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.

Of course, 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.

The API Design