FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of master's thesis

Title:	Interactive web documentation for Protocol Buffers
Student:	Bc. Jakub Dobrý
Supervisor:	Ing. Jiří Šmolík
Study program:	Informatics
Branch / specialization:	Web Engineering
Department:	Department of Software Engineering
Validity:	until the end of summer semester 2023/2024

Instructions

The aim of the work is to design and implement a static web presentation generator for gRPC API documentation. The input files are proto files that carry a description of services, calls and types. The output is an HTML page with documentation and the ability to call the API (provided that the API supports gRPC-web).

1. Research existing solutions for Protocol Buffers and compare features with similar tools for GraphQL or RESTful APIs.
2. Design and implement a documentation generator with the ability to call an existing gRPC-web API.
3. Discuss and possibly implement an alternative data source using gRPC reflection.
4. Test the application with automated tests and perform user testing.

Electronically approved by Ing. Jaroslav Kuchař, Ph.D. on 24 November 2022 in Prague.

Master's thesis

INTERACTIVE WEB DOCUMENTATION FOR PROTOCOL BUFFERS

Bc. Jakub Dobrý

Faculty of Information Technology
Department of Software Engineering
Supervisor: Ing. Jiří Šmolík
May 8, 2024

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis: Dobrý Jakub. *Interactive web documentation for Protocol Buffers.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2024.

Contents

List of Figures

List of Tables

List of code listings

I would especially like to thank my supervisor, Ing. Jiří Šmolík, for his help, time, and patience in answering my questions. I would like to thank all testers for taking the time to do the user testing. I would also like to thank my friends, who have been very supportive throughout. I am grateful to my girlfriend for her encouragement and support. Last but not least, I would like to thank my family for their support and patience while writing this thesis.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Article 46(6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the "Work"), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work in any way (including for-profit purposes) that does not detract from its value. This authorization is not limited in terms of time, location and quantity.

In Prague on May 8, 2024

Abstract

This master thesis presents the analysis of existing documentation tools, design, implementation, and testing of a static website generator tailored for gRPC APIs, enabling interactive API calls via gRPC-Web. The solution leverages a common JSON format for definitions and incorporates two generators: one for proto files and another for gRPC reflection. This system allows users to explore gRPC services, message types, and enums, complete with their documentation comments. It also facilitates live interaction with the gRPC API, allowing users to execute calls and view real-time results. Aiming to enhance the developer experience, this work will be publicly available to the gRPC community, providing a valuable tool for developers working with gRPC APIs.

Keywords static web page, gRPC, gRPC-Web, interactive calls, documentation generator, Protocol Buffers

Abstrakt

Tato diplomová práce představuje analýzu existujících nástrojů pro dokumentaci, návrh, implementaci a testování statického generátoru webových stránek určeného pro gRPC API, který umožňuje interaktivní volání API prostřednictvím gRPC-Web. Řešení využívá společný formát JSON pro definice a zahrnuje dva generátory: jeden pro proto soubory a druhý pro reflexi gRPC. Tento systém umožňuje uživatelům prozkoumávat gRPC service, message typy a enum, včetně jejich dokumentačních komentářů. Dále umožňuje interakci s gRPC API, což uživatelům umožňuje provádět volání a zobrazovat výsledky v reálném čase. S cílem zlepšit vývoj bude tato práce zpřístupněna veřejně pro gRPC komunitu, čímž poskytne cenný nástroj pro vývojáře pracující s gRPC API.

Klíčová slova statická webová stránka, gRPC, gRPC-Web, interaktivní volání, generátor dokumentace, Protocol Buffers

List of abbreviations

API	Application Programming Interface
CSS	Cascading Style Sheets
DOM	Document Object Model
EOF	End-of-file
GraphQL	Query Language for APIs
gRPC	Google Remote Procedure Call
HTML	HyperText Markup Language
HTTP	Hypertext Transfer Protocol
IDE	Integrated Development Environment
JSON	JavaScript Object Notation
NPM	Node Package Manager
REST	Representational State Transfer
RPC	Remote Procedure Call
TLS	Transport Layer Security
UI	User Interface
URL	Uniform Resource Locator
YAML	Yet Another Markup Language

Chapter 1
Introduction

Having a readily available client implementation and documentation for a server API significantly aids in testing and development. However, implementing clients can be a time-consuming effort. To address this, generic clients are often employed. Swagger[1] is a popular choice for RESTful APIs, while GraphiQL[2] often serves this purpose in the GraphQL world. There is an evident need for a similarly streamlined tool within the gRPC ecosystem.

While gRPC documentation tools exist, they often lack the user-friendliness found in their REST or GraphQL counterparts. These solutions may require separate servers and sometimes have limited support for comprehensively documenting services, calls, and types.

The central aim of this work is to design and implement a static web presentation generator focused on user-friendly gRPC API documentation. Input will consist of Protobuf Buffer files that describe the services, calls, and types. The desired output is a static HTML page that provides clear documentation and the ability to directly execute gRPC-web API calls (assuming the API supports gRPC-web).

I have chosen this topic because I believe I can offer a solution that will alleviate the challenges associated with documenting gRPC services. This is particularly relevant for developers who primarily work with RESTful APIs.

The thesis is divided into four main parts. The first part focuses on analyzing existing solutions. Subsequent parts cover design, implementation, and testing.

In the analysis phase, I will examine the features present in existing gRPC documentation solutions and compare them to similar tools designed for RESTful and GraphQL APIs. Based on the insights gained, I will define the specific requirements and use cases for the static web presentation generator.

In the design chapter, I will present a solution that leverages a static web page and tools for converting gRPC API definitions into a format readily understood by the web page. I will also delve into the options and potential limitations of using gRPC vs gRPC-web. Additionally, I will explore and design a mechanism to obtain a gRPC API definition using gRPC reflection.

The implementation chapter will address the selection of an appropriate technology stack, the implementation of the designed solution, and the overall architecture of the application. I will discuss any challenges encountered during implementation and the solutions devised. This chapter will conclude with a review of the licenses associated with any external libraries used.

Finally, the testing chapter will cover both manual and automated testing strategies. Automated testing will be conducted utilizing unit tests. Manual testing will be performed through user testing, with a strong emphasis on evaluating the user-friendliness of the application.

[1] `https://swagger.io/`
[2] `https://github.com/graphql/graphiql`

Chapter 2
Goals

The primary goal of this thesis is to develop a static HTML page that provides both developer-familiar documentation design and gRPC API interaction capabilities (assuming the API supports gRPC-web). This project will leverage Protocol Buffer files, which define services, calls, and data types as input. A key focus is to address the challenges of creating interactive gRPC API documentation by utilizing a static web page and tools facilitating conversion between the gRPC API definition and a format readily understood by the web page.

A comprehensive analysis of existing gRPC solutions, alongside comparable tools for RESTful and GraphQL APIs, will be conducted. This analysis will examine features, strengths, and weaknesses, informing the specific requirements and use cases for the proposed web-based solution.

The implementation chapter centers on the creation of the static HTML page, enabling direct gRPC API calls. Additionally, this page will incorporate the ability to automatically generate documentation leveraging gRPC reflection.

A testing phase will include both automated tests to verify functionality and user testing to evaluate the interface's overall usability and developer-familiar design.

The ultimate outcome of this work is to provide a valuable resource for developers building or using gRPC APIs. This solution will streamline the process of creating interactive documentation, enhancing understanding and efficient use of gRPC services, and hosting the documentation website.

Chapter 3
Analysis

In this chapter, I am going to introduce Protocol Buffers. Then, I will analyze existing solutions for website documentation with the ability to call APIs for Protocol Buffers, GraphQL, and RESTful APIs. I will start with Protocol Buffers, then move and explore the solutions for GraphQL, and finally analyze RESTful API.

3.1 Protocol Buffers

Protocol Buffers[1] are a mechanism for serializing structured data. They are language-neutral and platform-neutral. They encompass:

- definition language,
- compiler-generated code,
- language-specific runtime libraries,
- serialization format.

The definition language defines data structures expressed in .proto files. The compiler-generated code enables interaction with the defined data structures in a specific programming language. The language-specific runtime libraries facilitate the serialization and deserialization of data according to the Protocol Buffer format. The serialization format is a compact binary format for storing Protocol Buffer data in files or transmitting it across network connections. [1]

The mechanism of typical workflow is described in figure 3.1. For this work, my main focus will be generating the code (and website documentation) from the .proto files and using the generated code to interact with the final APIs.

There are two language versions of Protocol Buffers, version 2 and version 3. The versions share the same basic concepts using the same syntax, but version 3 improves version 2 in several ways [2]. As this work focuses on the latest technologies, I will focus on version 3 of Protocol Buffers. If no specific version is mentioned, it is assumed that version 3 is used.

3.1.1 Structures

The primary keywords in Protocol Buffers are *message*, *enum*, *service*, *method*, and *package*. The *message* is used to define a data structure. The *enum* defines a set of named constants. The *service* is a set of *methods* that can be called remotely. And, the *package* type is used to define a namespace for the defined *messages*, *enums*, and *services*. [2]

[1] `https://protobuf.dev/`

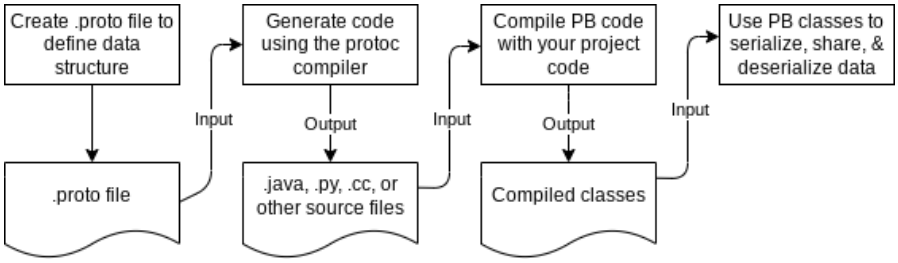


Figure 3.1 Protocol Buffers workflow [1]

3.1.1.1 Message

Messages are used to define data structures. They are defined using the *message* keyword followed by the name of the message and a block of fields. Each field has a name, a type, and a unique number across all fields in the *message*. The type of a field can be a scalar type or another *message* type. The possible scalar types are described in table 3.1 with their C++ programming language counterparts. The unique number is used to identify the field in the binary encoding. Reusing the same number for different fields is therefore highly discouraged. To avoid this, there is a *reserved* keyword, which I will describe later. [2]

.proto	C++
double	double
float	float
int32	int32
int64	int64
uint32	uint32
uint64	uint64
sint32	int32
sint64	int64
fixed32	uint32
fixed64	uint64
sfixed32	int32
sfixed64	int64
bool	bool
string	string
bytes	string

Table 3.1 Scalar types in Protocol Buffers [2]

Additional properties can be added to fields or types to alter their behavior. The possibilities are *reserved*, *optional*, *repeated*, *map*, and *oneof*. [2]

The *reserved* keyword is used to reserve a field number, preventing it from being used in the future. This is useful when a field is removed from a message, and the field number should not be reused. You can specify a single field number, a range of field numbers, or a list of field numbers and ranges. [2]

In Protocol Buffers version 3, fields are inherently optional, with omitted fields assuming their default values. This can create ambiguity when differentiating between a missing field and one explicitly set to its default. The *optional* keyword resolves this by providing a mechanism to explicitly mark fields as optional and track whether they have been set, even if the value is the default. [2]

The *repeated* keyword is used to define fields that can hold multiple values of the same type.

This is analogous to arrays or lists in common programming languages. A repeated field allows the representation of collections of data within the *message* structure. For example, a *message* representing an order might have a *repeated* field for line items, allowing multiple products within a single order. Protocol Buffers offer efficient encoding mechanisms for repeated fields, making them suitable for representing ordered data lists. [2]

The *map* keyword is employed to define fields encompassing key-value pairs, akin to dictionaries or hashmaps in programming contexts. A *map* field allows the flexible association of related data without the constraints of a rigid structure. For instance, a product attribute message could leverage a *map* field where keys denote attribute names ("color", "size") and their corresponding values provide the descriptions ("red", "large"). [2]

Finally, the *oneof* keyword provides a mechanism to define a *message* field where only one of several sub-fields can be set at a time. This is valuable when the *message* needs to represent mutually exclusive data variations. For example, a `payment_method` field within a *message* could use a *oneof* to support different payment types like `credit_card`, `debit_card`, or `paypal`. Setting one of these sub-fields automatically clears any previously set values within the *oneof*. This helps conserve memory and enforces a clear structure for alternative data representations. [2]

3.1.1.2 Enum

Another type of structure is *enum*. It is used to define a set of named constants. The constants are defined using the *enum* keyword followed by the name of the *enum* and a block of constants with their numeric values. The special numeric value 0 is used as the default value. Therefore, the first constant must have the value 0. [2]

The *enum* has a special option called *allow_alias*, which allows having the same numeric values for multiple names. This is useful when the same value is used in different contexts. [2]

3.1.1.3 Service and Method

The *service* defines a set of *methods* that can be called remotely. It is defined using the *service* keyword followed by the name of the *service* and a block of *methods*. Each *method* has a name, request, and response *message* type. The *method* can also have a *stream* keyword to define streaming of request, streaming of response, or both. [2]

Streaming is a feature that allows the client and server to send a sequence of *messages* back and forth until the stream is closed. This is useful when the client or server needs to send a large number of *messages* or just does not know the exact number of them in advance. [2]

There are four types of gRPC calls. When no streaming is involved, it is called *unary* call. When the request is streamed, it is called *client-streaming* call. When the response is streamed, it is called *server-streaming* call. And when both request and response are streamed, it is called *bidirectional-streaming* call. [3]

3.1.1.4 Packages and Options

The *package* is used to define a namespace for the defined *messages*, *enums*, and *services* in the .proto file. It is present at the beginning of the file using the *package* keyword followed by the name of the *package*. [2]

The other option to differentiate *messages* names is using nested types. The *message* can be defined inside another *message*. It is useful when the *message* is used only in the context of the parent *message* or is meaningful only in the parent *message* context. [2]

Both *package* and nested types are used to avoid name conflicts and can be used using the dot notation between names.

The *options* are used to alter the behavior of the .proto file, *message*, *enum*, *service*, *method*, or fields. They are defined using the *option* keyword followed by the name of the *option* and its

value. One of the most commonly used *option* is a *java_package*. The *java_package* is used to define the package for the Java programming language code generation. It is defined at the file level. Other *option* could be *java_multiple_files* for altering Java code generation to generate multiple files, or *optimize_for* for selection of C++ and Java generated client size, and more. [2]

3.1.2 Comments

The .proto files can contain comments. The comments can be single-line or multi-line. The single-line comments are started with the // characters. The multi-line comments start with the /* characters and end with the */ characters. The comments can be used to describe the purpose of the *message*, *enum*, *service*, *method*, or field. The comments can also describe the purpose of the *package* or the whole file. [2]

3.1.3 Code Generation

The .proto files are used as a source for generating a specific programming language code. Code generation can be done using various tools, the recommended one being the Protocol Compiler (protoc). It is used to generate the C++, C#, Dart, Go, Java, Python, Ruby, and JavaScript code. An example usage is described in the code snippet 3.1. Required classes and types are then generated, and the gRPC APIs can be called without extra coding work. [2]

Code listing 3.1 Protocol Buffers code generation [2]

```
protoc --proto_path=IMPORT_PATH --java_out=DST_DIR path/to/file.proto
```

3.1.4 Metadata

Metadata are key-value pairs sent with the initial or final request or response. They are used to provide additional information, such as authentication or tracing information. Two types of metadata are used: headers and trailers.

Headers are sent before the initial client request and before the initial response from the server. This applies only to the first message of the client and server. The figure 3.2 shows the gRPC headers in the request lifecycle.

Trailers are sent after the server gives the final response. They provide additional information about the response, such as the utilization or query cost. The figure 3.2 shows the gRPC trailers in the response lifecycle. [4]

3.1.5 gRPC-web

The gRPC is built on HTTP/2, using features like HTTP/2 framing [5]. As the HTTP/2 framing is not, and probably never be, directly exposed by any browser, the gRPC-Web protocol exists [6].

The design goals of the gRPC-Web are:

- to adopt the same framing as gRPC whenever possible,
- decouple from HTTP/2 framing,
- support text stream for cross-browser support [6].

The gRPC-Web clients require a proxy, which is added between the client and the server. The communication works as follows. The browser sends a request to the proxy using the gRPC-Web protocol. The proxy translates the gRPC-Web protocol to the gRPC protocol and sends the request to the server. The server processes the request and sends the response back to the proxy.

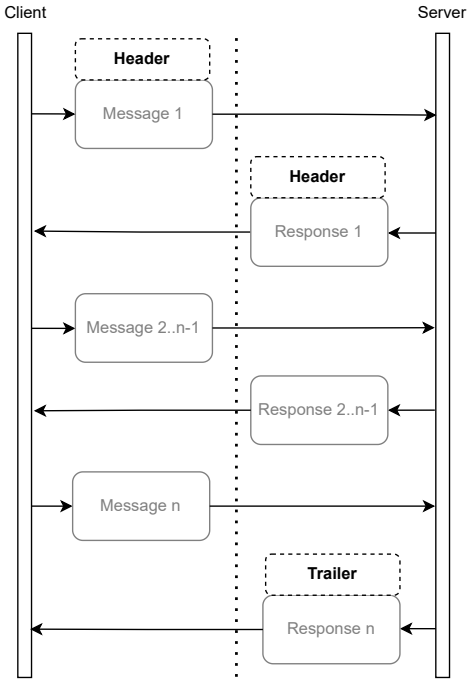


Figure 3.2 gRPC metadata

The proxy translates the gRPC protocol to the gRPC-Web protocol and sends the response back to the browser. The browser processes the response, and the communication is complete. [6]

The default proxy implementation is the Envoy[2] proxy. It supports the gRPC-Web protocol out of the box. Other options are, but not only, gRPC-Web Go proxy[3], APISIX[4], and Nginx[5].

Because of the proxy and browser implementation, there are a few differences and current limitations of the gRPC-Web [7]. The most important one is streaming support. Currently, the gRPC-Web does not support client-side streaming (effectively bi-directional streaming). It supports only unary calls and server-side streaming. Based on the streaming roadmap, the client-side streaming is planned for the future. It is planned for 2023+, but it has not been implemented yet [8].

3.1.6 gRPC Reflection

The gRPC protocol uses binary encoding. Therefore, it is impossible to query the server without knowing the protobuf definition of the service and both request and response messages in advance of the request. The gRPC reflection is a way to get this information using a standardized gRPC service that allows other clients to query the server for the protobuf-defined APIs. It includes all necessary information about the services, methods, enums, and messages. This information can encode requests, query the server, and decode responses. It is used by debugging tools such as grpcurl[6]. The gRPC reflection service is not exposed by default, so it must be explicitly enabled in the server configuration. The support for it varies across different gRPC implementations in different programming languages. [9]

[2] https://www.envoyproxy.io/
[3] https://github.com/improbable-eng/grpc-web/tree/master/go/grpcwebproxy
[4] https://apisix.apache.org/blog/2022/01/25/apisix-grpc-web-integration/
[5] https://www.nginx.com/
[6] https://github.com/fullstorydev/grpcurl

3.2 Existing Documentation Tools

This section examines popular tools for documenting gRPC, GraphQL, and RESTful APIs. I will assess the capabilities and shortcomings of each solution, addressing gRPC first, then GraphQL, and lastly, RESTful APIs. The section will culminate in a summary of findings, a discussion of issues, and a proposed solution for the static website generator.

3.2.1 Protocol Buffers

In the Protocol Buffers world, there are several tools for generating documentation or interactive calls from the .proto files. The most popular ones I have found are described in the following subsections.

3.2.1.1 Wombat

Wombat is a cross-platform gRPC desktop app client. The UI is in the figure 3.3. It is used to call gRPC services and inspect the responses. [10]

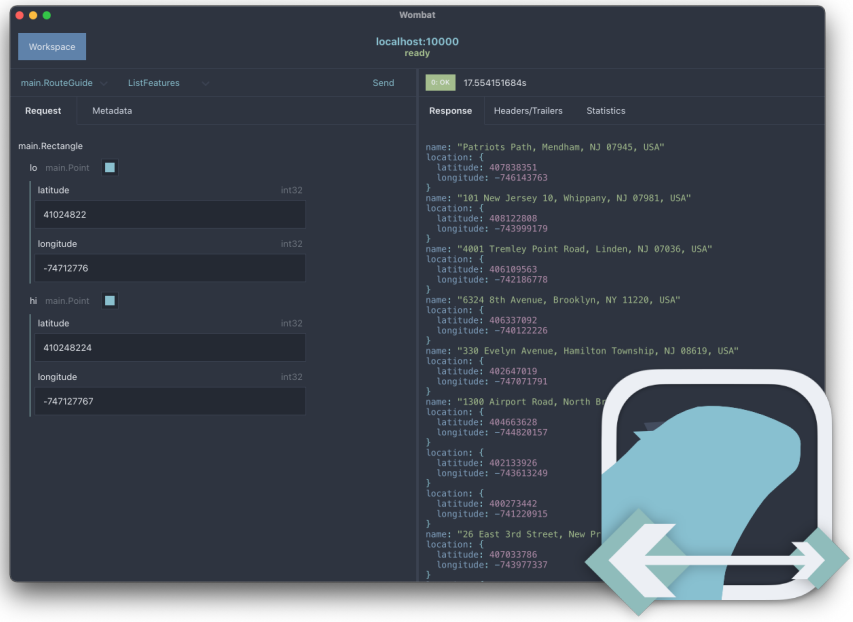


Figure 3.3 Wombat GUI [10]

The main features are:

- automatic parsing of proto definitions to render services and input messages,
- configuration of TLS,
- input form generation for all scalar types, nested messages, enums, repeated, oneof, and map,
- request metadata,
- execution of unary, server streaming, client streaming, bidirectional requests,
- pending request cancellation,

- sending EOF for client streaming,
- view response messages,
- view gRPC header and trailer,
- view RPC statistics,
- determine gRPC schema via reflection,
- support for Google Well Known Types,
- multiple workspace support [10].

The main disadvantages (compared to my task) are:

- desktop application (not a website),
- no support for documentation comments,
- last update 3 years ago [10].

