An API documentation system – pragmatic advice for program internals too

Sébastien Wilmet

First version: January 2020
Last update: May 2022

Introduction

This article looks at writing an API — either for libraries or programs — and the surrounding tooling that makes the programmers's life easier.

Note that not all software projects or development platforms follow the requisite ingredients that we will outline, so there is room for improvement.

Along the way, we will give practical advice, especially for an application codebase, to document at least the most important parts, that is, to have a good overview of a software project.

1 Working with libraries

A library usually has documentation to use its API. The tools that you use to read an API documentation depends from one development platform to another, but here are some desired functionality:

- The possibility to read the documentation without opening the source code of the library.
- A dedicated application for browsing and searching API documentation, with an advanced search entry to find symbols.
- Having the API browser linked with the text editor. From the text editor, when you want to look at the documentation of the symbol located under the cursor, a simple key press jumps to the corresponding symbol in the API browser[1].

Also, don't forget what does the "I" of "API" mean. API is the acronym for Application Programming Interface. The *interface* is the external surface of the library, what users need to know about it in order to use it. If properly done, the interface hides the implementation details (it's called information hiding, or encapsulation). For the library developer, it means that everything that is *not* part of the API can be changed without incrementing the major version number of the library, to keep

[1] An alternative is to have a popup window that is shown within the text editor to show the symbol's documentation. But the author of this article prefers the first approach with the API browser window maximized, to have more space to read the documentation, and also to quickly read other related APIs.

backward compatibility. Keeping certain details hidden gives more freedom for the library developer. Conversely, library users hate when there are API breaks. So it's not a good solution to make everything public, there needs to be a design balance between what is made public and what not.

An API without documentation is usually not an API. Just providing the functions's signatures (parameter types and return value type) is often not sufficient to know how to use the functions. But in a lot of simple cases, the function signature *is* enough, with the names and the types it's self-evident. So the library developer may be tempted to write documentation only for the functions that require it. The problem is the following: for the undocumented functions, can their signatures be extracted from the source code, to show it in the API browser? If not, then the library users need to read the source code, and this can lead to other problems (read the next paragraph).

If you need to open the source code of a library to read its API, there is the risk to read its implementation and see implementation details. Then the user of the library will be tempted to take advantage of the implementation details. This can lead to bad situations. If the library developer change that implementation detail, it will break the application. Then the application developer will be angry against the library developer. Or when the library developer knows in advance that if he or she changes that implementation detail it will break certain applications, then he or she may decide to *not* make the change, which brings the library developer with less freedom (or the need to create a new major version).

All in all, the user of a library is happy when the documentation is available in an API browser, or at least when there is no need to open the library source code. If the documentation is well structured, with a table of contents and related classes grouped together, it makes the library easy to learn. When needing to open the source code, there are way too much information to have a gist of the library API as a user, it takes more time to find the information, such a library is thus harder to learn and to work with.

2 Working on an application

We have just seen that using a library can be relatively easy, if it is well done there is no need to open the source code, and the information can be found quickly thanks to the help of an API browser application.

When developing an application, the developer should be aware that it's *also* possible to write API documentation, with that documentation available in an API browser. I think it is safe to say that most applications are not written like this, at least not thoroughly, except the applications that provide a plugin system. In other words, for most applications the developer always needs to open the source code of a class in order to know how to use that class.

We don't advocate for a thorough API documentation for the application internals, but if there is the need to always open the source code of a class in order to know how to use it, this leads to the following problems:

- It takes more time to find the information, there is too much information.
- For a newcomer who has never worked on the application before, it takes much more time to get accustomed with the codebase.

When a new application is written, when the project is young, the codebase is still small and the developer knows very well the code, so the developer doesn't feel the urge to write API documentation. Then it just continues like that, the codebase grows and grows, documentation comments are added to certain functions that need it, but that's it, the developers work by opening the source code, not by reading the API of a class in an API browser.

When developing a new application, what we recommend to do more or less early on during the development phase, is to write at least *some* of the API documentation. Especially the most important parts, which include the class descriptions, to have a good overview of the codebase. A ***pragmatic way*** of handling this is to not require to document every details for every individual functions, and to put the tools in place in order to have the API documentation system ready for that software project.

Conclusion

The emphasis, especially for documenting an application codebase, is to have a good overview of the piece of software. Having a table of contents and class descriptions are the most useful.

Good API tools make the life of developers easier, and certainly when a codebase grows and is a long-running project. Think about future developers working on the codebase, and also for your own future needs.

Other ways to put it is to lower the barrier to entry for newcomers, and to have a good way to manage complexity in a software system.

Revisions:

- January 2020: initial version, with as title "Trying to convince application developers to write API documentation".
- May 2022: new and better title, rewrite the introduction, add a conclusion and emphasize on a pragmatic way when targeting an application codebase.