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:

- perform 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.

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 /v1/bucket/{id}/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.

The API Design

Chapter 7. 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.