3.2.1.2 BloomRPC

BloomRPC is a cross-platform gRPC desktop app client. The UI is in the figure 3.4. It is used to call gRPC services and inspect the responses. [11]

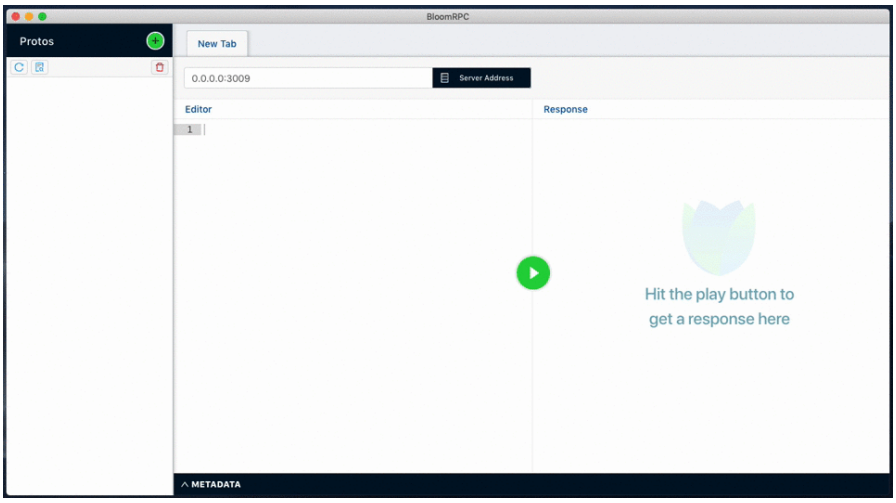


Figure 3.4 BloomRPC GUI [11]

The main features are:

- automatic parsing of proto definitions to list services and example messages,
- configuration of TLS,
- request metadata,
- execution of unary, server streaming, client streaming, bidirectional requests,
- selection between gRPC and gRPC-Web,
- pending request cancellation,

- sending EOF for client streaming,
- view response messages [11].

The main disadvantages (compared to my task) are:

- desktop application (not a website),
- no support for documentation comments,
- missing determine gRPC schema via reflection,
- missing gRPC headers and trailers preview,
- archived in 2023, usage is no longer recommended [11].

3.2.1.3 gRPC Docs and GenDocu

The gRPC Docs is a website API documentation generator by GenDocu. It provides RPC calls documentation for gRPC services. There is no option to call the services. This documentation is generated from the .proto files using protoc-gen-doc[7] utility with its custom format output as JSON. The example web UI is in the figure 3.5. [12]

GenDocu is a hosted gRPC Docs version. This version can call the gRPC services. But, at the time of writing, the GenDocu website is not working anymore, and the project looks abandoned (with the last commit being more than a year ago). [12]

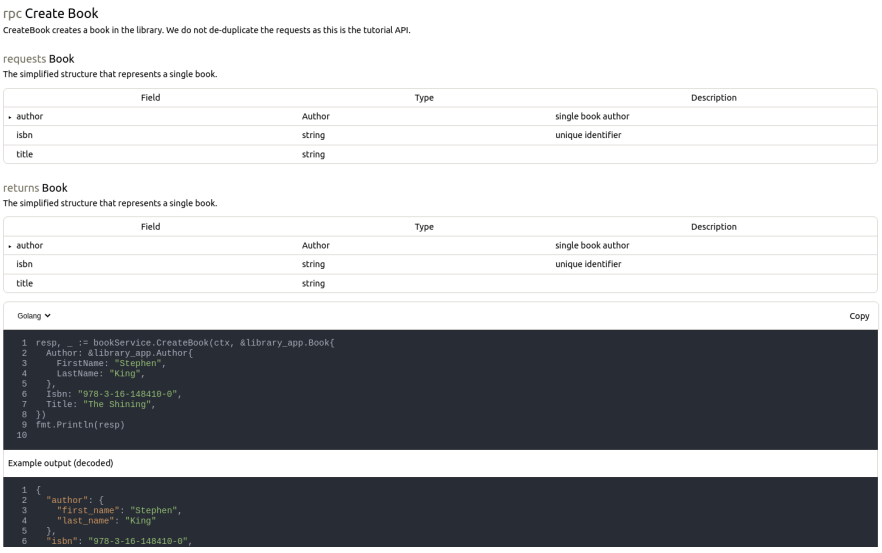


Figure 3.5 GenDocu Web UI [12]

The main features are:

- .proto files parsing and services, messages, enums generation,
- support for documentation comments,
- generation using common JSON format with the ability to change the source without redeploying the website [12].

[7] `https://github.com/pseudomuto/protoc-gen-doc`

The main disadvantages (compared to my task) are:

- missing determine gRPC schema via reflection,
- execution of unary, server streaming, client streaming, bidirectional requests,
- inactive and looks abandoned with no external websites working [12].

3.2.1.4 gRPC UI

The gRPC UI is a website that supports calling gRPC services. With this tool, you can browse the schema, which is presented as a list of available endpoints. The schema can be constructed by querying a server that supports server reflection, reading proto source files, or loading a compiled 'protoset' file (files containing an encoded file descriptor protos). The protoset file can be created using the protoc tool, which is used by the gRPC tooling for the client's code generation. The UI is in the figure 3.6. [13]

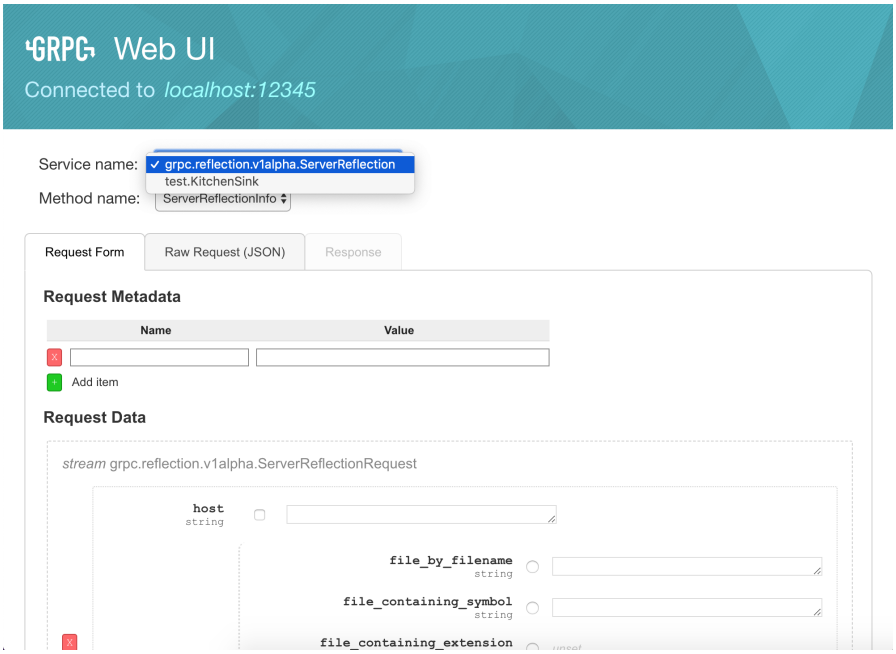


Figure 3.6 gRPC UI [13]

The main features are:

- listing services and methods,
- ability to construct messages with forms or raw JSON,
- execution of unary, server streaming, client streaming, bidirectional requests,
- request metadata, headers and trailers,
- configuration of TLS,
- rich support for well-known types,
- view response messages,
- actively maintained [13].

The main disadvantages (compared to my task) are:

- requires a server to run (not a static website),
- no support for documentation comments,
- requires you to construct the entire stream of request messages all at once, and then it shows the entire resulting stream of response messages all at once (so you can't interact with any streams interactively) [13].

3.2.1.5 letmegrpc

The letmegrpc is a website that supports calling gRPC services. It allows service methods to call using forms, where each method has its separate URL (`http://localhost:8080/ServiceName/MethodName`). It is constructed from a .proto file. The UI is in the figure 3.7. [14]

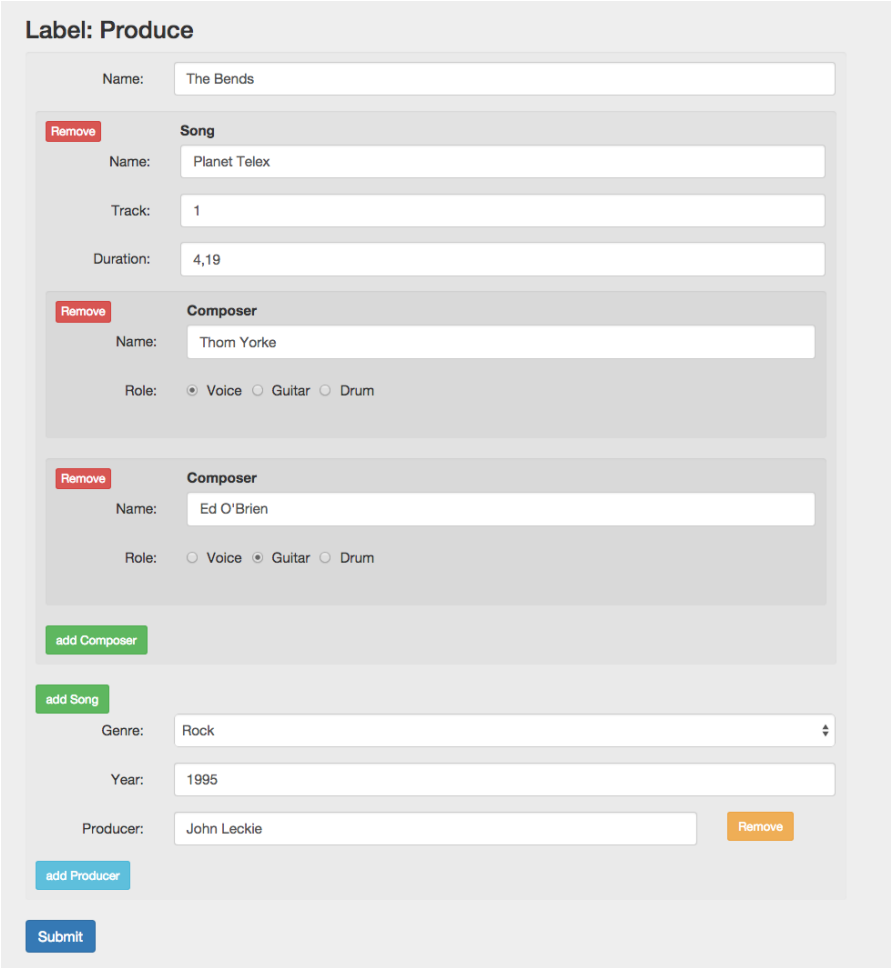


Figure 3.7 letmegrpc UI [14]

The main features are:

- comments to fields are in tooltips,
- ability to construct messages with forms,

- execution of unary, server streaming, client streaming, bidirectional requests,
- view response messages [14].

The main disadvantages (compared to my task) are:

- requires setup with manual go package downloads,
- no listing of services and methods,
- requires a server to run (not a static website),
- even though forms are powerful, they can be annoying to use, especially with complex requests,
- no support for request metadata, headers, and trailers,
- no direct support for gRPC reflection,
- custom proto file parser, which does not have implemented all language features ([15]),
- inactive, the last update was five years ago [14].

3.2.1.6 gRPC-swagger

The gRPC Swagger is a website. It is based on gRPC reflection and can list and call gRPC services. It uses Swagger UI design language (copy of the Swagger UI) but is not a part of the official Swagger. The UI is in the figure 3.8. The architecture is done using its own server as a proxy. The website's server forwards all requests to the gRPC server and back to the website. [14]

The main features are:

- listing of services and methods,
- familiar UI to the Swagger UI,
- support for request metadata, headers, and trailers,
- execution of unary, server streaming, client streaming, bidirectional requests,
- view response messages [16].

The main disadvantages (compared to my task) are:

- requires reflection enabled,
- no support for comments,
- requires a server to run (not a static website),
- it requires to construct the entire stream of request messages all at once, and then it shows the entire resulting stream of response messages all at once (so you can't interact with any streams interactively)
- request data are only in the form of JSON,
- inactive, last update two years ago, last release in 2020 [16].

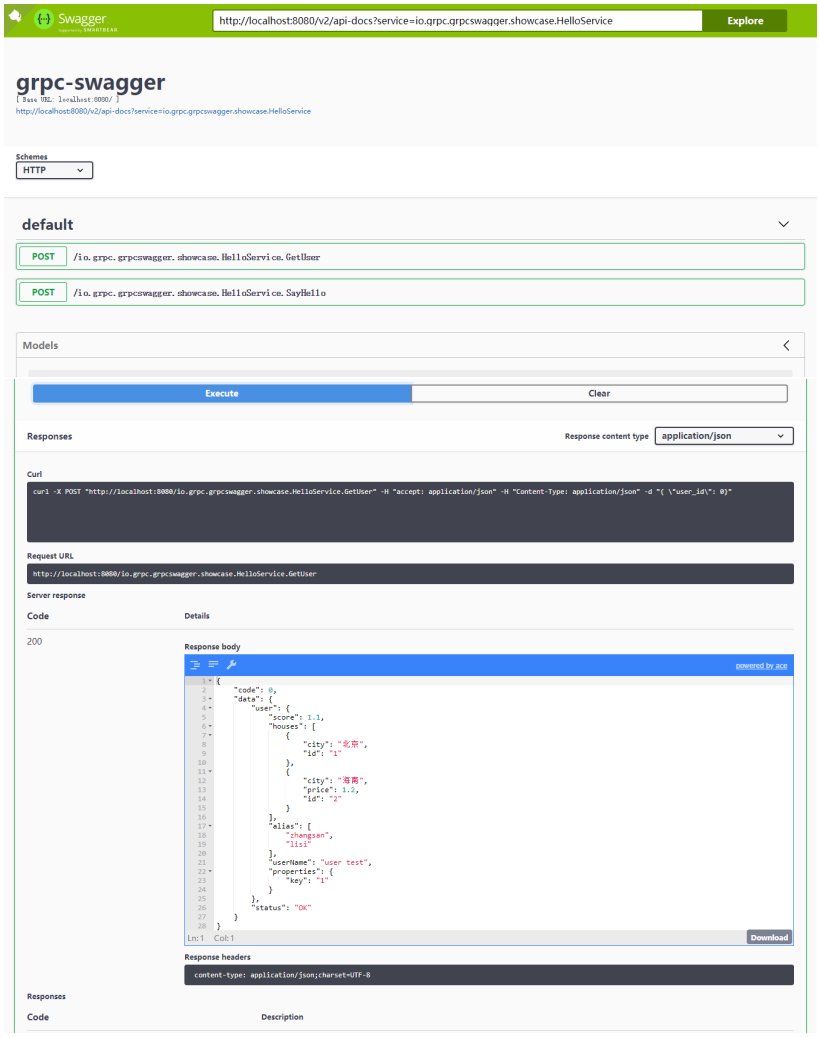


Figure 3.8 gRPC Swagger UI [16]

3.2.1.7 gRPC-Gateway

The gRPC-Gateway is a protoc compiler plugin. It generates a reverse proxy server, translating a RESTful JSON API into gRPC. So, it is a reverse proxy server, not a documentation website, nor a client for calling gRPC. It can also generate OpenAPI definitions using protoc-gen-openapiv2, which can be used with tools like Swagger UI. [14]

It uses annotations in the service definitions or a configuration file. An example of annotation is in the code snippet 3.2, and an example of the configuration file is in the code snippet 3.3. [14]

Code listing 3.2 gRPC-Gateway annotations [17]

```
import "google/api/annotations.proto";

rpc Echo(StringMessage) returns (StringMessage) {
  option (google.api.http) = {
    post: "/v1/example/echo"
    body: "*"
  };
}
```

Code listing 3.3 gRPC-Gateway configuration file [17]

```
type: google.api.Service
config_version: 3

http:
  rules:
    - selector: your.service.v1.YourService.Echo
      post: /v1/example/echo
      body: "*"
```

The main features are:

- support for request metadata, headers,
- ability to create OpenAPI definitions,
- execution of unary, server streaming, client streaming, bidirectional requests, but in a batched manner,
- and much more regards the specifics of the reverse-proxy layer [17].

The main disadvantages (compared to my task) are:

- dependent on OpenAPI/HTTP API - is not just a gRPC client, but much more,
- OpenAPI definitions do not have to reflect the gRPC service definitions,
- no support for trailers,
- no support for true bi-directional streaming,
- requires the underline gRPC service definitions,
- no support for reflection,
- no support for documentation/comments,
- requires a server to run (not a static website) [17].

3.2.1.8 Postman

Postman is the leading API platform [18]. It supports many different API types (e.g., HTTP API, WebSockets, GraphQL) but also includes gRPC. Upon loading .proto files or using gRPC reflection, it can understand the gRPC API and use the information for calls. It can do more than call the gRPC services, but it is not a documentation website. So, I will focus on its gRPC capabilities. The UI is in the figure 3.9. [19]

The main features are:

- listing of services and methods,
- execution of unary, server streaming, client streaming, bidirectional requests,
- broad range of features for API calls, including automated tests,
- support for request metadata, headers, and trailers,
- supports gRPC reflection,
- messages autocompletion,
- messages validation,

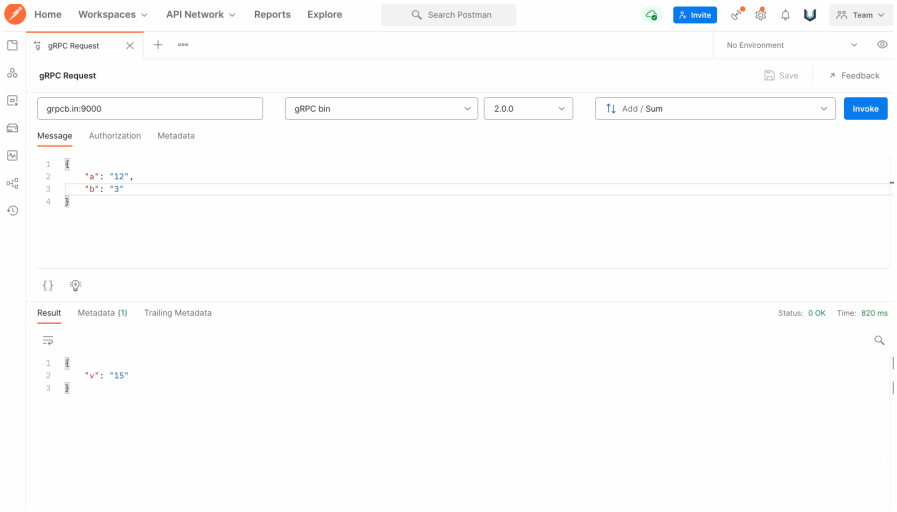


Figure 3.9 Postman UI [19]

- view response messages with full support for streaming [19].

The main disadvantages (compared to my task) are:

- it is a desktop application, not a static website,
- no support for comments,
- request data are only in the form of JSON [19].

3.2.1.9 proto2asciidoc

The proto2asciidoc is a plugin for the protoc tool that generates AsciiDoc documentation. Many tools can then parse the AsciiDoc format. One of these tools can be a static website. The main disadvantage is that it is not a website or application but a documentation file. Therefore, it does not allow calling the gRPC services at all. Another disadvantage is the special comment block formatting requirement, which differs from usual comments in proto files. And finally, it looks to be abandoned with the last commit more than two years ago. [20]

3.2.1.10 protoc-gen-doc

The protoc-gen-doc is a documentation generator for the protoc compiler tool. It can generate HTML, JSON, DocBook, and Markdown documentation from comments in .proto files. The main advantages are that it can take comments from the .proto files and generate a static documentation website or JSON, keeping the original structure of the input .proto files. The website contains the documentation of the services, methods, and messages. The JSON can then be used to create a custom website. For example, GenDocu mentioned earlier uses it. The disadvantage is that it does not allow the gRPC services to be called. [21]

3.2.2 GraphQL

In the GraphQL world, there are several tools for generating documentation or interactive calls. The most popular ones based on the state of GraphQL in 2022 and others I have found for documentation are in the following subsections [22].

3.2.2.1 GraphDoc

GraphDoc is a static website generator for GraphQL schemas. It supports live endpoint and .graphql definition files. The generated website contains the documentation of the queries, mutations, and types. They are shown in a file-like structure with interactive-type definition links. The UI is in the figure 3.10. [23]

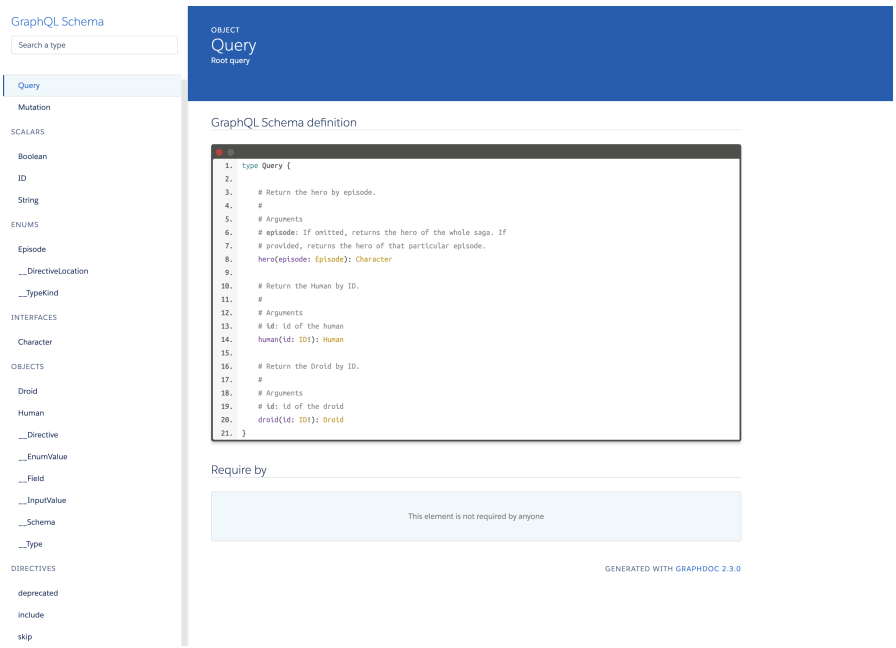


Figure 3.10 GraphDoc UI [23]

The main features are:

- static website,
- listing of queries, mutations, and types,
- generation from the live endpoint and .graphql definition files,
- documenting comments [23].

The main disadvantages (compared to my task) are:

- no requests execution,
- not actively maintained (last commit three years ago) [24].

