DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Information Systems

Improving the Quality of OpenAPI Specifications Using TypeScript Types and Annotations

Wolfgang Hobmaier

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Information Systems

Improving the Quality of OpenAPI Specifications Using TypeScript Types and Annotations

Verbesserung der Qualität von OpenAPI Spezifikationen mit Typescript Typen und Annotationen

Author:	Wolfgang Hobmaier
Supervisor:	Prof. Dr. Florian Matthes
Advisor:	Gloria Bondel
Submission Date:	2020-07-15

I confirm that this bachelor's thesis in information systems is my own work and I have documented all sources and material used.

Munich, 2020-07-15 Wolfgang Hobmaier

Acknowledgments

First of all, I want to thank the Software Engineering for Business Information Systems (sebis) chair at the TUM for enabling the creation of this thesis. Especially towards my advisor M.Sc. Gloria Bondel, I want to express my great gratitude for her help and supervision. Without her openness towards my proposal and her support throughout, this thesis would would never have come about.
I also want to thank my supervisor Prof. Dr. Florian Matthes for his valuable, constructive input especially early on while shaping the scope of this work, helpful discussions and allowing me to present this research.
I would also like to thank my employer AND Solution GmbH for providing an environment to test my assumptions and receive valuable feedback even on beta releases, especially Thomas Bayr, whose support was invaluable.
I also want to thank the OpenAPI and tsoa communities, who assisted with questions and sent contributions improving the project as a whole. A special thanks hereby goes to Phil Sturgeon for his expert feedback and productive input. Finally, I want to thank my family for their relentless and enduring support, especially during the most stressful times.

Abstract

The rise of enterprise APIs and the API economy have increased the demand for well documented APIs on the internet.Practices like microservices require well specified Private APIs in order to facilitate communication in complex application landscapes across language and system boundaries. In this area, the OpenAPI Specification has emerged as the most commonly used format to provide a standardized, language-agnostic format to describe Web APIs [Sma19]. Even when documentation is an explicit requirement, many Web APIs don't appear to meet these expectations in practice [Hos+18]. When asked about issues with regards to providing documentation, developers commonly cite the need for quick release cycles, lack of time and tooling [Sma19].

In order to provide high quality OpenAPI Specification documents, we propose a tighter integration between code and API descriptions by automating the process of creating API descriptions from code and therefore reducing the cost associated with providing high quality API descriptions. In this thesis, we compare existing approaches to documentation generation in typed systems and suggest a novel approach merging elements of existing approaches in literature with a less researched approach (Abstract Syntax Tree Parsing) and a novel approach which involves integration with a Type Checker.

Early evaluations with 2 participants show that OpenAPI specification documents automatically generated by our approach can increase the quality compared to state-of-the-art approaches, while reducing the amount of time needed for documentation annotations.

While our approach increases coupling between the documentation and source code and therefore reducing the applicability for different languages and frameworks, that coupling enables us to limit the amount of outdated or incorrect information in the generated API description. By depending on a well established standard, tooling developed for the OpenAPI ecosystem can enable additional quality assurance, including consistency and backwards compatibility.

Contents

1. Introduction

1.1. Motivation

IT Infrastructure is evolving. The popularity of the internet has moved applications "into the cloud", enabling interconnected and cross-company collaboration to solve complex business tasks. The shift from monolithic architectures to microservices has further aligned the communication mechanisms between company-internal business processes and cross-company collaboration.

"As the connective tissue linking ecosystems of technologies and organizations, APIs allow businesses to monetize data, forge profitable partnerships, and open new pathways for innovation and growth." [Iye+17]

The shift towards an open, cloud-based API economy has significantly increased the amount of Web APIs available and enabled new opportunities for value creation.

According to a survey conducted by Cloud Elements, "API integration continues to be critical to business strategy with 84% of respondents reporting that it's critical or very critical."[Clo20]

In order to succeed in an increasingly competitive API economy, or to succeed with a complex microservice landscape, high quality documentation is vital.

> "Clear, easy-to-access, and easy-to-grasp documentation is a prerequisite for API success. Documentation quickly becomes stale and out-of-date. When it does, users fail to derive value from it and, worse yet, lose trust in the company." [Fat19]

Unfortunately, high quality documentation for Web APIs is not a one time achievement, but a continuous, time consuming process. Documentation takes time, effort and buy-in from all parties involved (see Fig. 1.1), while numerous pitfalls or anti-patterns can hopefully be avoided.

> "Like cheap wine, long paper documentation ages rapidly and leaves you with a bad headache if you try to use it a year after it was created." [Adz11]

It's therefore not surprising that a large-scale study conducted by Aghajani et. al. "empirically confirms and complements previous research findings (and common

Figure 1.1.: The biggest obstacles to providing up-to-date API documentation [Sma19]

sense): Developers (and users) prefer documentation that is correct, complete, up to date, usable, maintainable, readable and useful." [Agh+19]. The most commonly observed issues during the study were completeness, up-to-dateness and correctness.

These documentation issues also apply to Web APIs. According to Maleshkova et. al, two thirds of the APIs do not state the data-type of the input and 40% of the APIs do not state the HTTP method, which characterize fundamental aspects of a HTTP Message. [MPD10]

In order to reduce the cost of creation and allow both humans and machines (with regards to tooling) to work with the same interface definition, companies around the world move towards describing their Web APIs in a standardized format. By relying on a standardized format for description Web API capabilities, integration with the infrastructure can be simplified, for example routing in gateways or EAM tooling. The most popular one of these standards is the OpenAPI Specification (OAS).

However, while reducing potential for certain mistakes through OpenAPI based validation, many OAS documents still do not provide entirely reliable documentation of the API: In a study conducted by Hosano et. al, of 67 publicly available endpoints with OpenAPI specification documents available, almost half of them were not correct. The most common discrepancies between implementation and specification were undocumented, dynamic or unreturned keys, and type mismatches. [Hos+18]

As a remedy for agile development teams, the notion of Living Documentation

was proposed by Martraire [Mar19], which incorporates documentation as continuous knowledge sharing alongside the continuous development practices of agile development teams.

> "Even in software development projects that claim to be agile, deciding what to build and doing the coding, testing, and preparing documentation are too often separate activities [...]. Separate activities induce a lot of waste and lost opportunities. Basically, the same knowledge is manipulated during each activity, but in different forms and in different artifacts—and probably with some amount of duplication. In addition, this "same" knowledge can evolve during the process itself, which may cause inconsistencies." [Mar19]

To minimize the potential for inconsistencies between description and implementation, we therefore propose a new approach that allows developers to use types and annotations to pre-describe the API in code and then automatically convert this representation into formal OpenAPI specification documents. Thus it is still possible to benefit from the advantages of an API design first strategy (such as early customer feedback and collaboration on the API definition via collaboration platforms or mocking tools), but the type system and the web framework ensure that the API description and actual implementation do not diverge, as the code remains the single source of truth.

1.2. Objective

The objective of this thesis is to outline and evaluate possible approaches to increase the of quality OpenAPI specification documents for Web APIs through abstraction and reuse of existing type definitions and framework code needed to power Web APIs, thereby decreasing the overhead introduced by seperating the activities of modeling, implementation and documentation using different tools as much as possible. While the major focus of our work is to enable accurate, correct and complete and up-to-date API Reference documentation, we intend to provide a clear path towards increased usability of the API. By relying on the existing ecosystem of standards and popular tools (HTTP, OpenAPI, JSON, JSON Schema, TypeScript), we intend to enable further automation at later stages of the API Lifecycle, notably by leveraging linters (to enforce consistent error messages, descriptions, examples and grouping of concepts) and Software development kit (SDK) generation to provide increased API usability, especially when generating statically checked SDKs [End+14] [Wit+17]. To provide additional incentives for developers, our approach removes the need for runtime validation.

1.3. Research Questions

In this section, three main research questions (RQ) will be outlined to serve as a overarching framework for this thesis.

RQ1: What are required elements of good API Reference Documentation for Web APIs?
In order to automate documentation, it is vital to define elements of good API Reference Documentation for Web APIs. Based on existing literature identifying elements of Documentation, adapt these suggestions for Web APIs and define the scope of a good Web API Reference Documentation in the context of this thesis. The goal of this research question is to define a comprehensive list of requirements good API Reference Documentation for Web APIs should fulfil, and how and where the OpenAPI specification makes it possible to formally define the knowledge needed to meet the requirements.

RQ2: What are possible approaches to ensure correct, complete and consistent API Reference Documentation of Web APIs?
Building on top of the requirements elicited in RQ1, we compare existing approaches to automated documentation from source code for traditional APIs with regards to their viability for Web APIs. We study where certain knowledge patterns can be found in source code while optimizing for correctness, completeness and usability (developer experience) in different scenarios and outline limitations of each approach. The result will be a mapping from knowledge sources to requirements and an overview, which technique is suited to provide extract the knowledge from these sources.

RQ3: Would developers use a OpenAPI driven framework?
Even if our steps towards higher quality API documentation may yield better documentation results, this question aims to put possible improvements into context. The best documentation approach likely would not be adopted if the developer experience suffers disproportionately, therefore we developed 3 hypotheses to question the impact regarding time, quality of documentation and developer experience:

- The OpenAPI-aware approach to development does decrease time spent on development
- Developers prefer our approach
- The quality of the OAS document improves

1.4. Research Approach

This thesis will apply a Design Science Research approach [Hev+04], as outlined in Fig.1.2. This approach is centered around development and evaluation of a new artifact informed by an existing knowledge base, which will be used to build a foundation of vocabulary and theories that can be used to define concrete requirements developed as part of answering RQ1. Based on these scientifically grounded, concrete requirements, we use RQ2 to argue which techniques are best suited to meets the requirements. This knowledge base will afterwards be applied to develop artifacts which can be used to address the demand for high quality documentation as a business need. Thereupon, we can assess this approach by building an implementation of this approach. By reapplying these developed artifacts in the appropriate environment during our evaluation, conclusions about viability in practice and future work can be drawn.

1.5. Outline

In chapter 2, fundamental knowledge will be introduced by defining key terminology used throughout this thesis, such as HTTP, REST APIs, JSON and the JSON Schema definitinon language, the OpenAPI Specification and "high quality API documentation". In chapter 3, we will show related work in the area of automated documentation generation. In chapter 4, we will present the results of our literature review of RQ1, whereas in chapter 5, we will determine where we can translate the knowledge patterns of source code to cover the requirements elicited. Based on our rationale as to which approach can be used to most accurately cover the requirements, chapter 6 will present the techniques and implementation chosen. Hereinafter, chapter 7 will present the results of an evaluation of the tool developed in comparison to other tools already used to generate API descriptions using the OpenAPI Specification with regards to cost and quality. We survey users of the tooling built to generate OpenAPI specification documents and investigate the effects on correctness and API developer experience. Based on our results, we determine potential effects on API consumer experience and time spent on documentation. Finally, chapter 8 will briefly summarize our findings, detail limitations of our approach and lay out future work.

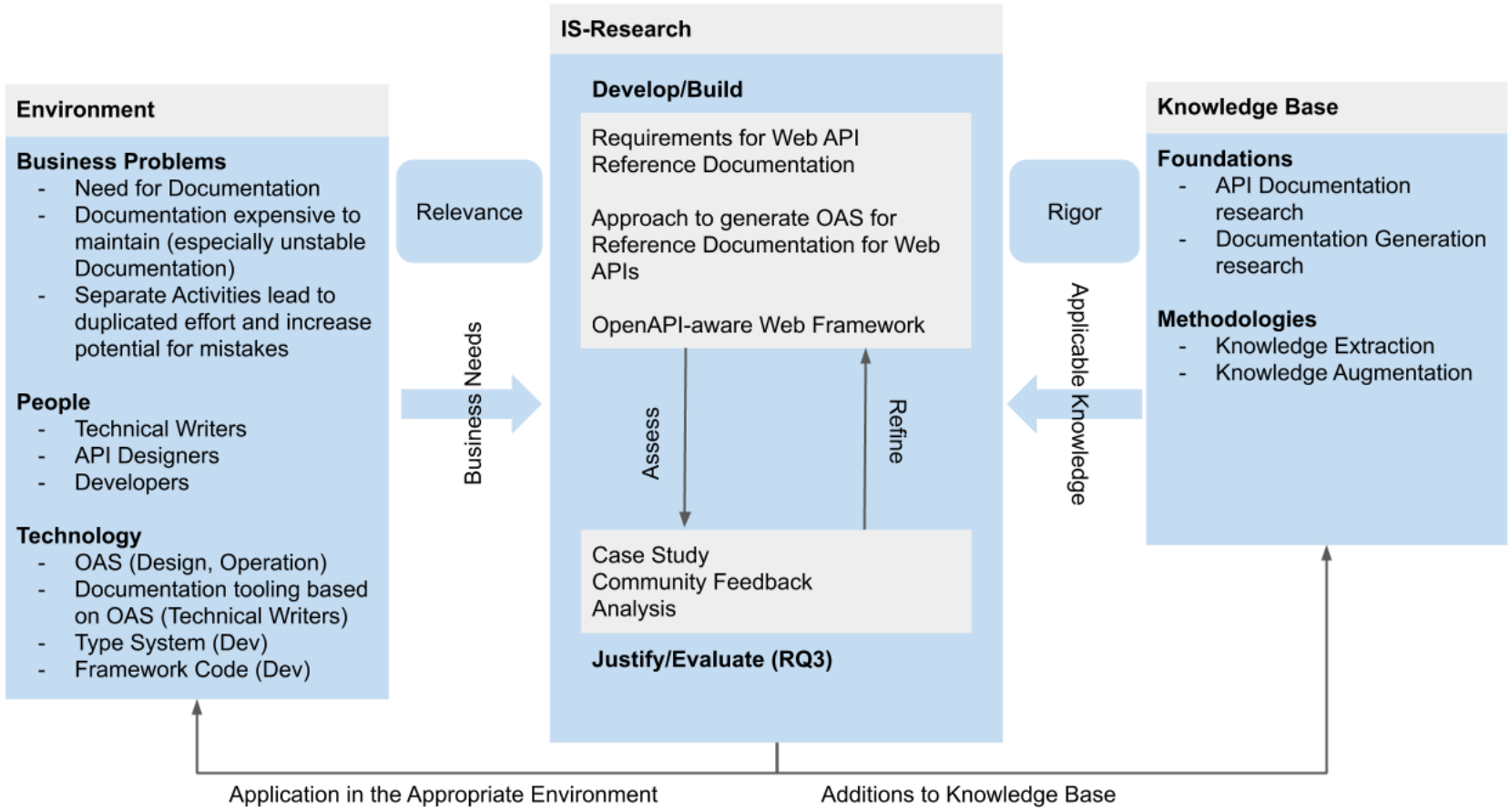


Figure 1.2.: Design Science Approach [GH13]

2. Foundations

This chapter outlines the fundamental concepts and assumptions which setup the context of the thesis. As design science always happens within an environment of existing knowledge, existing practices and a certain state of the art, we intend to provide a better understanding of this environment for the development of Web APIs.

2.1. HTTP

In order to better understand the documentation needs for Web APIs, a sufficient understanding of the protocol used for communication is required. Thus, this section will describe the mechanics of the Hypertext Transfer Protocol underlying the communication that is subject to documentation.

The Hypertext Transfer Protocol (HTTP) is the common language of the modern internet [Net11]. HTTP is a "stateless application-level request/response"[FR14a] protocol, based on TCP/IP (therefore providing certain guarantees around data-transmission). The goal of HTTP is to enable the exchange of documents, also called resources, between applications over the Web. It is so commonly used in Web servers that the terms HTTP server and web server are often used synonymously.

HTTP is specified in several RFCs:

- RFC 2616: Hypertext Transfer Protocol – HTTP/1.1 [1], the initial HTTP standard, obsoleted by 7230ff
- RFC 7230, HTTP/1.1: Message Syntax and Routing [2]
- RFC 7231, HTTP/1.1: Semantics and Content[3]
- RFC 7232, HTTP/1.1: Conditional Requests[4]

[1] `https://tools.ietf.org/html/rfc2616`
[2] `https://tools.ietf.org/html/rfc7230`
[3] `https://tools.ietf.org/html/rfc7231`
[4] `https://tools.ietf.org/html/rfc7232`

- RFC 7233, HTTP/1.1: Range Requests[5]
- RFC 7234, HTTP/1.1: Caching[6]
- RFC 7235, HTTP/1.1: Authentication[7]
- RFC 7540, HTTP/2: Hypertext Transfer Protocol Version 2[8]
- RFC 5789, PATCH Method for HTTP [9]

HTTP is used as a message based client/server computing model. Each message can be either a HTTP Request the client sends to the server, or a HTTP Response, which the server sends back to the client to service the Request. While the HTTP messages are embedded into a binary structure, called frame in HTTP 2, the semantics of the message are unchanged. [10].

"The target of an HTTP request is called a *resource*. HTTP does not limit the nature of a resource; it merely defines an interface that might be used to interact with resources. Each resource is identified by a Uniform Resource Identifier (URI), as described in Section 2.7 of RFC7230."[FR14a]

A HTTP Request therefore is a Tuple consisting of the HTTP start line containing the HTTP *Protocol Version*, a *Method*, a *Uniform Resource Identifier (URI)*, a field of *Headers* and a *Body*[11]. An example of a HTTP Request/Response is shown in Fig. 2.1.

The HTTP Version is denoted by `HTTP/<major>.<minor>`. The most common versions are 1.1 and 2.0. As both versions use the same semantics and message format, adherence to these either one of these versions of the HTTP specification will be assumed for the remainder of this thesis.

The HTTP standard defines 9 types of Methods (RFC 7231 + RFC 5789, section 2):

- GET: Transfer a current representation of the target resource.
- HEAD: Same as GET, but only transfer the status line and header section.

[5] `https://tools.ietf.org/html/rfc7233`
[6] `https://tools.ietf.org/html/rfc7234`
[7] `https://tools.ietf.org/html/rfc7235`
[8] `https://tools.ietf.org/html/rfc7540`
[9] `https://tools.ietf.org/html/rfc5789`
[10] `https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview#HTTP_Messages`
[11] Only allowed for certain *Methods*

- POST: Perform resource-specific processing on the request payload.
- PUT: Replace all current representations of the target resource with the request payload.
- DELETE: Remove all current representations of the target resource.
- CONNECT: Establish a tunnel to the server identified by the target resource.
- OPTIONS: Describe the communication options for the target resource.
- TRACE: Perform a message loop-back test along the path to the target resource.
- PATCH: Apply partial modifications to a resource.

In order to request resources by name, HTTP uses Uniform Resource Identifiers (URIs) as defined in RFC3986[12]. URI references are "used to target requests, indicate redirects, and define relationships"[FR14a].

In the context of Web APIs, a certain format of URIs is particularly common: One fixed base URL per API, which is comprised of a scheme (only "http" or "https") and a a static authority, most commonly a domain (i.e. "api.example.com") which serves resources addresses by paths (i.e. "/products"). Additionally, each URL may contain query parameters (i.e. "q=glass").
The resulting URL schema therefor may be denoted as

`http(s)://<domain><path><query>`

in our example:

`https://api.example.com/products?q=glass`

After the request start line including *Method*, the *URI* and the *Protocol Version*, *Headers* (zero or more *Header* fields) can be present: "Each header field consists of a case-insensitive field name followed by a colon (":"), optional leading whitespace, the field value, and optional trailing whitespace." [FR14a]
Headers usually contain request metadata such as content negotiation headers to inform the server about the format of the response, language information or authorization information such as API keys or JWT Tokens[13] etc.

[12] `https://tools.ietf.org/html/rfc3986`
[13] `https://jwt.io/`

The *Response* returned as a reply to a *Request* is a Tuple of the *Protocol Version*, a *Status Code*, the corresponding *Status message*, *Headers* (similar to the request headers), and, depending on the *Status Code*, a *Body*.

"The status-code element is a three-digit integer code giving the result of the attempt to understand and satisfy the request." [FR14b]
The first digit of each code describes the general class of status (1xx - Informational, 2xx - Successful, 3xx - Multiple Choices, 4xx Error, 5xx - Server Error) [Gou+02].
An exhaustive list of status codes defined in the HTTP specification and their meanings can be found in RFC 7231 Section 6 [14].

The main content of HTTP Messages is delivered via a body. In order for a client and a server to agree on a common format used, the format of the body is ususally determined via content negotiation. For detailed reasoning on why the main focus of this thesis is data exchange via JSON, (MIME type "application/json"), please refer to Section 2.3, where the format of a JSON Document will be described.

2.2. REST APIs

One of the key terms in the field of Web APIs is representational state transfer (REST). REST is an architectural pattern applied to create web services via HTTP. The term was initially used by Fieldings[Fie00] to describe a reusable, predictable way to implement client-server communication, where the client initially does not require any knowledge about the application in advance.

2.2.1. REST Constraints

By defining several constraints on top of HTTP, REST acts as a form of implicit documentation by convention:

- Client-server architecture: REST applications have a server that provides an API which can be accessed, regardless of the communication protocol. For Web APIs, REST is commonly applied to the HTTP protocol. The concrete implications of applying REST to HTTP will be outlined later. "Separation of concerns is the principle behind the client-server constraints. By separating the user interface concerns from the data storage concerns, we improve the portability of the user interface across multiple platforms and improve scalability by simplifying the server components. Perhaps most significant to the Web, however, is that the separation allows the components to evolve independently, thus supporting the Internet-scale requirement of multiple organizational domains"[Fie00].

[14] `https://tools.ietf.org/html/rfc7231#section-6`

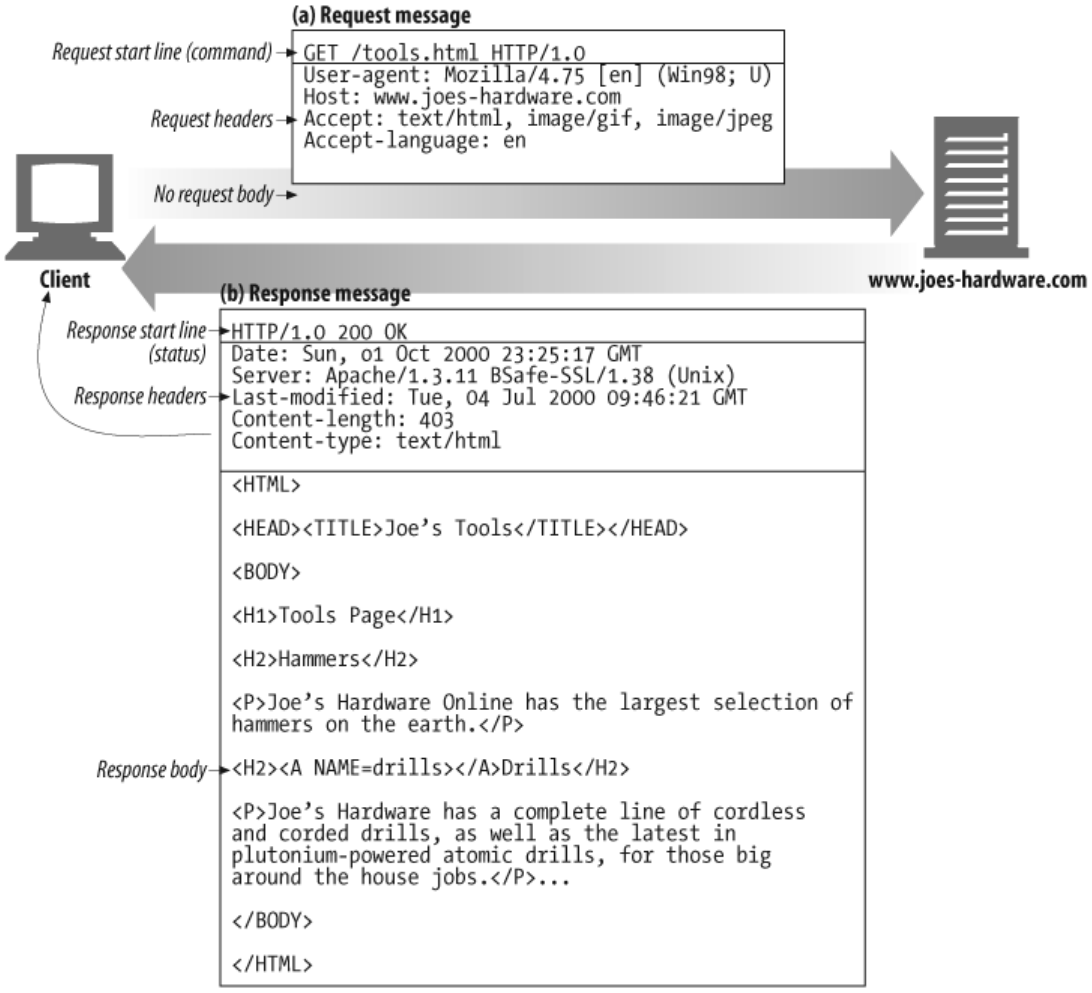


Figure 2.1.: The HTTP Message format [Gou+02]

- Stateless communication: Servers don't maintain any client state. Therefore, all client state required to fulfill the request must be provided on every call. "This constraint induces the properties of visibility, reliability, and scalability." [Fie00]

- Cacheable: The server indicates (via response headers) the cacheability of the response. "In order to improve network efficiency, we add cache constraints to form the client-cache-stateless-server style. Cache constraints require that the data within a response to a request be implicitly or explicitly labeled as cacheable or non-cacheable. If a response is cacheable, then a client cache is given the right to reuse that response data for later, equivalent requests."[Fie00] While the immediate advantage is improved performance by reducing the communication overhead, this constraint also increases the complexity due to the possibility of stale records being kept in cache.

