Documentation Generation as Information Visualization

Will Crichton
Stanford University
wcrichto@cs.stanford.edu

Abstract

Automatic documentation generation tools, or auto docs, are widely used to visualize information about APIs. However, each auto doc tool comes with its own unique representation of API information. In this paper, I use an information visualization analysis of auto docs to generate potential design principles for improving usability. Specifically, developers use auto docs as a reference by looking up relevant primitives given partial information, or leads, about its name, type, or behavior. I discuss how auto docs can better support searching and scanning on these leads.

2012 ACM Subject Classification Software and its engineering → Documentation; Human-centered computing → Information visualization

Keywords and phrases documentation generation, information visualization

Digital Object Identifier 10.4230/OASIcs.CVIT.2016.23

1 Introduction

Understanding other peoples' code is a fact of life in modern software development. With the rise of centralized package repositories, it's easier than ever to access third party libraries and frameworks. In 2019, the average JavaScript package had five direct dependencies and 86 transitive dependencies (vs. 3/7 respectively for Python) [5]. Developers cite the availability of libraries as the #1 reason to adopt a programming language [3]. Hence, making libraries easier to understand is a chief usability concern in today's programming landscape.

For the vast majority of libraries, their clients will never actually read the library source code. Instead, programmers rely on external resources: documentation, examples, tutorials, StackOverflow, blog posts, coworkers, and so on. However, these resources all have the same problem: they require a human. Someone has to write the blog, answer the StackOverflow post, and carefully craft the doc comments. Worse, that person probably has no training in technical communication, information visualization, computing education, or any discipline relevant to effectively explaining software. And most problematically, these resources can become out of date as soon as the library changes.

Automatically generated documentation, or auto docs, solve these problems by being derived directly from the source code. Tools such as Javadoc or Rustdoc take a codebase as input, and produce a wiki-like website as output. Auto docs are always up-to-date. The author of the auto doc software can (potentially) present information according to best practices. And all this comes for free — no extra effort needed by the library author! Every major programming language has a documentation generator, and recent languages consider one so vital that they ship it with the official toolset. Auto docs are arguably the most successful and widely-used software visualization tool in history.

Despite their apparent importance, auto docs are rarely the subject of research. There are few established best practices for auto docs. Each new language comes up with an entirely new format, as shown in Figure 1. Beyond aesthetics, each interface provides significantly different means of visualizing and searching information. This variation raises the question: is one format better than another? In this paper, I show how an information visualization analysis can generate potential design principles for auto docs.