3.2.2.2 GraphQL Playground

GraphQL Playground is an application for desktop and web. It allows developers to build and test GraphQL queries and mutations, explore an API schema, and view real-time results. GraphQL Playground offers features like automatic schema documentation, query history, and support for GraphQL subscriptions. It uses a live endpoint and can be hosted as a static website. The UI is in the figure 3.11. [24]

The main features are:

- context-aware autocompletion,

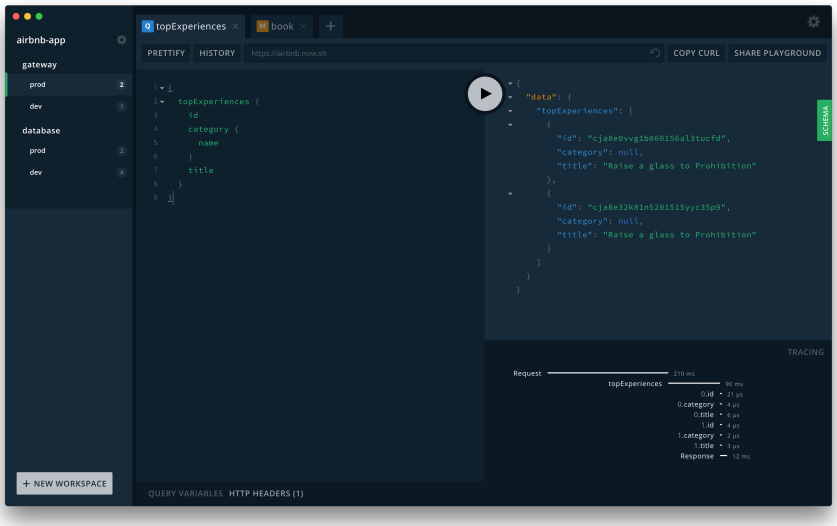


Figure 3.11 GraphQL Playground UI [24]

- listing of queries, mutations, and types,
- real-time GraphQL subscriptions,
- multiple projects and endpoints support,
- can be static website [24].

The main disadvantages (compared to my task) are:

- no documenting comments,
- not actively maintained (last commit two years ago, the last release in 2019) [24].

3.2.2.3 GraphiQL

GraphiQL is an interactive in-browser IDE. It allows GraphQL API documentation exploring and the ability to execute queries and mutations. It takes an endpoint and generates a static website. The UI is in the figure 3.12. [25]

The main features are:

- static website,
- documentation with comments,
- listing of queries, mutations, and types,
- execution of queries and mutations,
- autocompletion,
- metadata support [25].

The main disadvantages (compared to my task) are:

- disconnection between documentation and query execution,
- requires manual implementation - library usage [25].

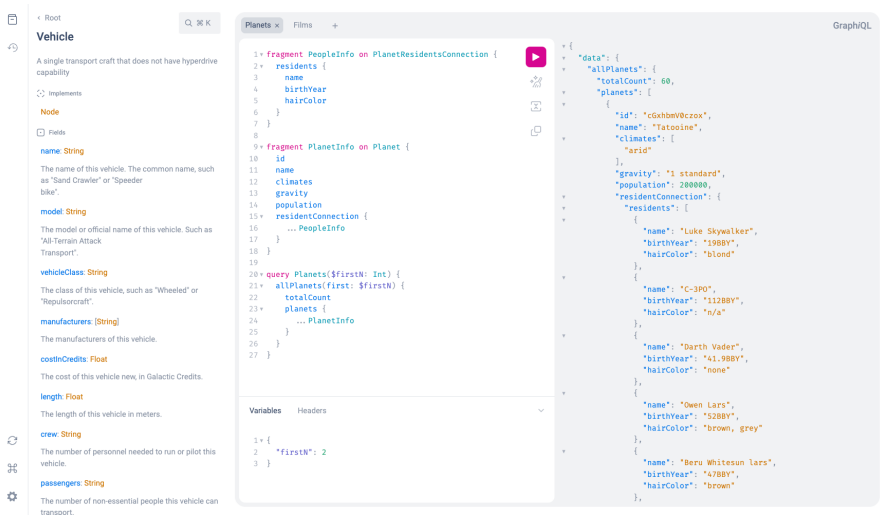


Figure 3.12 GraphiQL UI [25]

3.2.2.4 Apollo Studio

Apollo Studio is a cloud platform with features called Explorer or Schema. The Schema feature is for exploring the graph with documentation. The Explorer is a tool for executing queries and mutations, with the support of documentation, that is the closest to my task. It also contains features like monitoring, teams, changelog, deployment support, and more. The UI is in the figure 3.13. [26]

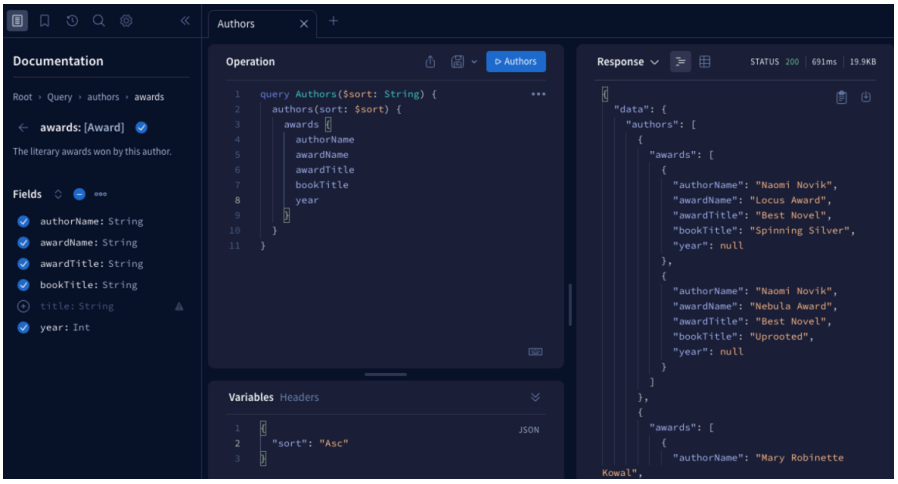


Figure 3.13 Apollo Studio UI [26]

The main features are:

- website,
- documentation with comments,
- listing of queries, mutations, and types,
- execution of queries and mutations,

- autocompletion,
- metadata support,
- option to construct queries using the schema with documentation,
- and much more [26].

The main disadvantage (compared to my task) is that it is not self-hosted and is a closed source [26].

3.2.3 RESTful API

In the RESTful API world, there are several tools for generating documentation or interactive calls. Swagger UI is one of the most popular ones, which I will mainly cover and compare with the gRPC options [27].

3.2.3.1 ReDoc

ReDoc is a tool for generating documentation based on the definition of OpenAPI. It is a static website with a responsive layout. It is used to preview documentation comments with example requests and responses and to show implementation examples. The UI is in the figure 3.14. [28]

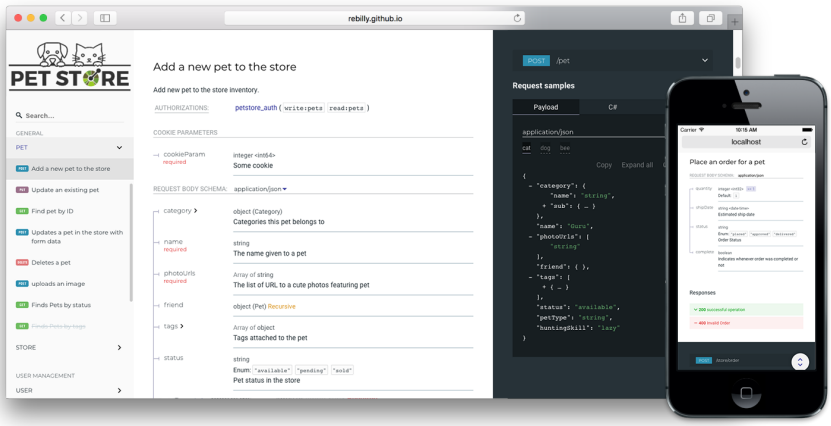


Figure 3.14 ReDoc UI [28]

The main features are:

- static website,
- responsive layout,
- documentation with comments,
- preview example requests and responses,
- implementation examples [28].

The main disadvantage (compared to my task) is no option of calling the endpoints [28].

3.2.3.2 RapiDoc

RapiDoc is interactive API documentation with OpenAPI definition. It is a static website, supports documentation comments, and allows calling the endpoints. The UI is in the figure 3.15. [29]

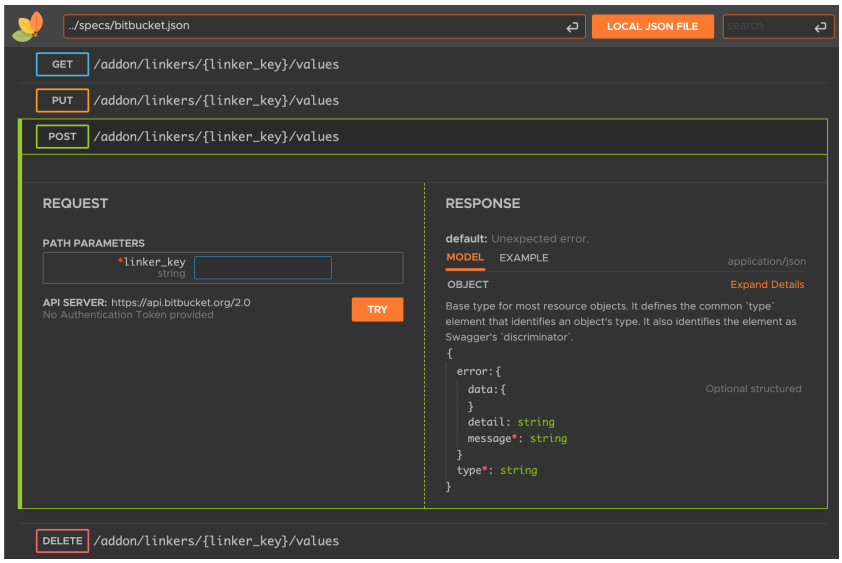


Figure 3.15 RapiDoc UI [29]

The main features are:

- static website,
- documentation with comments,
- preview example requests and responses,
- implementation examples,
- ability to call endpoints,
- support for metadata,
- support for authentication mechanisms [29].

I have not found any main disadvantages (compared to my task) of RapiDoc.

3.2.3.3 Swagger UI

Swagger UI is one of the most popular choices for RESTful API documentation [27]. The parent project is OpenAPI, a specification for general building HTTP APIs. This tool is used to generate documentation from the OpenAPI definition. It is a static website with support for comments and interactive endpoint calling. The UI is in the figure 3.16. [30]

The main features are:

- static website,
- documentation with comments,
- preview example requests and responses,

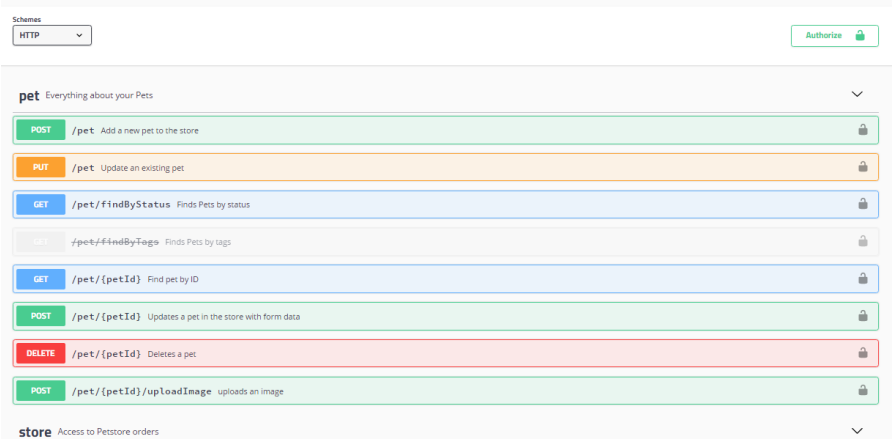


Figure 3.16 Swagger UI [30]

- implementation examples,
- ability to call endpoints,
- support for metadata,
- support for authentication mechanisms [30].

I have not found any main disadvantages (compared to my task) of Swagger UI.

3.2.4 Summary

I have compiled main tools for documenting and interacting with Protocol Buffers, GraphQL, and RESTful APIs. For each tool, I have described the overall image and the main features and disadvantages. While many of them excel in certain areas, they often fail to provide an experience that combines documentation rendering and interactive API exploration. The comparison of the main features of the Protocol Buffers tools is in the table 3.2.

For Protocol Buffers, the most dominant are desktop applications like Wombat, BloomRPC, and Postman, which facilitate the browsing and calling of gRPC services. However, these applications generally lack support for rendering documentation comments or determining gRPC schema via reflection.

Web-based solutions like gRPC Docs, gRPC UI, letmegrpc, and gRPC-swagger offer interactive gRPC calling capabilities. Still, they encounter limitations like interactive streaming support, documentation comments integration, or special server setup requirements. This means developers must set up and maintain a server infrastructure specifically for API documentation and interaction.

Additionally, static documentation generators like proto2asciidoc or protoc-gen-doc can render comments from .proto files as markup, but they do not allow direct interaction with the services. The gRPC-Gateway is a unique tool that generates a reverse proxy server for translating HTTP JSON APIs into gRPC. While it is not directly comparable to the other tools, it is worth mentioning due to its ability to generate OpenAPI definitions, which can be used with existing tools for HTTP APIs.

Also, it is important to point out that maintenance of the tools is generally not good, with most of them being inactive for more than a year. Only gRPC UI, gRPC-Gateway, protoc-gen-doc, and Postman are actively maintained.

Overall, the combination of having a static website with documenting comments and being able to call specific service methods interactively is lacking in current gRPC tools.

	Wombat	**BloomRPC**	**gRPC Docs**	**gRPC UI**	**letmegrpc**	**gRPC-swagger**	**gRPC-Gateway**	**Postman**	**proto2asciidoc**	**protoc-gen-doc**
Type	Desktop App	Desktop App	Static Web	Website	Website	Website	Proxy	Desktop App	AsciiDoc	Static Web, JSON
Listing of services & methods	x	x	x	x		x		x	x	x
Input as form	x			x	x					
Input as JSON		x		x		x		x		
Request metadata	x	x		x		x	x	x		
Headers & Trailers	x			x		x	x	x		
Execution of requests	x	x		x	x	x	x	x		
Determine schema via gRPC reflection	x			x		x	x	x	? (not specified)	? (not specified)
Schema from .proto files	x	x	x	x	x		x	x		
Documentation comments					∼ (in tooltips)				x	x
Maintained				x			x	x		x

Table 3.2 Protocol Buffers comparison

In the GraphQL realm, a documentation-focused tool graphdoc generates static documentation websites from schema definitions, providing a comprehensive view of the available queries, mutations, and types. However, this tool cannot execute queries and mutations, limiting its usefulness for interactive API exploration.

On the other hand, GraphQL Playground and GraphiQL are powerful web IDEs that do not require a specific server to be running and excel in executing queries and mutations. Still, they struggle to integrate documentation seamlessly and comprehensively in their UI, where only GraphiQL includes the documentation comments but is not able to connect them with the queries directly. The most advanced tool is Apollo Studio, a cloud platform that offers a wide range of features for exploring and executing GraphQL APIs. However, it still lacks a self-hosted solution for static website generation. As a result, none of the most popular tools I have found combine comprehensive documentation rendering with fully interactive query capabilities in a static website format.

In contrast, the RESTful API ecosystem is in a vastly different state. The ReDoc tool is a static website generator that renders documentation from OpenAPI definitions, providing a view of the available endpoints, request/response examples, and implementation examples. However, it does not support interactive API execution.

Solutions like Swagger UI and RapiDoc render documentation from OpenAPI definitions while allowing interactive API execution. These tools provide a cohesive experience for developers working with RESTful APIs, blending documentation and interaction capabilities while being able to run as a static website. I have found no significant disadvantages to either of these tools.

3.2.4.1 Comparison

Comparing the tools for Protocol Buffers, GraphQL, and RESTful APIs, I have found that the RESTful API tools are the most advanced in terms of combining documentation rendering with interactive API exploration. Tools like Swagger UI and RapiDoc provide a cohesive experience for developers working with RESTful APIs, blending documentation and interaction capabilities while being able to run a static website. They have features like listing endpoints, input as form or JSON (in a combined way, where the form is only to a first level of hierarchy deepness), request/response metadata, execution of requests, and documentation comments.

The GraphQL tools are also quite advanced, but they often struggle with integrating documentation comments into the interactive query execution. The Protocol Buffers tools are the least advanced, with most tools focusing on either documentation rendering or interactive API exploration, but not both. Also, the maintenance of the tools is generally not good, with most of them being inactive for more than a year.

3.2.4.2 Issues

Despite the strengths of existing tools for Protocol Buffers, there remains a notable gap for an integrated solution that can cater to a unified static web generator that can provide a cohesive experience by combining documentation and fully interactive request/response capabilities with support for features like streaming, headers and trailers metadata. This website generator should be able to create the static website from .proto files or gRPC reflection.

The main issues I will be addressing are:

- documentation comments,
- interactive streaming,
- gRPC reflection,
- request metadata, headers, and trailers.

3.3 Requirements

The main requirement is to create a static website documentation for Protocol Buffers, which supports API calls. The website should be able to render documentation comments, list services and methods, execute requests, and support request metadata, headers, and trailers. It has to understand all gRPC features like messages, services, methods, and enums. The website should be generated from .proto files or gRPC reflection.

Based on the issues and primary features described previously in the tools, I have compiled functional and non-functional requirements covering the required functionality of the static website interactive documentation generator.

3.3.1 Functional Requirements

F1 List services and methods
The website should list all services and methods available in the Protocol Buffers.

F2 List message types
The website should list all message types available in the Protocol Buffers.

F3 List enum types
The website should list all enum types available in the Protocol Buffers.

F4 Show comments for services, methods, message types, and enum types
Comments from the Protocol Buffers definitions should be rendered on the website for services, methods, message, and enum types.

F5 Show comments for fields
Comments from the Protocol Buffers definitions should be rendered on the website for fields in message or enum types.

F6 Execution of methods with unary requests
The website should allow for the execution of unary requests.

F7 Execution of methods with server streaming requests
The website should allow for the execution of server streaming gRPC method requests. Also, streaming cancellation should be allowed in the middle of the execution.

F8 Request body message input
The website should allow request data input as a form or JSON. It has to support all scalar types, including messages and enums. The input should eventually also contain an example of the message structure.

F9 Input form validation for the correct message structure
The website should validate the input form for the correct message structure and show an error if the structure is incorrect.

F10 Request metadata input
The website should allow for input of request metadata.

F11 Response headers and trailers
The website should show response headers and trailers.

F12 Response message
The website should display the response message or messages.

F13 Support for oneof fields
The website should support oneof the fields in the message types. They must be supported in the input request and the message definition.

F14 Support for map fields
The website should support map fields in the message types. They must be supported in the input request and the message definition.

F15 Support for repeated fields
The website should support repeated fields in the message types. They must be supported in the input request and the message definition.

F16 Support for nested messages
The website should support nested messages in the message types. They must be supported in the input request and the message definition.

F17 Support for well-known types
The website should support well-known types like Timestamp, Duration, FieldMask, etc. They must be supported in the input request and the message definition.

F18 Generate website from .proto files
The generator should be able to generate the website from .proto files. The .proto files can be a single file or a folder.

F19 Generate a website from gRPC reflection
The generator should be able to generate the website from gRPC reflection. The reflection can be from a server or a protoset definition file.

F20 Global metadata and authorization
The website should allow for global metadata definition and authorization metadata. It should be possible to set the metadata for all requests and methods.

F21 Options should be shown
The website should show the options for services and methods.

3.3.2 Non-Functional Requirements

N1 Static Website
The website should be static. It means only a combination of static files like HTML, CSS, and JavaScript without requiring a dynamic server to be run. This should allow for easy deployment and hosting on various platforms.

N2 Familiar UI
The website should have a familiar UI for developers, probably close to something already used in the RESTful or GraphQL worlds. It should be easy to navigate and understand, with a clear structure and layout.

N3 Input as form or JSON
The website should allow request data input as a form or JSON. Depending on the developer's preference, this should allow for flexibility in how the data is entered.

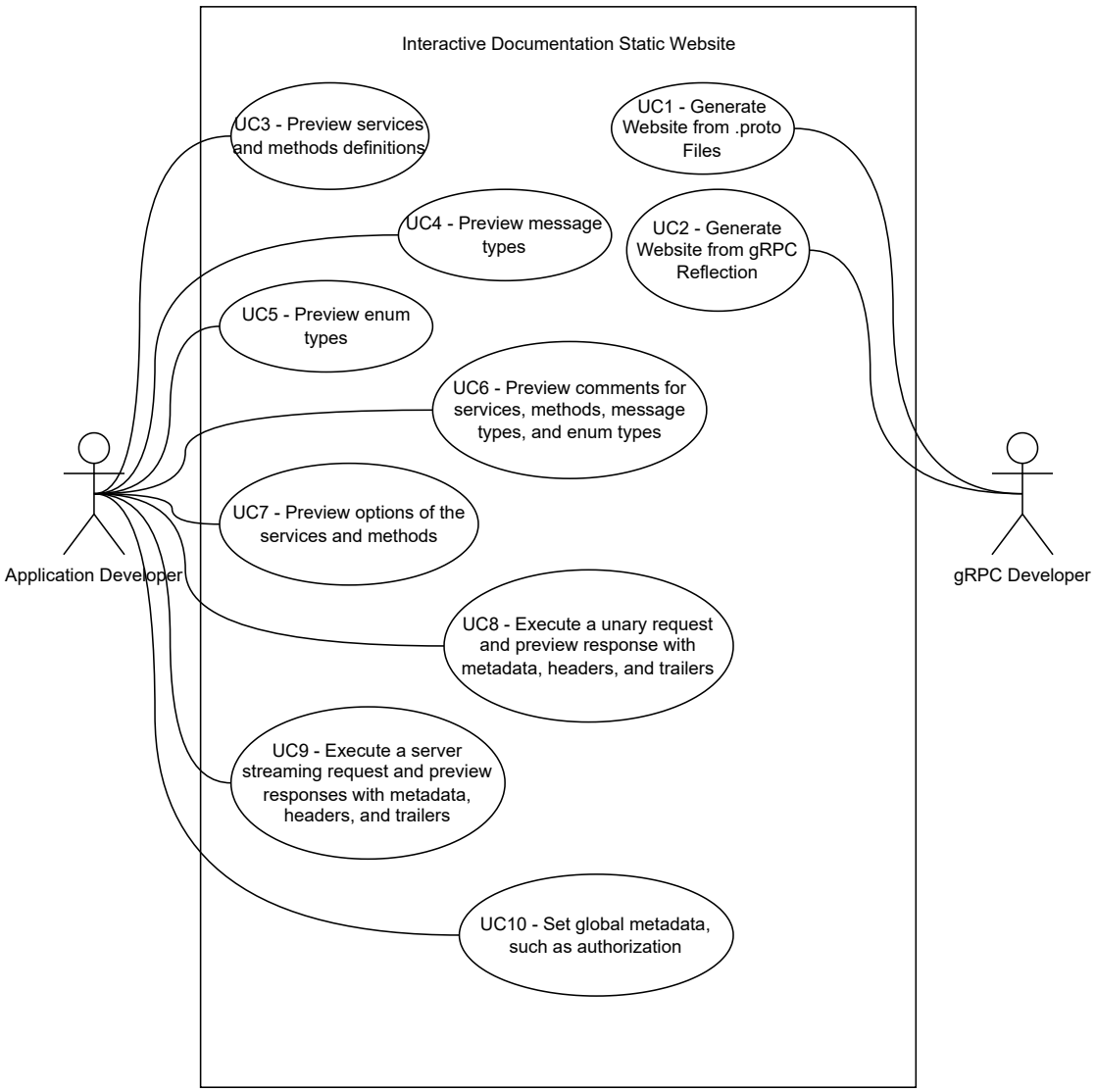


Figure 3.17 Use case diagram

3.4 Use Cases

Based on the requirements and comparing the existing tools, I have compiled a list of use cases that the static website interactive documentation generator should support. The figure 3.17 shows the diagram of the use cases and their relationships with the actors. In my case, I have two actors. One is the Application Developer, who uses the website to explore and interact with the gRPC services. The other is the gRPC Developer, who generates the static website from the .proto files or gRPC reflection.