- Uniform interface: REST outlines a consistent resource and endpoint naming: "The central feature that distinguishes the REST architectural style from other network-based styles is its emphasis on a uniform interface between components. By applying the software engineering principle of generality to the component interface, the overall system architecture is simplified and the visibility of interactions is improved. Implementations are decoupled from the services they provide, which encourages independent evolvability"[Fie00]. This decoupling of business logic from the services and data representations offered via the API allows both parts to evolve independently from each other. The cost of this decoupling is a decrease in efficiency as the uniform interface, subject to the principle of generality, is less efficitent (in terms of bytes transferred) than a specialized form which adapts to an application's need, therefore leading to *overfetching* of data. "The REST interface is designed to be efficient for large-grain hypermedia data transfer, optimizing for the common case of the Web, but resulting in an interface that is not optimal for other forms of architectural interaction"[Fie00].

- Layered System: In order to address scalability and encapsulation, REST adds a layered system constraint. As a client, this means that there is only one layer to communicate with, while deeper layers within the hierarchy are transparent. "The layered system style allows an architecture to be composed of hierarchical layers by constraining component behavior such that each component cannot "see" beyond the immediate layer with which they are interacting."[Fie00] This constraint is meant to decrease the complexity for the client, while allowing the addition of intermediaries (like (reverse-) proxies, gateways, firewalls, caches, load balancers, routing to legacy components etc.). "By restricting knowledge of the system to a single layer, we place a bound on the overall system complexity

and promote substrate independence."[Fie00]

- Code-on-demand: "The final addition to our constraint set for REST comes from the code-on-demand style."[Fie00] Code-on-demand means that a server may send or reference executable code in addition to data.[Ric+13] A very common example of code on demand is the HTML `<script>` tag, which is executed by a browser after downloading the containing HTML document. "REST allows client functionality to be extended by downloading and executing code in the form of applets or scripts. This simplifies clients by reducing the number of features required to be pre-implemented. Allowing features to be downloaded after deployment improves system extensibility. However, it also reduces visibility, and thus is only an optional constraint within REST.The notion of an optional constraint may seem like an oxymoron. However, it does have a purpose in the architectural design of a system that encompasses multiple organizational boundaries. It means that the architecture only gains the benefit (and suffers the disadvantages) of the optional constraints when they are known to be in effect for some realm of the overall system."[Fie00]

2.2.2. REST Applied to HTTP

Although the REST architecture style can be applied to may different communication protocols, it has been used to guide the standardisation of HTTP and URIs. [Fie00] "[T]he motivation for developing REST was to create an architectural model for how the Web should work, such that it could serve as the guiding framework for the Web protocol standards. REST has been applied to describe the desired Web architecture, help identify existing problems, compare alternative solutions, and ensure that protocol extensions would not violate the core constraints that make the Web successful." As the architecture influenced the standardisation of HTTP, many principles of REST are part of the standard itself, like Cache Control and Content Negotiation via headers. However, principles like Code on demand, modeling URLs to match resources and other constraints are not part of the standard and therefore results in API providers not adhering to them.

2.2.3. REST APIs in practice

In practice, the term REST has often been used without actually conforming to all the principles outlined in the original definition. While some requirements were already discussed within the publication ([Fie00]), some common misconceptions about the REST architectural style have been clarified later. In the words of the author: "I am getting frustrated by the number of people calling any HTTP-based interface a REST

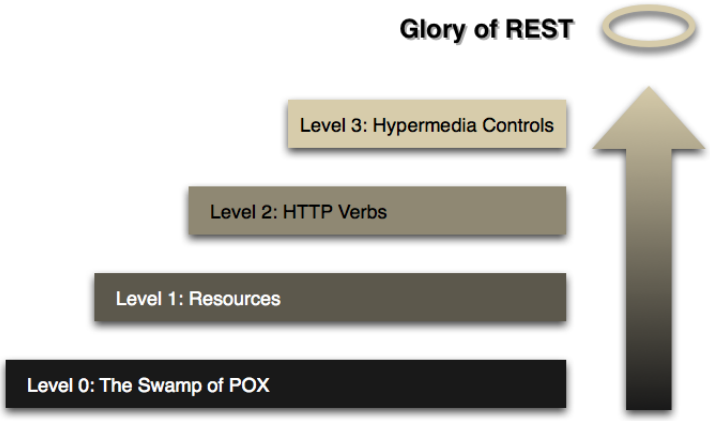


Figure 2.2.: Richardson Maturity Model [Fow10]

API" [Fie08]. According to Fieldings, "the [...] rules related to the hypertext constraint that are most often violated within so-called REST APIs. Please try to adhere to them or choose some other buzzword for your API"[Fie08].

In order to restore a concrete meaning to the terminology around REST, the Richardson Maturity Model (RMM) [15] is often used, although other more nuanced approaches have been proposed since [Alg10][SS15]. The RMM provides terminology do define states from HTTP APIs in their progress towards conforming to REST constraints through 4 levels, see Fig. 2.2.

While any HTTP API conforms to Level 0, the Richardson Maturity Model adds REST constraint iteratively:

1. Resources: Instead of RMI/RPC-style invocations imitating method calls, the server breaks down large service endpoints into multiple resources. [Fow10]

2. HTTP Methods: In addition to Level 1 requirements, level 2 introduces a standard set of verbs (HTTP Methods), so similar situations are handled in a predictable way, removing unnecessary variation.[Fow10]

3. Hypermedia Controls: Level 3 introduces discoverability, providing a way of making a protocol more self-documenting. Each response should contain links to other Endpoints which provide related or additional functionality. This behavior is commonly associated with the acronym HATEOAS (Hypertext As The Engine Of Application State)[Fow10]

[15] `https://martinfowler.com/articles/richardsonMaturityModel.html`

As the focus within this thesis is Web APIs, regardless of adhering to any level of the maturity model or RESTful principles, the name REST APIs does not fit the subject of the thesis. While the scope could have been limited to generating OpenAPI specification documents for REST APIs, allowing more assumptions and an overall more specific approach, the application in practice would have been very limited. Therefore, in order to apply our approach more broadly, adherence to the RESTful principles are explicitly not a requirement.

2.3. JSON(-Schema)

To better understand the documentation requirements of data exchange for Web APIs, it is necessary to understand common (validated) data exchange itself. In this section, we will define conventions used within this thesis in order to allow for better documentation of the data exchange process via Web API endpoints.

2.3.1. JSON

The JavaScript Object Notation (JSON) [Bra+14; ECM16] is the most common data-exchange format and media type used in Web APIs. As implied by the name, JSON is based on data types of the JavaScript programming language. The standard can now be found as ISO/IEC 21778:2017 [16]. Due to it's simplicity and great support in client side applications, JSON "nowadays plays a key role in web applications"[Pez+16], both in JavaScript and other languages interacting on the Web. "JSON has gained tremendous popularity among web developers, and has become the main format for exchanging information over the web."[Pez+16]

Using JSON, programmers can describe values (see Fig. 2.3), including objects (unordered key/value pairs, see Fig. 2.4), arrays (Fig. 2.5), numbers (see Fig. 2.6), strings (Fig. 2.7), booleans and null. An example can be found in Listing. 2.1.

Listing 2.1: JSON Example

```
{
  "id": 123,
  "active": true,
  "attributes": [],
  "created": 1592148288,
  "description": "",
  "name": "A simple glass of water",
}
```

[16] https://www.iso.org/standard/71616.html

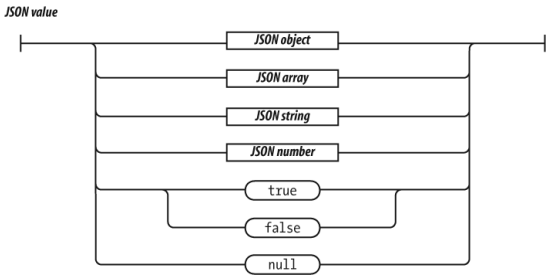


Figure 2.3.: JSON values [Cro08]

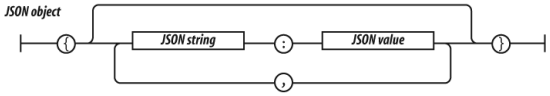


Figure 2.4.: JSON object [Cro08]

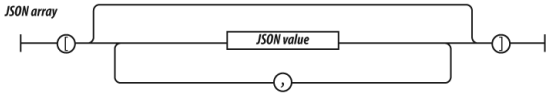


Figure 2.5.: JSON Arrays [Cro08]

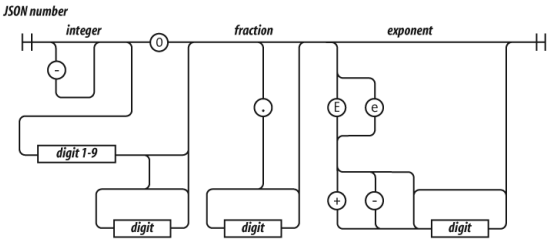


Figure 2.6.: JSON numbers [Cro08]

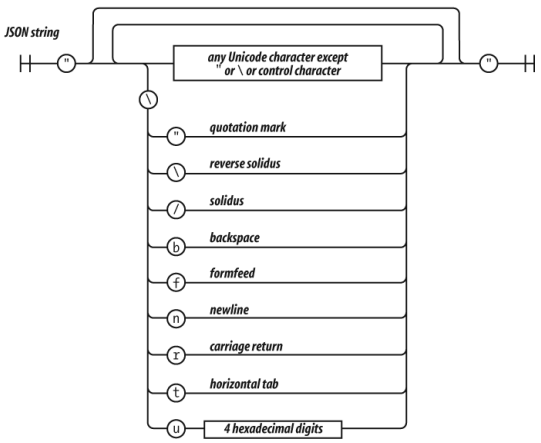


Figure 2.7.: JSON strings [Cro08]

In this thesis, JSON will be the default data exchange format, unless explicitly noted.

2.3.2. JSON Schema

Despite the popularity, JSON does not have a standardized meta-schema or vocabulary that allows specification of JSON document formats [Pez+16]. Given the popularity of JSON however, there is increasing demand for public Web APIs to implement an integrity layer, enabling a declarative way to describe valid inputs for elements of a HTTP request, like a body for POST requests to a Payment API. For instance, our public API could filter API requests before internal processing, which may increase security, allow the integrity layer to be reused and extracted, prevent data integrity issues or API usability concerns if the server crashes during processing with internal server error messages. In the worst case, that may lead to sensitive information being disclosed.
To fill the gap between lacking standardization and demand for language agnostic, reusable JSON declaration logic, JSON Schema[17] is a vocabulary that allows developers to annotate and validate JSON documents. For the JSON example shown in Fig. 2.1, a schema like the one provided in Fig. 2.2 could be used to specify the shape the JSON document in a declarative way.

In terms of standardization, "[t]he JSON Schema project intends to shepherd all four draft series to RFC status. Currently, we are continuing to improve our self-published Internet-Drafts. The next step will be to get the drafts adopted by an IETF Working Group. We are actively investigating how to accomplish this."[Org19]

[17] `https://json-schema.org`

The current draft can be found online[18], the most recent draft at the time of this writing is Draft 2019-09.

Listing 2.2: JSON Schema Exmample

```
{
  "$schema": "http://json-schema.org/draft-00/schema#",
  "type": "object",
  "properties": {
    "id": {
      "type": "integer"
    },
    "active": {
      "type": "boolean"
    },
    "attributes": {
      "type": "array",
      "items": {
        "type": "string"
      }
    },
    "created": {
      "type": "integer"
    },
    "description": {
      "type": "string"
    },
    "name": {
      "type": "string"
    }
  },
  "required": [
    "id"
  ]
}
```

2.4. OpenAPI Specification (OAS)

The OpenAPI Specification (formerly know as Swagger, OAS) is a community-driven, "programming language-agnostic interface description"[Ini+20] for HTTP APIs, which allows both humans and computers to discover and understand the capabilities of the service without access to source code, documentation, or through network traffic inspection[Ini+20]. The relationship is visualized in Fig. 2.8. API descriptions that conform to

[18] https://json-schema.org/specification.html

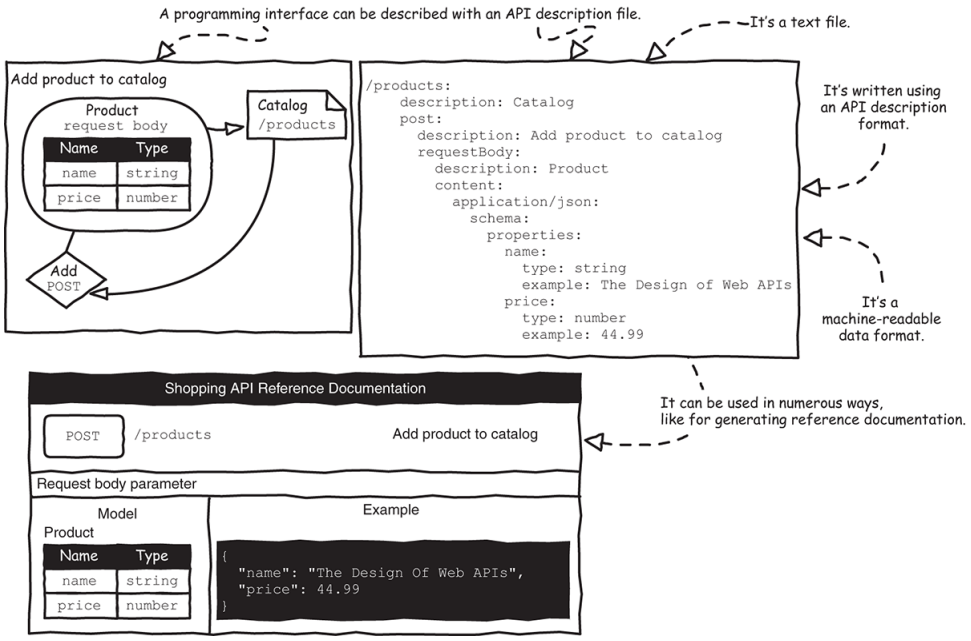


Figure 2.8.: Describing a programming interface with an API description format [Lau19]

the OpenAPI specification (format) are called OpenAPI specification documents (OAS documents, sometimes called (OpenAPI) specifications or specs).

As OpenAPI specification documents are used to describe APIs consumed via HTTP, the format uses HTTP naming conventions for the HTTP messages as introduced in section 2.1. Additionally, OAS documents contain general information about the API, the API provider and the servers which are used to service API requests. An OpenAPI specification document also describes the paths and each of the path's available methods, parameters, and responses [Lau19].

To provide generalizability when describing the *Path* to a request a Web API provides, OpenAPI divides the full path into 2 parts which are described using path templating: A list of *Servers* and *Paths*[19]. The query parts of the url are described as a query parameters, the concrete template instances in the path can be described via path parameter aswell. Therefore, a request with path `https://api.twitter.com/1.1/statuses/show/21023?include_entities=` should be split into a server (`https://api.twitter.com/{base_path}`, assigned to `#/servers/0/url`) with a base path (`1.1` at `#/servers/0/variables/base_path`), a path (`/statuses/show/{id}`) with one path parameter (`id`) and a query parameter (`include_entities`). In order to describe the underlying data model for JSON (and

[19] `https://spec.openapis.org/oas/v3.0.3#path-templating`

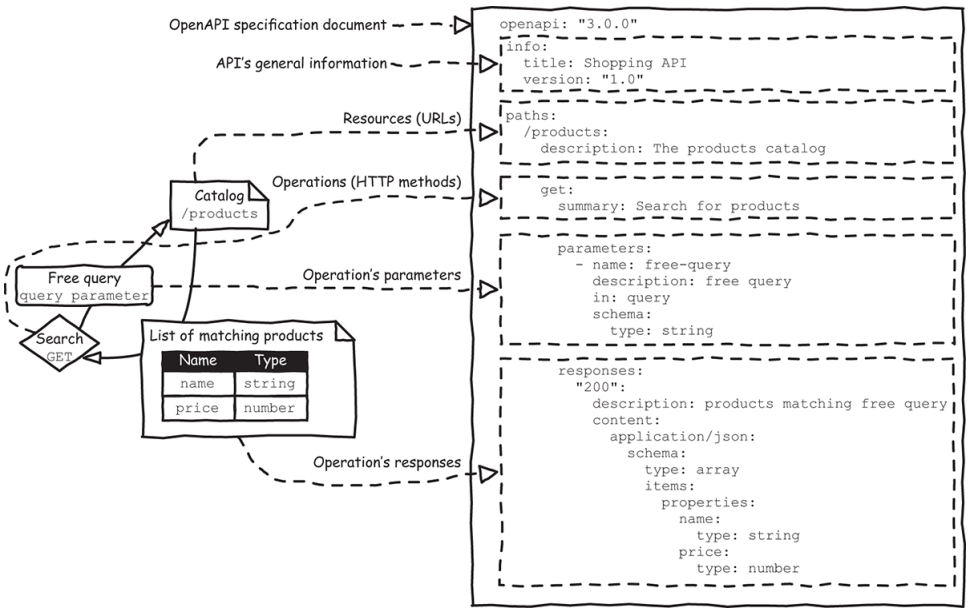


Figure 2.9.: An OAS document describing the search for products goal of the Shopping API [Lau19]

XML), OpenAPI uses a modified version of the JSON Data Schema defined in draft 00. [20]. While there are subtle differences, OpenAPI will adapt a newer draft (Draft 2019-09) [21] which consolidates differences in OAS v3.1, so we will refer to the OpenAPI data model and JSON schema interchangeably while generating code that is compatible with both Draft-00 with OpenAPI modifications and Draft 2019-09.

Due to the structured layout, the OpenAPI specification allow for a ecosystem of tooling. They include, but are not limited to: Client generation in various languages, Documentation generation, data validation layers and mock servers[22].

Because OpenAPI documents can be separated from the (eventual) implementation, the format is suited to document requirements for Web APIs without actual implementation and may even be used to power the logic of a mock server that can be used by clients involved who wish to get a deeper understanding of the proposed data format. [Bon+19]

[20] https://tools.ietf.org/html/draft-wright-json-schema-00
[21] https://json-schema.org/specification-links.html#2019-09-formerly-known-as-draft-8
[22] https://openapi.tools/

2.5. High quality API Documentation

As presented in the outline, the overarching objective of this thesis is to improve the quality of OpenAPI Specifications. In an effort to define the term *high quality*, Zhi et al. summarized a list of attributes which defines the term based on usage in the studies analyzed during their systematic mapping study of traditional APIs [Zhi+15]. In order to apply this list to Web APIs described using the OpenAPI Specification format, these attributes were sorted into three categories. Attributes that are the central focus of the tooling developed as part of this thesis are shown in table 2.1, attributes provided through the OpenAPI Specification format and the broader ecosystem of OpenAPI tools are listed in table 2.2, attributes which are not a strong focus of the work presented throughout the thesis and therefore still constitute responsibilities of the API provider are shown in 2.3.

Quality Attribute	Description
Accuracy	Accuracy measures describe the accuracy or preciseness of documentation content. Synonyms include 'preciseness'. The preciseness of documentation content is generally believed to have impacts on how easy it is for the exact information to be conveyed to the practitioners. If a document is written in a way that the phrasing is vague or the descriptions are too abstract without presenting concrete, exact examples, then it may create barriers for practitioners to retrieve the information and thus impacts the documentation quality.
Completeness	Completeness measures describe how complete document contents are in terms of supporting development/maintenance tasks. Software documentation is expected to contain all the information needed for the systems or modules described, so that when practitioners read documentation, they can retrieve the information needed for their tasks. If any necessary piece of information is missing, the documentation is perceived not being able to serve its purpose and not being useful in the scenario of need.
Correctness	Correctness measures describe whether the information provided in the documentation is correct or is in conflict with factual information. If the document presents incorrect information, it is likely to mislead practitioners and creates unnecessary barriers for them to finish the tasks. This attribute is included based on common sense.
Similarity	Similarity measures the similarity level in different documents and whether information is duplicated. Some papers use the following notions instead: 'uniqueness' and 'duplication'. Content duplication results in redundancy in the documentation content and leads to unnecessary mental efforts to read and process them.
Up-to-date-ness	Up-to-date-ness measures describe the extent to which the documents are kept updated during the evolution of software systems. Similar to the description of the attribute Traceability, technical documentation is expected to evolve together with software systems. In ideal case, each version of new software release is accompanied with a corresponding version of technical documents. Documentation contents that describe the past release of software systems may provide incorrect information, or miss new information, regarding the new system and thus mislead practitioners.

Table 2.1.: Documentation quality attributes goals provided by tooling (1/3), adapted from [Zhi+15]

Quality Attribute	Description
Accessibility	Accessibility measures describe the extent to which the content of documentation or document itself can be accessed or retrieved by the software practitioners. Synonyms include 'availability', 'information hiding' and 'easiness to find'. The attribute impacts how practitioners actually use the documentation. In our repository, quite a few papers discuss how this attribute impacts documentation quality, both quantitatively and qualitatively.
Informational organization	This attribute describe the extent to which information is organized in documents. If the documentation is organized in a way that is clear and in a structure that is natural to practitioners to understand, such documentation is like to be perceived as in high quality.
Consistency	Consistency measures describe the extent to which documentation, including information presented in documents, document format, etc. are consistent and have no conflict with each other. Synonyms include 'uniformity' and 'integrity'. If the documentation contents are presented inconsistently with conflicting elements, it may confuse practitioners and results in unnecessary mental efforts to resolve those artifacts during the usage of such documentation.
Format	This attribute refers to quality of documents' format, including writing style, description perspective, use of diagram or examples, spatial arrangement, etc. This attribute is included because practitioners may prefer certain types of writing styles which are easier for them to understand and use. For example, the decision of choosing to use graphical elements in the documentation is empirically investigated to have impacts on the programming understanding.
Trustworthiness	Trustworthiness measures describe the extent to which software practitioners perceive the documents are trustworthy and reliable. Similar to Readability, such attribute is subjective and up to the practitioners to evaluate.

Table 2.2.: Documentation quality attributes enabled by the OpenAPI specification format (2/3), adapted from [Zhi+15]

Quality Attribute	Description
Author-related	This attribute refers to those attributes related to document authors, including traces of who created the documents, author collaboration, etc. In practice, the authoring process is important for guarantee document quality.
Readability	Readability measures describe how easy documents can be read. Synonyms include 'clarity'. This is a subjective quality attributes that is up to the practitioners to decide. Several papers in our repository provide empirical evidence related to this quality attribute.
Spelling and grammar	This attribute refers to those attributes related to the grammatical aspects of documents. If a technical document is presented with a large number of spelling and grammatical errors, it will impact how practitioners read that document.
Traceability	Traceability measures describe the extent to which the document modification is able to be tracked; relevant information includes when/where/why the modification is performed and who performed. This attribute deals with the evolution of software documentation which requires special attention in technical documentation. This is because documentation needs to be kept up-to-date together with the software systems or code. The traceability attribute ensures that during the evolution, all the changes to the documentation should be justified and verifiable.
Other	Several other attributes related to documentation quality were mentioned in several papers, including abstractness, perceived goodness, etc.