© Will Crichton;
licensed under Creative Commons License CC-BY

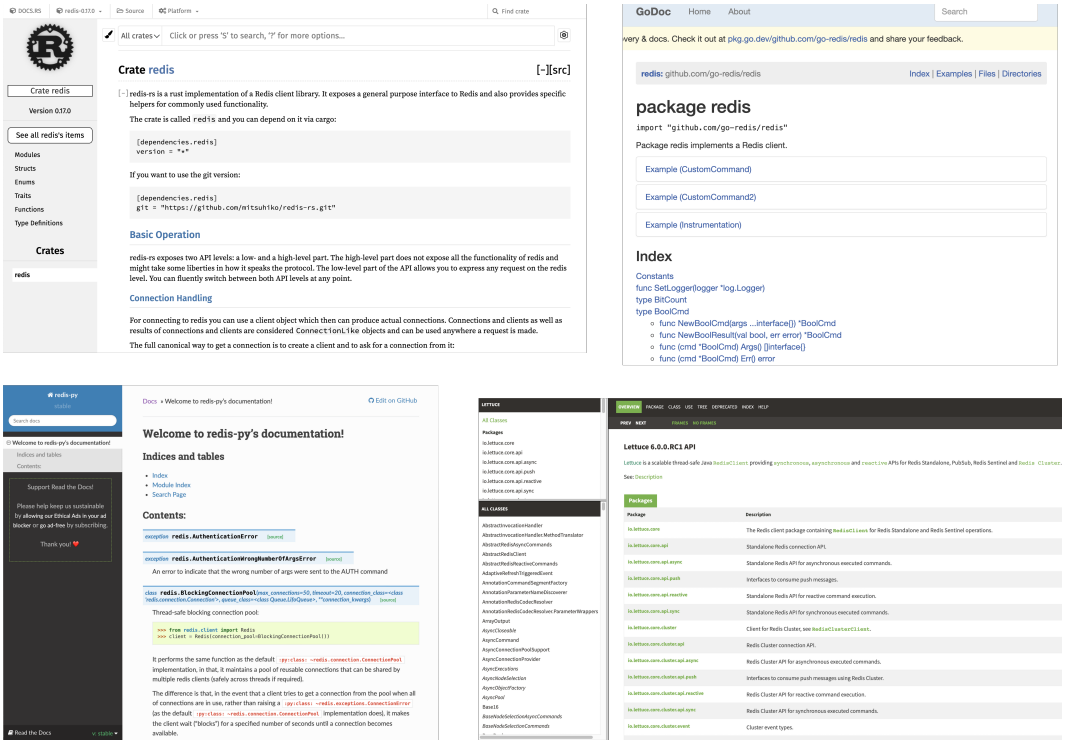

Figure 1 A screenshot of the frontpage of automatically-generated documentation for a Redis library in different languages. From top-left going clockwise: Rust (Rustdoc), Go (Godoc), Java (Javadoc), Python (Sphinx).

2 Task Analysis

To analyze auto docs from an information visualization perspective, we need to understand three things: the information, the tasks, and the representations that bridge the two. Then we can identify the gap between theory and reality to generate design ideas for future auto doc tools.

2.1 What information is being represented?

Auto docs generally present information about APIs, i.e. aspects that are externally visible to consumers of the library. APIs generally include:

- **Data structures:** structs, enums, classes
- **Functions:** constructors, methods, standalone functions
- **Interfaces:** abstract classes, traits, typeclasses
- **Hierarchies:** modules, files, namespaces

These code constructs collectively describe the primitives of an API. A developer can usually provide source-code comments further explaining their purpose, which may contain prose description or code examples. Additionally, many relations arise from the structure of these objects. For example:

- **Inputs:** a type is used as input to a function
- **Outputs:** a type is used as output of a function

- **Contains:** a type is a field of another type
- **Inherits:** a class inherits from another class
- **Implements:** a type implements an interface

Besides the explicit hierarchies within the language (e.g. namespaces), relations implicitly define hierarchies amongst API primitives. For example, the C++ auto docs tool Doxygen will automatically generate an inheritance diagram of all classes in a codebase. Rust's auto docs will also show all the types that implement a trait on the trait's generated page.

2.2 What tasks use this information?

Auto docs work best as a reference. That is to say, auto docs provide detailed information about individual constructs within an API. By contrast, tutorials are more effective at showing how multiple API constructs can be combined to accomplish a larger task. Therefore we will focus on tasks that developers have specifically for reference information.

The key feature of a reference is its ability to guide a reader with partial information about their object of interest. For example, if a person remembers the first letter of a particular word in a textbook, they can use a glossary to find the relevant word and corresponding pages. More generally, a person starts with a lead (e.g. a letter), and a reference provides some level of guidance in converting the lead into more information. For auto docs, the question is then: what leads do developers use when seeking information? Here are some example scenarios:

- A developer wants to turn a list into a set. Their lead is *functions that return the Set type and possibly take the List type as input.*
- A developer wants to find an element of a list that matches a predicate. Their lead is *the natural language keywords "list", "find", and "predicate".*
- A developer wants to know what immutable operations exist for lists. Their lead is *methods on the List type that have a read-only modifier.*

If you think about an API's information as a relational database, each lead is like a query on the database. The leads involve properties of atomic objects, e.g. whether a method's name or description contains keywords. They also involve relations between objects, such as methods that take a type as input.