3.4.1 UC1 – Generate Website from .proto Files

Primary Actor: gRPC Developer
Preconditions: The gRPC Developer has the .proto files ready. Either as separate files in a folder or as interlinked files using imports.

Goal: The website is generated.
Main Scenario: The gRPC Developer runs the generator. The generator reads the .proto files. The generator generates the website's necessary files for these definitions. The website is ready for the gRPC Developer to be hosted.
Alternative Scenario 1: The generator finds an error while parsing the .proto files. So, it shows an error message to the gRPC Developer. The gRPC Developer has to fix the error in the .proto files and rerun the generator.
Alternative Scenario 2: The .proto files do not exist. The generator shows an error message to the gRPC Developer. The gRPC Developer has to provide the .proto files and rerun the generator.

3.4.2 UC2 – Generate Website from gRPC Reflection

Primary Actor: gRPC Developer
Preconditions: The gRPC Developer has the gRPC server with reflection enabled, ready and running.
Goal: The website is generated.
Main Scenario: With reflection enabled, the gRPC Developer runs some tools to get the Protocol Buffer definitions from the target gRPC server. After the Protocol Buffer definitions are created, the gRPC Developer runs the generator and feeds it the definitions. The generator reads the definitions and generates the website's necessary files. The website is ready for the gRPC Developer to be hosted.
Alternative Scenario 1: The generator finds an error while parsing the definitions file. So, it shows an error message to the gRPC Developer. The gRPC Developer has to regenerate the definitions file and rerun the generator.
Alternative Scenario 2: The definition file does not exist. The generator shows an error message to the gRPC Developer. The gRPC Developer has to provide the definition file and rerun the generator.

3.4.3 UC3 – Preview services and methods definitions

Primary Actor: Application Developer
Preconditions: The Application Developer has opened the static website with the Protocol Buffers definition defined.
Goal: The services and methods are shown.
Main Scenario: The Application Developer opens the website. The website shows the services and methods available in the definitions of Protocol Buffers. The Application Developer can see the details of the services and methods with their relevant package paths.
Alternative Scenario 1: The website does not show the services and methods. The Application Developer sees an error message or empty page. The Application Developer has to ask the gRPC Developer to check the Protocol Buffers definitions and regenerate the website.

3.4.4 UC4 – Preview message types

Primary Actor: Application Developer
Preconditions: The Application Developer has opened the static website with the Protocol Buffers definition defined.
Goal: The message types are shown.
Main Scenario: The Application Developer opens the website. The website shows the message types available in the definitions of Protocol Buffers. The Application Developer can see the details of the message type with their fields and nested message types.

Alternative Scenario 1: The website does not show the message types. The Application Developer sees an error message or empty page. The Application Developer has to ask the gRPC Developer to check the Protocol Buffers definitions and regenerate the website.

3.4.5 UC5 – Preview enum types

Primary Actor: Application Developer
Preconditions: The Application Developer has opened the static website with the Protocol Buffers definition defined.
Goal: The enum types are shown.
Main Scenario: The Application Developer opens the website. The website shows the enum types available in the definitions of Protocol Buffers. The Application Developer can see the details of the enum type with their keys and values.
Alternative Scenario 1: The website does not show the enum types. The Application Developer sees an error message or empty page. The Application Developer has to ask the gRPC Developer to check the Protocol Buffers definitions and regenerate the website.

3.4.6 UC6 – Preview comments for services, methods, message types, and enum types

Primary Actor: Application Developer
Preconditions: The Application Developer has opened the static website with the Protocol Buffers definition defined.
Goal: The comments are shown.
Main Scenario: The Application Developer opens the website. The website shows all defined comments for services, methods, message types, and enum types. The Application Developer can see all the comments for each of them.
Alternative Scenario 1: The website does not show the defined comments. The Application Developer sees an error message or no comments on the page. The Application Developer has to ask the gRPC Developer to check the Protocol Buffers definitions and regenerate the website.

3.4.7 UC7 – Preview options of the services and methods

Primary Actor: Application Developer
Preconditions: The Application Developer has opened the static website with the Protocol Buffers definition defined.
Goal: The options are shown.
Main Scenario: The Application Developer opens the website. The website shows all defined options for services and methods. The Application Developer can see the last defined options for each of them.
Alternative Scenario 1: The website does not show the defined options. The Application Developer sees an error message or no options on the page. The Application Developer has to ask the gRPC Developer to check the Protocol Buffers definitions and regenerate the website.

3.4.8 UC8 – Execute a unary request and preview response with metadata, headers, and trailers

Primary Actor: Application Developer
Preconditions: The Application Developer has opened the static website with the Protocol Buffers definition defined.

Goal: Execution is successful, and the response is shown.
Main Scenario: The Application Developer opens the website. First, they define the target server URL. Then, they select a service and method. They fill in the request data and metadata. They execute the request. The website shows headers, then the response, and finally, the trailers.
Alternative Scenario 1: The request data are invalid. The website shows an error message. The Application Developer has to fix the request data and rerun the request.
Alternative Scenario 2: The request fails for any reason. The website shows an error message. The Application Developer has to check the server status, the request data, and metadata and rerun the request.
Alternative Scenario 3: The request is taking too long. The Application developer can cancel the request in the middle of the execution. They have to check the server status, the request data, and metadata and rerun the request.

3.4.9 UC9 – Execute a server streaming request and preview responses with metadata, headers, and trailers

Primary Actor: Application Developer
Preconditions: The Application Developer has opened the static website with the Protocol Buffers definition defined.
Goal: Execution is successful, and all responses are shown.
Main Scenario: The Application Developer opens the website. First, they define the target server URL. Then, they select a service and method. They fill in the request data and metadata. They execute the request. The website shows headers, then all responses while they are arriving from the server one by one, and finally, trailers.
Alternative Scenario 1: The request data are invalid. The website shows an error message. The Application Developer has to fix the request data and rerun the request.
Alternative Scenario 2: The request fails for any reason. The website shows an error message. The Application Developer has to check the server status, the request data, and metadata and rerun the request.
Alternative Scenario 3: The request is taking too long. The Application developer can cancel the request in the middle of the execution. They have to check the server status, the request data, and metadata and rerun the request.

3.4.10 UC10 – Set global metadata, such as authorization

Primary Actor: Application Developer
Preconditions: The Application Developer has opened the static website with the Protocol Buffers definition defined.
Goal: Execution is successful, and the global metadata is included in the request.
Main Scenario: The Application Developer opens the website. First, they define the target server URL. Then, they select a service and method. They fill in the request data and metadata. They execute the request. The website shows the executed request containing the global metadata.

3.5 Requirements to Use Cases Mapping

I have compiled a table of requirements and use cases mapping (see table 3.3) to verify that all requirements are covered by at least one use case and that no use case is unnecessary. The table shows that all requirements are covered.

	Use Cases									
	1	2	3	4	5	6	7	8	9	10
F1			x							
F2				x						
F3					x					
F4						x				
F5						x				
F6								x		
F7									x	
F8								x	x	
F9								x	x	
F10								x	x	
F11								x	x	
F12								x	x	
F13				x				x	x	
F14				x				x	x	
F15				x				x	x	
F16				x				x	x	
F17				x						
F18	x									
F19		x								
F20										x
F21							x			

Table 3.3 Requirements to use cases mapping

Chapter 4

Design

Based on the analysis, as one of the most advanced documentation generators is Swagger UI, it will be my main inspiration. Also, the Swagger UI is a well-known tool in the software development community, and many companies and projects use it. The Swagger UI works so the website is static by default, and the endpoints, types, and more are defined in a definition file. This file is an OpenAPI YAML or JSON file, which is then parsed and displayed on the website [31]. This is a good approach, as it separates the data from the view. This allows the data to be easily changed, and the website will regenerate dynamically in the browser. The deployment of the website is also more accessible, as it is just a static website, which can be hosted on any web server, and updating the definitions can be part of different deployment processes.

First, I will analyze if I could use Swagger UI directly or if I need to create my own website.

4.1 Swagger UI for gRPC

The first idea is to use Swagger UI and generate the OpenAPI definition file from the gRPC proto files. This approach would benefit from reusing a known tool with all its features. The issue is that OpenAPI Specification is made for HTTP APIs, which gRPC APIs, although using HTTP/2, are not [32]. This means the OpenAPI Specification does not support gRPC specifics, like the streaming or mapping service methods to HTTP endpoints. Implementation of such would require compromises on the translation layer between the two. As described in the analysis, the gRPC-Gateway is an example of this translation, but it is not a perfect solution, and it has its limitations, which I want to avoid.

Another option is using the Swagger UI plugin system [33]. This would allow me to use the swagger-ui package as a dependency and build all the gRPC specifics on top of it, including reading different specifications than OpenAPI. The positive side of this approach is that I would be able to use the Swagger UI design. On the other hand, I would need to reimplement almost every single part in order to be able to work with gRPC specifics and, in the future, maintain compatibility with possible Swagger UI updates. This would leave me only with a page skeleton. That is a toolbar and a background color. I do not see that as a significant advantage over creating my own website.

In the end, I have decided to create my own website, which will be inspired by the Swagger UI design, but it will be adjusted to the gRPC specifics.

4.2 gRPC-Web Limitations

Previously, I mentioned the limitations of the gRPC-Web. The most important influence will be based on the lack of support for client streaming and bidirectional streaming, even though its implementation is on the roadmap. This means that the website will not be able to execute these method types, and the execution will be limited to unary and server streaming methods only. However, it does not affect the design of the website. It can still show all relevant documentation and have the implementation ready when gRPC-Web developers add support.

For the possibility of gRPC-Web support for client streaming and bidirectional streaming not being added in the following years, I will prepare an interface to use a custom gRPC server proxy. If an implementation of that interface is provided, the website will be able to use it, and the user will be able to use all gRPC methods, including client streaming and bidirectional streaming. This solution will not be part of my thesis, but I will prepare the website so it can be added (or removed altogether) in the future.

The server streaming also has a limitation in gRPC-Web, which is that it works only in *application/grpc-web-text* format [7]. This means that the payload is base64 encoded, and it is not possible to use the *application/grpc-web+proto* content type, which uses payload in binary format. Only unary calls support both *an application/grpc-web+proto* and *application/grpc-web-text* content types. Therefore, unary requests will feature a selection of content types, which users will be able to change.

4.3 gRPC Reflection Possibility

Before I start designing the website architecture, I need to decide how the gRPC reflection as a data source could be used compared to the proto files. Any tool using gRPC reflection has to support the connection using gRPC itself. Therefore, it is not possible to use it directly in the browser. A server-side tool or client will need to be used to connect to the gRPC server and get the reflection data. The grpcurl tool mentioned in the analysis allows not only creating the client but also exporting a "protoset" file. This is a file containing the reflection metadata in its own format. This file can be used instead of a direct gRPC server as a reflection source. But it can also be read with tools other than grpcurl. For example, the gRPC UI tool elevates this in its own implementation [13]. This will allow for the documentation website to either use the protoset file directly or convert the protoset file to a different common format, which the website will then use.

The limitation of the gRPC reflection is that it does not provide comments from the proto files. This means the website will not be able to show the comments, which are part of the proto files, if the reflection is used. This is a significant limitation, as the comments are part of the documentation, and they are important for the user to understand the methods and types. The good thing is that it is being tracked on GitHub (`https://github.com/grpc/grpc/issues/22680`) by the gRPC developers, and when it is implemented, the documentation website will be able to use it. For now, if documentation comments are required, the proto files will have to be used as the data source.

4.4 Architecture

The architecture of the website will be based on the Swagger UI, but it will be adjusted to the gRPC specifics. The website will be static, and it will take a gRPC definition file to show all gRPC-related parts. The gRPC definition file from the proto files or gRPC reflection and its format will be most probably either JSON or YAML. These formats are common in software development, and they are also used by the Swagger UI, from which I take inspiration.

I have created a diagram of the website architecture, which is shown in the figure 4.1. There are two places where the generation of the common format starts. The first one is from the .proto file. The second one is from the gRPC reflection. Both methods will generate the same format that the website will use. The website will be static and generated using the common format on the fly using client-side programming languages like JavaScript. The user of the website will be able to see all the methods, types, and enums, which are defined in the gRPC definition file. They will also be able to call the methods and see the responses on the website. When they invoke the request, the website will select the correct backend (gRPC-Web or other implementation), and it will be the server. The server then returns the response, which is then displayed on the website (the responses can be shown gradually as, for example, the server streams responses).

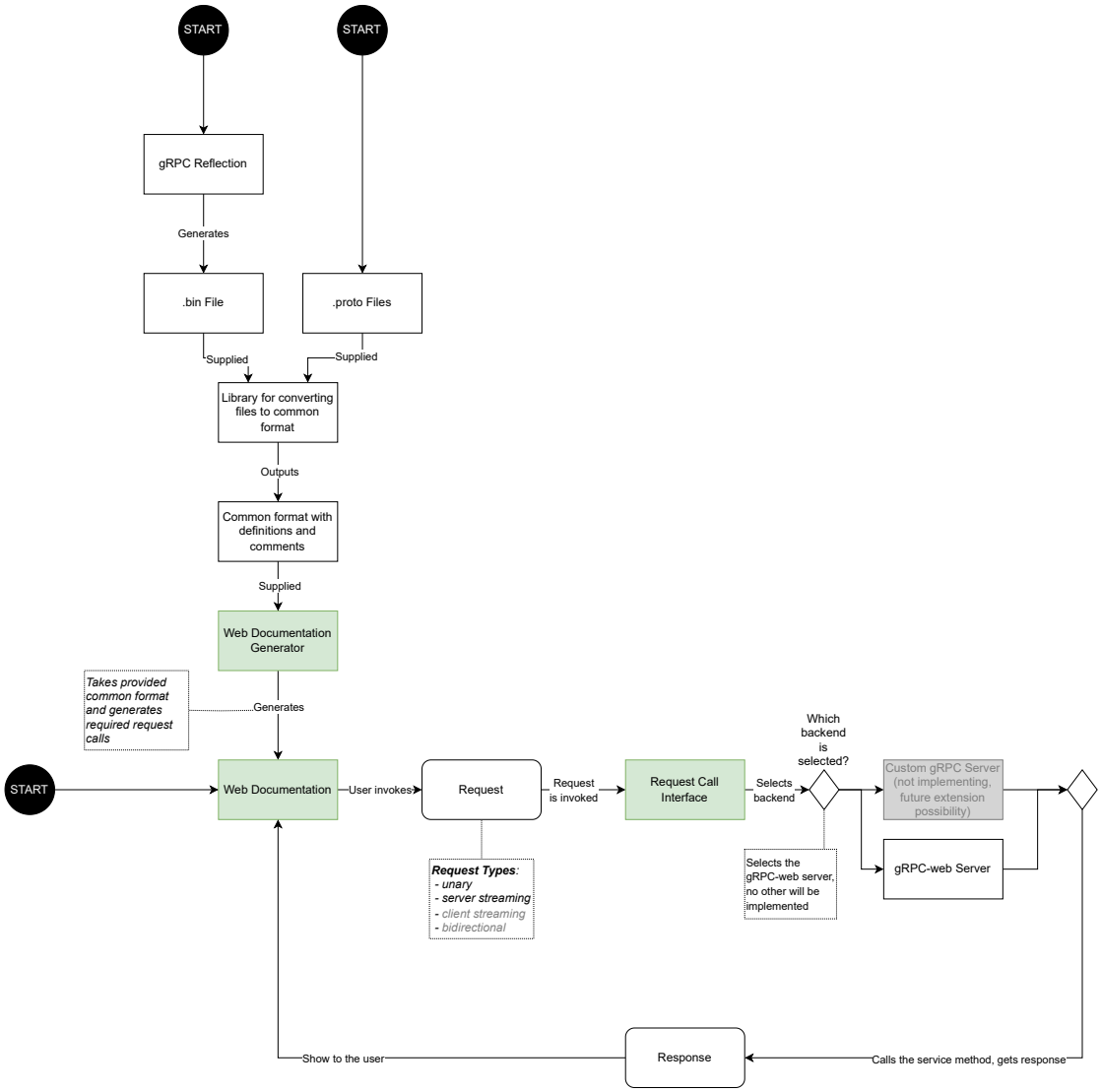


Figure 4.1 Architecture

This architecture should allow static website hosting, with the possibility of generating the website on the fly based on the common format definition. The deployment of that website could be then split into deploying the website itself and deploying the common format definition, which may allow more possibilities for gRPC API developers or maintainers. It will be also able

to execute queries and, more importantly, show server streaming responses right at the moment they are received from the server. The website will be able to work with both proto files and gRPC reflection, and it will be ready for future gRPC-Web support for client streaming and bidirectional streaming.

4.5 Common Format

As previously mentioned, the website will use a common format for the gRPC definitions, from which the website will be generated on the fly. In this section, I will discuss the possibilities of already existing common formats, and I will choose the most suitable one for the website. I will not consider the OpenAPI standard anymore, as I have discussed previously in the Swagger UI possibility.

4.5.1 grpc-protoc-gen-doc

The grpc-protoc-gen-doc[1] allows JSON output generation, including comments. This is great for documentation, but it does not allow the generation of the necessary gRPC metadata used for constructing gRPC requests. That means the website would not allow gRPC methods to be executed just from this file, and additional metadata would be needed.

4.5.2 gnostic

The gnostic[2] is a compiler for APIs described by the OpenAPI Specification. It generates JSON and YAML OpenAPI descriptors to (and from) Protocol Buffer representations. It uses gRPC options to specify HTTP paths and other information. [34]

This is a good option, as it is able to generate the OpenAPI Specification from the proto files. The issue is that the binary format of the proto files is defined by gnostic, so all generations have to be done from YAML or JSON, not from the proto files at all. This makes this library not suitable for use as a common format, as it would require the proto files to be converted to YAML or JSON first. Or implementing a script for converting proto files to the gnostic binary format, which is still missing all the implementation needed for the website itself.

4.5.3 protobufjs

The protobufjs[3] library is a JavaScript implementation with TypeScript support of the Protocol Buffers serialization with over fourteen million downloads per week. It is able to parse the proto files and query the data from them. The data can then be used to serialize and deserialize the messages in binary format when used in gRPC calls. It does not execute the gRPC calls directly, but this functionality can be implemented using other libraries. [35]

This library allows for listing all services, methods, types, enums, and all other related parts of proto files, such as fields, options, and, most importantly, also comments. It is able to parse the proto files and generate a JSON output, which can be then used instead of the proto files directly. This is a possible candidate for the common format, as it is able to generate the necessary data for the website, and it is also able to generate the binary format for the gRPC calls.

There is also a protobufjs/ext/descriptor[4] extension, which is able to parse and decode the descriptor.proto files used in the binary format of gRPC reflection. This means that the protobufjs library is able to work with both proto files and gRPC reflection. Using this extension, I

[1] https://github.com/pseudomuto/protoc-gen-doc
[2] https://github.com/google/gnostic
[3] https://github.com/protobufjs/protobuf.js
[4] https://github.com/protobufjs/protobuf.js/blob/master/ext/descriptor/README.md

can generate the common JSON format, and the website will be able to work with the reflection too.

4.5.4 Summary

Based on the options I have found, the protobufjs library looks like the best option for the common format. It is able to generate the JSON output, which can be used as a common format for the proto files and the gRPC reflection. It also allows the creation of the binary message format necessary for the gRPC calls. It is also able to parse the comments from the proto files, which is important for documentation purposes.

I could also create my own parser, which would parse the proto files and generate the JSON output. But this would require a lot of work, and it would not be as good as the protobufjs library, which is already broadly used. Therefore, I have decided to use the protobufjs library for the common format.

4.6 Website Design

Based on the architecture, the website generator will contain three parts:

- proto files generator,
- gRPC reflection generator,
- static website.

The proto files generator will generate the JSON output from the proto files. The gRPC reflection generator will generate the JSON output from the gRPC reflection. The static website will take the JSON output and generate the website on the fly in the browser.

The website design will be inspired by the Swagger UI design, but it will be adjusted to the gRPC specifics. This should allow the developers to get familiar with the website quickly, as they may already know the Swagger UI design.

I will describe each part in more detail in the following sections, and I will also show the wireframes of the website.

4.6.1 Proto Files Generator

The proto files generator will be a command line script that takes the proto files as input and generates the JSON output. It should also allow the user to use a folder of proto files. The protobufjs library will define the JSON output and should contain all the necessary data for the website, such as services, methods, types, enums, and comments.