Table 2.3.: Documentation quality attributes provided by the developers (3/3), adapted from [Zhi+15]

2.6. API-first Design

The idea of designing the API for a program first has a almost ten year history and has been applied with various success. "API-first design means identifying and/or defining key actors and personas, determining what those actors and personas expect to be able to do with APIs" [Tho09]. While this definition emphasizes the importance of identifying all the actors' expectations, other definitions use a process view to differentiate API-first Design: "Before you build your website, web, mobile or single page application you develop an API first" [Lan04].

A visualization of the API vs. Code first approach is shown in Fig. 2.10.

To better understand this shift in philosophy, what enables it, and how it manifests itself in the developer workflow, we will first look at traditional approaches and contrast it with the API first philosophy.

The classic code first approach to building APIs starts with identifying a business opportunity which leads to some form of documentation of the requirements of the API. After the requirements are identified, developers will implement the API and a technical writer will produce corresponding API descriptions used to document the

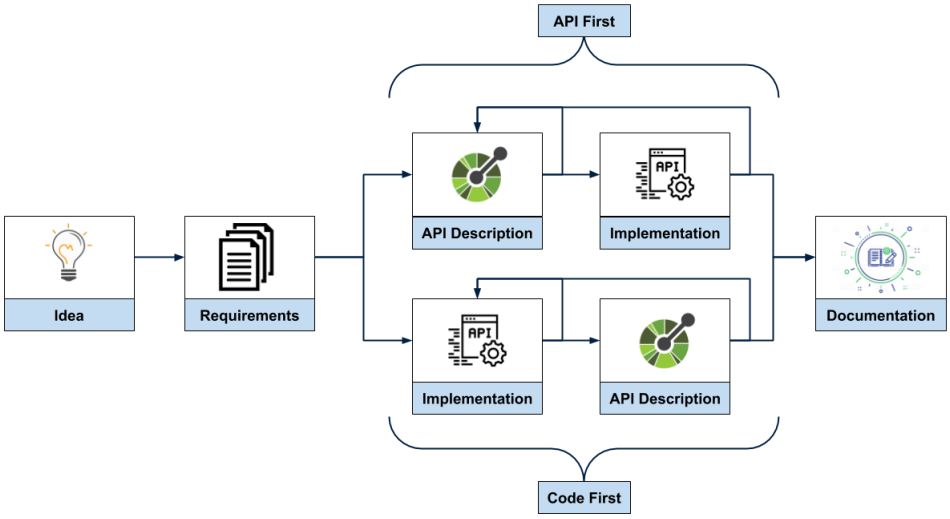


Figure 2.10.: API First vs. Code First

API after it was implemented.
The main concern the API first community frequently cites with regards to the code first approach is that API descriptions and API usability are treated like a secondary concern. Regardless of location and format of the API description, it requires an additional step, which may get overlooked, pushed to the side or hastily put together without proper care and verification.
If you put the consumer front and center, the API first philosophy argues, you have to put the API front and center and therefore focus on the API before focusing on the code that powers the API. The tangible difference usually is that API first frequently uses a specification language to produce a holistic overview of the API before an actual implementation is built. "Common practice is to create an API specification as an [OpenAPI Specification document], and define the details of the endpoints, including request and response formats. Using the API specification as basis, the actual product can be implemented afterwards" [Bui18].
Using mocking tools, this approach - unlike classic specification first approaches like the unmodified waterfall model - incorporates feedback cycles where potential customers can interact with a mock API that returns data conforming to the API description's schema. Also, the internals of the application implementing the API do not have to be specified up front. Existing research in this space "propose[s] a process for collaborative API proposal management using collaboration engineering". Bondel et al. "develop[ed]

and evaluate[ed] a prototype supporting this collaborative API proposal management process, which is designed using a design science approach and is evaluated in an action research case study. The evaluation results show, that the presented collaborative API proposal management prototype was perceived as useful and meets usability requirements" [Bon+19].

By using this approach, the API description created before the API is implemented, is often subsequently used as the source of truth for the API implementation, but also the documentation. One of the less discussed aspects of API first strategies are responding to inevitable change. Not only during the implementation of the initial specification, but also afterwards, each change has to be first specified, then implemented in order to use the API Specification Document as the source for documentation at a later point.

Another issue resulting from API first approaches are mismatches that occur because the modeling in the language of the implementation does not match the specification. API first advocates therefore promote automatic code generation to create the application shell that includes all the endpoints, parameters and models already defined in the specification. This approach however is only applicable to generate the first implementation, in response to change: "The main limitation of [generating code from a model] is the lack of round-trip engineering functionality. Once the model is specified, and the code is generated, the model and the code are not in synchronization anymore. When, for example, one of the consumed APIs changes (e.g., when a new API version was released) the developer has two options: either adjust the code manually or specify a completely new model. In the latter case, though, code that was added manually needs to be written all over again."[HSM18]

2.7. Living Documentation

The term *living documentation* first became popular in the book *Specification by Example* by Gojko Adzic [Adz11] and was further developed by Martraire in the book *Living Documentation* [Mar19]. Albeit the concept is commonly used to address internal documentation needs in agile development teams, many of the terminology and ideas can be used to inform external documentation strategies for Web APIs.

Living Documentation involves a set of four principles:

- **Reliable**: Living documentation is accurate and up-to-date with the software being delivered, at any point in time. Most of the knowledge is already present in the artifacts of the project, it just needs to be exploited, augmented, and curated for documentation purposes. [Mar19]

- **Low effort**: Living documentation minimizes the amount of manual work to be done on documentation, while ensuring reliability. By relying on standards, compatibility with existing tools can be preserved and the amount of work required will be decreased. [Mar19]
- **Collaborative**: Living documentation promotes conversations and knowledge sharing between everyone involved [Mar19].
- **Insightful**: Living documentation offers opportunities for feedback and encourages deeper thinking. It helps reflect on the ongoing work and helps in making better decisions [Mar19].

Applied to the business needs for Web API providers, the reliability aspect is covered by a broader set of attributes of *high quality* API Documentation as discussed in section 2.5. The goal of this thesis is to provide this documentation from knowledge present in source code therefore lowering the manual effort involved. As described in section 2.10, collaboration is important not only internally, but also externally, therefore collaboration with business partners is desirable [Bon+19]. In order to provide insightful *living Web API Documentation*, an API description must be able to be shared among all the parties involved, ideally as soon as the HTTP modeling (Endpoints, Resources, Methods etc.) is done. Despite the need for our work to address a broader set of goals, some of the concepts proposed in pursuit of living documentation are shared. The first of these concepts are *knowledge extraction* and *knowledge augmentation*. *Knowledge Extraction* is itself based on the observation that "most of the knowledge is already in the system itself"[Mar19]. Therefore, documentation should often be the process of sharing this knowledge in a uniform fashion by transforming the format of the knowledge, therefore making the knowledge accessible, explicit (curation) and less fragmented (consolidated). *Knowledge Augmentation* is a concept based on the observation that "most programming

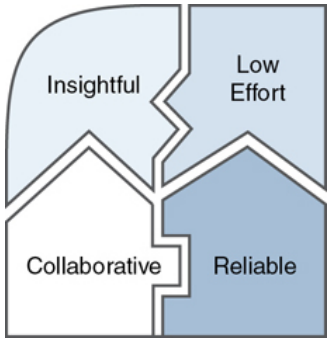


Figure 2.11.: Principles of living documentation [Mar19]

languages have no predefined way to declare the key decisions, to record the rationale, and to explain the choice made against the considered alternatives. Programming languages can never tell everything" [Mar19]. Implementation statements can be understood as the record of the result of discussions and trade-offs, the (current) final product, which lacks the context of the path towards that product. Programmers already use comments to provide this context, with close proximity to the relevant code statements, so that the augmented implementation can paint a bigger picture. Therefore, this additional, augmented knowledge would be beneficial in any automatic documentation approach as well. Unlike extracted knowledge however, augmented knowledge is usually not subjected to the same quality insurance implementation code is. Whenever possible, additional checks should therefore test this augmented knowledge whenever possible.

Another relevant categorization is the distinction between *stable*, or *evergreen* and *unstable* documentation. "Evergreen content is a kind of content that remains of interest for a long period of time, without change, for a particular audience. Evergreen content does not change, and yet it remains useful, relevant, and accurate. Obviously, not every kind of document contains evergreen content."[Mar19] Evergreen content focuses on goals and intentions, therefore describing business goals instead of technical details. Higher-level technical knowledge can also be expressed in evergreen content. In Web APIs, evergreen, or stable knowledge can often be found in longer (hyper-)text documents containing high-level guides or tutorials, presented alongside an API Reference. The API Reference itself should be considered unstable, which implies costly to maintain, documentation.

The final important discussion by Martraire revolves around the preference towards a single source of truth. Whenever knowledge is duplicated or separated, i.e. between implementation and API description, these knowledge sources must be reconciled, imposing an additional burden on the API developer. A reconciliation strategy for manually created API descriptions to insure correctness must therefore incorporate contract testing between the API description and the API implementation.

3. Related Work

In this chapter, an overview of the existing studies of automatic or tooling supported generation of technical documentation more broadly will be presented. Approaches to generating documentation automatically from source code will be examined in chapter 5, but there are several other approaches to generating (parts of) software documentation from other sources.

3.1. Generating Documentation from API usage or examples

Besides source code, the most commonly studied source for documentation is API usage data [NAP18].
Nasehi et al. [NM10] suggested this approach as a general concept for APIs. In their evaluation, subjects had difficulty finding relevant examples by browsing or searching the unit test code repository did not always result in helpful examples. For Web APIs, Sohan et al. developed SpyREST [SAM15a][SAM15b][SAM17], a tool to generate REST API documentation from on API calls. SpyREST hereby adds a proxy between the client and the server which inspects traffic and infers an API specification from the traffic. The architecture is displayed in Fig. 3.1. In order to add additional metadata, SpyREST extracts meta information like descriptions from headers that can be supplied: "The HTTP headers *x-spy-rest-version*, *x-spy-rest-resource*, and *x-spy-rest-action* can be used to override autodetection of these respective fields. Additionally, API developers can use *x-spy-rest-desc* header to attach human readable descriptions for each API example so that the web interface can tag the examples against meaningful descriptions."[SAM15b] A SaaS version of SpyREST is available online[1].

Compared to other approaches, SpyREST is language agnostic and initially requires no workflow changes for the developer. This setup could be used to ensure all integration tests that hit the server, regardless of the Test DSL or Framework are observed and used. As SpyRest treats the Web API like a black box, the approach also has downsides. One of these issues is indeterministic model reduction. Given one or more examples, inferring a minimal valid schema for inputs is not deterministic nor can be proven

[1] `http://www.spyrest.com/`

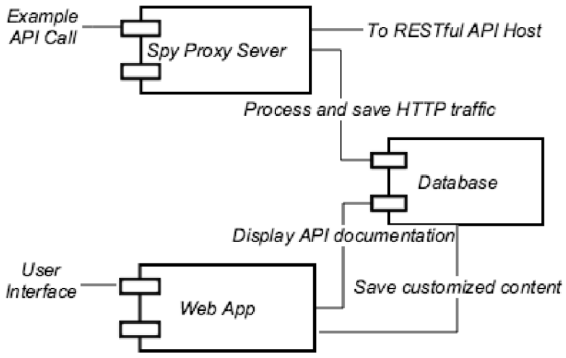


Figure 3.1.: SpyREST Design, from [SAM15b]

correct. In code of typed languages, this type inference indeterminism can usually be avoided by developer provided type information.

One of the limitations of this approach is the ability to reduce the inputs and outputs to a minimal mapping. The quality attribute that describes this behavior is accuracy, which will be detailed in Section 2.5. Like any other approach that tries to achieve specification through examples, "despite occasional claims to the contrary, a set of examples is rarely a complete specification, for the same reason that testing cannot prove a program correct. There are significant advantages to a formal specification: precision, completeness, and machine processability to name a few. In particular, preconditions and non-determinism are difficult to express with test cases. Nonetheless, it is important to recognize the role that examples can play and, in fact, have played for centuries in mathematics"[HS03]. Similar findings were described by Suter et. al in their publication on inferring Web API Descriptions from usage data using trained binary classifiers [SW15]: "Reflecting the results of our evaluation, we find inferring web API descriptions from examples to be a hard problem. While our methods improve upon the, to our knowledge, only existing tool with the same goal, results are still impeded by incomplete or noisy input data caused partly by lax API implementations, which forgive faulty requests" [SW15].

While the major upside of documentation from exemplary usage data are a generally low cost associated with this approach and a good generalizability across languages and frameworks, accuracy/precision and correctness are 2 major problems for these approaches.

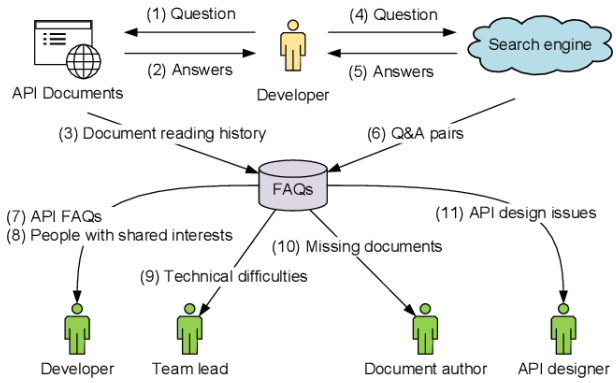


Figure 3.2.: The proposed approach [CZ14]

3.2. UML Representations from OpenAPI Specifications

As the OpenAPI document is just structured text that can be used to render documentation, Ed-douibi et al. proposed a tool (OpenAPItoUML) to visualize OpenAPI specification document using the unified modeling language (UML)[EIC18]. Their approach was later refined in through WAPIml, "an OpenAPI round-trip tool that leverages model-driven techniques to create, visualize, manage, and generate OpenAPI definitions. WAPIml embeds an OpenAPI metamodel but also an OpenAPI UML profile to enable working with Web APIs in any UML-compatible modeling tool."[Ed-+19] WAPIml currently supports OpenAPI 2.0 and works as an Eclipse plugin. We hope future work can integrate the UML models into a Web based OpenAPI 3 documentation generator, which is commonly used to display OpenAPI documents.

3.3. Collecting crowdsourced documentation

StackOverflow is an online developer community used to asked development related questions and receive answers. StackOverflow questions may be tagged, signalling that the question pertains to a certain topic. Chen et. al therefore proposed enhancing provider authored API documentation with frequently asked questions (FAQs) into API documents. The publication presents a prototype of their proposed tool called Crowdsourced Online FAQs (COFAQ), whose approach is visualized in Fig. 3.2).

A broader approach to collect relevant knowledge to enhance API documentation by Treude et. al "present[s] an approach to automatically augment API documentation with "insight sentences" from Stack Overflow — sentences that are related to a particular API type and that provide insight not contained in the API documentation of that type" [TR16].

The contribution includes SISE, a novel machine learning based approach that uses as features the sentences themselves, their formatting, their question, their answer, and their authors as well as part-of-speech tags and the similarity of a sentence to the corresponding API documentation.

With SISE, the authors were able to achieve a precision of 0.64 and a coverage of 0.7 on the development set of over 1500 sentences. Furthermore, "[i]n a comparative study with eight software developers, we found that SISE resulted in the highest number of sentences that were considered to add useful information not found in the API documentation. These results indicate that taking into account the meta data available on Stack Overflow as well as part-of-speech tags can significantly improve unsupervised extraction approaches when applied to Stack Overflow data."[TR16]

4. Standardized API Reference Documentation using OpenAPI

4.1. Elements of Web API Documentation in Literature

While API documentation has been extensively studied in literature, studies of Web API Documentation are comparatively very limited. As the OpenAPI format can be used to describe a broad set of HTTP based Web APIs, including but not limited to REST APIs, and does not impose strict constraints on the described API or the way in which it is described (i.e. conventions around reuse or accuracy requirements), guidance from the specification is limited.

In order to apply components of good Documentation, we therefore based our initial list on the taxonomy presented by Cummaudo et. al [CVG19], see Fig. 4.1.

Our general approach to adopt the original publication for Web APIs consists of two reductive steps and a final transformative step (detailed in section 4.2), as visualized in Fig. 4.2.

Step 1: Removing sub-dimensions irrelevant in the context of Web APIs

For the first step, we removed elements that do not apply to API reference documentation from the proposed dimensions.

- A8: Debugging: For Web APIs, debugging by the API consumers is usually not intended.
- A10: System requirements: The overarching standard (HTTP) sets the requirements. Regardless of the preferred (potentially language specific) HTTP client or library, the API provider should not impose any additional requirements.
- A11: Installation Instructions: One of the main factors attributed to the success of Web APIs is that they do not rely on complex installations in order to get started as an API consumer. While many Web APIs require some form of authentication, providing this information fits best within a quick-start or an authentication guide, that can be presented in evergreen content.

Key	Description: Dimensions A=Usage Description; B=Design Rationale; C=Domain Concepts; D=Support Artefacts; E=Documentation Presentation	Primary Studies	Total (%)
[A1]	Quick-start guides to rapidly get started using the API in a specific programming language.	[S4, S9, S10]	3/21 (14%)
[A2]	**Low-level reference manual documenting all API components to review fine-grade detail.**	**[S1, S3, S4, S8, S9, S10, S11, S12, S15, S16, S17]**	**11/21 (52%)**
[A3]	Explanations of the API's high-level architecture to better understand intent and context.	[S1, S2, S4, S11, S14, S16, S19, S20]	8/21 (38%)
[A4]	Source code implementation and code comments (where applicable) to understand the API author's mindset.	[S1, S4, S7, S12, S13, S17, S20]	7/21 (33%)
[A5]	**Code snippets (with comments) of no more than 30 LoC to understand a basic component functionality within the API.**	**[S1, S2, S4, S5, S6, S7, S9, S10, S11, S14, S15, S16, S18, S20, S21]**	**15/21 (71%)**
[A6]	**Step-by-step tutorials, with screenshots to understand how to build a non-trivial piece of functionality with multiple components of the API.**	**[S1, S2, S4, S5, S7, S9, S10, S15, S16, S18, S20, S21]**	**12/21 (57%)**
[A7]	Downloadable source code of production-ready applications that use the API to understand implementation in a large-scale solution.	[S1, S2, S5, S9, S15]	5/21 (24%)
[A8]	Best-practices of implementation to assist with debugging and efficient use of the API.	[S1, S2, S4, S5, S7, S8, S9, S14]	8/21 (38%)
[A9]	An exhaustive list of all major components that exist within the API.	[S4, S16, S19]	3/21 (14%)
[A10]	Minimum system requirements and dependencies to use the API.	[S4, S7, S13, S17, S19]	5/21 (24%)
[A11]	Instructions to install or begin using the API and details on its release cycle and updating it.	[S4, S7, S8, S9, S11, S13, S16, S19]	8/21 (38%)
[A12]	Error definitions that describe how to address a specific problem.	[S1, S2, S4, S5, S9, S11, S13]	7/21 (33%)
[B1]	**A brief description of the purpose or overview of the API as a low barrier to entry.**	**[S1, S2, S4, S5, S6, S8, S10, S11, S15, S16]**	**10/21 (48%)**
[B2]	Descriptions of the types of applications the API can develop.	[S2, S4, S9, S11, S15, S18]	6/21 (29%)
[B3]	Descriptions of the types of users who should use the API.	[S4, S9]	2/21 (10%)
[B4]	Descriptions of the types of users who will use the product the API creates.	[S4]	1/21 (5%)
[B5]	Success stories about the API used in production.	[S4]	1/21 (5%)
[B6]	Documentation to compare similar APIs within the context to this API.	[S2, S6, S13, S18]	4/21 (19%)
[B7]	Limitations on what the API can and cannot provide.	[S4, S5, S8, S9, S14, S16]	6/21 (29%)
[C1]	Descriptions of the relationship between API components and domain concepts.	[S3, S10]	2/21 (10%)
[C2]	Definitions of domain-terminology and concepts, with synonyms if applicable.	[S2, S3, S4, S6, S7, S10, S14, S16]	8/21 (38%)
[C3]	Generalised documentation for non-technical audiences regarding the API and its domain.	[S4, S8, S16]	3/21 (14%)
[D1]	A list of FAQs.	[S4, S7]	2/21 (10%)
[D2]	Troubleshooting suggestions.	[S4, S8]	2/21 (10%)
[D3]	Diagrammatically representing API components using visual architectural representations.	[S6, S13, S20]	3/21 (14%)
[D4]	Contact information for technical support.	[S4, S8, S19]	3/21 (14%)
[D5]	A printed/printable resource for assistance.	[S4, S6, S7, S9, S16]	5/21 (24%)
[D6]	Licensing information.	[S7]	1/21 (5%)
[E1]	Searchable knowledge base.	[S3, S4, S6, S10, S14, S17, S18]	7/21 (33%)
[E2]	Context-specific discussion forum.	[S4, S10, S11]	3/21 (14%)
[E3]	Quick-links to other relevant documentation frequently viewed by developers.	[S6, S16, S20]	3/21 (14%)
[E4]	Structured navigational style (e.g., breadcrumbs).	[S6, S10, S20]	3/21 (14%)
[E5]	Visualised map of navigational paths to certain API components in the website.	[S6, S14, S20]	3/21 (14%)
[E6]	**Consistent look and feel of documentation.**	**[S1, S2, S3, S5, S6, S8, S10, S15, S20]**	**9/21 (43%)**

Figure 4.1.: An overview of the 5 dimensions and categories (sub-dimensions) within proposed taxonomy [CVG19]

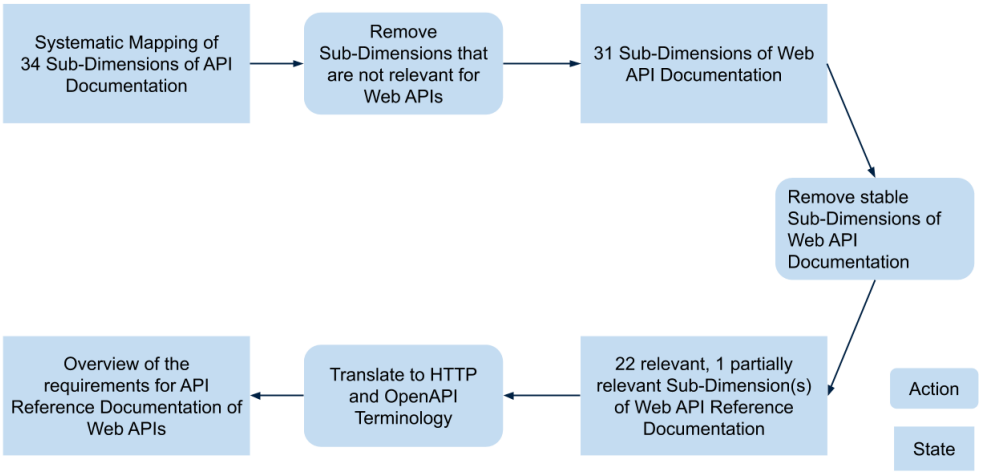


Figure 4.2.: Adapting the initial taxonomy to Web APIs

Step 2: Removing or limiting sub-dimensions that aren't part of API reference documentation, or evergreen content

In this step, we analyze the remaining sub-dimensions. While documentation of these sub-dimensions is necessary as part of a broader documentation strategy, providing this knowledge may not be suitable for automation or presentation within the API reference documentation.

- A1: Quick-start guide to rapidly get started using the API in a specific language: This getting started guide usually constitutes the entry point (or the first page after a landing page) for potential API consumers. For Web APIs, it usually contains information on how to obtain API keys used for authentication in subsequent API Requests. Due to the high variance in procedures involved and high amount of assumed free flow text, but also a higher degree of stability, the best place for a getting started guide would be in form of a supplementary resource alongside the API Reference. Therefore, this element will not be a requirement for the automatic generation of an OpenAPI description.
- A3: Explanations of the API's high-level architecture to better understand intent and context: For Web APIs, this section is usually called an *introduction guide*, which contains free form text following the getting started guide. As laid out

in the motivation and the foundations 2, the context of this thesis are Web APIs which revolve around the modification of resources. Documenting the relation of resources, with free form descriptions and correct properties is a vital aspect of API Reference Documentation and will be considered a relevant requirement. Unlike traditional APIs, the architecture of the API is usually confidential and should be opaque to the API customer. This implies that any automatic approach should treat the inner workings of the API as a black box and only document the surface exposed to the customer.