2.3 What is the best information representation for each task?

Consider a developer with a lead of natural language keywords, e.g. the "find" and "predicate" example. They might adopt a range of strategies to forage for this information, for example in this order:

1. **Search engine:** they go to Google and type "<language> list find predicate". No relevant results show up.
2. **Browser search:** they go to the auto doc page for the List data structure. They Ctrl+F in the browser for each keyword, but don't find a relevant method.
3. **Scanning:** on the auto doc page, they scan through the list of method names and description. Realizing the method was called "indexOf" instead of "find", they locate the appropriate method.

Figure 2 Prototype of a scanning-oriented interface for Rust's auto docs. By displaying methods in a table and grouping by the first argument, developers can quickly search for relevant methods.

These examples highlights two key needs of a developer following a lead. First, they need to be able to encode the lead into a search. Second, if the search fails, they need a visualization of many possibly relevant objects which can be manually searched. Then the developer can make fuzzier associations between their lead and the provided information, e.g. the semantic relationship between "indexOf" and "find".

2.4 Where do current information representations fall short?

For search, most auto docs tools support (at best) natural language keywords. But many relational leads are difficult to encode in a query. "All functions that return a list" has a formal representation: the data type `* -> _ list` where `*` means any type, and `_ list` means a list of any kind of element. Notably, Haskell has a type-based search engine, Hoogle [1], which enables these kinds of queries. But no other language or auto doc tool supports these queries.

For scanning, auto docs tools have two issues: first, scanning requires an initial filter or anchor to limit the set of possible matches. For most tools today, functions are anchored around their class. That means a developer can easily get a webpage of all the methods on the List class, but as mentioned above, they cannot easily get a page of all methods that return a List. Even within a class's method list, it can be valuable to filter. For example, Rust's documentation for `Vec<T>` [2] lists well over 100 methods. The docs provides no way to e.g. filter for read-only methods that take `&self` as input.

Second, auto doc tools do not necessarily provide efficient information representations to facilitate rapid scanning. Most auto docs today present methods in a one-dimensional list. However, a developer may more easily scan a large number of method names if they can all be on screen at once, e.g. in a two-dimensional table like the prototype in Figure 2. Alternatively, organizing methods hierarchically could turn a linear search into logarithmic.

For example, a the methods of a stateful class could be grouped around which states they apply to [4].

3 Conclusion

My goal in this paper is to provide an initial framework for generating ideas about improving auto docs. Like many programming tools, auto docs are widely used, but have yet to be examined critically from a human-centered perspective. As Bret Victor pointed out in his article "Learnable Programming" [6], an API resource should "dump the parts bucket onto the floor." However, developers shouldn't have to spend hours searching through a sea of parts. With a careful understanding of what developers are looking for and what leads they use, we can build better tools for searching and visualizing API information.

References

1 Hoogle, 2020. URL: `https://hoogle.haskell.org/`.

2 std::vec::Vec - Rust, 2020. URL: `https://doc.rust-lang.org/std/vec/struct.Vec.html`.

3 Leo A Meyerovich and Ariel S Rabkin. Empirical analysis of programming language adoption. In *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*, pages 1–18, 2013.

4 Joshua Sunshine, James D Herbsleb, and Jonathan Aldrich. Structuring documentation to support state search: A laboratory experiment about protocol programming. In *European Conference on Object-Oriented Programming*, pages 157–181. Springer, 2014.

5 Ruturaj K Vaidya, Lorenzo De Carli, Drew Davidson, and Vaibhav Rastogi. Security issues in language-based sofware ecosystems. *arXiv preprint arXiv:1903.02613*, 2019.

6 Bret Victor. Learnable programming, 2012. URL: `http://worrydream.com/LearnableProgramming/`.