4.6.2 gRPC Reflection Generator

The gRPC reflection generator will be a command line script that takes the gRPC reflection definition file as input and generates the JSON output. The input bin file will be the protoset file generated by the grpcurl tool. The protobufjs library will define the JSON output and should contain all the necessary data for the website, such as services, methods, types, enums, and comments.

4.6.3 Website Wireframe

The main website wireframe is in the figure 4.2. It starts with a toolbar at the top, which contains the name of the website in the format of a logo and a definition file link with a preview button. The definition file link will allow the user to upload the definition file, and the preview button will generate the website on the fly. This link can be either a local file or a remote path.

Next, there is a part with a brief explanation of what this website is and what it allows. This part may be used in the future for general gRPC API information of the particular definition file.

Then, there are settings fields and options that apply to the whole website. The first one is a method selection. This is a dropdown list that allows users to choose which backend implementation they want to use. Right now, only gRPC-Web is supported, but in the future, there may be more options, such as custom gRPC server proxy or anything else. Next is a gRPC server URL. This is a text field where the user can enter the URL of the gRPC server, which is later used as the backend URL for gRPC calls. For the gRPC-Web implementation, this URL will be the URL of the gRPC-Web proxy, such as Envoy. The last setting is metadata. The button will open a modal with a table where the user can enter the metadata key-value pairs. There will be an authorization field that will allow the user to enter the authorization token, which will be used in the gRPC calls. This field will only define an extra metadata pair with a predefined key and value prefix.

The main part of the website consists of three sections: services, types, and enums. Each service name contains a full package path, documentation comment, and options, like *java_package*. Services are expandable, and when expanded, they show the methods. Each method contains the type (unary, client streaming, server streaming, or bidirectional streaming), name, and part of the documentation comment. Methods are expandable as well, and when expanded, they show more information, which I will describe in the following section.

The message types and enums sections are similar to the services section. Each message type or enum contains the full package path and documentation comment and can be expanded to show more information. Enums are prefixed with the *[Enum]* to distinguish them from message types, but otherwise, they share the same design.

4.6.3.1 Method Wireframe

The method wireframe is shown in the figure 4.3. It contains the method name, type, full documentation comment, and options. Then, it is divided into two sections. The first one contains the request data, and the second one contains the response data.

The request data section contains a table with all top-level fields and their respective names, types, and comments. All fields are empty by default because all fields in the gRPC request are optional by design. The only pre-filled fields are message types because the structures can be complex, and I assume they will be filled with at least some data. If not, they can be cleared by the user, which I consider to be easier than creating the structure from scratch. Additional options are shown for these fields, too. These options allow the user to preview an example input and the target message type model. The input of the message type is done using a JSON format, which is validated on the fly. The form itself is generated only for the top-level fields, and the example input is done only for one level deeper. This is an intentional design decision because the depth of nested message types can be infinite, but the user input always starts with the top-level one. Therefore, any nested message type is treated as being part of the top-level message type. This should allow quick input for common use cases, such as one string or number of inputs, but also be able to define complex structures using the JSON format for message types. For fields using oneof, repeated, or map, respective implementations, such as an array, are shown.

Before the request execution, all fields are validated, and the user is informed about any errors. Message type errors that do not conform to the message type structure are shown, too.

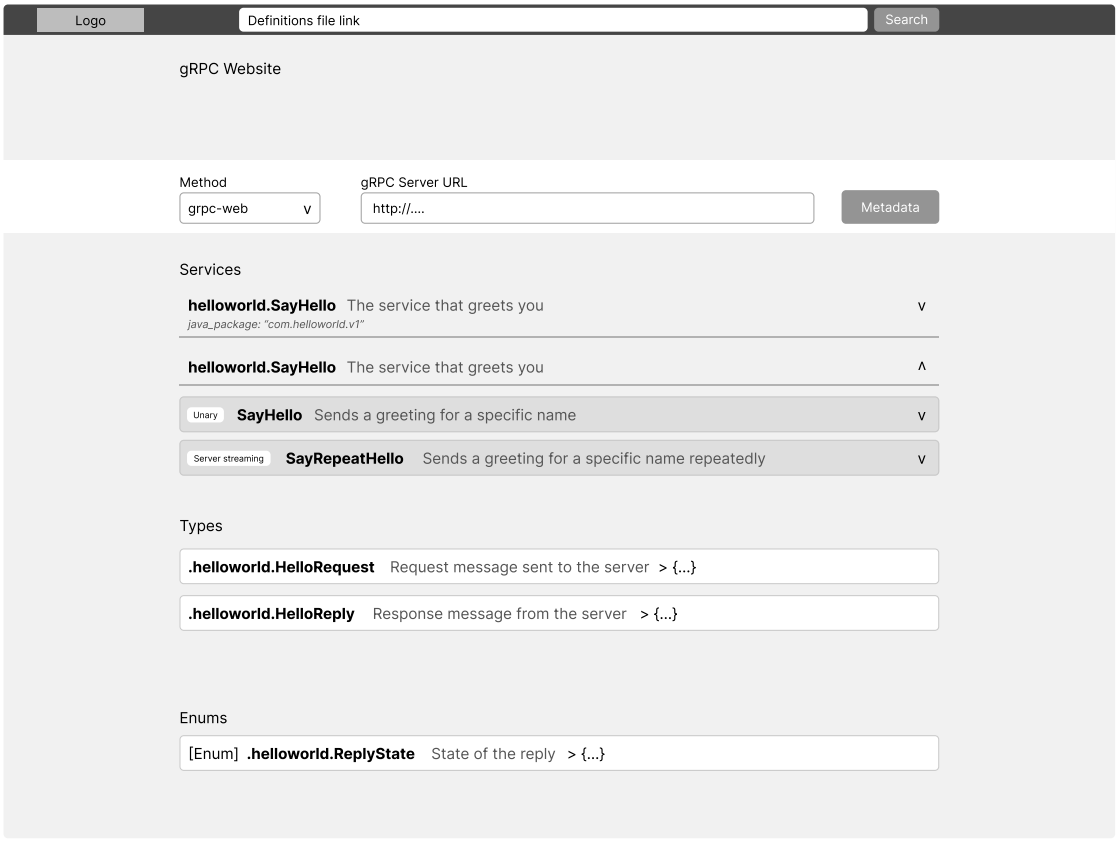


Figure 4.2 Main layout wireframe

This should allow the user to correct the errors and execute the request again.

The responses section contains content type selection, which is used mainly for the gRPC-Web and sets the content type of the communication, and four subsections:

- request JSON,
- request server,
- server responses,
- responses.

The content type selection place is based on the Swagger UI content type selection, hence part of the responses section. The request JSON section contains the request data in JSON format with metadata, which is sent to the server. This is useful for debugging purposes, as the user can see what is sent to the server in case the UI is not clear enough. The request server section contains the information about the request server to remind the user that this server is requested because its URL is set in the global settings at the top of the page. The server responses section contains information about the server responses, such as headers, status codes (if an error occurred), messages, and trailers. The messages are dynamically added in this section in the case of server streaming-type requests. The responses section contains an example response from the server in the case of success with the example message type. The message type is expandable and behaves the same as I will describe in the following message type detail section.

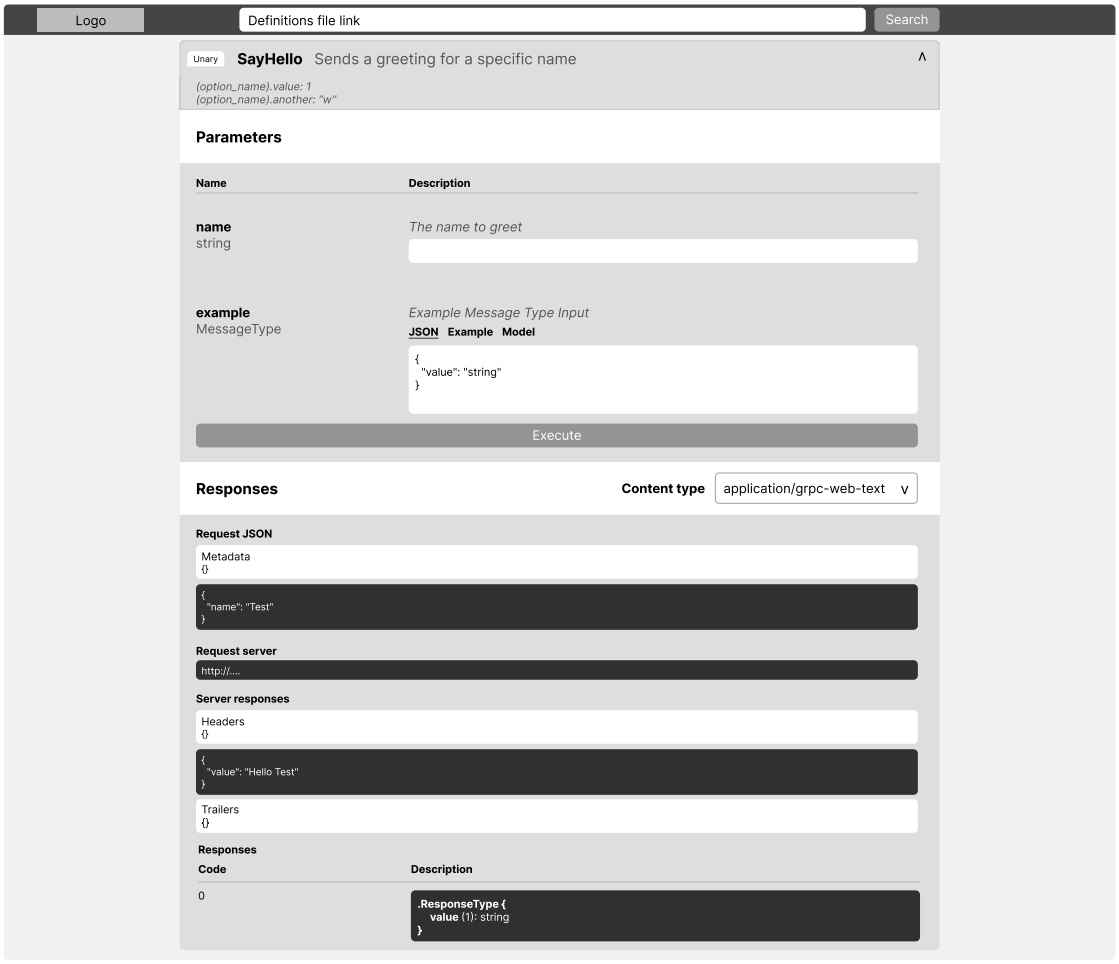


Figure 4.3 Method wireframe

4.6.3.2 Type and Enum Wireframe

The method wireframe is shown in the figure 4.4 and the enum wireframe in the figure 4.5. Both message types and enums contain the full package path with a name and a documentation comment and can be expanded to show more information. They also include options like *allow_alias* for enums.

For the message types, all fields are shown with their names, documentation comments, unique identifiers, and types. Also, fields that are part of oneof, repeated, or map are shown in their respective manner. If the field type is a message type, it is expandable as well, and it shows the same information as the message type itself. Recursive message types allow infinite expansion.

For the enums, all keys are shown with their values and value options. The value options can be used, for example, to mark the key as deprecated or define custom properties.

Both message types and enum types work the same when used in the method requests and response examples, and they are expandable on demand with the same design. The support for well-known types is achieved by traversing all imported dependencies and showing their definitions. They are shown in the same way as the user-defined types.

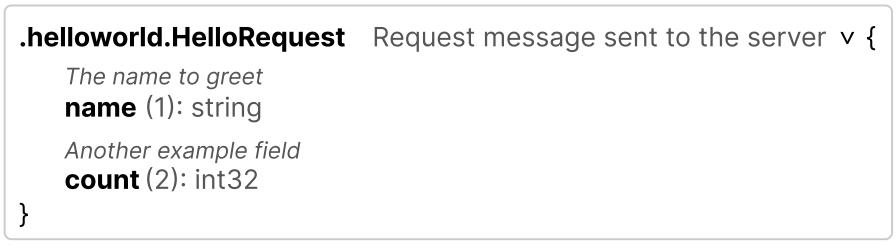


Figure 4.4 Type wireframe

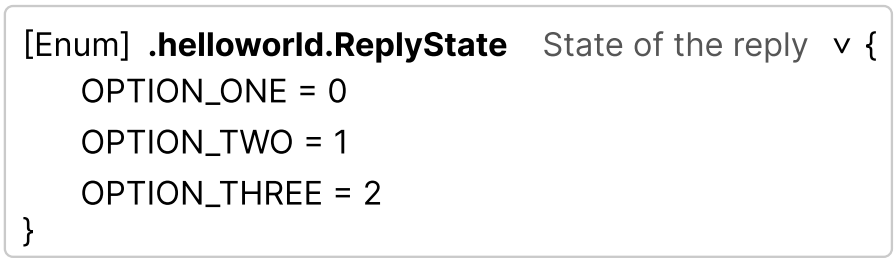


Figure 4.5 Enum wireframe

4.7 Fulfillment of Requirements

I have gone through all the requirements and added a description of how the website design fulfills them. It is shown in the table 4.1.

Based on the table, the website generator design fulfills all the requirements defined in the analysis.

	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13	F14	F15	F16	F17	F18	F19	F20	F21	N1	N2	N3
Main webpage design	x	x	x																					
Showing comments for the services, methods, message types, fields, and enums				x	x																			
Execution of methods using direct gRPC-Web implementation						x	x																	
Method design with inputs for all fields								x																
Form validation									x															
Global metadata settings, with the ability to set the authorization token										x										x				
Showing the response headers, messages as they come, and trailers											x	x												
Supporting oneof, map, repeated, and nested message types in the method input and in the message type													x	x	x	x								
Supporting all imported types, including well-known types, and showing them in the types section																	x							
Generator for the proto files																		x						
gRPC reflection generator, combined with grpcurl																			x					
Showing options for services, methods, message types, and enum values																					x			
Website is static and all content is generated on the fly																						x		
UI is inspired by the Swagger UI design, and it is adjusted to the gRPC specifics																							x	
Message request input is in a combination of form and JSON format																								x

Table 4.1 Fulfillment of requirements

Chapter 5

Implementation

In this chapter, I will describe the implementation of the website generator. I will start by discussing the choice of technology, followed by the project settings. Then, I will describe the code structure, user interface, and functionality. Finally, I will discuss the licensing of the libraries used in the project.

5.1 Choosing the Technology

The main factor in choosing the technology is the ability to generate a website that can be easily deployed and accessed by users. In the design chapter, I concluded that I would use a common format with all gRPC definitions, and the website would be generated from this format in the browser. Therefore, I must choose a technology for the website and the common format generators.

As for the website, the basics are done using HTML and CSS. There is no other choice. For the programming language, because of the dynamic rendering based on the common format, the only possibilities are JavaScript and WebAssembly. Because I will need to be updating the website (meaning the DOM), which is not supported directly by WebAssembly, I have chosen JavaScript [36].

JavaScript was initially designed to manage basic website interactions using simple scripts. The website will not be a simple script though, but many scripts with complex functions and classes. Complexity can lead to subtle bugs that are hard to track down in plain JavaScript. Adding type-checking ensures that errors are caught at the development stage, reducing bugs and improving code quality. Additionally, type annotations make the code more readable and easier to understand, which is crucial when collaborating with others or for future modifications. [37]

Based on the State of JavaScript 2022 survey [38], the most popular language flavor of JavaScript is Typescript[1]. Because of the popularity, the benefits of type-checking, and the readability of the code, I have decided to use TypeScript for the website.

The browser does not limit the common format generator language selection because the generators are run on the developer's machine. Therefore, the language choice is based on the required libraries for the generators. But, because I will be using the protobufjs JavaScript library, I will use JavaScript for the generators.

[1] `https://www.typescriptlang.org/`

5.1.1 Web Framework

To create the website with extensive logic, I will use a front-end framework. The most popular options for the web framework based on the State of Javascript 2022 survey are React[2], Angular[3], Vue.js[4], and Svelte[5] [39]. I want the page to exist for a long time and be maintained. For this reason, I will choose the most popular framework, which is React.

React is a JavaScript library for building user interfaces. It is maintained by Meta and a community of individual developers and companies. React can be used as a base in the development of single-page web applications. It allows developers to create large web applications that can change data without reloading the page, making the website faster. [40]

Using React is a great option, but it requires a lot of manual setup (routing, code splitting, and more). For this reason, there is a React framework called Next.js[6]. It simplifies the setup, development, static site page generation, routing, and a lot more [41]. It is also the first recommended way to build a new React application by the React team [42]. Therefore, I will use Next.js for the website. The key features, except the setup of Next.js, that I will use are static site generation and TypeScript support.

5.1.2 Styling Libraries

Instead of using my own CSS classes to style the website, I will use a framework to speed up development. The most popular CSS frameworks based on the State of CSS 2023 survey are Bootstrap[7], Tailwind CSS[8], and Materialize CSS[9] [43]. All of them are good options, but as for choosing the front-end framework, I will choose the most popular one, which is Bootstrap.

5.1.3 Protobufjs Library

Based on the design chapter, the protobufjs library is used to serialize and deserialize the data in a common JSON format. The library also offers a tool called protobufjs-cli[10]. It is a command-line tool that can be used to generate the JSON format from the proto files [44]. It has an issue, though. It only allows individual proto files as input, not the entire folder, which is in the earlier defined use case 3.4.1. Also, the correct parameters need to be specified in order to get the correct output. For this reason, I will use the protobufjs-cli, but only as a library. This will allow me to have potential features in the future, fix the current ones, and seal the required parameters.

There are two more issues with this library that need to be addressed. The first one is that there is a bug in the library when parsing the value options for enums. I found the issue in the library's repository source code and fixed it locally using the package manager patch functionality. I have also reported the issue to the library's repository at `https://github.com/protobufjs/protobuf.js/issues/1961` with a way to fix it and created a pull request. The patch change is in the code snippet 5.1.

[2] `https://react.dev/`
[3] `https://angular.io/`
[4] `https://vuejs.org/`
[5] `https://svelte.dev/`
[6] `https://nextjs.org/`
[7] `https://getbootstrap.com/`
[8] `https://tailwindcss.com/`
[9] `https://materializecss.com/`
[10] `https://www.npmjs.com/package/protobufjs-cli`

Code listing 5.1 protobufjs library enum comments bug fix

```
Enum.fromJSON = function fromJSON(name, json) {
    // This line is removed
    var enm = new Enum(name, json.values, json.options, json.comment, json.comments);
    // This line is added
    var enm = new Enum(name, json.values, json.options, json.comment, json.comments,
    json.valuesOptions);
    enm.reserved = json.reserved;
    return enm;
};
```

The second issue is that the protobufjs-cli library does not support comments. I have found an already existing issue (`https://github.com/protobufjs/protobuf.js/issues/1145`), which hints at how to patch the library locally until it is added to the library itself. To make the protobufjs-cli library support comments, I have slightly updated how JSON exports work and created a local patch. When this feature is added to the library, I can remove the patch and use the library as it is. The patch change is in the code snippet 5.2.

Code listing 5.2 protobufjs-cli comments support

```
function json_target(root, options, callback) {
    // This line is removed
    callback(null, JSON.stringify(root, null, 2));
    // Theis line is added
    callback(null, JSON.stringify(root.toJSON({ keepComments: true }), null, 2));
}
```

5.1.4 gRPC-Web Client Library

The gRPC-Web library only offers client-side code generation from the proto files (stubs). However, it does not allow the client to make the requests only with the possibility of implementing the data serialization and deserialization. I have tried to find a library offering this functionality, but I have not found any. For this reason, I have decided to analyze how the stubs are generated in order to extract the gRPC-Web client. I already know that I can serialize and deserialize the data using the protobufjs library, which I will use for that.

There are two generated gRPC-Web client files. One generates the client, and the other generates the message types and methods. Note that the client is still specific to the proto files.

Based on the generated code, I have found that the actual call is done using initialization of *GrpcWebClientBase* and then calling *rpcCall* method for unary requests and *serverStreaming* method for server streaming requests. The client initialization does not require any parameters but can be supplied with the option object, which may contain a format (binary or text). The *rpcCall* method requires a method name, a request message type object, a metadata of the request, a methodDescriptor (more on that later), and a callback. The *rpcCall* function returns a stream, which may be used for additional headers or trailer handling. The *serverStreaming* method requires the same parameters, but the callback is not present, and responses are handled only using the returned stream.

The code snippet 5.3 shows an example of the method with parameters. The method name is a string with a full path to the method. The request message type is defined by the generated client and is required to contain specific attributes and methods. Because I am using the protobufjs library, I have found out I am able to supply only an empty object ($\{\}$) there and do the rest of the work in other functions supplied in the method descriptor.

The method descriptor is an object containing the method name, the method type, the request message type, the response message type, and the request serialize and response deserialize

functions. The method name is just the method name without a path. The method type is either unary or server streaming. Because I am using the protobufjs library, a fake class with a constructor can replace the request and response message types. Finally, the request serialize, and response deserialize functions are functions that take the message object and return the serialized or deserialized message. Because I have replaced the request object with an empty object, the request serialize function is not supplied with the message. However, I can directly access the variable from the outer scope using the protobufjs library and return the encoded message. For the response deserialize function, I can use the protobufjs library again to decode the available message, this time as a byte's array parameter, and return it. The callback function or the stream receives the deserialized message in the protobufjs format.

Code listing 5.3 gRPC-Web extracted client unary call example

```
const client = new GrpcWebClientBase({ format: options?.format });

const unaryStream = client.rpcCall(
    methodPath,
    // Ignored, using protobufjs directly
    {},
    options?.metadata ?? {},
    new MethodDescriptor(
        method.name,
        MethodType.UNARY,
        // Ignored, using protobufjs directly
        DummyRPCType,
        // Ignored, using protobufjs directly
        DummyRPCType,
        () => {
            return typeEncode.encode(message).finish();
        },
        (bytes: Uint8Array) => {
            return typeDecode.decode(bytes);
        },
    ),
    (err, response: protobuf.Message<MessageData>) => {
        if (err) {
            reject(err);
        } else {
            completeResponse.data = [response];
        }
    },
);
```

I have extracted the gRPC-Web client from the generated stubs, and I can use it on the website for any request. The gRPC-Web library still handles the communication with the server, though, so any future updates to the library should be automatically applied to the client as well.