- A6: Step-by-Step tutorials, with screenshots to understand how to build a non-trivial piece of functionality with multiple components of the API: Web APIs are called via HTTP Request, so screenshots do not play an important role. For Web APIs, a similar notion as step-by-step tutorials exist, usually referred to as scenarios, a composition of usually sequentially executed HTTP requests. While detailed scenarios are best provided in a separate document (similar to the high level architecture), with links from the steps to the API Reference of the method used, simple links could also be shown if it's clear given the context which request is related and should most likely called next. We will therefore consider simple links, but move more complex scenarios that require persistent state over several requests to a different approach. In order to allow for operations to be linked, each *Operation* should define a unique *OperationId*, see Fig. 4.4. As outlined in section 2.2, possible links should be included in the response payload of REST APIs themselves (HATEOAS requirement of REST APIs). Any other Web API, which does not adhere to the REST constraints may instead choose to only document static references via OpenAPI Links to other Operations.

- C3: Generalized documentation for non-technical audiences: We consider the target audience of API Reference to be generally technically versed and therefore consider non-technical documentation to be provided alongside the API Reference Documentation.

- D1: A list of FAQs: Not considered due to the assumed low likelihood of finding this kind of knowledge in source code (even if present). As presented in chapter 3, there are existing approaches to generate this knowledge using crowd-sourced approaches, however, we think this kind of documentation may best be suited for additional resources alongside the API reference documentation as part of a broader API documentation strategy, similar to the *Getting Started Guide* or long form tutorials.

- D2: Troubleshooting suggestions: Whenever an API consumer encounters unexpected behavior, an API should "help users recognize, diagnose, and recover

from errors" [MS16]. While the intention of good documentation should be to prevent these mistakes in advance, good guidance on how to proceed in case of an issue could reduce frustration and increase user retention. As the original publication refers to error handling both in a more general sense (D2) and specific terms (A12), we consider general error handling in dealing with Web APIs or the specific Web API to be outside the scope of this thesis and focus on specific error handling as outlined in A12, which is usually associated with a specific endpoint. On a more general level, we would recommend adopting a standardized error format interface, i.e. the format proposed in RFC 7808 [1], which aims to improve the usability of the API itself. Enforcing this style is also not a documentation concern, but defining an error message interface that can be reused and therefore referenced throughout the documentation would even improve the API reference documentation usability in an indirect way.

- D3: Diagrammatically representing API components using visual architectural representations: For approaches to generating UML diagrams from OpenAPI specifications that can be presented alongside textual information in the API Introduction, we refer to work done by Ed-douibi et al. in [EIC18] and [Ed-+19] as detailed in Chapter 3 and mentioned in Chapter 3.

- D5: A printed/printable resource for assistance: There are existing command-line tools in the OpenAPI ecosystem that generate offline API Reference Documentation given an OAS document, providing a downloadable OAS document should be up to the API Developers.

- E2: Context-specific discussion forum: Omitted as an API Reference Documentation should not incorporate a discussion forum.

From the remaining 31 sub-dimensions proposed in the taxonomy, we removed 9 sub-dimensions according to our rationale as to which sub-dimensions can or should not be provided through API Reference documentation. For these 9 dimensions, we provided general guidance as to which alternative approaches may be taken to provide other forms of, or strategies to address or mitigate potential lack of documentation. In order to allow for a successful linking between Operations or from a Tutorial to an Operation, only the generation of a unique *OperationId* and that can be referenced using the OpenAPI *Link Objects*, as shown in Fig 4.9, remain as a requirement.

[1] `https://tools.ietf.org/html/rfc7807`

4.2. Specifying Web API Reference Documentation using OpenAPI

In this transformative step, the remaining 22 sub-dimensions will be translated into the corresponding elements of the OpenAPI Standard. When necessary, appropriate extensions using the Specification Extension mechanism [2] will be proposed. In order to reference locations within the OAS, JavaScript Object Notation (JSON) Pointer notation, as proposed in RFC 6901 [BZN13], is used.

- A2: Low-level reference manual documenting all API components to review fine-grade detail: This sub-dimension describes the core of every API Reference Documentation. In Web APIs, accessible via HTTP, this involves description of all paths the API exposes to the API customers. As outlined in 2.1, the structure of the HTTP messages to be documented is well defined and based on the HTTP Message Specification. Translating the Elements of a HTTP Request/Response to OpenAPI descriptions is presented in Chapter 2, Section 2.4.
- A4: Source code implementation and code comments (where applicable) to understand the API author's mindset: Alongside every *Path*, *Parameter*, *Response* and *Body Property*, OpenAPI supports a description field to allow for expression of free flow text in markdown notation as specified by the CommonMark Markdown Standard 0.27 [3].
- A5: Code snippets (with comments) of no more than 30 LoC to understand a basic component functionality within the API: Beyond the descriptions, the OpenAPI Specification adds an example(s) field similar to the description fields, see Fig. 4.7.
- A7: Downloadable source code of production-ready applications that use the API to understand implementation in a large-scale solution: Due to the language agnostic nature of Web APIs, the code invoked to call the API may vary. Instead of transforming example parameters to several client examples, standardizing the output format allows to hook into the OpenAPI ecosystem and generate entire SDKs. For quick feedback cycles, the tool used to generate the rendered API Reference Documentation from the API description should transform the example into a usable example the consumer can try out (i.e. a cURL example or a "Try it out" feature).

[2] `https://spec.openapis.org/oas/v3.0.3#specification-extensions`
[3] `https://spec.commonmark.org/0.27/`

- A9: An exhaustive list of all major components that exist within the API: In order to encourage reuse and promote a better understanding of the resources the API operates on, instead of defining these components "in-line" and duplicating descriptions, examples and the data model, the OpenAPI Specification defines a mechanism called Schema Objects. A Schema Object allows the definition of input and output data types. These types can be objects, but also primitives and arrays. If a component of the API is considered "major", every part of the API that uses that schema can instead reference a Schema Object through a *$ref* Pointer according to the JSON Schema Pointer specification, see section 2.3.

- A12: Error definitions that describe how to address a specific problem: According to the HTTP Standard, the *Status* should give the initial indication whether a call was successful. While the Error message, most commonly provided in the body of the Error Response is determined by the implementation, the OpenAPI Specification allows each documented response to have a description that should provide information on how to address issues. A good approach to API reference documentation should therefore ensure complete coverage of all responses and provide all the fields shown in Fig. 4.9 for both successful and erroneous responses.

- B1-7: A description of the purpose or overview of the API as a low barrier to entry, types of applications the API can develop, types of users who should use the API, types of users who will use the product the API creates, success stories and comparisons to other APIs and limitations on what the API can and cannot provide: The Design Rationale dimension will not be considered as part of automatic API Reference Documentation for reasons similar to the ones given in A3. Generating high level rationale from source code while ensuring correctness goes beyond the scope of the thesis and may be suited best for additional resources beyond the API Reference Documentation. However, we would recommend a brief summary of the rationale behind the API at the beginning of the API Reference Documentation as part of the *Information Object*, see the description property of Fig. 4.5.

- C1/2: Description of the relationship between API concepts and domain concepts or terminology, with synonyms, if applicable: For a given Operation, this information should be provided within the Operation's description. See Fig 4.4. As the description is free form text, we will not apply validation of this augmented knowledge. Additional knowledge beyond simple relationships, which are not expected to change frequently (stable documentation) may be provided in documentation alongside the API Reference.

- D4: Contact information for technical support: In order to contact the API provider, it's useful to provide contact information to the consumer of the API. This information may be provided within the OpenAPI specification's *Information Object*. A diagram of this object can be found in Fig. 4.5.
- D6: Licensing information: As an API consumer, it's critical to know what constitutes appropriate and inappropriate usage of the API. While the legal details should not be displayed in full detail within the API Reference Documentation of the API, links to the relevant location should be included. Within the OpenAPI spec, links and the name of the license and the terms of service can be included in the *Information Object* as shown in Fig. 4.5.
- E3: Quick-links to other relevant documentation frequently viewed by developers: This kind of knowledge may be included using knowledge augmentation at the Operation level, see the *externalDocs* property of Fig. 4.4.

Requirements E1, E4, E5 and E6, while important, are not a requirement of the API description document. Instead, the tooling that transforms the API description into a rendered API reference documentation should be evaluated according to these sub-dimensions.

Although the source publication by Cummaudo et. al provides a good indication about the general availability through a percentage indicator of occurrence, it does not list popularity or importance of these elements as perceived by the API consumers reading the documentation. Therefore, we compared our transfomed of the Elements of Web API reference documentation to a survey of the most important "things" developers look for in API documentation to validate the relevance of the requirements elicited as published in the "State of API 2019" survey conducted by SmartBear [Sma19]. The results are shown in Fig. 4.3.

As our elements - which will now be considered to be requirements - cover documentation of examples, status codes and error codes, authentication, error message format, http request format and parameters, the top 6 of the most important "things" developers look for in API documentation are appropriately covered. Methods and code samples are also included. For changelogs and SDKs, while not included in our approach, OpenAPI tooling provides possible generation approaches. If resources refers to REST resources, resources would be included as well, however, since the term may also refer to additional documentation resources, we can't confirm they would be covered by an approach according to our requirements.

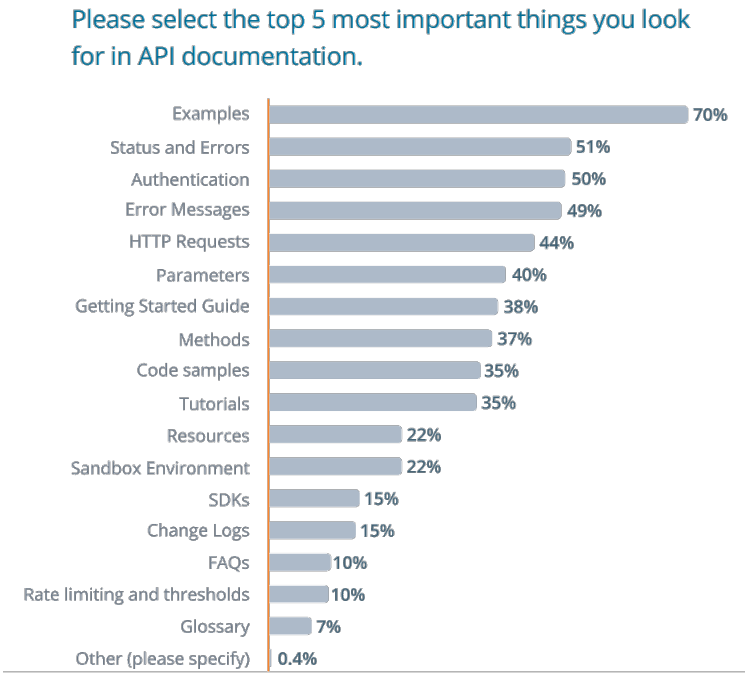


Figure 4.3.: API Documentation priorities for API consumers [Sma19]

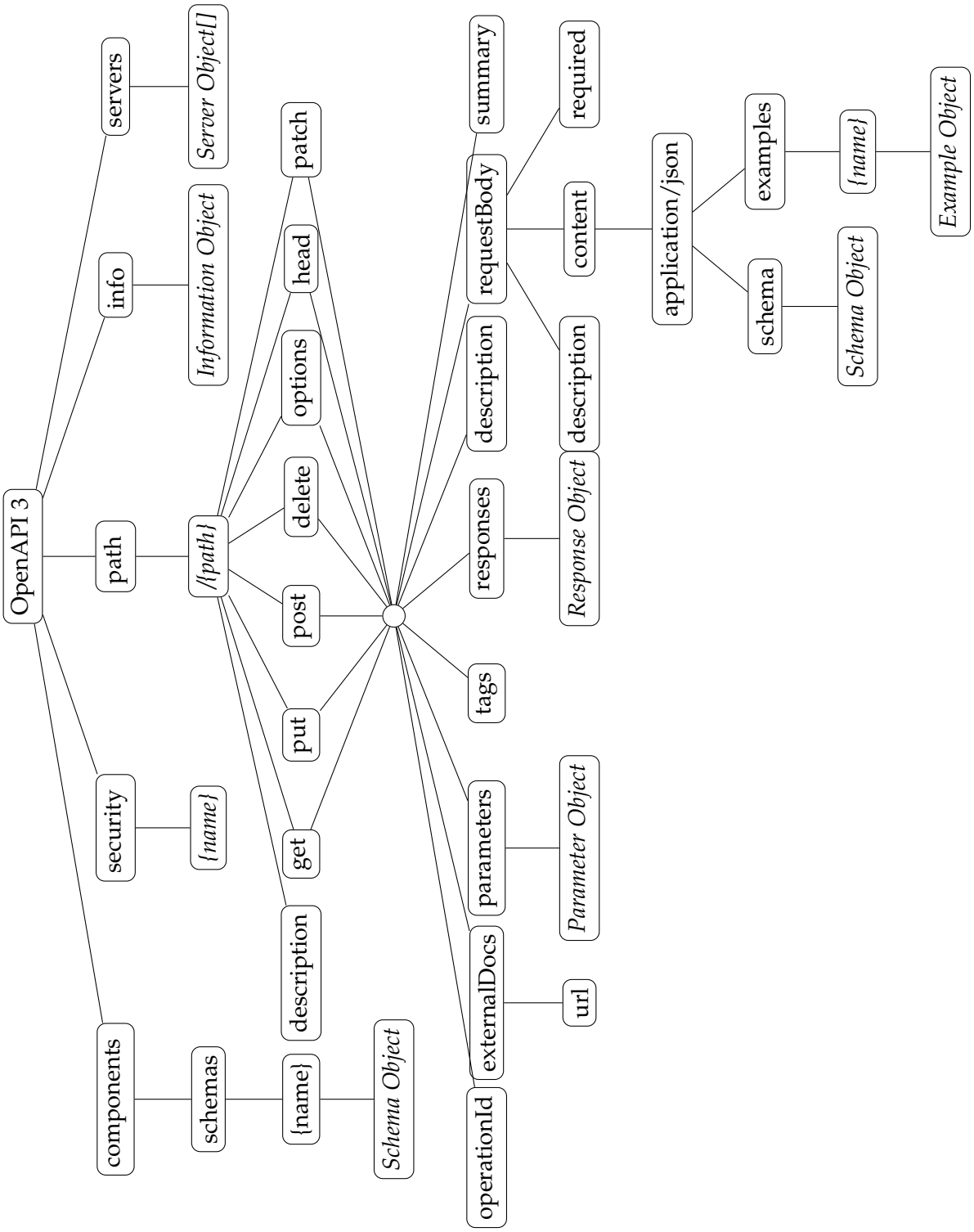


Figure 4.4.: OpenAPI Fields according to requirements

Figure 4.5.: Proposed Coverage of the Information Object

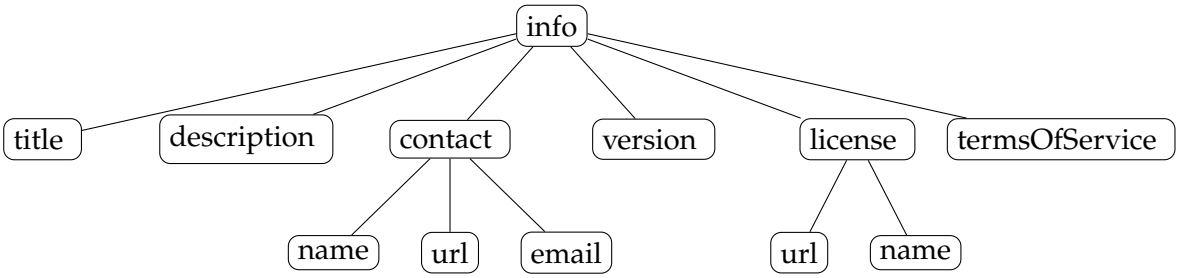


Figure 4.6.: Proposed Coverage of the Server Object

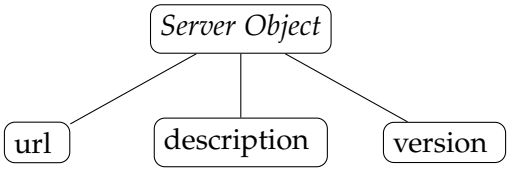


Figure 4.7.: Proposed Coverage of the Example Object

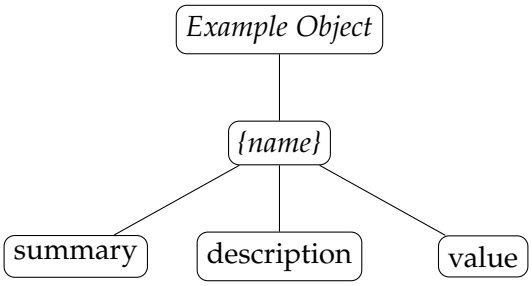


Figure 4.8.: Proposed Coverage of the Parameter Object

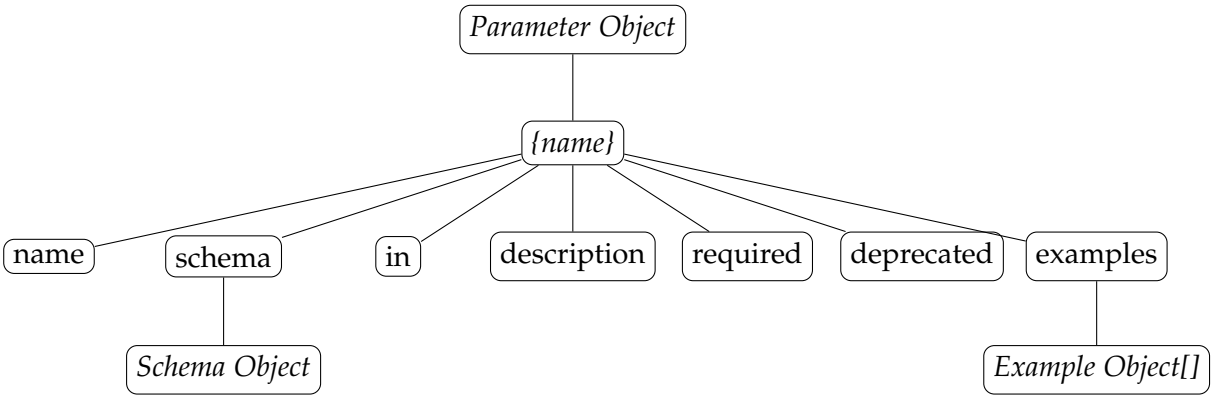
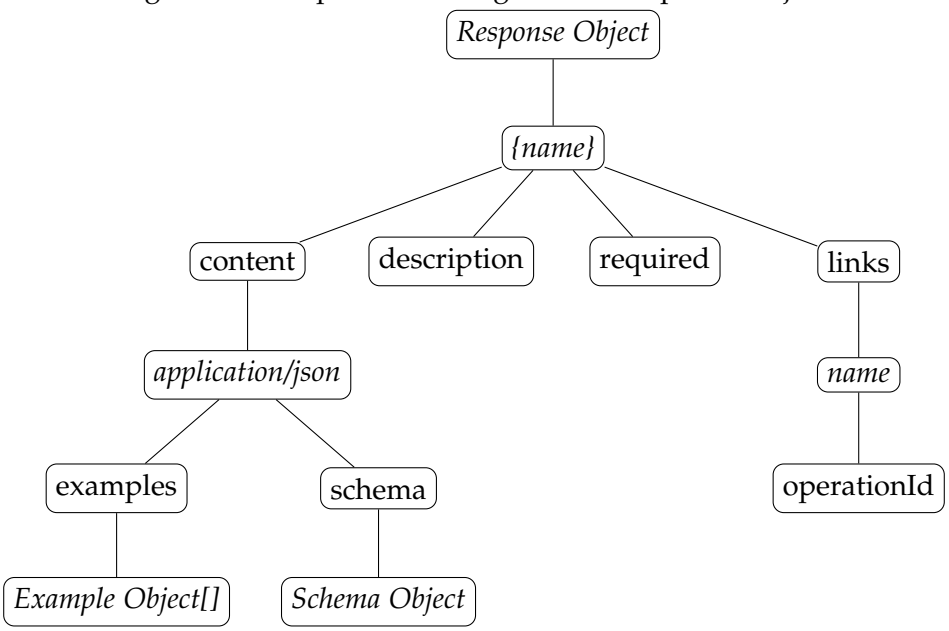


Figure 4.9.: Proposed Coverage of the Response Object



5. Approaches to generating API Reference Documentation

Generating Documentation from source code is not a new concept. While techniques vary, a systematic mapping study conducted by Nybom et al. compared sources for documentation generation and identified the approach targeting implementation code to be the most commonly used general approach (see Fig. 5.1). [NAP18] Although the study gives a general idea about the popularity, but it does not draw distinctions based on the techniques used to extract documentation from code.

Additionally, efforts to document Web APIs from source code using these techniques have, to the best of our knowledge, not been the subject of scientific literature.

In this section, we will present the general approaches used to generate documentation and compare their strengths and weaknesses with regards to extracting OpenAPI specification documents from code powering the logic of Web APIs. Compared to the approaches studied by Nybom et al., generating Web API documentation from source code adds additional complexity as the source knowledge has to be transformed into a programming language agnostic format (the OpenAPI Schema).

In order to evaluate the viability of any approach, we choose TypeScript code as the source, due to the popularity of the language, which could increase the likelihood of adoption an general relevance of the project, but also because TypeScript's type system targets JavaScript, which increases the compatibility between JavaScript objects and JSON(-Schema).

In order to successfully implement any approach, the following metrics are of special interest:

- Operation generation: Each Operation describes the format of the HTTP Message Specification. This includes path, parameters (excluding the schema), responses (excluding the schema). This corresponds to the `OperationItem` and it's parent items up to the `Paths` object, as defined in the OpenAPI Specification.
- JSON Schema generation: As every parameter and every response body needs to be specified using the OpenAPI version of the JSON Schema Draft, the automatic inference of this schema is a special metric within our evaluation. This includes generation of major, reusable components (C1)

Source	Count	Primary Studies	Description
Source Code & Examples	18	S1, S2, S5, S7, S8, S9, S11, S12, S13, S16, S19, S22, S24, S28, S31, S32, S34, S35	Source code, code comments and annotations, example code
API Usage & API calls	9	S1, S3, S14, S17, S20, S23, S25, S30, S33	API usage scenarios, traces of method calls in existing software
API Documentation & Tutorials	8	S5, S6, S10, S23, S26, S27, S35, S36	Natural language description and API tutorials
Online Information	7	S4, S11, S18, S29, S31, S32, S34	Websites such as Stack Overflow
Manual Input	5	S9, S12, S14, S15, S21	Requires manual input of API user
Databases & Search Engines	4	S4, S7, S21, S34	Results from search engines, data found in databases

Figure 5.1.: Approaches to documentation generation for traditional APIs [NAP18]

- Usability aid generation: Descriptions, Links and Examples, Tags, or a short summary are all elements of an API description intended to contribute towards a better understanding of the API description and therefore improved usability. As a vital part of documentation, each approach should be able to include this type of information.
- Metadata generation: Elements such as Contact Information (D4) and Licensing Information (D6) or Tag descriptions used to group concepts (C1) or a brief description of the purpose of the API as a whole (B1), which are not directly related to Operations are considered here.

5.1. Sources of documentation in programs

Pieces of documentary knowledge can be found throughout programs, in various forms and different levels of hierarchy (Fig. 5.2).

