new 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. While possessing some experience in designing APIs it's quite hard to make an API core which fails to meet obviousness, readability, and consistency criteria.

Problems begin we 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 future completely: 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 make 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 keep essential growth points mean 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 developments 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 they 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 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) 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.

Terms ‘major API version’ and ‘new API version, containing backwards incompatible changes to functionality’ are therefore to be considered as equivalent.

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).

For 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 in general will be provide 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. 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:

- 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}` should refer 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 semantic of an entity value (field, header, parameter);
- a specific JSON, containing a `some_parameter` field with `example value` value 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; status might be omitted (treat it like `200 OK` if no status is provided);
- response could possibly contain additional notable headers;
- 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 expressions like ‘`POST /v1/bucket/{id}/some-resource` method’ (or simply ‘`bucket/some-resource` method’, ‘`some-resource`’ method if no other `some-resources` are specified throughout the chapter, so there is no ambiguity) to refer to such endpoint definition.

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 API comprises four steps:

- defining an application field;
- separating abstraction levels;
- isolating responsibility areas;
- describing final interfaces.

This for-step algorithm actually builds an API from top to bottom, from common requirements and use case scenarios down to refined entity nomenclature. In fact, moving this way you will eventually get a ready-to-use API — that's why we value this approach.

It might seem that the most useful pieces of advice are given in a last chapter, but that's not true. The cost of a mistake made at certain levels differs. Fixing naming is simple; revising wrong understanding 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 emphasis on another word.

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

2. What *problem* we solve? Are we sure that 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' 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 while 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 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 is no simple answers to those questions. Ideally, you should give answers having all relevant metrics measured: how much time is wasted exactly, and what numbers we're going to achieve having this coffee machines density? Let us also stress that in real life obtaining these numbers is only possibly when 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 developing APIs, we should look at all those questions from different angle: why solving those problems specifically requires an API, not simply specialized software? In terms of our fictional example we should ask ourselves: why provide a service to developers to allow brewing coffee to end users instead of just making an app for end users?

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: your writing some software to allow other companies to develop software to solve users' problems. You must possess an expertise in both domains (API and user products) to design your API well.

As for our speculative example, let us imagine that in near future some tectonic shift happened on 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 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 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 imprtant and what goals we trying to achieve by separating them.

Let us remember that software product is a medium connecting two outstanding context, 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 take 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 elaboration 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 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 coffee-machine specified
// 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 order state.

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

This solutions intuitively looks bad, and it really is: it violates all abovementioned principles.

In first, to solve the task ‘order a lung’ a developer need to refer to the ‘recipe’ entity and learn that every recipe has an associated volume. Then they need to embrace the concept that order is ready at that particular moment when beverage volume becomes equal to reference one. This concept is simply unguessable and bears to particular sense in knowing it.

In second, we will automatically got problems if we need to vary 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 will have to performs 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 mass producing a bunch of recipes only different in volume, or introduce some recipe ‘inheritance’ to be able to specify ‘base’ recipe and just redefine the volume.