5.1.5 Other Libraries

Other essential libraries that I will use are react-hook-form[11] for creating the request input form validation, yup[12] for the schema validation (used in connection with forms), and fontawesome[13]

[11] https://react-hook-form.com/
[12] https://github.com/jquense/yup
[13] https://fontawesome.com/

for the icons. These libraries have the features I need from them, have over a million weekly downloads (which is significant compared to other NPM packages), and are maintained. Again, I am choosing the most popular and maintained libraries for the long-term existence of the static documentation website.

5.2 Project Settings

The project is using the Lerna[14] and pNpM[15] package manager. Lerna is a tool that optimizes the workflow around managing multi-package repositories [45]. And pNpM is a fast, disk-space efficient package manager with the support of workspaces and libraries patching [46]. Both tools cooperate and are compatible with each other.

The project is set up as a monorepo. It is divided into three packages:

- **proto-to-json** – contains the generator from proto files,
- **reflection-to-json** – contains the generator from the reflection file,
- **web** – contains the website.

The project also contains *example* and *patches* folders.

The *example* folder contains example proto and reflection files with a pre-generated common format, as well as Envoy proxy configuration and gRPC server implementation in JavaScript. The Envoy proxy can then be started using a Docker image *envoyproxy/envoy* and the gRPC server using the *node server.js* command. This setup can be used to test the website's functionality.

The *patches* folder contains the patches for the protobufjs library described in the 5.1.4. They are split into two files, each containing the diff of a concrete package file. These files are generated using the *pnpm patch <pkg name>* command and applied automatically when the package is installed.

The generators from proto and reflection files are set up as NPM runnable packages. This means the packages can be installed globally and run from the command line.

The website is set up as a Next.js project using the command *npx create-next-app@latest*. It uses an App directory for routing, TypeScript as the language, ESLint[16] for linting, Jest[17] for testing, and Prettier[18] for code formatting. The website is set up to be statically generated using the *output: "export"* option, which means that the website is generated at build time and served as static files. For building, it uses the Next.js Compiler, which is run using the *pnpm build* command.

5.3 JSON from Proto Files Generator

The generator from proto files is a command-line tool that generates the common JSON format. It takes one or more proto files or a directory with proto files as input and outputs the JSON format to the console. The script uses the proto files from the command or recursively traverses the directory for all proto files, which are then passed to the protobufjs-cli library with correct arguments and writes output to the console. The output can be then redirected to a file.

The code snippet 5.4 shows an example of the command. The symbol > redirects the output to a file. The *SOURCE_PROTO_FILES* can be any number of proto files or folder combinations split by space. If invalid proto files are passed, an error will be thrown, which is then displayed in the console.

[14] `https://lerna.js.org/`
[15] `https://pnpm.io/`
[16] `https://eslint.org/`
[17] `https://jestjs.io/`
[18] `https://prettier.io/`

Code listing 5.4 proto-to-json command example

```
gf-proto-to-json ${SOURCE_PROTO_FILES} > ${EXPORTED_NAME}.json
```

The generated JSON output can then be used on the website.

5.4 JSON from gRPC Reflection Generator

The generator from the reflection file is a command-line tool that generates the common JSON format. The script uses the reflection bin file from the command, which is then processed by the protobufjs library and serialized to the JSON format. The output can be then redirected to a file.

The bin format can be generated using the grpcurl tool. The example command is shown in the code snippet 5.5. The *BIN_FILE* is the target reflection bin file, and *GRPC_SERVER* is the gRPC server address. The *-protoset-out* is used to output the reflection to the file, and the *describe* tells the grpcurl to output the reflection definitions. Other parameters and features of the grpcurl tool can also be used (e.g., TLS settings, filtering only parts of the reflection interface, etc.). This is just an example. Also, there may be other tools for creating the reflection bin file. There is no limitation on the tool used as long as the bin file is in the correct format.

Code listing 5.5 proto-to-json command example

```
grpcurl -protoset-out ${BIN_FILE}.bin -plaintext ${GRPC_SERVER} describe
```

The code snippet 5.6 shows an example of the command. The symbol > redirects the output to a file. The *SOURCE_BIN_FILE* is the bin file. If an invalid bin file is passed, an error will be thrown, which will then be displayed in the console.

Code listing 5.6 proto-to-json command example

```
gf-reflection-to-json ${SOURCE_BIN_FILE} > ${EXPORTED_NAME}.json
```

The generated JSON output can then be used on the website.

5.5 Static Website

The website is static and generated using the Next.js framework. The code structure is divided into:

- **src/app** – contains the pages of the website,
- **src/components** – contains the React components,
- **src/contexts** – contains the React contexts,
- **src/scss** – contains the global styles of the website,
- **src/services** – contains the functions with logic for the website,
- **src/types** – helper types and constant definition,
- **public** – contains the public files (e.g., images).

After the website is built, the output is in the *out* directory. The directory contains static HTML, CSS, JavaScript files, and all files from the public directory. It can then be hosted by any web server that supports static file hosting. It is enough to copy only the contents of the *out* directory to the server. The default definitions file is in the root of the directory and is called *definitions.json*. It is shown on the website page by default. Any other file can replace it with the same structure and name, or it is possible to host the file anywhere and specify the file URL in the *url* query parameter of the website.

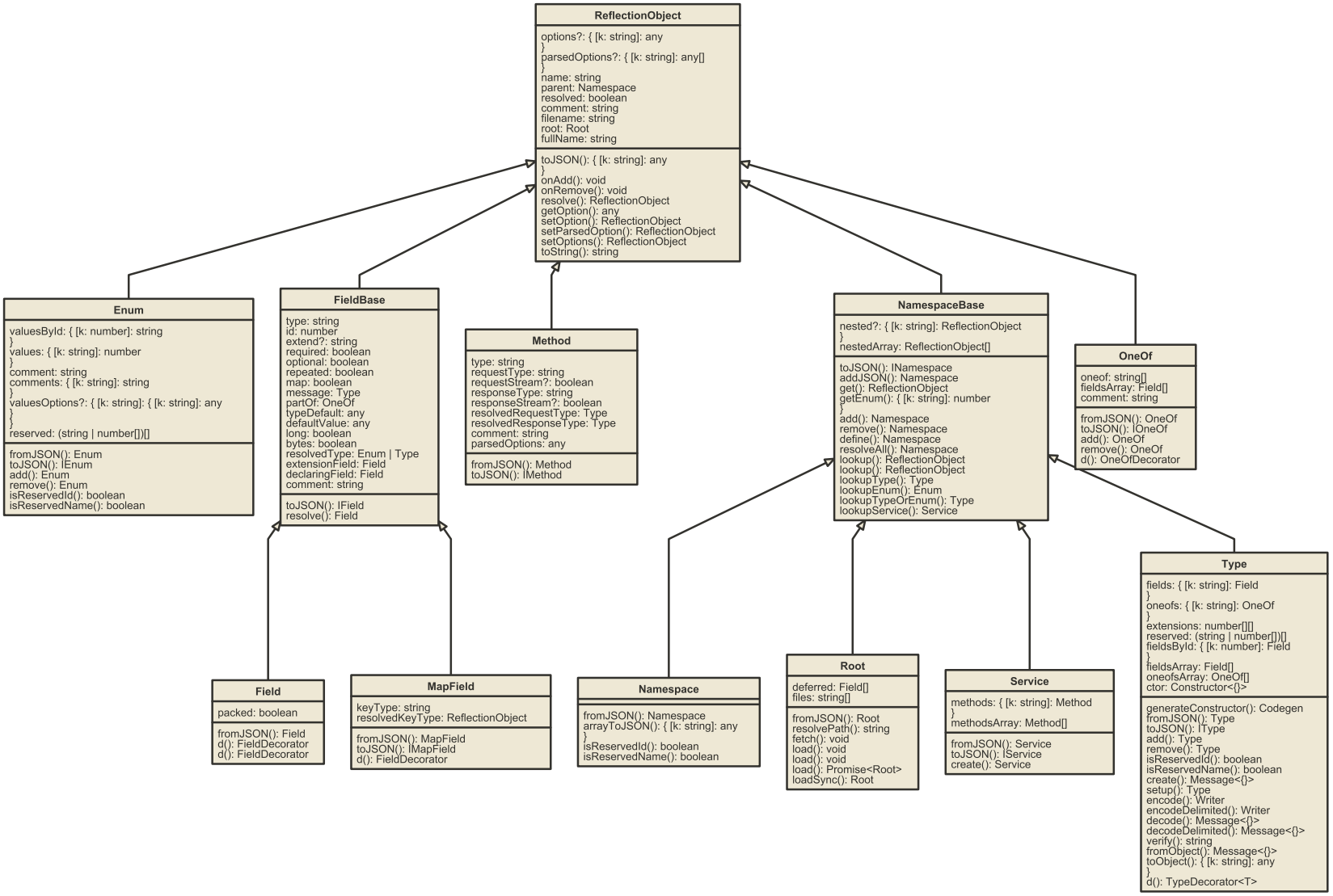


Figure 5.1 Protobufjs class diagram [35]

5.5.1 Protobufjs Data Structure

I am using the protobufjs library with its data structure. The data class diagram is shown in the figure 5.1.

The base is the *Root* class, which is returned by parsing the common format. Other classes like service or type create a tree structure. For example, using the field *nested* (or *nestedArray*) from the *NamespaceBase*, I can traverse the tree and find all services, message types, or enums. This way, I find all the data I need for the website.

Each class has fields related to the protobuf features. For example, the *Service* class has a *methods* field containing all methods for that service, and the *Method* class has a field of what type of streaming it is. Also, all classes have a *comment* field containing the proto file's comment. Using these fields, I can display the data on the website.

5.5.2 Design and Functionality

The website design starts from the wireframes in the design chapter. The whole page is in the figure 5.2. At the top, the toolbar contains the URL input for definitions JSON files by default. If the user clicks on the URL dropdown button, they can change the input to a file input (shown in the figure 5.3). The file can then be a JSON definition file from the local machine or the gRPC reflection bin file. The file is then parsed, and the data is displayed on the website.

Under the toolbar is a short description of the website's purpose. Under the description is a section with global settings. The first is the selection of the backend method, which is, by default, the gRPC-Web. This is a place where it is possible to add more backends in the future, which may have full implementation of client streaming. If the gRPC-Web library eventually has client streaming support, this dropdown can be easily removed.

Next to the method selection is a backend gRPC server selection. The user can change the server address and port. This is a global setting for the website and is used for all requests.

Next is the global metadata definition. Upon clicking the button, a modal dialog, figure 5.4, is shown. The user can then define the metadata as key-value pairs for the request. It also allows defining the authorization token, effectively creating a new metadata entry with a predefined format.

The list of services is under the global settings. The services are displayed as described in the design chapter. In the figure 5.2, the user can also see the difference between the unary and server streaming methods, as well as service options and comments.

The expanded method is shown in the figure 5.5. Its design is based on the wireframes, where the user can see the request input form and the response message type. After executing the method, the response is shown as in the figure 5.6. It contains the request JSON with the metadata, server URL, and the response with headers and trailers. When the method type is server streaming, the response is shown as in the figure 5.7 with all responses gradually showing as soon as they arrive. If the user wants to cancel the method execution while it is still running, they can click the cancel button, which is shown in the figure 5.8. If there is an error in the response, it is shown as in the figure 5.9 with the gRPC error code status, as well as the HTTP status (for gRPC-Web) and metadata.

The input of a method is as a form. Different input field types are shown in the figure 5.10. There is special handling for bytes input (file selection), strings, numbers, enums (dropdown selection), booleans (dropdown selection), oneof the fields grouping, and JSON input fields. The JSON input fields are shown for repeated (arrays), maps, and messages fields. Each JSON input contains also a tab with the JSON schema example and the interactive message type model (with expandable nested message types). The same example and model also apply for the enum types, but the input is a dropdown selection (if it is not part of repeated). Each field has a validation based on the schema definition. The examples are shown in the figure 5.11. The errors are

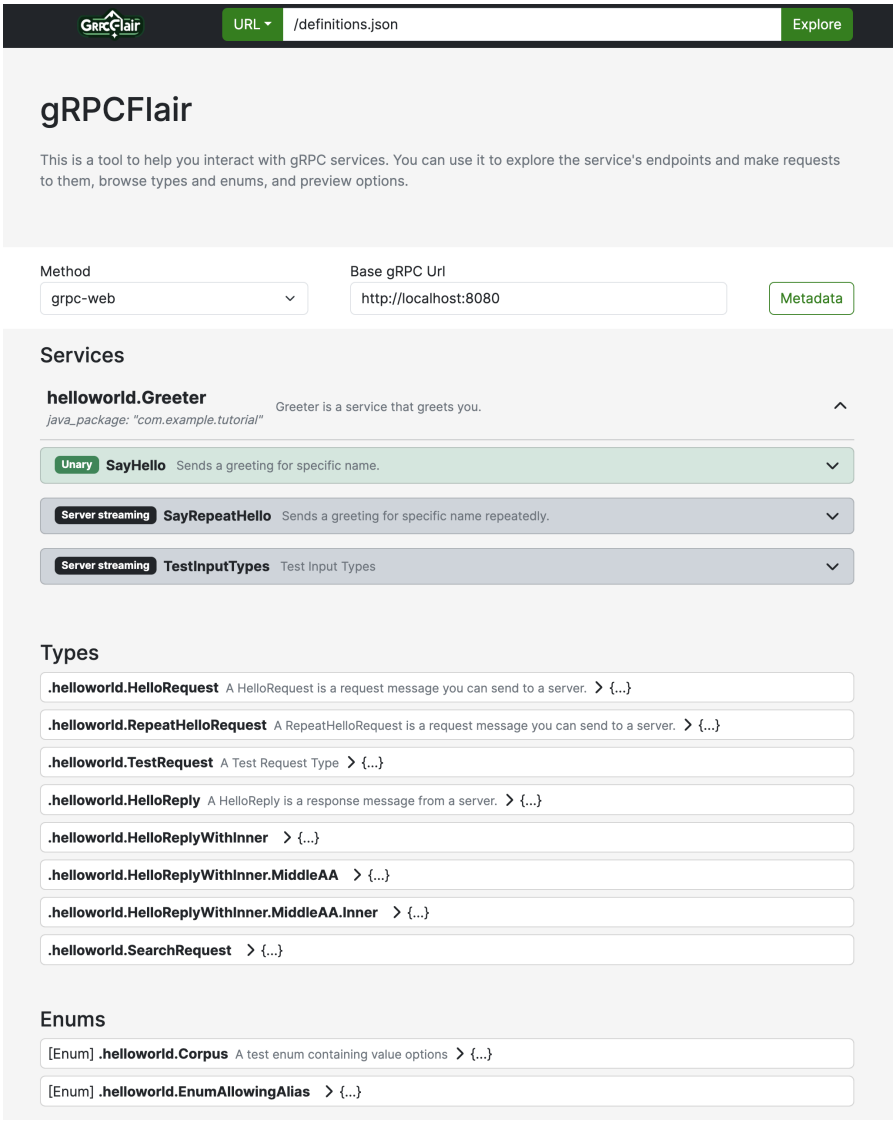


Figure 5.2 Website overview

Figure 5.3 Input using file

shown after execution under the field with the error message and above the execution button. The execution button is disabled until all fields are valid.

Message types are the next section on a website. The expanded message type is shown in the figure 5.12a. It shows the documentation comments and options for the message type itself, as well as for the fields. All types of fields are shown, including repeated (as an array), maps, and oneof the fields. Green text with underlining indicates an expandable message type. The expanded message type is shown in the figure 5.12b. If the expanded message type contains another nested message type, it is also expandable. Each expansion rendering is done after clicking on the field, which prevents issues with recursive types.

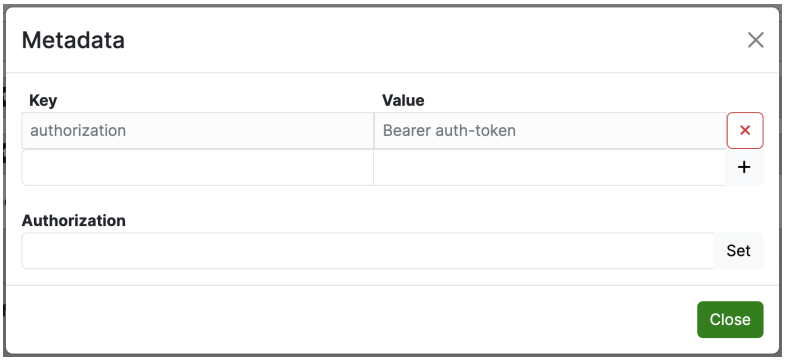


Figure 5.4 Metadata modal dialog

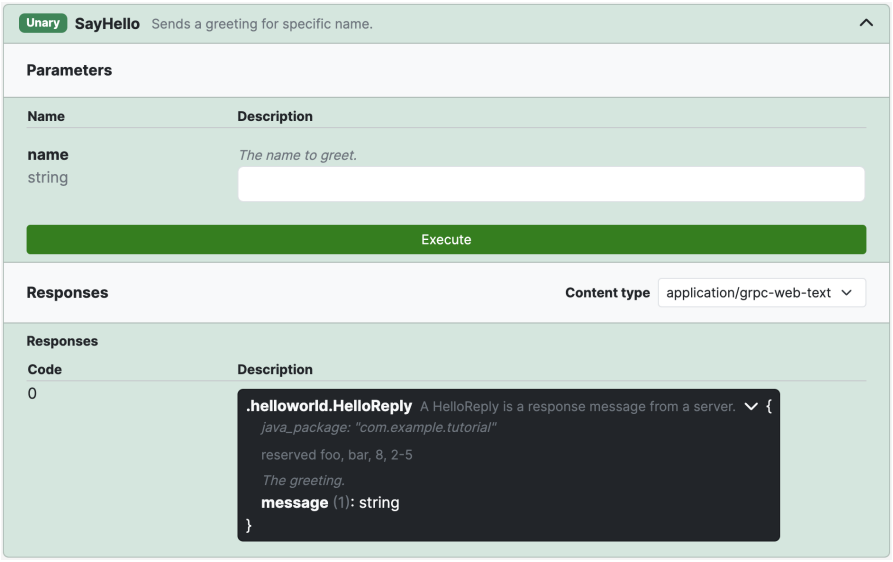


Figure 5.5 Method overview

The website ends with enums. The expanded enum is shown in the figure 5.12c. It shows the documentation comments and options for the enum itself, as well as for the values. The values are shown as keys with their respective numbers. The enum value options are shown in brackets after the value name, copying the protobuf syntax.

All the functionality and design allow the user to navigate through the proto file services, methods, message types, and enums and see the documentation comments. When the user selects a method, they can execute it with the input form and see the response. For streaming requests, the user can see the responses gradually. They can see the error message for errors. This way, the user can test the gRPC server without implementing any client and preview documentation.

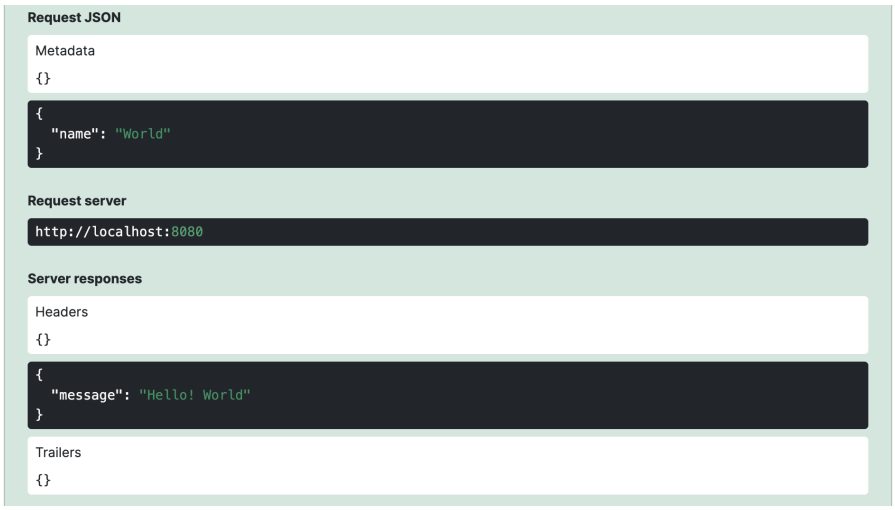


Figure 5.6 Method execution response

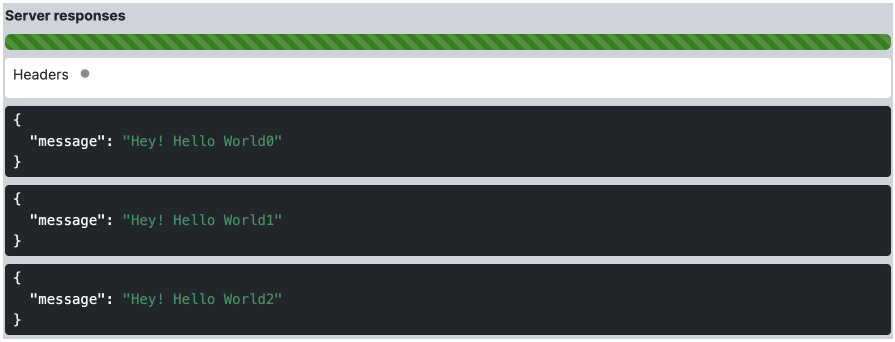


Figure 5.7 Method execution response – server streaming

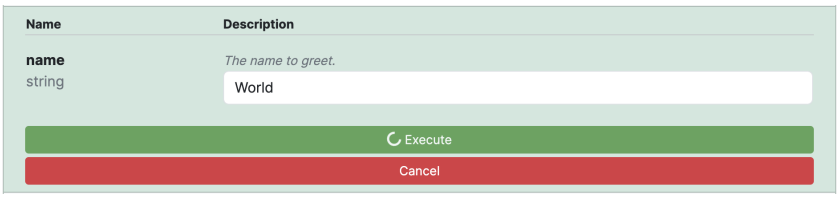


Figure 5.8 Method execution pending state

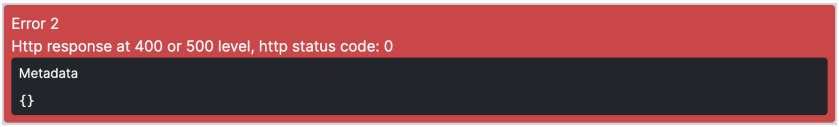
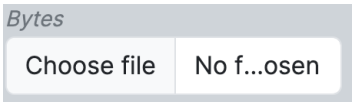
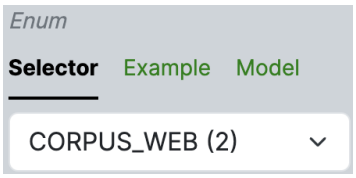