Broadly speaking, when looking at the hierarchy of a program, we can identify at least 3 levels where documentation can be found. At the highest level, documentation about the API as a whole can be found at the application level. Typical examples of this kind of information are configuration files like a `package.json` file in the root folder of the application. Below the application level, many frameworks and languages support modules to group related functionality. A module for a MVC-style framework may include references to the Controllers or shared authentication logic. If available, this module may be used to tag (and therefore group) all endpoints defined within the module with a common tag. Within modules, depending on the programming style, we will find classes and methods (object oriented programming) or functions (functional programming), or a mix, to handle HTTP requests. The set of all API request handlers are referred to as the API surface. At this level, we can often identify detailed descriptions of one Operation, including a description of the operation, parameters, and possibly the shape of the response expressed through type annotations. At the lowest level, the statement level, the actual implementation can be used to derive

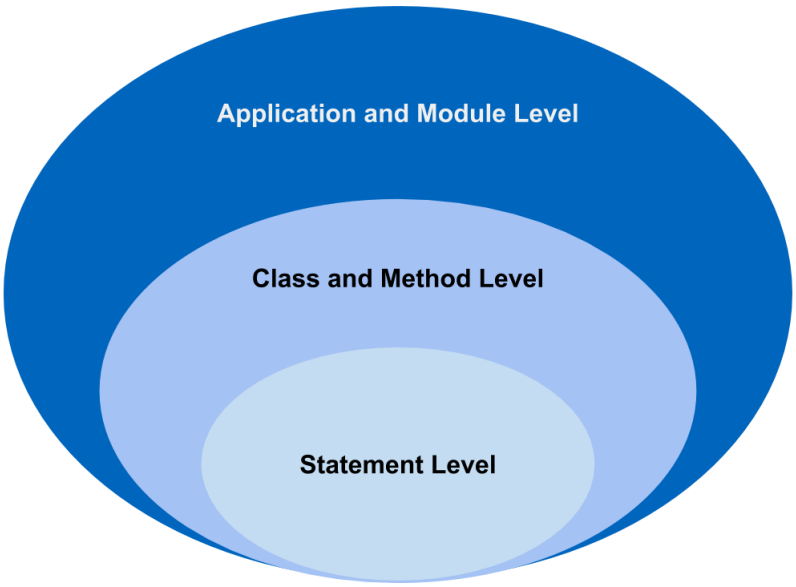


Figure 5.2.: Hierachy of an API Implementation

information about the knowledge about the API. In traditional APIs, the statements in combination with some degree of context is often used for source code summary, which provides a summary or a description of the method. In this section, we intend to provide information about benefits and issues associated with each source as it relates to documentation quality.

5.1.1. (Structured) Comments

Comments seem like a natural fit for developer provided documentation, as it is a very basic form of documentation almost all developers will be familiar with. Furthermore, most development environments provide some level of integration for documentation blocks (doc blocks, doc comments) above methods that will i.e. be rendered in a tooltip when hovering method references. A key property of docomentation comments (doc comments, doc blocks) is their largely unstructured nature. While this property of doc comments makes them very flexible, and enable custom domain specific languages (DSLs) in comments, like Markdown or OpenAPI DSLs (see Fig.5.1, this flexibility comes at a price. In the cases we studied, these comments were largely unchecked and therefore provide no guarantees that they do or do not match the implementation. As a result, comments are a good fit to apply knowledge augmentation, but cannot be used for knowledge extraction and should therefore be verified whenever possible.

Listing 5.1: A DSL for OpenAPI in doc blocks using an @api doc tag

```
/**
  * @api [post] /users
  * produces:
  * - "application/json"
  * requestBody:
  * required: true
  * content:
  * "application/json":
  * schema:
  * $ref: "#/components/schemas/UserLoginDto"
  * responses:
  * "200":
  * content:
  * "application/json":
  * schema:
  * $ref: "#/components/schemas/User"
  * "400":
  * content:
  * "application/json":
  * schema:
  * $ref: "#/components/schemas/ErrorMessage"
  * ...omitted
  **/
```

One of the most prominent abstractions on top of pure comments is a format first introduced in Java called JavaDoc. This syntax used in Doc blocks adds some structure via the use of Doc-Tags. A similar markup language can also be used to annotate JavaScript and is called JSDoc. As JavaDoc was intended to provide more structured hints for documentation generation approaches, using the JavaScript equivalent seems like a good fit for applications of the knowledge augmentation pattern.

5.1.2. Annotations

Another language level feature some generation approaches rely on are *Annotations*. In the context of the Java language, these Annotations offer some benefits compared to (JavaDoc) Comments and are not removed at runtime. This allows for metadata set via annotations to be read and modified at runtime using reflection 5.2.1. The most popular library using this feature for OpenAPI Annotations is Swagger-Code [1]. One of the benefits in term of correctness is some limited syntax validation, which can ensure the supported OpenAPI specification fields are used with the correct structure,

[0] https://jsdoc.app/

[1] https://github.com/swagger-api/swagger-core

therefore preventing invalid OpenAPI specifications. The downside of Annotations, compared to structured comments, is the lack of editor integration. As annotations themselves are purely descriptive, they may be used to augment the documentation but must be subject to additional checking whenever possible. However, the combination of annotations with knowledge about how the metadata will be used to modify the program behavior, relying on the knowledge through knowledge extraction is possible. For example, given we know a web framework that recognizes an `@Post("/example")` annotation and registers a request handler that invokes the annotated method for POST requests on path `/example`, we can use the annotation to extract knowledge about the program. We will call these annotations "functional annotations", indicating that the metadata provided has significant impact on the annotated code.

In the case of Swagger-Core, annotations are used in combination with reflection at Runtime (meaning these annotations are functional), for example to reflect the properties and types on class definitions. This part of Swagger-Core will be thoroughly examined in the Reflection Section and for the scope of this section, only the Annotations features will be considered. Through coupling with the Web Framework, the swagger-jaxrs2 reader engine, which is part of the swagger-core package infers elements of the specification like *Path* or the *Method* from the annotations that the Web framework uses to generate the routing. This means, both of these parts are correctly documented by default. *Parameters* will not be documented by default, a special `@Parameter` annotation is required. In order to ensure all *Parameters* are correctly documented, special care is required. An example of a Parameter annotation can be seen in Listing 5.2.

Listing 5.2: Documenting Path, Method and Parameter using Swagger-Code, adapted from [2]

```
@Path("/subscription/{subscriptionId}")
@GET
public Response getSubscription(
    @Parameter(
        in = "path",
        name = "subscriptionId",
        required = true,
        description = "parameter description",
        allowEmptyValue = true,
        allowReserved = true,
        schema = @Schema(
            type = "string",
            format = "uuid",
            description = "the generated UUID"
        )
    ) String subscriptionId) {
            // ...
}
```

5.1.3. Statements

At the statement level, the actual implementation statements can be used to derive information about the knowledge about the API. For traditional APIs, this approach called source code summarization has been shown to be a promising, but inconsistent approach to generate summaries or descriptions of the implementation. “Unfortunately, there is no agreed upon understanding of what makes up a "good summary.""[MM16]. More advanced source code summarization approaches, that include contextual information [MM14], seem to improve the performance, however, as the inner workings of an API should mostly be opaque, this approach may disclose too much information about the code to be summarized. In order to provide clear boundaries, this thesis will not use statements in method implementations.

5.1.4. Type Systems

In computer science, a type is a concise, formal description of the behavior of a program fragment. [Rém15] Types are useful for quite different reasons: They first serve as machine-checked documentation. More importantly, they provide a safety guarantee. [Rém15] [End+14] The combination of both reasons make them an interesting research topic for correct software documentation. The safety guarantees, especially within the

[2] https://github.com/swagger-api/swagger-core/wiki/Swagger-2.X---Annotations#parameter

boundaries of the system (the API), can aid in making sure variables are assigned and returned correctly. However, one of the main challenges is enforcing type safety at the boundaries of a statically typed system at runtime. More generally, this could be database operations, reading files from disk, or, in Web APIs, accepting HTTP requests.

TypeScript is a popular superset of the JavaScript programming language, that extends the JavaScript language with a rich gradual type system, that can be used to statically analyze programs.

"Despite its success, JavaScript remains a poor language for developing and maintaining large applications" [BAT14]. In order to address scalability concerns of larger JavaScript applications, TypeScript "aims to provide lightweight assistance to programmers, the module system and the type system are flexible and easy to use."[BAT14] In fact, a substantial amount of TypeScript's popularity can be attributed to two major factors: The ease of adoption and support for gradual adoption and the strong focus on developer productivity and editor integration, resulting in TypeScript being the 2nd most beloved programming language of 2020 according to the 2020 Developer Survey conducted by StackOverflow [Ove20].

A list of type constructs TypeScript offers can be found in table 5.1.

One of the major reasons TypeScript was presumed to yield good results is the focus on type annotations for JavaScript which is the origin for JSON and therefore provides good modeling capabilities for JSON objects. However, TypeScript is not a sound type system. For type systems, soundness means "that all of type-checked programs are correct (in the other words, all of the incorrect program can't be type checked)"[Chi14]. If a sound type system "says" that a variable has a particular type, then it is ensured that type correctly describes the value at runtime. As TypeScript, like many languages, allows type casting or type expectations that may not hold up at runtime and does not do type validation at runtime, the runtime type of a particular value may be incorrectly described. While sound type systems would therefore lead to better results in terms of correctness, it was determined the price of adoption may be significantly decreased due to a severe impact presumed with regards to developer experience. Therefore, in order to address the lack of soundness, implementation code should be present to reject unintended assignments at runtime, especially since the schema of a HTTP request is determined by the client sending the request and can not be assumed to match the type specification as expected by the API developers.

5.1.5. Configuration

Similar to annotations, it is possible to extract documentation from configuration or project files, if assumptions about the format or can the way the program uses this configuration can be made. As an example for NodeJS projects, the author and license

Name	Description	Example Value	Example Type Definition
Boolean	Logical Operators	true, false	boolean
Number	JavaScript floating point number	3.1415, 0xf00d	number
String	Textual data	"Hello", 'World'	string
Array*	Array Collections	[1,2,3]	number[], Array<number>
Tuple	Fixed length, fixed type ordered Arrays	[1, true]	[number, boolean]
Enum	Fixed Enumerations	Color.RED	enum Color {RED, BLUE}
Void	absence of a type	-	void
Undefined	unassigned value	undefined	undefined
Unknown	Type without information	1, "", false	unknown
Any	unknown type without type checking	1, true, ""	any
Null	explicit unavailability of data	null	null
Never	type of values that never occur	-	never
Object	JavaScript object type	{}	object
Object Literal	Key/value mapping	{a: true}	{a: boolean}
Interfaces*	Reusable, named object literals	{a: true}	interface A {a: boolean}
Classes*	Object Literals from JavaScript Classes	{a: true}	class A { a: boolean }
Literal Types	Single fixed value	'fixed'	'fixed'
Bigint	Safe representation for big numbers	9007199254740991n	bigint
Symbol	anonymous, unique value	Symbol("unique")	symbol
Intersection	intersecting combination type	{a: true, b: ""}	{a: boolean} & {b: string}
Union	combination type	true	boolean | string
typeof Operator	Infer type from data	-	typeof "string"
Mapped Type*	type mapping for object types	StringifyValues <a: boolean> = {a: ""}	type StringifyValues<T> = { [P in keyof T]: string}
Conditional Type*	non-uniform type mappings	true	string extends number ? string : boolean
Type Alias	renames type definition	"a word"	type Word = string

* Generic Types

Table 5.1.: TypeScript types

information can be read from a `package.json` file, where this information is usually preserved in a structured form. Similarly, many frameworks accept some form of configuration which provides reliable information about the program behavior. All of this static information should be used for knowledge extraction.

5.2. Extracting Documentation

After identifying the most common sources of (augmented) knowledge, it's important to compare popular approaches on extracting this knowledge in order to acquire these relevant inputs that can be transformed into an API description.

5.2.1. Extracting Documentation using Reflection

Several popular OpenAPI Projects in various languages take advantage of a program's ability to inspect itself at runtime, called reflection. In practice, this constitutes the overall most common OpenAPI documentation technique. Reflection capabilities are especially common in object oriented programming languages that heavily rely on class based program organization. In these languages, it is therefore very common to wrap the response handlers as class (instance) methods and data transfer objects (DTO) in classes with properties. By relying on metadata from annotations, reflection can be used to implement aspect oriented programming paradigms to enable developers to define complex http request/response interactions using familiar programming constructs in combination with via the use of annotations. Regardless of language, comments are usually removed at runtime. Subsequently, (structured) comments can not be extracted via reflection. In order to address this issue, frameworks use non-functional annotations to provide descriptions summaries etc. and accept the lack of editor support as outlined in 5.1.2. Another limitation of reflection capabilityes is rooted in type erasure. Type erasure is an optimization strategy that removes type information before execution. Depending on the language, this optimization may occur during compilation/transpilation or during load time, but in both cases it limits the ability to access type information required to accurately describe type schemas. As TypeScript targets JavaScript runtimes like V8[3], all TypeScript type annotations are removed. To combat type erasure, TypeScript provides a compiler flag called "emitDecoratorMetadata" [4] [5]. If enabled, during transpilation, TypeScript

[3] `https://v8.dev/`
[4] `https://github.com/Microsoft/TypeScript/issues/2577`
[5] `https://github.com/Microsoft/TypeScript/issues/2577`

adds functions to the transpiled code which use the Reflect API[6] and add metadata for decorated properties. The relevant metadata is then persisted in the metadata object of the property available under the "design:type" key, which includes a basic type information. Possible values are references to the String, Boolean, Number, Object, Array, or to user defined class constructor functions. Any additional information, which may already be present in the type annotations must still be provided via Decorator arguments, leading to a high volume of duplicated modeling effort which may introduce mismatches.

An example of the transpiled code of a class (Fig. 5.3 using this setting is shown in Figure 5.4.

While type erasure is less impactful on the ability to reconstruct expressive schemas in other typed languages, most statically typed languages rely on this optimization, especially when working with Generics. For example, in Java, Generics are only checked at compile-time for type-correctness. The generic type information is then removed, i.e. `List<String>` will be converted to type `List` [Doc19] [Ben18].

Although modeling through the use of classes in a different language may look convenient, it presents its own set of challenges. As classes are very dynamic, static analysis often can not be used to correctly analyze the serialized schematic representation of a class instance, requiring manual annotation to help the documentation tool. TypeScript interfaces therefore provide a more powerful way to express the schema of a JavaScript object which itself makes it more trivial to correctly produce a schematic representation.

[6] `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Reflect`

Listing 5.3: Decorated class before transpilation

```
import { ApiProperty } from "@nestjs/swagger";
import { AnEnum } from "../enum";
import { AnotherClass } from "./anotherclass.dto";

export class CreateCatDto {
  @ApiProperty()
  readonly name!: string | null;

  @ApiProperty()
  readonly age!: number;

  @ApiProperty()
  readonly breed!: string;

  @ApiProperty()
  readonly tags?: string[];

  @ApiProperty()
  createdAt!: Date;

  @ApiProperty()
  readonly options?: Record<string, unknown>[];

  @ApiProperty()
  readonly enum!: AnEnum;

  @ApiProperty()
  readonly tag!: AnotherClass;

  nested!: {
    first: string;
    second: number;
  };
}
```

Listing 5.4: Decorated class property after transpilation with metadata

```
"use strict";
var __decorate = // Omitted, calls Reflect.decorate() or Object.defineProperty
    ();
var __metadata = // Omitted, wraps Reflect.metadata(k, v);
// Omitted
const enum_1 = require("../enum");
const anotherclass_dto_1 = require("./anotherclass.dto");
class CreateCatDto {
}
__decorate([
    swagger_1.ApiProperty(),
    __metadata("design:type", Object)
], CreateCatDto.prototype, "name", void 0);
__decorate([
    swagger_1.ApiProperty(),
    __metadata("design:type", Number)
], CreateCatDto.prototype, "age", void 0);
__decorate([
    swagger_1.ApiProperty(),
    __metadata("design:type", String)
], CreateCatDto.prototype, "breed", void 0);
__decorate([
    swagger_1.ApiProperty(),
    __metadata("design:type", Array)
], CreateCatDto.prototype, "tags", void 0);
__decorate([
    swagger_1.ApiProperty(),
    __metadata("design:type", Date)
], CreateCatDto.prototype, "createdAt", void 0);
__decorate([
    swagger_1.ApiProperty(),
    __metadata("design:type", Array)
], CreateCatDto.prototype, "options", void 0);
__decorate([
    swagger_1.ApiProperty(),
    __metadata("design:type", String)
], CreateCatDto.prototype, "enum", void 0);
__decorate([
    swagger_1.ApiProperty(),
    __metadata("design:type", anotherclass_dto_1.AnotherClass)
], CreateCatDto.prototype, "tag", void 0);
exports.CreateCatDto = CreateCatDto;
```

5.2.2. Extracting Documentation using Abstract Syntax Tree Parsing

As laid out in the previous section, reflection capabilities vary between Languages, but in almost all cases, some amount of information is lost at runtime. In order to avoid this issue, a method that works closed to the source code itself is required. While working on the source code files directly is possible, intuition suggests there may be a better representation: The Abstract Syntax Tree (AST).
An AST is a language specific representation of the syntax of a programming language in a hierarchical tree-like data-structure. The tree represents all of the constructs in the language and their subsequent rules. While not every character may be preserved, an AST is required to contain all structural information. In typed languages, this data-structure is also used as the input to type checkers, therefore, all type annotations are present in their entirety. ASTs are specific to programming languages, but research for universal syntax trees is being conducted. [Tec18]

In TypeScript, the compiler API can be used to generate the AST based by passing one or more entry files to the program. Different methods of the API then provide the ability to traverse the AST. Visualization tools like an AST viewer are often useful to gain a better understanding (see footnote [7]).

One property specific to this approach is that the modeling of the type schema has to be mostly static. This is a benefit in the sense that it is easier to generate the OpenAPI Specification without starting the application, however the downside of this approach is that it is harder to change formulate a highly dynamic model that depends on the runtime environment. As an example we could imagine a scenario, where a bidding API exposes an endpoint at which bids are submitted. In this case, only bids higher than the last highest price can be submitted. A runtime approach could return a specification which includes the minimum price that can currently be submitted, whereas a build time approach can not express this Schema requirement without an additional runtime component, which updates the previously generated specification in a similar fashion. It should however be noted that TypeScript only evaluates decorators once, so additional steps have to be taken in order to allow for this behavior (for example if the documentation allows the developer to provide a function that recalculates the minimum price every time), but enabling this behavior less complex than static, AST approaches.

[7] `https://bit.ly/3iUOl3c`

5.2.3. Extracting Documentation from a Type Checker

As the Abstract Syntax Tree only contains a representation of the structure of the program, additional APIs, often provided by a compiler or other language tooling are used to reduce the amount of complex work for example editor integrations have to perform in order to provide rich editor experiences like providing type information or autocomplete functionality. In TypeScript, a wrapper around the Type Checker API called *TSServer* implements a server to provide type information. Given a Node in the Abstract Syntax tree, it's possible to work with the type checker API directly to gather type information using the `getTypeAtLocation()` method. The interface of the returned type information object (TypeScript v3.9.3) is displayed in Fig. 5.5, possible type flags can be found in the Appendix A.2.

Listing 5.5: The `Type` interface

```
export interface Type {
    flags: TypeFlags;
    symbol: Symbol;
    aliasSymbol?: Symbol;
    aliasTypeArguments?: readonly Type[];
    // more properties here

    getFlags(): TypeFlags;
    getSymbol(): Symbol | undefined;
    getProperties(): Symbol[];
    getProperty(propertyName: string): Symbol | undefined;
    isUnion(): this is UnionType;
    isIntersection(): this is IntersectionType;
    isUnionOrIntersection(): this is UnionOrIntersectionType;
    isLiteral(): this is LiteralType;
    isStringLiteral(): this is StringLiteralType;
    isNumberLiteral(): this is NumberLiteralType;
    isTypeParameter(): this is TypeParameter;
    isClassOrInterface(): this is InterfaceType;
    isClass(): this is InterfaceType;
    // some methods removed
}
```

5.3. Comparison

The following tables 5.2 and 5.3 give an abbreviated, high level overview of our previous findings. A checkmark (✓) indicates overall good viability, a dash (–) indicates limited viability, a cross (✗) signals limited or no viability.

Target	Type System	Annotations	Structured Comments	Configuration
JSON Schema	✓	–	✗	✗
OpenAPI Schema	–	✓	✗	✗
Usability aid	✗	–	✓	✓
Metadata	✗	–	✗	✓

Table 5.2.: Comparison of documentation sources according to their viability for API description elements

Target	Type System	Annotations	Structured Comments	Configuration
AST Parsing	–	✓	✓	✓
Type checker API	✗	✗	✗	✗
Reflection	–	✓	✗	✓

Table 5.3.: Viability of extraction techniques with regards to source format

To summarize, there is no single approach that is able to satisfy all requirements on it's own. Although popular approaches across multiple languages usually leverage a combination of Annotations and Reflection, this approach is often cumbersome, as it requires a lot of additional work and repetition, duplicating knowledge between type system and annotations for the documentation tool. Also, a deep understanding of reflection capabilities is required in order to recognize when additional annotations may of may not be necessary."In-Editor" developer experience benefits associated with JSDoc / JavaDoc are unavailable. Interestingly, we noticed that the languages with best support for Reflection had the most limited support for describing JSON.

In the following chapter we will therefore investigate a different (hybrid) approach, that combines AST Parsing and working with the type checker API as a compile time technique instead of Reflection.

6. Building and integrating an OAS Generation Framework

While we would've ideally built the entire framework on our own, due to limited time and potential re-implementation of more advanced existing, publicly available work, our approach builds on top of the tsoa [1] framework. The existing implementation provides an approach that already does limited work on AST parsing.

We added and merged additional work that improves the OAS generation and incorporates information from the type checker, as outlined in 5, to resolve OpenAPI definitions for TypeScript types where an AST based approach would be too complex or inflexible.
The goal of these additions was to provide a broader coverage of all the requirements as presented in Chapter 4.

6.1. Building the OAS

The overall goal of the tsoa project is to infer a correct OpenAPI specification based on functional framework annotations and TypeScript types. To allow for a better understanding of the architecture and API of a tsoa project, we refer to the *Getting Started Guide*[2], which was developed as part of this thesis. More details will be presented in Section 6.5. At a high level, the tsoa cli command takes a TypeScript program as the input, traverses the AST and it's nodes, and returns both the validation and routing "glue code" between the controller layer and the underlying web framework (i.e. express[3] or koa[4]), and an OpenAPI Specification document. Internally, tsoa leverages the TypeScript Compiler API to construct the AST in order to collect `ClassDeclaration` AST Nodes, filtering `ClassDeclarations` with a `@Route(<basePath>)` Decorator, therefore identifying all Entrypoints into the application's controller layer.

[1] https://github.com/lukeautry/tsoa
[2] `https://tsoa-community.github.io/docs/getting-started.html`
[3] `https://expressjs.com/`
[4] `https://koajs.com/`

6.1.1. Operation resolution

Each of the controller class AST Nodes is subsequently passed into a method analysis, which checks the `ClassDeclaration` for child nodes of kind `MethodDeclaration`. All the methods declared withing the class are filtered, only retaining relevant method definitions (`public` or no modifier, not explicity ignored, with a HTTP Method decorator (`@Get(<path>)`, `@Post(<path>)` etc.).

In combination with the `basePath`, the `path` will be used to construct an OpenAPI `Path Item Object`. Each of the HTTP Method decorators will then be used to fill the corresponding fixed fields and prepare an OpenAPI `Operation Object`. The `Operation Object` object is composed of several properties, displayed in the lower half of Fig. 4.4, below the link node. In addition to descriptive properties like `tags` (set in tsoa using the `@Tags(...string[])` decorator), `description` (parsed from the JSDoc of the `MethodDeclaration`), `summary` (parsed from the JSDoc `@summary` tag), and `operationId` (defaults to the method name, may be overridden by a `@OperationId(name: string)` decorator, the `Operation Object` contains 5 functional properties. If an operation requires Authentication/Authorization, the corresponding `security` field will be set by parsing the `@Security(name: string)` annotation at the class or method level. As these security definitions are reusable and applied for several Operations, according to requirement A9, this definition should also be reused and therefore only referenced (using JSON Pointers) in the `Operation Object`. In order to fill the remaining 3 non-descriptive fields (parameters, responses and requestBody), tsoa internally uses a `TypeResolver` to transform the TypeScript type definitions into JSON Schema definitions.