Variant II: we modify an interface, pronouncing volumes stated in recipes being just 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 arbitrary volume requested a developer will need to obtain 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 a code working with the `POST /v1/orders` handler, but forget to make corresponding changes in an 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 simply be renamed to ‘default volume’, 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 parameters). And which is even worse, developers will have to change logics 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 context which 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 physical coffee machine
    ]
  }
  …
}
```

We call this approach ‘naïve’ not because its wrong; on the contrary, that's quite 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 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 method exists. Before abstraction 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, and the only customizable options are some beverage parameters, like desired volume, syrup flavor and 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": "01",
  // 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
// 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 real life you really get 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 requested volume

with volume sensor state.

Now the picture becomes more apparent: wee 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 kinds belong to different abstraction levels themselves: first one provide a higher level API than 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 got if our interim level missed.

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 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 an 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 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 will check 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 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. stateful execution context) to run a program and controls 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
  "command_sequence_id",
  // How the exectuion 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 recipe, coffee machine model, and 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 handlers will request pieces of data they need 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 operation would work? To obtain a status the following call chain is to be performed:

- user initiate a call to `GET /v1/orders` method;
- `order` handler completes operations on its level of responsibility (for example, checks user authorization), finds `program_run_id` identifier and performs a call to `runs/{program_run_id}` endpoint;
- `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 `GET /execution/status` method of a physical coffee machine API, gets coffee volume and compares it to the reference value;
  - or invokes `GET /v1/runtimes/{runtime_id}` to obtain `state.status` and convert it to order status;
- in case of the second API kind the call chain continues: `GET /runtimes` handler invokes `GET /sensors` method of a physical coffee machine API and performs some manipulations on them, like comparing cup / ground coffee / shed water volume with those requested upon command execution and changing state and status if needed.

NB: ‘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, 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 that 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 springs through our abstraction level. In this case the call chain will look like that:

- user initiates a call to `POST /v1/orders/{id}/cancel` method;
- the method handler completes operations on its level of responsibility:
  - checks the authorization;
  - solves money issues, whether a refund is needed;
  - finds `program_run_id` identifier and calls `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 `POST /execution/cancel` method of a physical coffee machine API;
  - or invokes `POST /v1/runtimes/{id}/terminate`;
- in a second case the call chain continues, `terminate` handler operates its internal state:
  - changes `resolution` to `"terminated"`;
  - runs `"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 every abstraction level the idea of ‘order canceling’ is reformulated:

   - at `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;
   - while at a second API kind physical level a ‘cancel’ operation itself doesn't exist: ‘cancel’ means executing a `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 principes.

2. From a high-level point of view, cancelling an order is a terminal action, since no further operations are possible. From a low-level point of view processing a request continues until the cup is discard, and then the machine is to be unlocked (e.g. new runtimes creation allowed). It's a task to execution control level to couple those two states, outer (the order is canceled) and inner (the execution continues).

It might look like 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, not mentioning and 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.

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 entities 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 flows we have in our coffee API?

1. Sensor 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 ‘coffee is being shed’ or ‘cup is being set’ notions. It's our software which treats incoming sensor data and introduces new terms: if the volume of coffee or water is less than target one, then the process isn't over yet. If the target value is reached, then this synthetic status is to be switched and next command to be executed.
   It is important to note that we don't calculate new variables out from sensor

data: we need to create new data set 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 on program execution state we can (again via creating new, high-level data context) merge two different data streams from two different kinds of APIs into a single stream in a unified form of executing a beverage preparation program with logical variables like recipe, volume, and readiness status.

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

We may also traverse the tree backwards.

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

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

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

Also, if we take a 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 an order context ‘leaked’ physical data (beverage volume prepared) is injected, therefore stirring abstraction levels irreversibly;
- from other side, an order context itself is deficient: it doesn't provide new meta-variables non-existent on low levels (order status, in particular), doesn't initialize them and don't provide game rules.

We will discuss data context in more details in Section II. There we will just state that data flows and their transformations might be and must be examined as an 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

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

- user level (those entities users directly interact with and which are formulated in terms, understandable by user: orders, coffee recipes);
- program execution control level (entities responsible for transforming orders into machine commands);
- runtime level for the second API kind (entities describing command execution state machine).

We are now to define each entity's responsibility area: what's the reason in keeping this entity within our API boundaries; which operations are applicable to the entity itself (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 over the API and formulate in subject area terms what every object is. Let us remind that abstraction levels concept implies that each level is some interim subject area per se; a step we take to traverse from describing a task in first connected context terms (‘a lungo ordered by a user’) to 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 immutable entities for us, which means we could only read it.
    - A `coffee-machine` is a model of a real world device. From coffee machine description we must be able to retrieve its geographical location and the options it support (will be discussed below).
2. Program execution control level entities.

    - A ‘program’ describes some general execution plan for a coffee machine. Program could only be read.
    - A 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 each object, 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 level terms (`program_run_id` for instance). This is totally fine: connecting context 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 `GET /v1/recipes`;
- retrieve a list of all available coffee machines from `GET /v1/coffee-machines`;
- write a code to 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 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 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 cafes in the vicinity and types of coffee they offer (‘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 order 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 write, 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 purposes 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 order price question. Our search function returns some ‘offers’ with prices. But ‘price’ is volatile; coffee could cost less during ‘happy hours’, for example. Implementing this functionality, developers could make a mistake thrice:

- 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 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 displayed price in the order creation request, and return an error if it differs from the actual one. (Actually, any APY working with money *shall* do so.) But it isn't helping with first two problems, and makes user experience to degrade. Displaying actual price is always much more convenient behavior than displaying errors upon pushing 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 a user 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 specific cafes.

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

The main rule of error interface in the APIs is: 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 user readable message.

Error response content must address the following questions:

1. Which party is the problem's source, client or server?
   HTTP API traditionally employs `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?
   Invalid price is resolvable: client could obtain new price offer and create new order using it. But if the error occurred because of a client code containing a mistake, 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. 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, 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 mistake, a client is to check its kind (‘some problem with offer’), check specific error reason (‘order lifetime expired’) and send 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 a 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 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 never operate more than 10 entities at a time.

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

```
{
  "results": [
    {
      "coffee_machine_type": "drip_coffee_maker",
      "coffee_machine_brand",
      "place_name": "Кафе «Ромашка»",
      // 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 seven, and even 9. Fields are being mixed in a 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 following 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 in a some generalized object.

It is important to say that readability is achieved not only by simply 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 split into 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.