Figure 5.9 Method execution response error



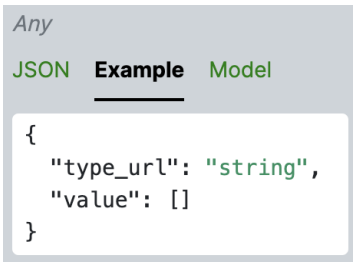
(a) Bytes

(b) Number

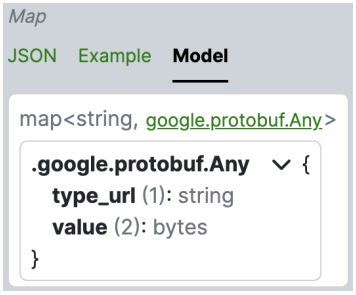


(c) Enum

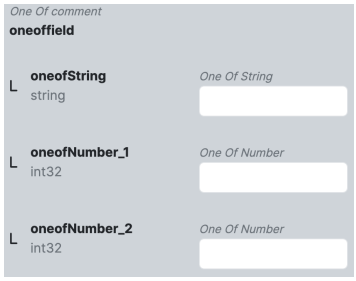
(d) JSON (message type, repeated, map)



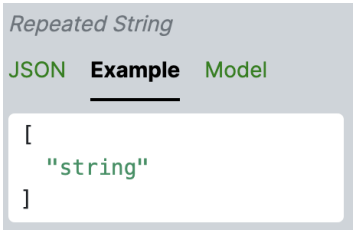
(e) JSON - example



(f) Model

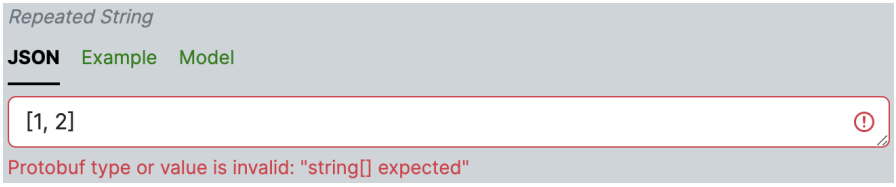


(g) Oneof

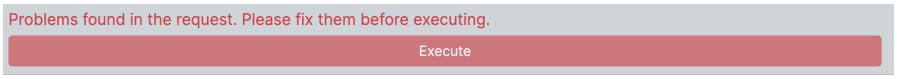


(h) Repeated - example

Figure 5.10 Method execution input fields

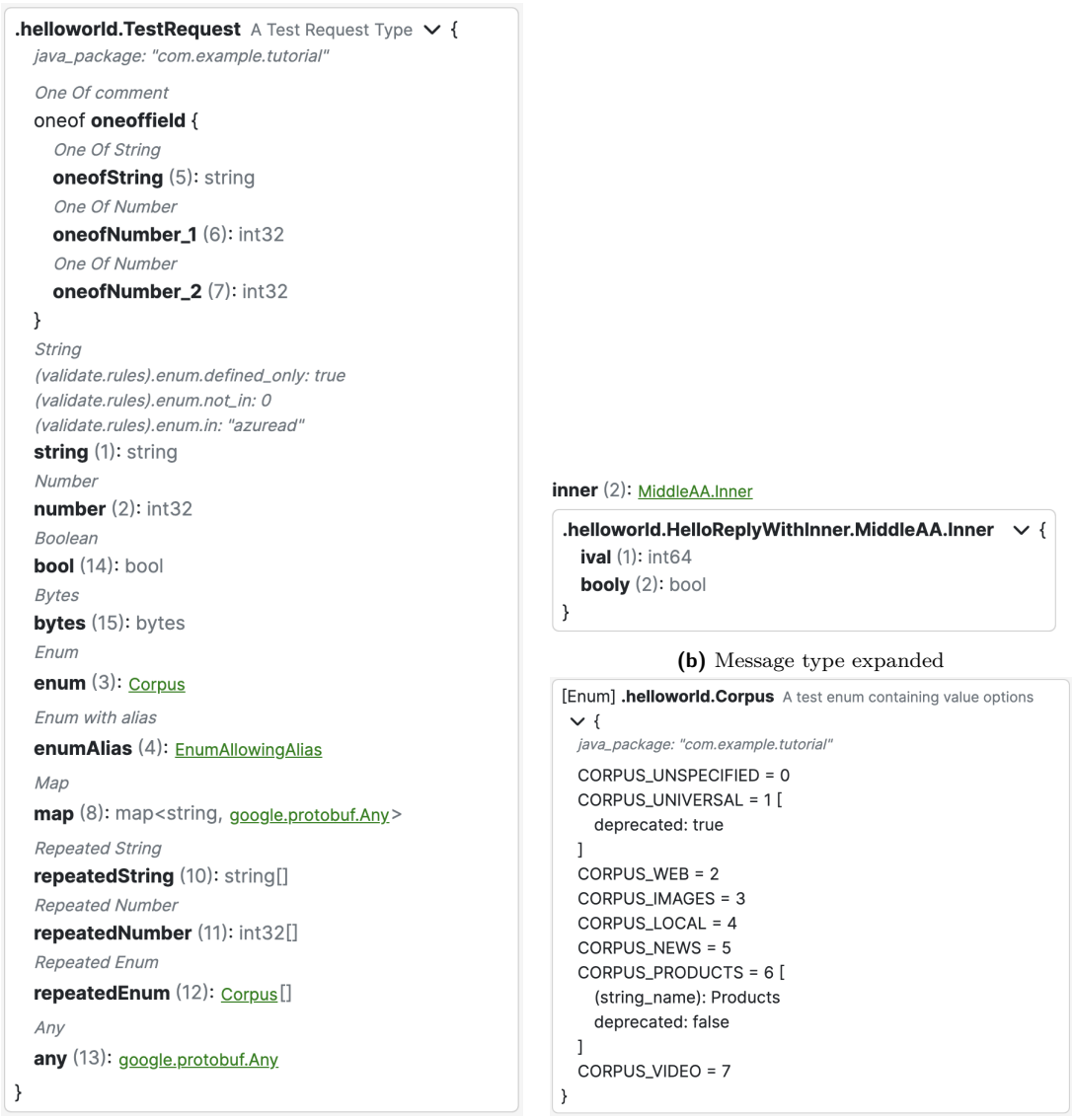


(a) Field



(b) Button

Figure 5.11 Input validation



(a) Message type overview

(b) Message type expanded

(c) Enum overview

Figure 5.12 Message type and enum type

5.6 Licensing

For the implementation, I have used libraries that use the following licenses:

- MIT[19],
- BSD-3-Clause[20],
- Apache 2.0[21],
- ISC[22],
- CC-BY-4.0[23].

The list of libraries and their licenses is captured in tables 5.2 and 5.3. The rights and limitations of these licenses are then shown in table 5.1.

	MIT	**BSD-3-Clause**	**Apache 2.0**	**CC-BY-4.0**	**ISC**
Permissions					
Commercial use	✓	✓	✓	✓	✓
Modification	✓	✓	✓	✓	✓
Distribution	✓	✓	✓	✓	✓
Patent use	-	-	✓	x	-
Private use	✓	✓	✓	✓	✓
Conditions					
License and copyright notice	✓	✓	✓	✓	✓
State changes	-	-	✓	✓	-
Limitations					
Trademark use	-	-	x	x	-
Liability	x	x	x	x	✓
Warranty	x	x	x	x	✓

Table 5.1 Overview of licenses and their limitations

All licenses allow private and commercial use, including distribution and possible modifications. The only requirement is to state changes if made (I have not made any changes to the libraries that require it).

Because I have met the requirements of the licenses and their limitations, which allow me to use the libraries for free, I can use them for my work.

[19] https://choosealicense.com/licenses/mit/
[20] https://opensource.org/license/bsd-3-clause
[21] https://choosealicense.com/licenses/apache-2.0/
[22] https://www.isc.org/licenses/
[23] https://creativecommons.org/licenses/by/4.0/

Library	License
@fortawesome/fontawesome-svg-core[24]	MIT
@fortawesome/free-regular-svg-icons[25]	CC-BY-4.0, MIT
@fortawesome/free-solid-svg-icons[26]	CC-BY-4.0, MIT
@fortawesome/react-fontawesome[27]	MIT
@hookform/resolvers[28]	MIT
bootstrap[29]	MIT
grpc-web[30]	Apache-2.0
lodash[31]	MIT
next[32]	MIT
p-cancelable[33]	MIT
protobufjs[34]	BSD-3-Clause
protobufjs-cli[35]	BSD-3-Clause
react[36]	MIT
react-bootstrap[37]	MIT
react-dom[38]	MIT
react-hook-form[39]	MIT
react-syntax-highlighter[40]	MIT
sass[41]	MIT
yup[42]	MIT

Table 5.2 List of libraries and their licenses

[24] https://www.npmjs.com/package/@fortawesome/fontawesome-svg-core
[25] https://www.npmjs.com/package/@fortawesome/free-regular-svg-icons
[26] https://www.npmjs.com/package/@fortawesome/free-solid-svg-icons
[27] https://www.npmjs.com/package/@fortawesome/react-fontawesome
[28] https://www.npmjs.com/package/@hookform/resolvers
[29] https://www.npmjs.com/package/bootstrap
[30] https://www.npmjs.com/package/grpc-web
[31] https://www.npmjs.com/package/lodash
[32] https://www.npmjs.com/package/next
[33] https://www.npmjs.com/package/p-cancelable
[34] https://www.npmjs.com/package/protobufjs
[35] https://www.npmjs.com/package/protobufjs-cli
[36] https://www.npmjs.com/package/react
[37] https://www.npmjs.com/package/react-bootstrap
[38] https://www.npmjs.com/package/react-dom
[39] https://www.npmjs.com/package/react-hook-form
[40] https://www.npmjs.com/package/react-syntax-highlighter
[41] https://www.npmjs.com/package/sass
[42] https://www.npmjs.com/package/yup

Development Library	License
@testing-library/jest-dom[43]	MIT
@testing-library/react[44]	MIT
@types/jest[45]	MIT
@types/lodash[46]	MIT
@types/node[47]	MIT
@types/react[48]	MIT
@types/react-dom[49]	MIT
@types/react-syntax-highlighter[50]	MIT
cross-env[51]	MIT
eslint[52]	MIT
eslint-config-next[53]	MIT
eslint-config-prettier[54]	MIT
jest[55]	MIT
jest-environment-jsdom[56]	MIT
prettier[57]	MIT
typescript[58]	Apache-2.0
lerna[59]	MIT
rimraf[60]	ISC

Table 5.3 List of development libraries and their licenses

[43] https://www.npmjs.com/package/@testing-library/jest-dom
[44] https://www.npmjs.com/package/@testing-library/react
[45] https://www.npmjs.com/package/@types/jest
[46] https://www.npmjs.com/package/@types/lodash
[47] https://www.npmjs.com/package/@types/node
[48] https://www.npmjs.com/package/@types/react
[49] https://www.npmjs.com/package/@types/react-dom
[50] https://www.npmjs.com/package/@types/react-syntax-highlighter
[51] https://www.npmjs.com/package/cross-env
[52] https://www.npmjs.com/package/eslint
[53] https://www.npmjs.com/package/eslint-config-next
[54] https://www.npmjs.com/package/eslint-config-prettier
[55] https://www.npmjs.com/package/jest
[56] https://www.npmjs.com/package/jest-environment-jsdom
[57] https://www.npmjs.com/package/prettier
[58] https://www.npmjs.com/package/typescript
[59] https://www.npmjs.com/package/lerna
[60] https://www.npmjs.com/package/rimraf

Chapter 6
Testing

For the website testing, I have prepared automated unit tests and manual scenarios. The automated tests are written in the TypeScript programming language using the Jest [1] framework. The manual scenarios serve as a guide for the user testing of the website and for verifying the correctness of the website's behavior.

6.1 Automated Testing

Using the Jest framework, I have created unit tests for the most critical parts of the website. The tests cover the services and components of the website. They are located in the `__tests__` directory. The test coverage statistics are in the table 6.1.

	Statements	Branches	Functions	Lines
Coverage	95.07%	95.65%	89.71%	96.36%

Table 6.1 Test coverage statistics

6.2 Manual Scenarios

For the manual user testing of the website, I have created scenarios that cover all use cases. Each scenario consists of:

- initial state,
- steps,
- covered use cases.

In the individual steps, I describe how to achieve the result defined in the title of the testing scenario.

[1] `https://jestjs.io/`

6.2.1 T1 – Generating the website from proto files and validating the data

Initial state: A folder with proto files
Steps:

1. Open the terminal
2. Write a command for translating proto files to JSON *gf-proto-to-json ./ > common.json*
3. Open the website (locally or hosted), change the input to file, and upload the JSON file
4. Check that the data is correctly displayed, including the services, methods, and messages
5. Check that the comments and options are present

Use cases covered: UC1, UC3, UC4, UC5, UC6, UC7

6.2.2 T2 – Generating the website from the gRPC reflection and validating the data

Initial state: A gRPC server URL with reflection enabled
Steps:

1. Open the terminal
2. Write a command for getting the bin file from the gRPC server *grpcurl -protoset-out descriptors.bin -plaintext ${GRPC_SERVER_URL} describe*
3. Write a command for translating the bin descriptors file to JSON *gf-reflection-to-json descriptors.bin > common.json*
4. Open the website (locally or hosted), change the input to file, and upload the bin file
5. Check that the data is correctly displayed, including the services, methods, and messages
6. Check that the comments and options are present

Use cases covered: UC2, UC3, UC4, UC5, UC6, UC7

6.2.3 T3 – Executing unary request

Initial state: Hosted website with loaded definitions and gRPC server running with the Envoy proxy
Steps:

1. Open the website with definitions loaded
2. Open the unary request
3. Fill in the required fields
4. Click on the “Execute” button
5. Check that the response is displayed with headers and trailers

Use cases covered: UC8

6.2.4 T4 – Executing server streaming request

Initial state: Hosted website with loaded definitions and gRPC server running with the Envoy proxy
Steps:

1. Open the website with definitions loaded
2. Open the server streaming request
3. Fill in the required fields
4. Click on the “Execute” button
5. Check that the responses are being displayed as they arrive
6. Check that the headers and trailers are displayed

Use cases covered: UC9

6.2.5 T5 – Setting global metadata

Initial state: Hosted website with loaded definitions and gRPC server running with the Envoy proxy
Steps:

1. Open the website with definitions loaded
2. Click the metadata button
3. Set any key-value metadata pair
4. Set the authorization token
5. Close the metadata dialog
6. Open any unary request
7. Fill in the required fields
8. Click on the “Execute” button
9. Check that the request contains the set metadata

Use cases covered: UC10

6.3 User Testing

My goal for the user testing is to determine the website’s intuitiveness and clarity and identify any issues. As for the target user group, I have chosen developers.

I have created the following scenarios for the user testing:

- Common Format Generation from Proto Files,
- List Services, Methods, Message Types, and Enum Types,
- Comments and Options,
- Execute Unary Request,

- Execute Server Streaming Request,
- Complex Method Input,
- Global Metadata.

The scenarios are different from those for testing the functionality of the website and do not cover the entire functionality because the primary goal of user testing is the intuitiveness and clarity of the website.

Each user will be briefed about the website's general use case, asked about their experience with similar pieces of software, and then they will be asked to complete the scenarios and provide feedback on the website's functionality and clarity.

The pre-questionnaire will be used to gather information about the user's experience with similar software. The questions are:

- What is your main focus in programming? (software, hardware, web development, etc.)
- What is your experience with REST APIs or GraphQL?
- What is your experience with Swagger UI, GraphiQL, or similar tools?
- What is your experience with the gRPC?

The post-questionnaire will be used to gather feedback on the website's functionality and clarity. The questions are:

- How did you find the common format generation process? (rate 1–5, 1 being the best)
- How did you find the overall website design? (rate 1–5, 1 being the best)
- How difficult was it to find the methods, message types, and enums? (rate 1–5, 1 being the easiest)
- How difficult was it to control the method execution? (rate 1–5, 1 being the easiest)
- Is there anything else to add?

6.3.1 Common Format Generation from Proto Files

The tester is provided with proto files and a script for generating a common format with its documentation.

The following instructions are given:
Based on the provided proto files, generate a common format that will be used later on for the documentation website.

The expected result is:
A generated common format JSON file.

6.3.2 List Services, Methods, Message Types, and Enum Types

The tester is provided with the common format file. The website is running on localhost on port 3000.

The following instructions are given:
You have a website running on the localhost on port 3000. List all services, methods, message types, and enum types from the common format file. Find the method `SayHello` and tell me what type it gives as a response.

The expected result is:
The method is found, and the type is `HelloReply`.

6.3.3 Comments and Options

The tester is provided with the common format file. The website is running on localhost on port 3000.

The following instructions are given:
You have a website running on the localhost on port 3000. What does the method `ListFeatures` in the service `RouteGuide` do, and what is the *java_package* of that service?

The expected result is:
The method obtains features within the given rectangle, and the *java_package* is *guiding.route*.

6.3.4 Execute Unary Request

The tester is provided with the common format file. The website runs on localhost on port 3000, and the gRPC-Web proxy with the gRPC server is set up.

The following instructions are given:
You have a website running on the localhost on port 3000. Execute method `SayHello` with their parameters and tell me the response with their headers and trailers.

The expected result is:
The method is executed with any parameter and the response is "Hello Name" in the `message` field of the response message type.

6.3.5 Execute Server Streaming Request

The tester is provided with the common format file. The website runs on localhost on port 3000, and the gRPC-Web proxy with the gRPC server is set up.

The following instructions are given:
You have a website running on the localhost on port 3000. Execute method `SayRepeatHello` with their parameters and tell me the responses with their headers and trailers.

The expected result is:
The method is executed with any parameter and the response is "Hello Name" multiple times in the `message` field of the response message type.

6.3.6 Complex Method Input

The tester is provided with the common format file. The website runs on localhost on port 3000, and the gRPC-Web proxy with the gRPC server is set up.

The following instructions are given:
You have a website running on the localhost on port 3000. Execute method `ListFeatures` with their parameters and tell me the responses. The parameters are the following:

- `lo`: latitude: 400000000, longitude: -750000000,
- `hi`: latitude 420000000, longitude -73000000.

The expected result is:
The method is executed with the parameters, and responses are returned.

6.3.7 Global Metadata

The tester is provided with the common format file and authorization token. The website runs on localhost on port 3000, and the gRPC-Web proxy with the gRPC server is set up.

The following instructions are given:
You have a website running on the localhost on port 3000. Execute method `SayHello` with the authorization token.

The expected result is:
The method is executed, and the authorization token is present in the request metadata section.

6.3.8 Testing Results

The testing has been conducted with five developers, regardless of their experience with gRPC. The goal was to determine the website's intuitiveness and clarity and identify any issues. The individual testing reports follow in the next sections.

6.3.8.1 Tester 1

The first tester is an undergraduate student in computer science.

Pre-questionnaire answers:

1. Mobile applications and hardware.
2. Using them every day.
3. Using them at work for documentation, checking, and testing the APIs.
4. Heard of, but never used.

Post-questionnaire answers:

1. 1 – It is clear and simple to use, but he would prefer a button for the generation in the UI.
2. 2 – Good, clear overview. The metadata button is not clear. It feels like submitting the row.
3. 1 – All good.
4. 1 – All clear, familiar design and controls to the Swagger UI.
5. No.

Individual scenarios:

1. All done as expected.
2. All done as expected.
3. All done as expected.
4. The first method execution was done without any parameters (that is acceptable by gRPC, so it is not an issue). The rest was done as expected.
5. All was done as expected. He did not expect the number after a name in the response message (not part of my implementation; it is related to the backend).
6. All done as expected.
7. All done as expected.

6.3.8.2 Tester 2

The second tester is an undergraduate student in computer science.

Pre-questionnaire answers:

1. Web development, full-stack.
2. Using them actively.
3. Rarely used, but when used, he uses them for documentation and API design checking. The testing of endpoints is usually done using Postman.
4. None.

Post-questionnaire answers:

1. 1 – All was clear.
2. 1 – All clear, similar to Swagger UI. The only complaint is that the red color in client streaming invokes there is an error.
3. 1 – All was clear.
4. 1 – All was clear.
5. The metadata button was also expected in the method request, but he does not know how it is done in Swagger UI. (He found it very quickly regardless.)

Individual scenarios:

1. All done as expected.
2. He could not say the returned message type frame is the returned type. But he guessed the correct place quickly without any help.
3. All done as expected.
4. When he executed the method, he thought the request shown body was the response body. The rest was done as expected.
5. The execution was done as expected. The only confusion was that he was unsure if the response was a list or multiple responses. He has suggested adding a title to each response.
6. He could not find the input for the `lo` parameter before executing the method. It was caused by him playing with the application previously and accidentally changing the tab to the model. After pointing him to the correct tab, he found the input, and the rest was done as expected. It was more of a testing preparation issue than an application issue connected with the stress of being a tester.
7. All done as expected.

6.3.8.3 Tester 3

The third tester is an undergraduate student in computer science.

Pre-questionnaire answers:

1. Software development, management, and mobile development.

2. Using REST APIs for mobile and web applications. Also, using GraphQL web applications.
3. None experience with Swagger UI. Some experience with Apollo Studio (not much).
4. None.

Post-questionnaire answers:

1. 1 – All was clear.
2. 2 – Server streaming responses were too long, so the page became too long. The suggestion is to add scrolling, or a max height, or collapse the responses. The rest was clear.
3. 1 – All was clear.
4. 1 – All was clear.
5. He would add the metadata button in the method request, not only globally.

Individual scenarios:

1. All done as expected.
2. All done as expected.
3. All done as expected.
4. When he executed the method, he thought the request shown body was the response body. The rest was done as expected.
5. All done as expected.
6. All done as expected, but when he had sent the request, he was waiting for the loading on the executed button to end. So, he did not see the responses as they were coming.
7. He could not find the global metadata settings because he did not connect metadata with authorization. The rest was done as expected.

6.3.8.4 Tester 4

The fourth tester is a software developer with a master's degree in computer science.

Pre-questionnaire answers:

1. Web engineering, backend development in Typescript.
2. Every day.
3. Not much usage, but he knows what they are. He uses the OpenAPI specification for REST APIs instead of Swagger UI.
4. He was using it about two years ago for mobile development.