6.1.2. Schema resolution

While there are other type construct TypeScript uses, only a broader subset of these types are relevant to determine the Schema of the return type of the Operations defined via Controllers. Returning instances of function types or types used to model JavaScript's `this` are not useful HTTP Responses, therefore types for specification of those values are not be resolved and will instead result in a compilation error. For NodeJS data types like Dates, Buffers and Streams, which are class instances that have a custom serialization process, but can be transformed into HTTP Responses, special transformations are required. A list of special interfaces, as an amendment to table 5.1 presented in section 5.1.4, can be found in table 6.1.

Name	Description	Example Value	Type Definition
Date	Date(time) class	new Date()	Date
Buffer	Buffer clase	new Buffer()	Buffer
ReadableStream*	Readable stream class	new ReadableStream()	Readable
Promise*	Asynchronous operation result	Promise.resolve(5)	Promise<number>

* Generic Type

Table 6.1.: TypeScript Controller return types

After defining the input types, we can now formulate a grammar for the output domain, a JSON Schema derivation with OpenAPI extensions, reduced to a minimal subset which can be used to map the input type domain. The grammar for this OpenAPI schema (tables 6.2, 6.3, 6.4, 6.5, 6.6, 6.7, 6.8) is based on the formal Grammar for JSON Schema as presented by Pezoa et. al [Pez+16].

JSDoc	:=	{ (defs,)? JSch }
defs	:=	{ "components": { "schemas": { kSch (, kSch)*}}}
kSch	:=	kword: { JSch }
JSch	:=	(res (, res)*)
res	:=	type | strRes | numRes | arrRes | objRes | multRes | refSch | description | example | nullable | default | format
type	:=	"type" : typename
typename	:=	"string" | "integer" | "number" | "boolean" | "null" | "array" | "object"
description	:=	"description": string
example	:=	"example": Jval
nullable	:=	"nullable": bool
default	:=	"default": Jval
format	:=	"format": string

Table 6.2.: Base Schema

strRes	:=	minLen | maxLen | pattern
minLen	:=	"minLength": n
maxLen	:=	"maxLength": n
pattern	:=	"pattern": "regExp"

Table 6.3.: String Schema

numRes	:=	min | max | multiple
min	:=	"minimum": r (,exMin)?
exMin	:=	"exclusiveMinimum": bool
max	:=	"maximum": r (,exMax)?
exMax	:=	"exclusiveMaximum": bool
multiple	:=	"multipleOf": r (r >= 0)

Table 6.4.: Numeric Schema

objRes	:=	prop | addprop | req | minprop | maxprop
prop	:=	"properties": { kSch (, kSch)*}
kSch	:=	kword: { JSch }
addprop	:=	"additionalProperties": (bool | JSch)
req	:=	"required": [kword (, kword)*]
minprop	:=	"minProperties": n
maxprop	:=	"maxProperties": n

Table 6.5.: Object Schema

arrRes	:=	items | additems | minitems | maxitems | unique
items	:=	(sameitems | varitems)
sameitems	:=	"items": { JSch }
varitems	:=	"items": [{ JSch }(,{ JSch })*]
additems	:=	"additionalItems": (bool | { JSch })
minitems	:=	"minItems": n
maxitems	:=	"maxItems": n
unique	:=	"uniqueItems": bool

Table 6.6.: Array Schema

multRes	:=	allOf | anyOf | enum
anyOf	:=	"anyOf": [{ JSch } (, { JSch }) *]
allOf	:=	"allOf": [{ JSch } (, { JSch }) *]
enum	:=	"enum": [Jval (, Jval)*]

Table 6.7.: Meta and Enum Schema

refSch	:=	"$ref": "uriRef"
uriRef	:=	(address)? (# / JPointer)?
JPointer	:=	(/ path)
path	:=	(unescaped | escaped)
escaped	:=	~0 | ~1

Table 6.8.: Referenced Schema

The target (OpenAPI) schema includes type mappings, which can not expressed through TypeScript. As an example, a TypeScript string can not be used to define a pattern. Therefore, tsoa supports the ability to fall back to annotations in these cases (which will be validated on incoming Requests via the integrity layer) using JSDoc annotations.

```
interface Password {
  /**
   * @minLength 8
   * @maxLength 20
   * @pattern ((?=.*\d)|(?=.*\W+))(?![.\n])(?=.*[A-Z])(?=.*[a-z]).*$
   */
  password: string;
}
```

Using these annotations, we can allow more fine-grained modeling while enforcing parameter integrity at runtime. However, as of the writing of this thesis, response integrity for these annotations is not enforced. A list of annotations can be found in the documentation online [5].

6.2. Contributions

As we based the work done within this thesis on an existing project, some of the TypeScript type to JSON Schema transformations were already present, so our contributions included: Type Aliases, Conditional Types, Mapped Types, as well as the Unknown

[5] https://tsoa-community.github.io/docs/annotations.html

Top Type, which we will describe in more detail. A full list of contributions to the tsoa codebase can be found online[6].

Reusable Schema naming

One of the requirements elicited in Chapter 4 is A9: An exhaustive list of all major components that exist within the API. This requirement intends to promote a better understanding of the Resources the API operates on. OpenAPI allows these reusable schemas to be defined as *reusable component schemas* which can be referenced via JSON Pointer. This approach is similar to type references TypeScript supports to name types:

"Classes, interfaces, enums, and type aliases are named types that are introduced through class declarations [...], interface declarations [...], enum declarations [..], and type alias declarations [...]. Classes, interfaces, and type aliases may have type parameters and are then called generic types. Conversely, named types without type parameters are called non-generic types."[Mic16]

In order to allow for type alias and proper generic interface support, we initially needed to change the resolution of type aliases as a referenceable type with a consistent, OpenAPI compatible naming scheme (matching the RegEx `^[a-zA-Z0-9\.\- _]+$`).

The existing naming scheme was not suitable for any reasonably complex TypeScript types with Type Arguments.

For example,

```
let a: MyModel<T | U>
```

would be transformed to an OpenAPI component named *MyModelobject* and therefore clash with a

```
let a: MyModel<T | U | V>
```

definition.

Multiple type arguments were not supported either.

Therefore, a new naming scheme was proposed. The new naming scheme applies OpenAPI compatible escapes for TypeScript reference names while preventing distinct references to share a common name (reference clashes). A pseudo-implementation would look like this:

[6] https://github.com/lukeautry/tsoa/issues?q=label%3Aba+is%3Aclosed

```
function getRefTypeName(name: string): string {
  return name
    .replace(/<|>/g, '_') // Replace | with _
    .replace(/\s+/g, '') // Trim whitespace
    .replace(/,/g, '.')
    .replace(/\'([^']*)\'/g, '$1') // Strip ' around string literals
    .replace(/\"([^"]*)\"/g, '$1') // Strip " around string literals
    .replace(/&/g, '-and-')
    .replace(/\|/g, '-or-')
    .replace(/\[\]/g, '-Array')
    .replace(/{|}/g, '_')
    .replace(/([a-z]+):([a-z]+)/gi, '$1-$2')
    // Replace 'propertyName: type' with 'propertyName-type'
    .replace(/;/g, '--')
    .replace(/([a-z]+)\[([a-z]+)\]/gi, '$1-at-$2');
    // Replace member access: 'MyModel["prop"]' => 'MyModel-at-prop'
}
```

With a 1:1 mapping of a unique TypeScript Reference Type name to a unique OpenAPI Schema Object name, we introduced a naming algorithm that transforms reference names for type aliases to an intermediate representation withing tsoa that is compatible with existing reference types like interfaces.

Type Aliases

A type alias serves as an alias for the type specified in the type alias declaration. Unlike an interface declaration, which always introduces a named object type, a type alias declaration can introduce a name for any kind of type, including primitive, union, conditional, mapped and intersection types. [Mic16]

A type alias may have type parameters that serve as placeholders for actual types to be provided when the type alias is referenced via a type reference. A type alias with type parameters is called a generic type alias. The type parameters of a generic type alias declaration are in scope and may be referenced in the aliased Type. [Mic16]

Type aliases are used via type references. A type reference is composed of the name of the type alias with a list of comma separated type arguments matching the type parameters of the type alias declaration. Type references to generic type aliases produce instantiations of the aliased type with the given type arguments. Writing a reference to a non-generic type alias has exactly the same effect as writing the aliased type itself, and writing a reference to a generic type alias has exactly the same effect as writing the resulting instantiation of the aliased type [Mic16].

```ts
type StringOrNumber = string | number;
type Text = string | { text: string };
type NameLookup = Dictionary<string, Person>;
type ObjectStatics = typeof Object;
type Callback<T> = (data: T) => void;
type Pair<T> = [T, T];
type Coordinates = Pair<number>;
type Tree<T> = T | { left: Tree<T>, right: Tree<T> };
```

As tsoa provides additional JSON Schema annotations, our approach allows for attaching these annotations above the type alias declaration. A common application of annotation-enhanced type alias translation to OpenAPI schema is visualized in Fig. 6.1.

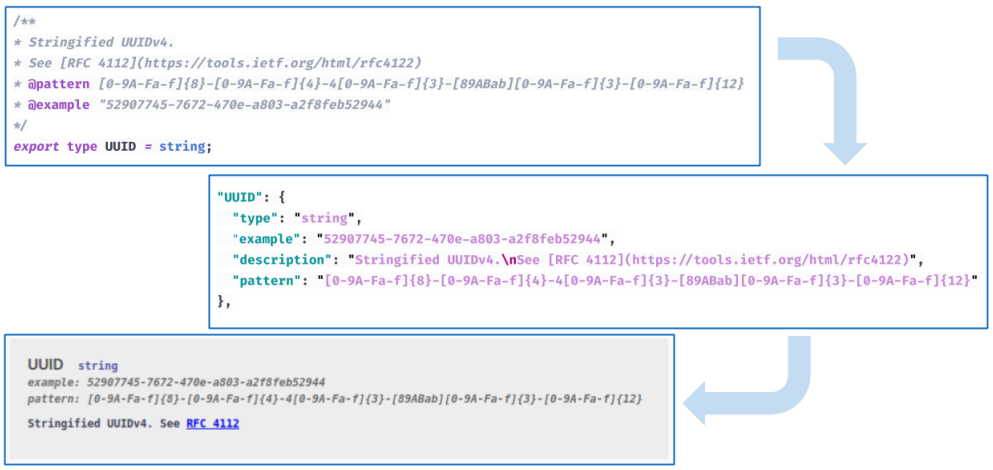


Figure 6.1.: Declaration and translation of an annotation enhanced type alias

As the usage of type aliases suggests a certain amount of reusability within the program, they are considered a reusable component (A9). However, unlike regular references, the schema of a generic type alias reference depends on the type arguments. Due to limitations of the schema underlying OpenAPI 3, no mechanism to provide arguments to a schema is available. This means in order to represent the schema of a generic type alias reference, a new version of the generic schema has to be rendered. As the previous subsection introduced the naming scheme for this kind of schema reference, this subsection will provide the concepts used to implement the translation process between a generic type alias reference to a correct schema definition.

Whenever tsoa's type resolver encounters a type node of kind `TypeReference`, a new

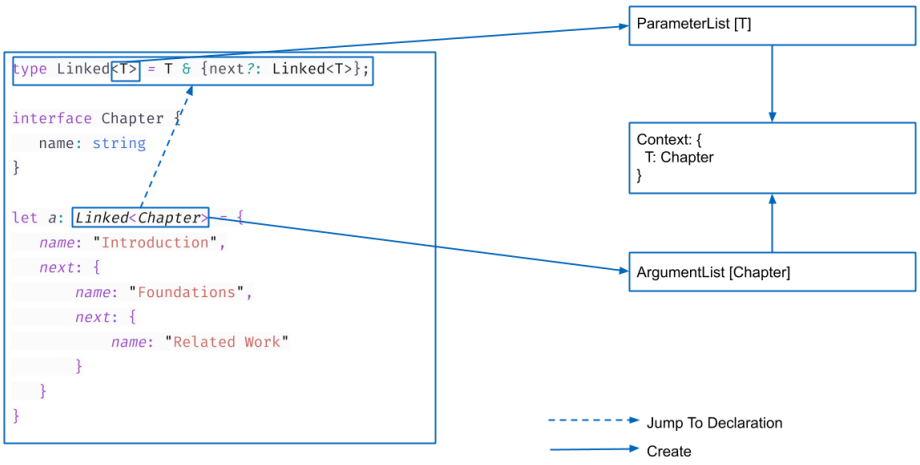


Figure 6.2.: Context creation for generic type alias resolution

resolution context is created. During processing of the type alias reference, which contains all the type arguments, the resolution algorithm jumps to the type alias declaration and collects all the type parameter declarations of the type alias declaration based on the type parameter declaration name. If the declaration defines defaults, these defaults are added to the context. Now, each value of the type arguments is added to the context, overriding the default if available. After the context is created (see Fig. 6.2), the declaration can be resolved, replacing each occurrence of a type parameter with the type argument (see Fig. 6.3). The circular reference detection used to escape infinite recursion is not displayed. As generic type aliases may be nested, this context is created until the resolution is complete, so a nested alias resolution can access the parent context if the parameter is not applied in the child context.

As this contextualized resolution and generic naming can also be used for generic interface references, the interface resolution was adapted to use the same implementation as well.

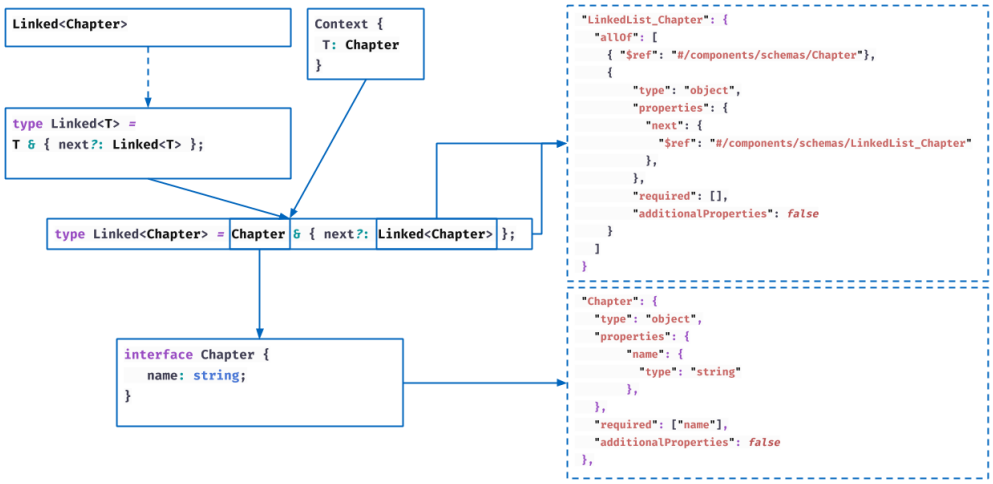


Figure 6.3.: Context utilization for generic type alias resolution

Conditional Types

One of the types most recently added to TypeScript (in version 2.8) are conditional types, which "add the ability to express non-uniform type mappings"[7]. Related to the if/else statements, "a conditional type selects one of two possible types based on a condition expressed as a type relationship test"[8]. As the TypeScript Specification [9] does not mention conditional types at the point of writing, we will use the relevant parts of the specification of conditional types which refers to value types as presented in the announcement post of TypeScript 2.8.

The shape of a Conditional type C can be denoted as `T extends U ? V : W`, where `T extends U` is called a type relationship test, which checks if T can be assigned to U. A simplified explanation of the Conditional type C would then be: Given type instances T, U, V, W, C can be resolved to V if T `extends` U, meaning $t \in U \forall t \in T$ (T can be

[7] https://www.staging-typescript.org/docs/handbook/release-notes/typescript-2-8.html#conditional-types

[8] https://www.staging-typescript.org/docs/handbook/release-notes/typescript-2-8.html#conditional-types

[9] https://github.com/Microsoft/TypeScript/blob/master/doc/spec.md

assigned to U), W otherwise.

$$f(T, U, V, W) = \begin{cases} V : T \subseteq U \\ W : T \not\subseteq U \end{cases} \tag{6.1}$$

Conditional Types can be partially applied, using type aliases. A type alias

```
type C<T> = T extends string ? 'true' : 'false';
```

therefore is a partial application of f, where $U = string, V = "string", W = "other"$, therefore `C<T>` $= f(T, string, "string", "other")$. In practice however, TypeScript's implementation deviates from our previous definition. First, the `extends` keyword does not work exactly like $\subseteq$. Instead, if the checked type is a naked (without being wrapped in another type, i. e., an array, a tuple, a function, a promise or any other generic type) type parameter, TypeScript distributes over union types during instantiation. The conditional type is then also called a distributive conditional type. This means a conditional type instantiation such as

```
type C<T> = T extends string ? 'true' : 'false';
```

would lead to `C<string | boolean>` being distributed over the conditional type, meaning `C<string | boolean> = 'true' | 'false'` as `string` $\subseteq$ `T = string | boolean`, `string` $\subseteq$ `U = string` $\implies$ $'true' \in$ `C<string | boolean>` and `boolean` $\subseteq$ `T = string|boolean`, `boolean` $\not\subseteq$ `U = string` $\implies$ $'false' \in$ `C<string | boolean>`.

The naked type parameter pre-condition for distribution of conditional types means e.g.

```
type C<T> = T extends Promise<string> ? 'true' : 'false';
```

evaluates `C<Promise<string|number>>` to `"false"` as `(string | number[])` is not naked, therefore will not be distributed and TypeScript asserting that `Promise<string|number>` $\not\subseteq$ `Promise<string>`.

Similarly, the distribution seems to be currently (version 3.9.5) limited to explicitly user defined type unions, as

```
type C<T> = T extends 1 ? 2 : 3;
```

evaluates to `C<number> = 3`, where 3 is a numeric literal type, although $1 \subset number \implies 2 \in$ `C<number>`.

This union distribution is the concept that enables some of powerful predefined TypeScript Utility types:

```
/**
 * Exclude from T those types that are assignable to U
 */
type Exclude<T, U> = T extends U ? never : T;
```

```
/**
 * Extract from T those types that are assignable to U
 */
type Extract<T, U> = T extends U ? T : never;
```

While it may have been tempting to implement parsing the Conditional Node in context based on the AST nodes, the additional definitions and current limitations make it considerably more complex to parse Conditional Types. It would incorporate re-implementing the type resolution as presented in the Changelog, including type relationship tests and union distribution. Additionally, TypeScript's implementation may be subject to change (e.g. addressing some og the current limitations) in the future and therefore the parsing may have to be adapted. Instead, an approach that works with TypeScript's type checker was deemed a better fit given the complexity of the problem. Once tsoa detects a type reference node, which instantiates a conditional type, the TypeScript compiler API is used to retrieve an instance of a Compiler `Type`. The interface of the result this API provides is listed in Fig. 5.5.

Once the `Type` is acquired, the provided methods on the type instance are used to narrow down the corresponding schema. If a `Symbol` is present on the type instance, the Type's symbol can be used to "jump" to the declaration node in the AST and proceed from there. In other cases, the type instance methods can be used to to transform the `Type` instance into a virtual `TypeNode` (virtual TypeNodes imitate the format of an AST Node, but have no real position) so the logic otherwise used to resolve real Nodes in the AST can be applied.

Mapped Types

Mapped types are type constructs to create new object types from existing object types and a type mapping expression. The new type hereby transforms each property from the the existing type in the same way, according to the type mapping expression. For Web APIs, the probably most common use case is the `Partial` type alias, provided by TypeScript, which is used to create a new version of an object type where all properties are optional. The `Partial` type can often be found as the body type for `PATCH`, `PUT` or `POST` requests.

```
/**
 * Make all properties in T optional
 */
type Partial<T> = {
    [P in keyof T]?: T[P];
};
```

In order to resolve these more complex type constructs, similar to Conditional Types, support for mapped types passes some of the work on to the Type Checker API. In order to retain all required information, the implementation was modified to capture the the referencing AST type reference node, including type arguments, if the mapped type is used via reference to a type alias declaration wrapping the mapped type. This allows the mapped type resolution, which operates on the mapped type declaration to ask the Type Checker for all the properties on the new type via a method call.

The unknown type

Similar to the `any` type, the `unknown` type does not impose any limitations on the values that can be assigned to a variable of that type.

"TypeScript 3.0 introduces a new top type `unknown`. `unknown` is the type-safe counterpart of `any`. Anything is assignable to `unknown`, but `unknown` isn't assignable to anything but itself and `any` without a type assertion or a control flow based narrowing. Likewise, no operations are permitted on an `unknown` without first asserting or narrowing to a more specific type. "[10]

The JSON Schema equivalent of the `unknown` type is an empty schema. During validation, inputs of type `unknown` are not validated.

The respective Pull Requests are linked in footnotes [11] [12] [13] .

Type-Checked Alternative Responses

Before the addition of this PR, controller methods had one return type, used for the success response type and schema. While the framework provided a mechanism to describe alternative response types, it was required to throw an Error, catch that error and transform the error into a JSON response matching annotated shape. This was not type-safe and indeed error-prone because TypeScript does not perform any checks on thrown Errors.

With the addition of this PR, it's now possible to return these additional responses on a type-checkable channel (an involable function injected by declaring a decorated parameter in the request handler) that does not impact the regular return type.

The discussion regarding the proposed API of type-checked alternative responses[14]

[10] `https://www.typescriptlang.org/docs/handbook/release-notes/typescript-3-0.html#new-unknown-top-type`

[11] `https://github.com/lukeautry/tsoa/pull/559`

[12] `https://github.com/lukeautry/tsoa/pull/640`

[13] `https://github.com/lukeautry/tsoa/pull/729`

[14] `https://github.com/lukeautry/tsoa/issues/617`

and the implementation[15] can be found on GitHub, the respective description can be found in the API (Reference) Documentation of tsoa itself[16] [17] [18].

Author Information

Requirement D4 of API Reference documentation suggests providing contact information for technical support. In NodeJS projects, this information is located in the package declaration file called `package.json`. For information about the format of this file, we refer to the NodeJS guides [19] and the npm documentation [20]. Our contibution[21] allows tsoa to parse this information to fill the Contact Object of the OpenAPI specification. If the author is not the person responsible for providing technical support, this information can be provided via tsoa configuration (`tsoa.json`) instead.

6.3. Limitations

While we tried to outline several interesting upsides of an AST based approach, various limitations still exist. First, there are some limitations imposed by the modified JSON Schema Draft-00 that OpenAPI uses to formalize schemas. One of them is the inability to specify the order of Array item schemas. This currently makes it impossible to generate a correct 1:1 mapping from a TypeScript Tuple type to the OpenAPI 3.0 schema. While newer drafts of JSON Schema would support this construct, we will have to terminate generation of the OAS document if tsoa detects tuple types exposed to the client.

Another limitation is the lack of type checking TypeScript imposes on throw/catch statements. Given a method definitions it is very complex to statically analyze which Errors may occur during execution which makes it hard to track errors, especially since the underlying web framework may choose to not handle errors or transform them to an arbitrary JSON response. Therefore, no guarantees around correctly documenting responses created by throwing Errors can be provided. While developers can address some of these issues by using type-checked responses instead of throwing an error, using a global error handling to transform the Error into a response, and adding

[15] `https://github.com/lukeautry/tsoa/pull/699`
[16] `https://tsoa-community.github.io/docs/error-handling.html#typechecked-alternate-responses`
[17] `https://tsoa-community.github.io/reference/globals.html#res`
[18] `https://tsoa-community.github.io/reference/globals.html#tsoaresponse`
[19] `https://nodejs.org/en/knowledge/getting-started/npm/what-is-the-file-package-json/`
[20] `https://docs.npmjs.com/files/package.json`
[21] `https://github.com/lukeautry/tsoa/pull/710`

`@Response` annotations manually, tsoa can not provide proper tooling to verify the implementation against the annotation at this point.

Another limitations is based on lacking support for `@link` tags in JSDoc. While this issue is being tracked in the TypeScript issue tracker[22], until the issue is resolved, we decided to not use the link tag as a source for OpenAPI links.

A minor inconvenience is the handling of the `@SuccessResponse` decorator. While it can be used to document the status code of the default response, the developer currently has to manually make sure the status code is actually set. Instead, this decorator should is actually set without any additional effort by the developer. As this change would be considered a breaking change, it's being tracked as a remaining issue to be resolved in the future [23].

6.4. Integrating the generated API descriptions into a holistic API strategy

As we have seen, tsoa can be used to compile the is-state of our application into an OpenAPI specification. However, this is-state may neither match our requirements (for example because the body properties are in snake case, while the requirements dictate the use of camel case), nor established best practices in HTTP APIs, such as offering various HTTP Methods to interact with resources or best practices regarding OpenAPI specifications like grouping conceptually related Endpoints using Tags.
In order to ensure this behavior, it's been very common to use linting, which refers to to tooling that flags suspicious code in software. Similar tooling exists in for OpenAPI specifications and while there are currently several openly available packages, we decided to outline the process using spectral[24] as it is actively maintained and popular in the OpenAPI community.

Spectral can be configured using a configuration file with yaml syntax (*.spectral.yml*). Internally, spectral checks the provided OpenAPI specification against a set of rules, which are intended to increase the quality or consistency of the OpenAPI specificiation under review.

Out of the box, spectral will apply the default ruleset that checks several more common recommendations [25]. Additionally, custom rulsets can be added[26].

[22] `https://github.com/microsoft/vscode/issues/57495`
[23] `https://github.com/lukeautry/tsoa/issues/723`
[24] `https://github.com/stoplightio/spectral`
[25] `https://stoplight.io/p/docs/gh/stoplightio/spectral/docs/reference/openapi-rules.md`
[26] `https://stoplight.io/p/docs/gh/stoplightio/spectral/docs/guides/4-custom-rulesets.md`

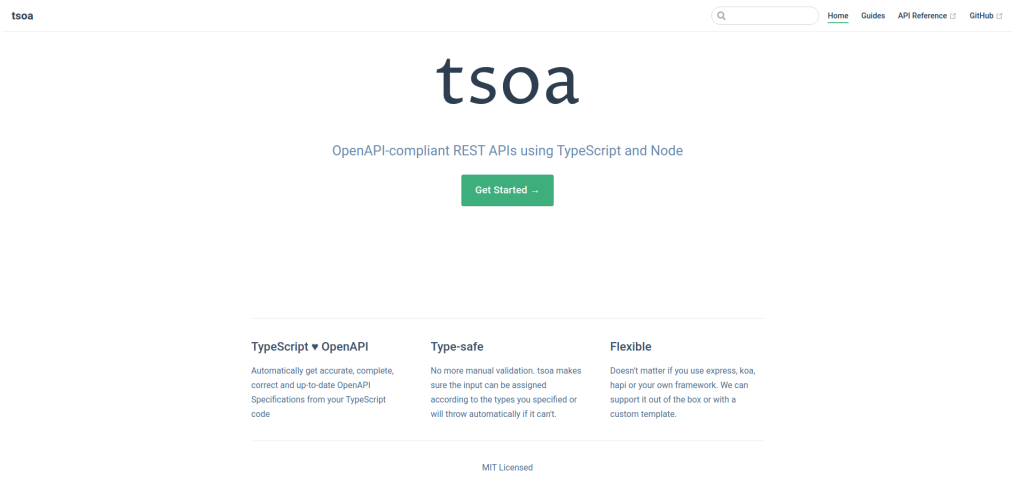


Figure 6.4.: Preview of the tsoa documentation

While it is impossible to list all the potentially viable custom rules, some more commonly known additional rules shall be pointed out:

- Limiting the amount of allowed status codes
- Requiring descriptions and tags
- Allowing only examples that match the JSON Schema
- Enforcing consistent casing for bodies, header, path and query parameters

6.5. Documenting the Approach

Similar to the recommendation developed for providers of Web APIs, tsoa should be documented using a two-pronged approach aswell: An API Reference documentation listing the details of all the options as well as a more high level, textual description to on-board new users. Previously, tsoa itself only offered a long, single page Readme and a reference to the tests. In order to address this lack of documentation, a Documentation was developed and an API Reference generation tool for TypeScript was integrated.

The repositories can be found online, both for the generated API Reference [27] and the high-level Documentation[28]. The rendered versions are available as well[29][30].

[27] `https://github.com/tsoa-community/reference`
[28] `https://github.com/tsoa-community/docs`
[29] `https://tsoa-community.github.io/reference/globals.html`
[30] `https://tsoa-community.github.io/docs/`

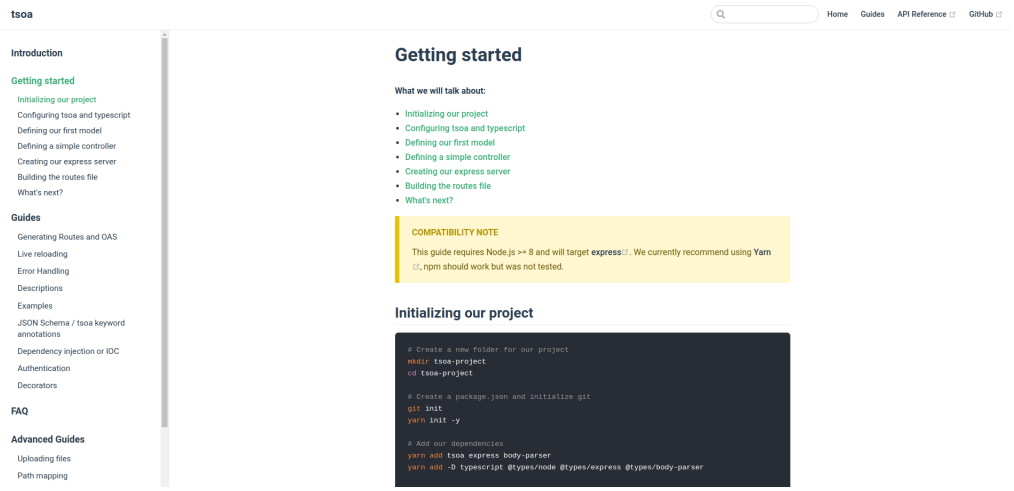


Figure 6.5.: Preview of the getting started guide of the tsoa documentation

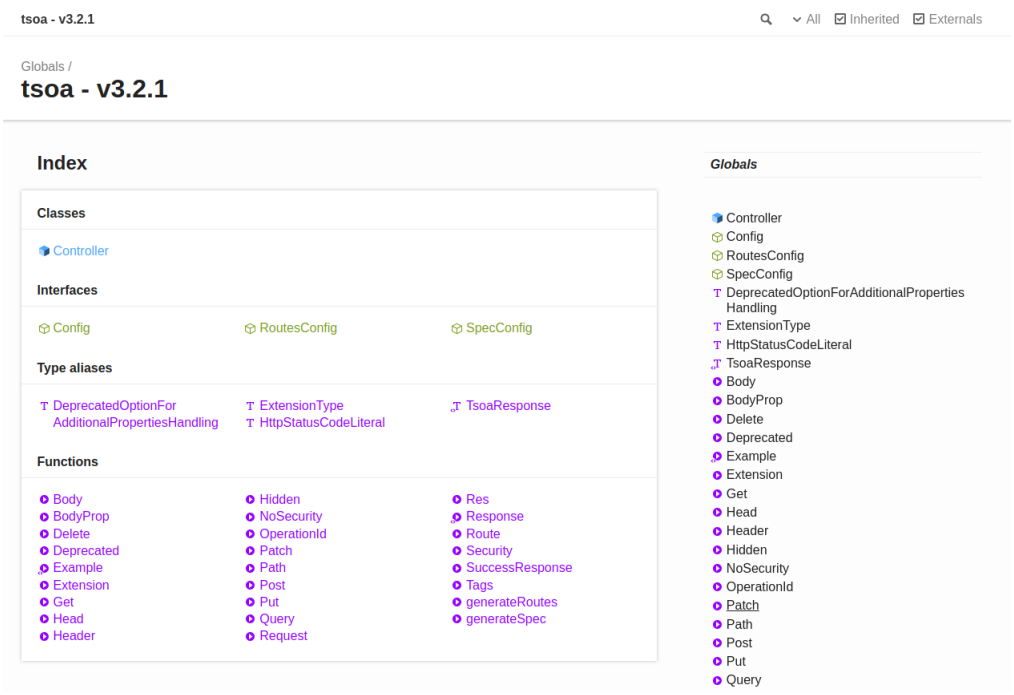


Figure 6.6.: Preview of the tsoa API Reference Documentation

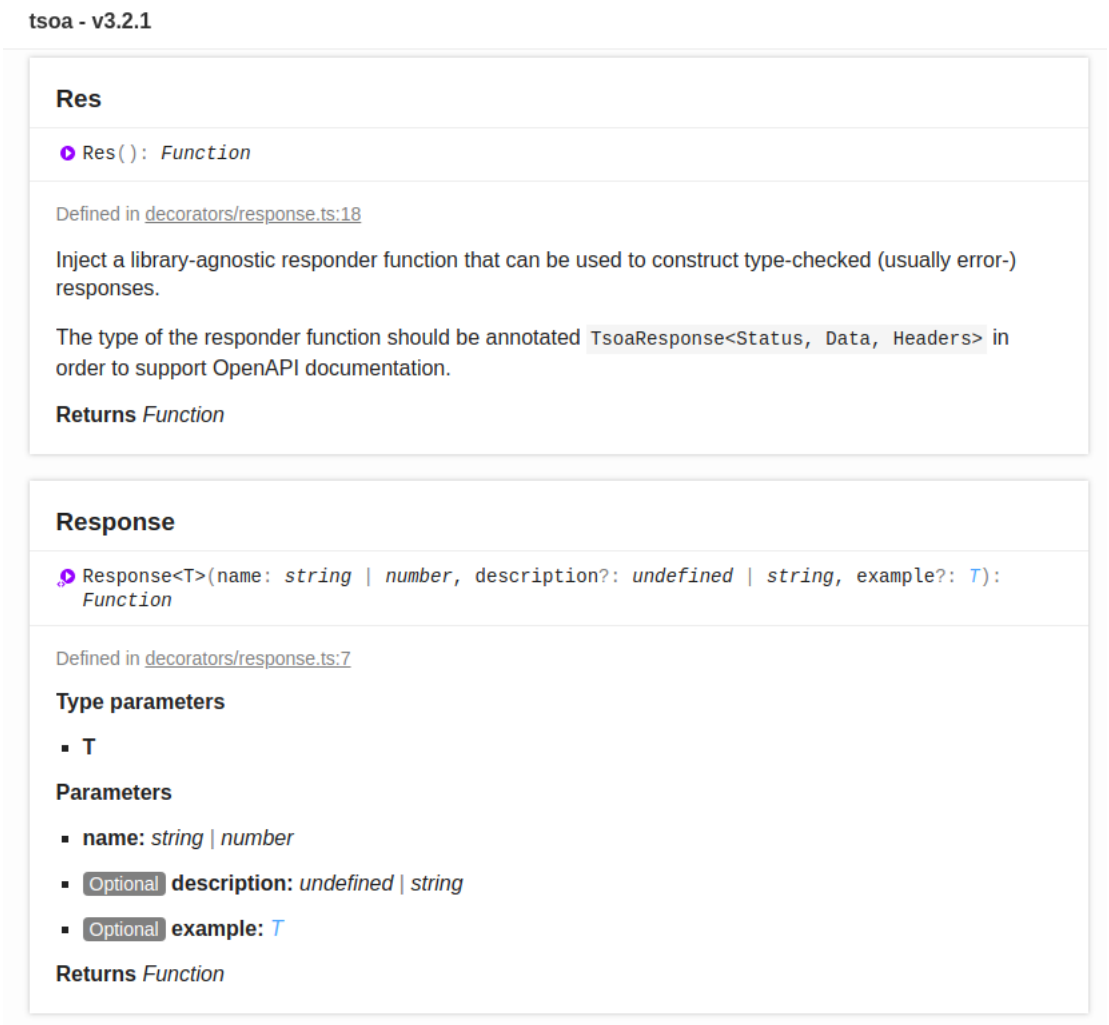


Figure 6.7.: Snipped of the tsoa API Reference Documentation

6.6. Case Study

We will outline the approach of our integration with an exemplary endpoint that describes the tooling used to improve the OpenAPI specification generated from Code.

The requirements are similar to the endpoint described in the modeling assignment of the evaluation A.1.3.

A mock of this controller can be found online [31]

In order to provide high quality documentation, the imaginary rental company decided to adopt the pipeline as proposed in the thesis.

Therefore, when implementing this controller, TypeScript will ensure that:

- All services are accessed properly (using static analysis).
- Type mismatches are prevented within the boundaries of the system.
- No unsafe operations are performed on services and parameters if they match the type annotations.
- The return matches the return annotation (here: the *Order* interface.

When building the controller using tsoa's command line:

- An OpenAPI specification will be generated from the TypeScript types and the JSDoc annotations.
- A runtime wrapper that ensures requests which don't match the TypeScript types will be rejected will be generated.

However, tsoa does not impose any limitations on the usability of the documentation. OpenAPI specifications can be very permissive and vague in many ways. The documentation after only defining the Controller and Models can be found online [32]. As we have outlined throughout the thesis, descriptions, examples, endpoint summaries or additional grouping of endpoints with related functionality contribute towards an easier understanding of the concepts used within the API.

Therefore, as proposed, the imaginary rental company uses a custom spectral ruleset to ensure these additional documentation artifacts are in place using a custom spectral ruleset (*.spectral.yml*).

[31] `https://gist.github.com/WoH/9e8778bbaefa3c4e60cbc0a5ecd8aff2`
[32] `https://woh.github.io/redoc-rentals/redoc-rentals-minimal.html`

```
extends: spectral:oas
rules:
  contact-properties: error
  oas3-valid-example: warn
  oas3-schema: warn
  oas3-parameter-description: error
  no-$ref-siblings: off
```

Using this ruleset, spectral suggests the following additions:

- Operation 'description' must be present and non-empty string.spectral(operation-description)
- Operation should have non-empty 'tags' array.spectral(operation-tags)
- OpenAPI object should have non-empty 'tags' array.spectral(openapi-tags)

While tsoa provides ways to infer that information from JSDoc (description) and the `@Tags()` decorator, this information was not present, however, since the program itself was valid, a correct OpenAPI spec could be produced. Now, our imaginary rental company decides to add a description for the endpoint, a short summary and tags to the endpoint's source code implementation and consistent specifications are ensures [33]:

```
@Tags("order")
export class OrdersController extends Controller {
  // ...omitted

  /**
   * This endpoint is used to rent a boat or a ship.
   * @summary Add a new rental order.
   * @param badRequest Bad Request
   * @param paymentRequired Insufficient funds available
   * @param notFound Not Found
   * @param requestBody The Create Order payload
   */
  @Post()
  public async createOrder(
```

The documentation after adding descriptions and tags can be found online [34]. In the future, the rental company may chose to apply some more opinionated rules to their specification that may not be useful for other companies, but, if applied, may help consumer satisfaction. Some of our ideas include, but are not limited to:

[33] As the "orders" tag applies to all endpoints of this controller, it was moved to the class level, not the method level.

[34] `https://woh.github.io/redoc-rentals/redoc-rentals-full.html`

- Ensuring response parameter keys are at least 2 characters long and use camelCase:

```
response-property-names:
  severity: error
  message:
    "Invalid response property name: {{property}} (expected: camelCase, length
        >1)"
  recommended: true
  given: "$..components.schemas.*.properties[*]~"
  then:
    function: pattern
    functionOptions:
      match: "^[a-z][a-zA-Z]+$"
```

- Non 2xx codes respond with an *Error Message* schema.
- Every response with a non-empty parameter array has a 422 response for validation errors.
- All requests with security schemes have a 401 Unauthorized response.

Especially the last 2 items are results of the type checking limitations and throw/catch pattern we discussed in section 6.3.

7. Evaluation

In order to validate our approach, we conducted a coding exercise with TypeScript developers that were tasked to implement a service that produces an OpenAPI specification. We intended to compare 3 approaches (annotation based, reflection based, macro based) over 3 versions of a todo application. A ready-to-use scaffolded project, including db and data-access layer were provided. The participants were subsequently tasked to implement the Controller layer using express with no additional tooling but an annotation parser, the most popular TypeScript framework for NodeJS (nest), which provides support for reflection based OpenAPI documentation[1] and tsoa, which uses a build-time approach as outlined in chapter 6.

The instructions for each task/framework can be found in the Appendix A. Due to time constraints and feedback from the participants, Task 3 was removed from the evaluation, and, as both our dummy implementation (~1:20h) and the first candidate (over 2.5h) required approximately double the time for the pure JSDoc (oas) implementation and annotation, this approach was omitted aswell. As the participating candidates all were not very familiar (4-5) with the OpenAPI specification itself, it was deemed unlikely the evaluation would yield comparable results. Unfortunately, from the 4 planned evaluations, 1 participant was not able to complete the evaluation for personal reasons, another candidate did not submit in time.

7.1. Participants

At the beginning of the evaluation, every participant was asked questions about their background. This includes information about their role, years of programming experience and a self-rating of their familiarity with the frameworks and techniques used throughout the evaluation (1-5, 5 being best).

All times are based on the evaluation author's time needed to complete these tasks. The time needed to familiarize with the framework and the time spent reading respective documentation is not included.

[1] nest also added an AST parsing plugin, which aids in the annotation process. Therefore, participants were asked to not use the plugin to provide a better distinction between reflection and AST parsing. However, the plugin is currently only able to annotate classes, modeling using TypeScript types is currently not supported

Role	Programming	TypeScript	Nest	tsoa	OpenAPI
Author	9	4	4	5	4
Director Development Product Owner	30	2	1	2	3
Senior Software Engineer	8	3	1	3	1

Table 7.1.: Participant overview

7.2. Time

Task	nest	tsoa
1	1:40	1:25
2	0:15	0:15
1	1:20	1:30
2	0:10	0:10

Table 7.2.: Time

In order to provide at least some additional context, the times for an implementation by the evaluation author are provided below.

Style	nest	tsoa	oas
Without Annotations	0:33	0:35	0:35
With Annotations	0:53	0:39	1:21

Table 7.3.: Dummy times

7.3. Quality

In order to assess the quality of the produced API descriptions, a scoring system that grades each approach with up to 5 points for every operation was developed. Depending on the severity of accuracy between API description and implementation, either 0.5 points (property schema inaccuracies) or 1 point (missing response, missing parameter, type mismatches) were subtracted from the score. If the documentation for the Validation Error (400) response was missing, 2 points were subtracted, as it affects 4 endpoints. If the documentation for the Unauthorized (401) response was missing, 2 points were subtracted, as it affects 4 endpoints. If the documentation for the Not Found (404) response was missing, 1 point was subtracted, as it affects 2 endpoints.

If the API implementation did not match the requirements, but the implementation matched the API description, no points were subtracted from the score.

nest	tsoa
14	18.5
9	17.5

Table 7.4.: Correctness

In order to provide a lower bound for the expected quality of the approaches, three dummy implementations were conducted. Dummy nest represents a placeholder for a nest project that only relies on reflection capabilities, enabled through class-based modeling and decoration of the class properties, but does not include any additional knowledge augmentation. Dummy tsoa represents a placeholder for a tsoa implementation that only relies on the AST parsing capabilities of our approach, but does not include any additional augmentation. This dummy implementation uses type-checked alternative responses over throw/catch. Dummy OpenAPI similarly represents a placeholder to provide a baseline for an express implementation with TypeScript, without any annotations.

nest	tsoa	oas
8	14	-

Table 7.5.: Dummy Correctness

7.4. Discussion

Does the OpenAPI-aware approach to development decrease time spent on development (including documentation)?
As intuition suggests, the overall development time tends to decrease. As even the dummy implementation required significantly more time, the pure JSDoc (oas) approach will likely not be as fast, given an equal amount of familiarity.

Do developers prefer our approach?
Based on the feedback we received, it was very clear our candidates prefer TypeScript code and types over comments yaml or json description. This seems intuitive, as we chose developers to participate. Technical writers may have a very different perception, but one of the candidates in our evaluation explicitly mentioned they would not consider writing OpenAPI "by hand". While the UI was generally well perceived, the lack

of efficiency still drew criticism. While some developers preferred nest, others were of the opinion the additional documentation to understand what had to be annotated was too confusing. One candidate mentioned, that, while declaring parameters instead of accessing them via a request object required more thinking up front, the validation and documentation benefits were more important and therefore reasonable and, all things considered, worth it.

Does the quality of the OAS document improve?
While, as shown in the nest dummy candidate, some information regarding parameters can be correctly identified, the accuracy is very limited. Manual annotations are error prone, developers often missed 4xx response annotations. While tsoa tended to include more of these responses, the validation and authorization error responses required manual annotation and indeed were sometimes missed. As the evaluation did not explicitly require descriptions, tags or and summaries, only the "easily assessable" or less subjective attributes of high quality documentation were evaluated. Drawing conclusions from this data should therefore be avoided, but the improvements seem to be sufficiently promising to collect further data in the future.

8. Conclusion and Outlook

The goal of the work described in this thesis was to improve the quality of OpenAPI specification documents using existing knowledge already present in source code. By using TypeScript as a modeling language for API specifications, we reduced duplication and, as the evaluation shows, we may able to reduce time spent on API implementation + documentation, compared to other approaches. This enables an API first approach (in code) to API documentation that can be used to gather feedback early - which is usually only available to APIs developed using a separate, unverified API description - while being able to directly use the models in the implementation for type checking and request validation (see Fig. 8.1). However, there are still several limitations, some of which are the result of trade offs, others as a result of time limitations or restrictions of the TypeScript, or the OpenAPI modeling language based on JSON Schema Draft-00.

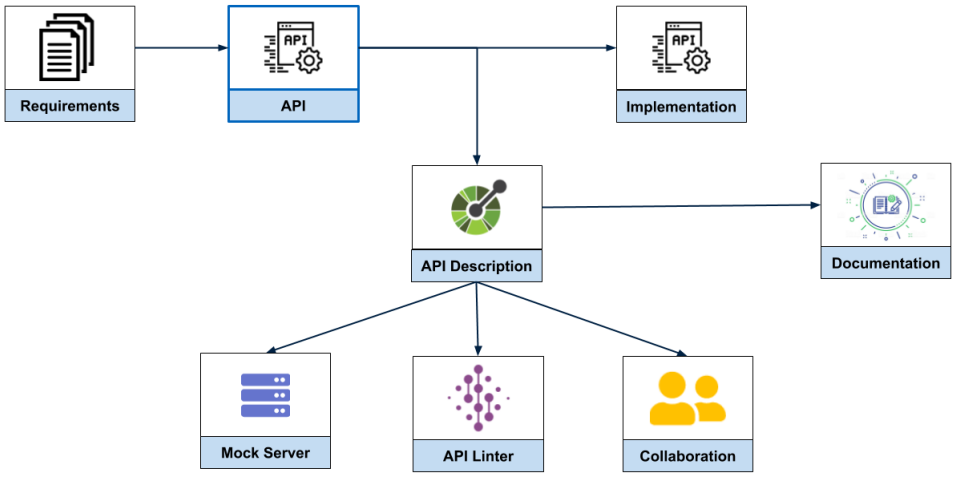


Figure 8.1.: API Model first approach

One of the major trade-offs is the focus on one language and one framework only. While necessary to improve correctness, this limits the applicability to other web frameworks and moreover seemingly hard to port the approach to other languages.

As TypeScript has no way of expressing certain limitations (data type integer, string patterns etc.), a small DSL using annotations was needed. This means TypeScript will not be able to check if statements assign an invalid value to a variable of this type, and only incoming requests will be checked. Extending TypeScript to improve this behavior may be possible, but was not explored yet due to time constraints. Also, our correctness guarantees are tied to the ability of TypeScript to check the API implementation code. As discussed, type casting and safety after throwing errors is limited in order to provide a better gradual adoption. Incorrectly modeling the shape of e.g. a database response may propagate throughout the application and impact the OpenAPI Specification document. Even though version 4.0 of TypeScript will allow catching errors as `unknown` instead of `any`, which forces the developer to manually narrow down the type, this opt-in feature - unless enforced by a TypeScript linter - may not be used and subsequently still lead to mismatches. A stricter language with similar JSON modeling capabilities like Hegel[1] would likely improve the correctness.

Literature also recommended an integration of links between Endpoint documentation. As the '@link' annotation was not properly supported by TypeScript, implementing this requirement was deemed too time consuming.

We conducted an evaluation, but due to the small sample size of participants and varying familiarity with the frameworks and approaches, the external validity is low. Applying this comparison with a broader set of endpoints and more participants that are already familiar with all 3 approaches would therefore be helpful to better understand the impact of each approach with regards to API description quality.

In the OpenAPI community, our approach was positively received. In a comparison between different approaches, the verdict of one industry expert was particulatly clear: "Other frameworks have first-party or third-party support for annotations, which are purely descriptive repetitions of the actual code they sit above at best. At worst they're just lies" [Stu20]. In contrast, the notion of using the code as a source for descriptions was deemed an improvement: "There is a new category of API description integration popping up in some web frameworks which is somewhat like Annotations or DSLs, but instead of being purely descriptive it's actually powering logic and reducing code, giving you one source of truth. [...] Instead of descriptive annotations or comments shoved in as an afterthought, the API framework has been designed around the use of annotations. [...] [T]his new approach for making annotations useful is very much closing the gap. If you're going to use a code-first approach, you should absolutely try and find a

[1] `https://hegel.js.org/`

framework like TSOA to power your API and reduce the chance of mismatches". [Stu20]

Although one possible integration strategy into a holistic API strategy was briefly described, more research should study needs to be done before before one could obtain better recommendations in this space.

As OpenAPI is in the process of integrating a newer, more expressive JSON Schema Draft for OpenAPI 3.1, modeling TypeScript using the 2019-07 draft requires significantly less workarounds. This alignment in the standards ecosystem allows future implementations of our general strategy to depend on tools from the JSON Schema ecosystem, as there will not be any more OpenAPI special cases with regards to the schema (instead, there is a well specified, optional OpenAPI vocabulary). This helps with combining efforts, possibly across languages in (typed) language to JSON Schema translation, but also in JSON Schema based validation for API Requests.

A. Appendix

A.1. Evaluation Instructions

Thanks for participating in our evaluation! Today, we want to test different approaches to generating OpenAPI specifications from Code. This means we would like you to implement and evolve the same application with 3 different controller layers. In order to get started quickly, we provide the application shell with a database, a data-access layer and required configuration in advance.

A.1.1. Initial survey

Before we get started, we'd like to know some general information about you:

- What's your current role?
- How many years of programming experience do you have?
- How familiar are you with TypeScript? (1-5, 5 is very familiar)
- How familiar are you with nest's swagger capabilities? (1-5)
- How familiar are you with tsoa? (1-5)
- How familiar are you with express? (1-5)
- How familiar are you with the OpenAPI Specification? (1-5)

Along with these instructions, you will be provided with an order to complete this tasks in. We would kindly ask you to respect that order.

A.1.2. Getting familiar with the approaches

In case you are not familiar with NestJS:

NestJS[1] uses Controllers[2] to handle requests, which will return Data Transfer Objects (DTOs), which are classes with (public) properties. In order to document these Classes,

[1] `https://github.com/nestjs/nest`
[2] `https://docs.nestjs.com/controllers`

NestJS uses property decorators [3]. In order to validate requests, usually validation decorators are required [4]. The Auto Validation pipe is already set up in the starter project.

In case you are not familiar with tsoa:

Tsoa[5] compiles your code into an OpenAPI specifications and a small runtime layer at build time. This means tsoa can make use of TS interfaces and type definitions to generate documentation and validation logic. Additionally, tsoa uses JSDoc annotations to enhance documentation and validation[6] [7] [8].The starter project already handles serialization of Validation and Authorization Errors.

In both cases, methods with decorators and decorated parameters are used to inject parts of the request at runtime[9] [10]. Similarly, `@Request()` will inject the entire request but not produce documentation for access.

In case you are not familiar with swagger-inline/OpenAPI:

swagger-inline[11] is a small utility that allows you to write the OpenAPI specification side-by-side with your express code. If you do not feel comfortable writing OpenAPI by hand, we'd suggest you use a web UI[12].

Here's the list of starter projects:

- Approach 1: tsoa
- Approach 2: nest
- Approach 3: express + swagger-inline (oas)

For all 3 projects, the basic structure is already in place. Additionally, authentication is already implemented and ready to be used. All projects expose a Swagger UI

[3] `https://docs.nestjs.com/recipes/swagger#decorators`
[4] `https://docs.nestjs.com/techniques/validation`
[5] `https://github.com/lukeautry/tsoa`
[6] `https://tsoa-community.github.io/docs/annotations.html`
[7] `https://tsoa-community.github.io/docs/descriptions.html#endpoint-descriptions`
[8] `https://tsoa-community.github.io/docs/examples.html`
[9] `https://docs.nestjs.com/controllers#request-object`
[10] `https://tsoa-community.github.io/docs/getting-started.html#defining-a-simple-controller`
[11] `https://github.com/readmeio/swagger-inline`
[12] `https://stoplight.io/p/studio`

on `http://localhost:3001/api`. As a point of reference, please take a look at the `UserController`.

The only requirement for running the projects is Docker (with docker-compose). To start, run `docker-compose up` inside the folder of each project.

A.1.3. Coding

In this step, we want to define a controller and data transfer objects for `Todos`. Each `Todo` belongs to a `User` who can create, retrieve, update, delete them.
Please note down the time needed to complete the objective for each task.

Task 1: Implementing

First, we need to define some data transfer objects. In tsoa, we can use classes, but usually (annotated) TypeScript interfaces/type aliases will be sufficient.

The shape of the DTOs is based on the `TodoEntity`. Additionally, the requirements are:

- The title must have a `minLength` of 3.
- The description is returned as `null` if it wasn't set.

Todo DTO
The title, description and progress properties of the `TodoEntity`. (user is optional)

CreateTodo DTO

- A title of type string with a `minLength` of 3.
- An optional description of type string.
- A progress of type `TodoProgress`

UpdateTodo DTO

- Optional title of type string with a `minLength` of 3
- An optional description of type `string`.
- An optional progress of type `TodoProgress`.

All of the Todo Endpoints require an authorized `User`. Otherwise, the request should respond with a Status of 401 and a JSON Object with a message of type string. The User is defined on the request object and is provided:

- In tsoa by declaring `@Request() @InjectUser() user: User` as a parameter for the request handler
- In nest by using `@InjectUser() user: User`
- In express/swagger-inline as `request.user`

The Update and Delete Endpoint will be called with the UUID as a Path parameter and should respond with a Status of 404 and a JSON body with a message property of type string if the Todo to update is not present (or does not belongs to the user). All endpoints should respond with a Http response with status 400 and a JSON body with at least a message property of type `string`) (or `string[]`) if validation fails.

While the GET /todos endpoint responds with an array of Todos, the Create, Update and Delete Endpoints should respond with a single Todo Entity. Please implement the controller/data transfer layer and document the API using OpenAPI.

Tips:
While there are a lot of similarities, all 3 approaches handle returning non-successful responses differently. While nest promotes throwing errors (with names based on the eventual status code) which require annotating the request handlers with `@ApiResponse` et al., which are caught and transformed to JSON responses, tsoa similarly uses `@Response<T>(status, description, example)` to document responses as a result of error handling (Validation and Authorization Errors in middleware), but promotes injecting responders (`@Res() errorResponse: TsoaResponse<Status, Res, Headers>`) which can be called in a checked way instead. This is very similar to calling `res.status(400).json({})` in express directly.

Task 2: Improving

In this part, we will change the implementation of the GET /todos endpoint. We will introduce 2 optional query parameters, *progress* and *search*. The *progress* query param is of type *TodoProgress[]* and will be used to filter *Todos* by progress. The *search* of type string with a `minLength` of 3 can be used to search for a text. Add these 2 parameters to the endpoint, merge them into a *GetTodosDto* and pass it to the *getAll()* method of the `TodoService`.

Task 3: Modeling

In this part of the evaluation, we would like to explore modeling with all of these frameworks. Therefore, we created 2 requirements for Endpoints. The Documentation can be found here.

We have also provided stubs here.

Your task will be to add missing models and controllers.

The endpoint we use may belong to a rental company for boats and ships. There are multiple ships that can be rented (by passing the ship's id), or a boat, and while there are several boats, they are of the same type, so providing an id is not neccessary. To enable integration with other marketplaces beyond their homepage, the decision was made to publish this endpoint via API. The endpoint can be used by POSTing a request to /orders. The body should contain a JSON object with information about the order:

- a configuration, either for a ship or for a boat
- start date/time of the rental
- end date/time of the rental
- in case of a boat, the configuration must be an object which may contain an amount of lifevests, up to 8, which is the capacity of a boat
- in case of a ship, the configuration must contain the shipId annd a captainId, a reference to a captain the renter chose to accompany the trip. In case the renter is allowed to navigate the ship (will not be verified by the API), the capitainId should be explicitly set to null.
- a chargeAccountId, which references the account to charge for that purchase

Additionally, in order authorize the request, an Authorization header with a JWT must be provided.

There are several ways, the API will respond to these requests:

- 200: Ok: The rental was successful
- 400: The request failed because the rental could not be made (i.e. because a ship is not available that day). The response body will contain an object with a message explaining why the rental could not be processed.
- 401: Unauthorized: The Authorization header was not provided or incorrect
- 404: Not Found: One of the provided id's was not found. A message with details will be provided in the body

- 422: Unprocessable Entity: The request did not match the specification

A.1.4. Final survey

Thanks and congrats on completing these tasks. We would like you to answer a few questions about the approaches you got to know today.

- Would you prefer explicitly listing Parameters in Controllers (instead of grabbing them off the request object directly) in exchange for documentation and validation?
- Did you prefer to write descriptions/metadata in Decorator arguments or JSDoc?
- Did you enjoy using JSDoc as the source for descriptions?
- Did you enjoy using JSDoc for OpenAPI/JSON Schema modeling? Please provide reasons if you want to share any.
- Did you prefer classes over interfaces and type aliases to define DTOs?
- If your TS types were validated at runtime, would you still use class based DTOs? If the answer is yes, we would like to hear why.
- Which approach to writing the controller layer did you enjoy best overall?
- If requirements changed, which approach would be the least error-prone?
- Please share the time you needed for each task and framework.

A.2. TypeScript TypeChecker Type Flags

Listing A.1: TypeScript Type Checker Flags

```
export enum TypeFlags {
    Any = 1,
    Unknown = 2,
    String = 4,
    Number = 8,
    Boolean = 16,
    Enum = 32,
    BigInt = 64,
    StringLiteral = 128,
    NumberLiteral = 256,
    BooleanLiteral = 512,
    EnumLiteral = 1024,
    BigIntLiteral = 2048,
    ESSymbol = 4096,
    UniqueESSymbol = 8192,
    Void = 16384,
    Undefined = 32768,
    Null = 65536,
    Never = 131072,
    TypeParameter = 262144,
    Object = 524288,
    Union = 1048576,
    Intersection = 2097152,
    Index = 4194304,
    IndexedAccess = 8388608,
    Conditional = 16777216,
    Substitution = 33554432,
    NonPrimitive = 67108864,
    Literal = 2944,
    Unit = 109440,
    StringOrNumberLiteral = 384,
    PossiblyFalsy = 117724,
    StringLike = 132,
    NumberLike = 296,
    BigIntLike = 2112,
    BooleanLike = 528,
    EnumLike = 1056,
    ESSymbolLike = 12288,
    VoidLike = 49152,
    UnionOrIntersection = 3145728,
    StructuredType = 3670016,
    TypeVariable = 8650752,
    InstantiableNonPrimitive = 58982400,
    InstantiablePrimitive = 4194304,
    Instantiable = 63176704,
    StructuredOrInstantiable = 66846720,
    Narrowable = 133970943,
    NotUnionOrUnit = 67637251,
}
```

List of Figures

List of Tables

Listings

Bibliography

[Adz11] G. Adzic. *Specification by Example: How Successful Teams Deliver the Right Software*. 1st. USA: Manning Publications Co., 2011. ISBN: 1617290084.

[Agh+19] E. Aghajani, C. Nagy, O. L. Vega-Márquez, M. Linares-Vásquez, L. Moreno, G. Bavota, and M. Lanza. "Software Documentation Issues Unveiled." In: *Proceedings of the 41st International Conference on Software Engineering*. ICSE '19. Montreal, Quebec, Canada: IEEE Press, 2019, pp. 1199–1210. DOI: 10.1109/ICSE.2019.00122.

[Alg10] J. Algermissen. *Classification of HTTP-based APIs*. 2010. URL: http://algermissen.io/classification_of_http_apis.html (visited on 05/03/2020).

[BAT14] G. Bierman, M. Abadi, and M. Torgersen. "Understanding typescript." In: *European Conference on Object-Oriented Programming*. Springer. 2014, pp. 257–281.

[Ben18] E. Bendersky. *Type erasure and reification*. 2018. URL: https://eli.thegreenplace.net/2018/type-erasure-and-reification/ (visited on 06/28/2020).

[Bon+19] G. Bondel, D. H. Bui, A. Faber, D. Seidel, and M. Hauder. "Towards a Process and Tool Support for Collaborative API Proposal Management." In: *The 25th Americas Conference on Information Systems (AMCIS), Cancun, Mexiko* (2019).

[Bra+14] T. Bray et al. "The javascript object notation (json) data interchange format." In: *URL https://www. rfc-editor. org/rfc/rfc7159. txt* (2014).

[Bui18] D. H. Bui. "Design and Evaluation of a Collaborative Approach for API Lifecycle Management." In: (2018).

[BZN13] P. Bryan, K. Zyp, and M. Nottingham. "JavaScript object notation (JSON) pointer." In: *RFC 6901 (Proposed Standard)* (2013).

[Chi14] T.-h. Chien. *Soundness and Completeness of the Type System*. 2014. URL: https://logan.tw/posts/2014/11/12/soundness-and-completeness-of-the-type-system (visited on 06/26/2020).

[Clo20] Cloud Elements. *The State of API Integration 2020*. 2020. URL: https://offers.cloud-elements.com/2020-state-of-api-integration-report (visited on 04/01/2020).

[Cro08] D. Crockford. *JavaScript: The Good Parts: The Good Parts*. " O'Reilly Media, Inc.", 2008.

[CVG19] A. Cummaudo, R. Vasa, and J. Grundy. *What should I document? A preliminary systematic mapping study into API documentation knowledge*. 2019. arXiv: 1907.13260 [cs.SE].

[CZ14] C. Chen and K. Zhang. "Who Asked What: Integrating Crowdsourced FAQs into API Documentation." In: *Companion Proceedings of the 36th International Conference on Software Engineering*. ICSE Companion 2014. Hyderabad, India: Association for Computing Machinery, 2014, pp. 456–459. ISBN: 9781450327688. DOI: 10.1145/2591062.2591128.

[Doc19] O. J. Documentation. *Type Erasure*. 2019. URL: https://docs.oracle.com/javase/tutorial/java/generics/erasure.html (visited on 06/28/2020).

[ECM16] J. ECMA. "404 the json data interchange standard." In: *ECMA International* (2016).

[Ed-+19] H. Ed-douibi, J. L. Cánovas Izquierdo, F. Bordeleau, and J. Cabot. "WAPIml: Towards a Modeling Infrastructure for Web APIs." In: *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*. Sept. 2019, pp. 748–752. DOI: 10.1109/MODELS-C.2019.00116.

[EIC18] H. Ed-Douibi, J. L. C. Izquierdo, and J. Cabot. "OpenAPItoUML: a tool to generate UML models from OpenAPI definitions." In: *International Conference on Web Engineering*. Springer. 2018, pp. 487–491.

[End+14] S. Endrikat, S. Hanenberg, R. Robbes, and A. Stefik. "How Do API Documentation and Static Typing Affect API Usability?" In: *Proceedings of the 36th International Conference on Software Engineering*. ICSE 2014. Hyderabad, India: Association for Computing Machinery, 2014, pp. 632–642. ISBN: 9781450327565. DOI: 10.1145/2568225.2568299.

[Fat19] F. Fatemi. "3 Keys To A Successful API Strategy." In: *Forbes* (2019).

[Fie00] R. T. Fielding. "REST: architectural styles and the design of network-based software architectures." In: *Doctoral dissertation, University of California* (2000).

[Fie08] R. T. Fielding. *REST APIs must be hypertext-driven*. 2008. URL: https://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven (visited on 06/28/2020).

[Fow10] M. Fowler. *Richardson maturity model*. 2010. URL: http://martinfowler.com/articles/richardsonMaturityModel.html (visited on 06/21/2020).

[FR14a] R. Fielding and J. Reschke. *Hypertext transfer protocol (HTTP/1.1): Message syntax and routing*. Tech. rep. RFC 7230, June 2014, 2014.

[FR14b] R. Fielding and J. Reschke. *Hypertext transfer protocol (HTTP/1.1): Semantics and content*. Tech. rep. RFC 7231, June 2014, 2014.

[GH13] S. Gregor and A. R. Hevner. "Positioning and presenting design science research for maximum impact." In: *MIS quarterly* (2013), pp. 337–355.

[Gou+02] D. Gourley, B. Totty, M. Sayer, A. Aggarwal, and S. Reddy. *HTTP: the definitive guide*. " O'Reilly Media, Inc.", 2002, p. 33.

[Hev+04] A. R. Hevner, S. T. March, J. Park, and S. Ram. "Design science in information systems research." In: *MIS quarterly* (2004), pp. 75–105.

[Hos+18] M. Hosono, H. Washizaki, Y. Fukazawa, and K. Honda. "An Empirical Study on the Reliability of the Web API Document." In: *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE. 2018, pp. 715–716.

[HS03] D. Hoffman and P. Strooper. "API documentation with executable examples." In: *Journal of Systems and Software* 66.2 (2003), pp. 143–156.

[HSM18] A. Hernandez-Mendez, N. Scholz, and F. Matthes. "A Model-driven Approach for Generating RESTful Web Services in Single-Page Applications." In: *MODELSWARD*. 2018, pp. 480–487.

[Ini+20] O. Initiative et al. "The OpenAPI Specification." In: *URL: https://github.com/OAI/OpenAPI-Specification/blob/master/versions/3.0.3, accessed: 14.07.2020* (2020).

[Iye+17] K. Iyengar, S. Khanna, S. Ramadath, and D. Stephens. "What it really takes to capture the value of APIs." In: *McKinsey & Company* (2017).

[Lan04] K. Lane. *What Is An API First Strategy? Adding Some Dimensions To This New Question*. 2004. URL: https://apievangelist.com/2014/08/11/what-is-an-api-first-strategy-adding-some-dimensions-to-this-new-question (visited on 06/03/2020).

[Lau19] A. Lauret. *The Design of Web APIs*. 1st. USA: Manning Publications Co., 2019. ISBN: 9781617295102.

[Mar19] C. Martraire. *Living Documentation: Continuous Knowledge Sharing by Design*. 1st. USA: Addison-Wesley Professional, 2019. ISBN: 0134689321.

[Mic16] Microsoft. *TypeScript Language Specification*. 2016. URL: https://github.com/Microsoft/TypeScript/blob/master/doc/spec.md (visited on 06/21/2020).

[MM14] P. W. McBurney and C. McMillan. "Automatic Documentation Generation via Source Code Summarization of Method Context." In: *Proceedings of the 22nd International Conference on Program Comprehension*. ICPC 2014. Hyderabad, India: Association for Computing Machinery, 2014, pp. 279–290. ISBN: 9781450328791. DOI: 10.1145/2597008.2597149.

[MM16] P. W. Mcburney and C. Mcmillan. "An Empirical Study of the Textual Similarity between Source Code and Source Code Summaries." In: *Empirical Softw. Engg.* 21.1 (Feb. 2016), pp. 17–42. ISSN: 1382-3256. DOI: 10.1007/s10664-014-9344-6.

[MPD10] M. Maleshkova, C. Pedrinaci, and J. Domingue. "Investigating web apis on the world wide web." In: *2010 eighth ieee european conference on web services*. IEEE. 2010, pp. 107–114.

[MS16] B. A. Myers and J. Stylos. "Improving API usability." In: *Communications of the ACM* 59.6 (2016), pp. 62–69.

[NAP18] K. Nybom, A. Ashraf, and I. Porres. "A systematic mapping study on API documentation generation approaches." In: *2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE. 2018, pp. 462–469.

[Net11] M. D. Network. *HTTP Messages*. 2011. URL: https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview#HTTP_Messages (visited on 04/03/2020).

[NM10] S. M. Nasehi and F. Maurer. "Unit tests as API usage examples." In: *2010 IEEE International Conference on Software Maintenance*. IEEE. 2010, pp. 1–10.

[Org19] J. S. Organisation. *The home of JSON Schema*. 2019. URL: https://json-schema.org/ (visited on 05/13/2020).

[Ove20] S. Overflow. *2002 Developer Survey*. 2020. URL: https://insights.stackoverflow.com/survey/2020 (visited on 06/21/2020).

[Pez+16] F. Pezoa, J. L. Reutter, F. Suarez, M. Ugarte, and D. Vrgoč. "Foundations of JSON Schema." In: *Proceedings of the 25th International Conference on World Wide Web*. WWW '16. Montréal, Québec, Canada: International World Wide Web Conferences Steering Committee, 2016, pp. 263–273. ISBN: 9781450341431. DOI: 10.1145/2872427.2883029.

[Rém15] D. Rémy. "Type Systems for Programming Languages." Course notes, available electronically. 2015.

[Ric+13] L. Richardson, M. Amundsen, M. Amundsen, and S. Ruby. *RESTful Web APIs: Services for a Changing World*. " O'Reilly Media, Inc.", 2013.

[SAM15a] S. M. Sohan, C. Anslow, and F. Maurer. "Spyrest: Automated restful API documentation using an HTTP proxy server (N)." In: *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2015, pp. 271–276.

[SAM15b] S. Sohan, C. Anslow, and F. Maurer. "Spyrest in action: An automated RESTful API documentation tool." In: *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2015, pp. 813–818.

[SAM17] S. Sohan, C. Anslow, and F. Maurer. "Automated example oriented REST API documentation at Cisco." In: *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*. IEEE. 2017, pp. 213–222.

[Sma19] SmartBear Software. *State of API 2019*. 2019. URL: https://static1.smartbear.co/smartbearbrand/media/pdf/smartbear_state_of_api_2019.pdf (visited on 04/01/2020).

[SS15] I. Salvadori and F. Siqueira. "A maturity model for semantic restful web apis." In: *2015 IEEE International Conference on Web Services*. IEEE. 2015, pp. 703–710.

[Stu20] P. Sturgeon. *There's No Reason to Write OpenAPI By Hand*. 2020. URL: https://apisyouwonthate.com/blog/theres-no-reason-to-write-openapi-by-hand (visited on 04/13/2020).

[SW15] P. Suter and E. Wittern. "Inferring web API descriptions from usage data." In: *2015 Third IEEE Workshop on Hot Topics in Web Systems and Technologies (HotWeb)*. IEEE. 2015, pp. 7–12.

[Tec18] Techopedia. *Abstract Syntax Tree (AST)*. 2018. URL: https://www.techopedia.com/definition/22431/abstract-syntax-tree-ast (visited on 06/20/2020).

[Tho09] K. Thomas. *API-First Design*. 2009. URL: http://asserttrue.blogspot.com/2009/04/api-first-design.html (visited on 06/21/2020).

[TR16] C. Treude and M. P. Robillard. "Augmenting API Documentation with Insights from Stack Overflow." In: *Proceedings of the 38th International Conference on Software Engineering*. ICSE '16. Austin, Texas: Association for Computing Machinery, 2016, pp. 392–403. ISBN: 9781450339001. DOI: 10.1145/2884781.2884800.

[Wit+17] E. Wittern, A. T. Ying, Y. Zheng, J. Dolby, and J. A. Laredo. "Statically checking web API requests in JavaScript." In: *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE. 2017, pp. 244–254.

[Zhi+15] J. Zhi, V. Garousi-Yusifoğlu, B. Sun, G. Garousi, S. Shahnewaz, and G. Ruhe. "Cost, benefits and quality of software development documentation: A systematic mapping." In: *Journal of Systems and Software* 99 (2015), pp. 175–198. ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2014.09.042.