Post-questionnaire answers:

1. 1 – All was clear.
2. 1 – Similar to Swagger UI, so all was clear.
3. 1 – He was not sure about the returned type model. The rest was clear.

4. 1 – He was looking for a while for the button to execute the method, but he missed it because it was too wide. The rest was clear.
5. If the gRPC API were larger, it would be good to have collapsible services/types/enums sections.

Individual scenarios:

1. All done as expected.
2. He could not say if the returned type is the message body or the whole message type. But he eventually remembered it was the whole message type. The rest was done as expected.
3. All done as expected.
4. He could not find the execution button because it was too long, and thought it was a heading. But he had eventually found it. The rest was done as expected.
5. All done as expected.
6. All done as expected.
7. He was looking for metadata in the method but quickly found it at the top. He has suggested making the button or header sticky at the top. He has also pointed out that the Bearer prefix addition seems unnecessary as there might be other types of authorization that this might not work with. The rest was done as expected.

6.3.8.5 Tester 5

The fourth tester is a software developer with a master's degree in computer science.

Pre-questionnaire answers:

1. Web engineering, full-stack with a focus on backend development.
2. 6–7 years of experience designing backend APIs, experience with REST, GraphQL, and GRPC APIs.
3. Most experience with Swagger UI than Apollo Studio. Writing backend specifications and using these tools as a platform for team sharing.
4. Two years of usage on a larger production project. Wrote a framework for Node.js and gRPC. Worked a lot with the core library.

Post-questionnaire answers:

1. 1 – He would like to have the reflection URL directly. Otherwise, it was all good.
2. 1 – Similar to Swagger UI, so all was clear. An idea is adding a badge to the metadata button to symbolize an authorization header set state.
3. 1 – All good. He has stressed the importance of being able to use the `ctrl+F` (find text on the page) shortcut. He also appreciated the model preview in the input form while writing the JSON data.
4. 1 – All good, similar to the Swagger UI. He appreciated the instant response feedback.
5. All good. A suggestion is to add a link to a user guide at the top of the page.

Individual scenarios:

1. All done as expected.
2. All done as expected.
3. He had skipped the comment and told what the method does based on the input and output types, which was correct, but overlooked the comment message. The rest was done as expected.
4. First, he was confused by the request with the response. Then, confused the headers with the response message. The rest was done as expected.
5. Confused what the *streaming* text in response title means, though did not have any effect on the expectations or actions. The rest was done as expected. Suggested scrollbar or collapse of the server responses as the server streaming requests are usually used for monitoring data.
6. All done as expected.
7. He has tried to set the authorization headers manually, which is fine and also works. The rest was done as expected.

6.3.9 Found Issues and Their Solutions

The overall feedback was positive, and the testers found the website intuitive and straightforward. However, a few issues were found, and they are listed in the table 6.2 with their respective solutions. The changes in the UI are shown in the figure 6.1.

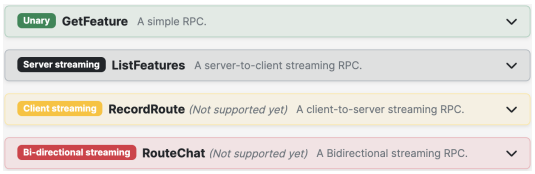
The slightly lighter color scheme and the change of color for client streaming are shown in figure 6.1a to solve the confusion with the method being in the error state. Then the figure 6.1b shows added “OK” text to explain the “0” response status, alongside added “Response message type” title to hint the user about the response type. The figure 6.1c shows a dark border around the execute button to emphasize it, and also the cancel button to the right instead of being underneath.

The metadata renaming to the “Metadata & Authorization” with the larger space on the left and making the method input and the base gRPC URL input smaller, shown in the figure 6.1d, is there to prevent a submit button perception and to explain the authorization definition action. Also, a count badge was added to the metadata button to show the number of set metadata and authorization headers. After clicking the metadata button, a modal dialog is opened. Changes to that dialog are shown in the figure 6.1e. The colors for the set and close buttons have been switched to underline the primary actions, compared to the secondary close. Another change is in the authorization section, where additional types were added with the preview of the final metadata key-value pair that will be added when the authorization is set.

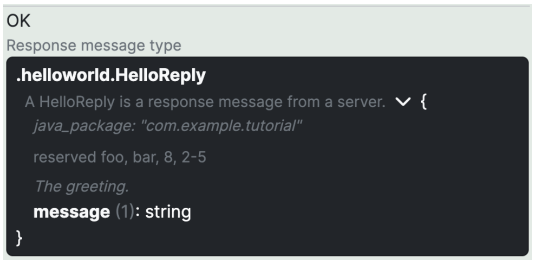
The responses list has been redesigned with the addition of the title to each response message and the total response count. It is shown in the figure 6.1f. Another change is the addition of a maximum height for the response container. Its height is fixed to 50% of the view height, and it automatically scrolls to the bottom, if not scrolled, as new messages arrive. This also helps with having too long message responses.

Issue	Solution	State
The metadata button does not show a clear connection with authorization.	Renamed the button to “Metadata & Authorization” (6.1d).	Fixed
The metadata button invokes the feeling of submitting the row (not opening a modal).	Added greater space between the button and the input fields. Also, make the input fields smaller so the button is much larger. (6.1d)	Fixed
Swapping the request with the response after the method execution.	Changed the request background color to white to match the metadata and highlight the heading.	Fixed
The metadata is also expected in the execution of the method.	Having only global metadata is a design decision based on the Swagger UI. The idea is that the metadata is usually the same for all methods, such as authorization.	Will not fix
The red color background in client streaming invokes there is an error.	The background colors have been redone to be lighter. Also, the client streaming color was changed to yellow. (6.1a)	Fixed
It is hard to distinguish the returned message type from the body of the message type.	Added description of the response status code and the title for the returned message type. (6.1b)	Fixed
Confusion if the responses are a response as a list or a list of responses.	Added a title to each response with the message number and total count at the end. (6.1f)	Fixed
Server streaming methods can have too many responses, making the page too long.	The responses are now scrollable in a fixed-height container. (6.1f)	Fixed
The responses are not distinguished from the headers.	Added titles to each response and the total count. (6.1f)	Fixed
Not seeing the responses as they are coming, waiting for the loading on the executed button to end.	Relocated the cancel button to the right side of the execution button. Changed the responses to be shown in a fixed-height container. (6.1c) Also, this was more related to the tester’s stress, where, in reality, they would scroll sooner.	Partially fixed
If the gRPC API were larger, it would be good to have collapsible services/types/enums sections.	Added collapsible sections for services, message types, and enum types.	Fixed
The button for execution is too long, so it is mistaken for the heading.	The button’s width is designed based on the Swagger UI, but its color, border, and icon (in front of the text) were changed to increase visibility. (6.1c)	Fixed
The close button on the metadata modal is mistaken for the set button in the authorization.	The close button is now a secondary color, and the set button is a primary color. (6.1e)	Fixed
There may be other types of authorization than Bearer.	Added more authorization types, such as Basic and API Key. Otherwise, the use of authorization is optional, and the user can always use the metadata directly. (6.1e)	Fixed
Have the gRPC reflection URL directly on the website.	That is not possible because of the gRPC browser limitations and the static website (backendless) hosting.	Will not fix
Add authorization metadata set state badge.	Added a badge of the set metadata headers count.	Fixed
Add a user guide link at the top of the page.	This might be added in the future if the solution is public (the user guide is part of the GitHub repository), but as of now, it will not be added.	Will not fix

Table 6.2 Found issues and their solutions



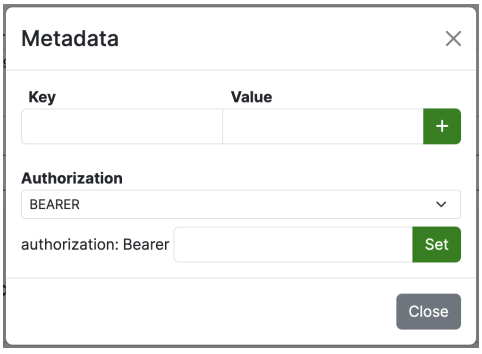
(a) Background colors



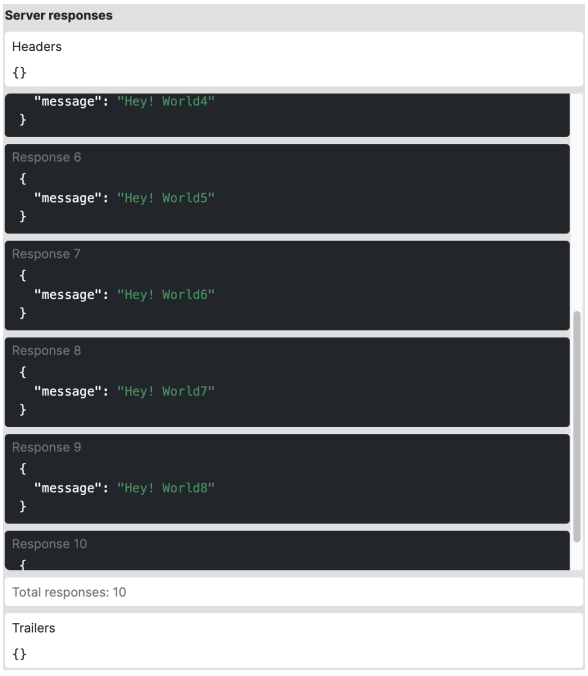
(b) Response description

(c) Execute and cancel buttons

(d) Metadata button, global settings



(e) Metadata dialog



(f) Scrollable responses with titles and total count

Figure 6.1 Changes after testing

6.4 Testing Summary

The user testing feedback was positive, and the testers found the website intuitive and straightforward. It was also appreciated that the process and documentation presentation are familiar to the Swagger UI. The generation of the common format was simple, but it was suggested to have a button for the generation in the UI. The generation is designed to be performed by automatic deployment or other tools, so having the UI would change its purpose. Therefore, it is not planned to be implemented.

The user testing showed that the resulting website fulfills the documentation purpose, allows the method calls, and contains no functionality-affecting bugs. After fixing the found usability issues (see subsection 6.3.9), the intuitiveness of the website is met.

The automatic and manual testing confirmed the fulfillment and functionality of all the static website generator requirements.

Chapter 7
Conclusion

In my thesis, I have dealt with the analysis of existing solutions for Protocol Buffers and compared them with similar tools for GraphQL and RESTful APIs, as well as the design, implementation, and testing of the static website generator with interactive API call support.

As part of the analysis, I found out what tools are used for the Protocol Buffers and how they work, including their features and shortcomings. I have then done the same for GraphQL and RESTful APIs. Finally, I compared the tools and analyzed the state of the static website with interactive API call support. The main outcome is that there are tools for RESTful APIs with almost no issues from my task perspective. The existing tools for GraphQL come close but have a few issues, like the lack of a self-hosted solution. On the other hand, the tools for Protocol Buffers are not as advanced. They have many issues, like not being a static website, lack of support for documenting comments, or lack of support for interactive API calls. None of the analyzed tools for Protocol Buffers support all the features I have defined as necessary for my task. Based on the information found, I then compiled requirements and use cases.

In the next chapter, I discussed the design of the solution. I have analyzed the possibility of using Swagger UI, but I have found that it is not suitable for my task. This is because the OpenAPI Specification is made for HTTP APIs, which gRPC APIs, although using HTTP/2, are not. Therefore, the specification and the web UI do not support specific gRPC features, like streaming. Then, I analyzed the gRPC-Web library and gRPC reflection limitations, which influenced the design of the solution (gRPC reflection does not support documentation comments as of right now). I have then proposed a solution that uses the gRPC-Web library and is able to use the gRPC reflection. The architecture's central part uses a common file format and static website, which is then rendered on the fly based on the provided file. I have found an existing library called protobufjs, which can convert proto files and the gRPC reflection to the JSON format with small improvements from my side. This has allowed me to design the website the same way as the Swagger UI works, which should allow familiarization with the tool.

In the implementation, I then put the designs together, and using several libraries, I built two generators (one for proto files, one for the gRPC reflection) and a static website with the ability to call the gRPC-Web methods. I had also overcome several issues. One was successfully extracting the gRPC-Web client from the gRPC-Web library, which was not designed for this purpose, and there is no other solution for this. Then, for one issue with the protobufjs library, I had it fixed locally, and I created an issue on the library's GitHub page and opened a pull request with a fix proposal.

As part of the testing, I created automated unit tests that covered the website functionality. Then, I built test scenarios covering all functionality and use cases, which were used to test the static website and generators. I also conducted user testing, which revealed several minor bugs. By resolving these, I removed the barriers to intuitiveness and fixed minor issues.

The static website with both generators created meets all the requirements and solves the issues of existing solutions by creating a static website based on a common file format and two generators from proto and gRPC reflection files. Together, they create a unique solution and option for the gRPC API documentation.

7.1 Possible Future Development

For the future development, I see several possibilities. One of them is to add support for more backends than just gRPC-Web. This could be done by creating a custom proxy to translate the requests from the UI to the gRPC server. The implementation could be done using Websockets or HTTP/2, as it has to support client streaming. This would not only allow the website to be used with any gRPC server but also not limit its use to any specific backend.

Another possibility is to add support for automatic generation from the gRPC API server source codes. This could be done by creating a Gradle or Maven plugin (for Java or Kotlin projects), which will generate the JSON file from the project proto files and then host the static website along the project server. This would allow the website to be used without extensive deployment setup and would allow it to be used in the development phase — for example, the automatic JSON definition file re-generating on each update.

Appendix A

Website Guide

This tool helps you interact with gRPC services. You can use it to explore the service's endpoints and make requests to them, browse types and enums, and preview options.

A.1 Prerequisites

To run this application, you need to have the libraries installed.

```
pnpm install
```

A.2 Usage

Here are the commands you can use to run the application or generate JSON common format for the proto files.

A.2.1 Website

Information about the website compilation and development or production build.

A.2.1.1 Development

To run the website in development mode, use the following command.

```
pnpm -C web run dev
```

A.2.1.2 Production

To run the website in production mode, use the following commands. First, build the website and then start the server. The server will be available at `http://localhost:3000` by default.

```
pnpm -C web run build
pnpm -C web run start
```

A.2.2 Proto Files to JSON Generation

Generates a JSON from the proto files. The source files can be a single file or a list of files separated by a space or a folder/folders.

```
gf-proto-to-json ${SOURCE_PROTO_FILES} > ${EXPORTED_NAME}.json
```

A.2.3 Reflection to JSON Generation

Generates a JSON from the reflection. The source file should be a single file in the .bin format.

In the following example, the source file is a protoset file. The protoset file can be created using the grpcurl[1] tool.

1. Create a protoset file (it can be done using different ways, this is just an example).

   ```
   grpcurl -protoset-out descriptors.bin -plaintext localhost:8980 describe
   ```

2. Generate a JSON from the protoset file. The file should be a single file in the .bin format.

   ```
   gf-reflection-to-json ${SOURCE_BIN_FILE} > ${EXPORTED_NAME}.json
   ```

A.3 Testing Server

To test the application, you can use the example testing server and Envoy proxy. It is a simple gRPC server that has a few endpoints and types.

(Source of the Node.js server and Envoy proxy configuration: `https://github.com/grpc/grpc-web/tree/master/net/grpc/gateway/examples/helloworld`)

1. Go to the example folder.

   ```
   cd example
   ```

2. Start the Envoy proxy.

   (Linux users: Use address: localhost instead of address: host.docker.internal in the bottom section.)

   ```
   docker run -d -v "$(pwd)"/envoy-proxy.yaml:/etc/envoy/envoy.yaml:ro -p
       8080:8080 -p 9901:9901 envoyproxy/envoy:v1.22.0
   ```

3. Run the gRPC server.

   ```
   node server.js
   ```

4. Access the (already running) website and set the server URL to `http://localhost:8080`.

[1] `https://github.com/fullstorydev/grpcurl`

Bibliography

1. GOOGLE LLC. *Overview | Protocol Buffers Documentation* [online]. 2024. [visited on 2024-03-11]. Available from: `https://protobuf.dev/overview/`.
2. GOOGLE LLC. *Language Guide (proto 3)* [online]. 2024. [visited on 2024-03-11]. Available from: `https://protobuf.dev/programming-guides/proto3/`.
3. GOOGLE LLC. *Core concepts, architecture and lifecycle* [online]. 2022. [visited on 2024-03-16]. Available from: `https://grpc.io/docs/what-is-grpc/core-concepts/`.
4. GOOGLE LLC. *Metadata* [online]. 2024. [visited on 2024-03-17]. Available from: `https://grpc.io/docs/guides/metadata/`.
5. GOOGLE LLC. *gRPC over HTTP2* [online]. 2023. [visited on 2024-03-16]. Available from: `https://github.com/grpc/grpc/blob/master/doc/PROTOCOL-HTTP2.md`.
6. GOOGLE LLC. *gRPC Web* [online]. 2023. [visited on 2024-03-16]. Available from: `https://github.com/grpc/grpc/blob/master/doc/PROTOCOL-WEB.md`.
7. GOOGLE LLC. *gRPC for Web Clients* [online]. 2024. [visited on 2024-03-16]. Available from: `https://github.com/grpc/grpc-web`.
8. GOOGLE LLC. *Streaming Roadmap* [online]. 2023. [visited on 2024-03-16]. Available from: `https://github.com/grpc/grpc-web/blob/master/doc/streaming-roadmap.md`.
9. GOOGLE LLC. *Reflection* [online]. 2024. [visited on 2024-03-16]. Available from: `https://grpc.io/docs/guides/reflection/`.
10. CHAPMAN, Roger. *Wombat* [online]. 2021. [visited on 2024-03-17]. Available from: `https://github.com/rogchap/wombat`.
11. BLOOMRPC. *BloomRPC* [online]. 2023. [visited on 2024-03-17]. Available from: `https://github.com/bloomrpc/bloomrpc`.
12. GENDOCU CLOUD. *gRPC Docs* [online]. 2022. [visited on 2024-03-17]. Available from: `https://github.com/gendocu-com/grpc-docs`.
13. FULLSTORY. *gRPC UI* [online]. 2024. [visited on 2024-03-18]. Available from: `https://github.com/fullstorydev/grpcui`.
14. GOGOPROTOBUF. *letmegrpc* [online]. 2019. [visited on 2024-03-19]. Available from: `https://github.com/gogo/letmegrpc`.
15. KRÄMER, Benjamin. *protoc-gen-letmegrpc does not support subtypes as repeated* [online]. 2018. [visited on 2024-03-19]. Available from: `https://github.com/gogo/letmegrpc/issues/44`.
16. ZHANG, Jikai; LIU, Zhengyang. *gRPC-swagger* [online]. 2020. [visited on 2024-03-20]. Available from: `https://github.com/grpc-swagger/grpc-swagger`.

17. BRANDHORST, Johan et al. *gRPC-Gateway* [online]. 2023. [visited on 2024-03-20]. Available from: https://github.com/grpc-ecosystem/grpc-gateway.

18. G2, INC. *Grid Report for API Platforms | Winter 2024* [online]. 2023. [visited on 2024-03-26]. Available from: https://www.g2.com/reports/grid-report-for-api-platforms-winter-2024.embed?secure%5Bgated_consumer%5D=cc626f0f-1e59-4bd9-a005-1d41885ec07d&secure%5Btoken%5D=5c23561e75cffcd402334a5730b2e29401c52ed08015396aae5936d18f19c986&tab=profile-postman.

19. WISE, Joshua. *Postman Now Supports gRPC* [online]. 2022. [visited on 2024-03-20]. Available from: https://blog.postman.com/postman-now-supports-grpc/.

20. PRODUCTSUP. *protoc-gen-proto2asciidoc* [online]. 2022. [visited on 2024-03-18]. Available from: https://github.com/productsupcom/protoc-gen-proto2asciidoc.

21. MUTO, David. *protoc-gen-doc* [online]. 2022. [visited on 2024-03-18]. Available from: https://github.com/pseudomuto/protoc-gen-doc.

22. GREIF, Sacha. *Other Tools* [online]. 2023. [visited on 2024-03-26]. Available from: https://2022.stateofgraphql.com/en-US/other-tools/.

23. RAMIREZ, Fede. *Static page generator for documenting GraphQL Schema* [online]. 2020. [visited on 2024-03-20]. Available from: https://github.com/2fd/graphdoc.

24. GRAPHCOOL. *GraphQL Playground* [online]. 2022. [visited on 2024-03-20]. Available from: https://github.com/graphql/graphql-playground.

25. GRAPHQL. *GraphiQL* [online]. 2024. [visited on 2024-03-20]. Available from: https://github.com/graphql/graphiql/tree/main/packages/graphiql.

26. APOLLO GRAPH INC. *Essential GraphQL developer tooling* [online]. 2022. [visited on 2024-03-27]. Available from: https://www.apollographql.com/tutorials/fullstack-quickstart/06-connecting-graphs-to-apollo-studio.

27. SMARTBEAR SOFTWARE. *API Tools, Technologies, and Methodologies* [online]. 2023. [visited on 2024-03-26]. Available from: https://smartbear.com/state-of-software-quality/api/tools/.

28. REBILLY, INC. *Redoc* [online]. 2024. [visited on 2024-03-21]. Available from: https://github.com/Redocly/redoc.

29. MAJUMDAR, Mrinmoy. *RapiDoc* [online]. 2023. [visited on 2024-03-21]. Available from: https://rapidocweb.com/.

30. SMARTBEAR SOFTWARE. *Swagger UI* [online]. 2024. [visited on 2024-03-20]. Available from: https://swagger.io/tools/swagger-ui/.

31. SMARTBEAR SOFTWARE. *Basic Structure* [online]. 2024. [visited on 2024-03-25]. Available from: https://swagger.io/docs/specification/basic-structure/.

32. SMARTBEAR SOFTWARE. *OpenAPI Specification* [online]. 2021. [visited on 2024-03-29]. Available from: https://swagger.io/specification/.

33. SMARTBEAR SOFTWARE. *Plugin system overview* [online]. 2024. [visited on 2024-03-25]. Available from: https://swagger.io/docs/open-source-tools/swagger-ui/customization/overview/.

34. GOOGLE LLC. *gnostic* [online]. 2023. [visited on 2024-03-31]. Available from: https://github.com/google/gnostic.

35. WIRTZ, Daniel. *protobufjs* [online]. 2022. [visited on 2024-04-01]. Available from: https://www.npmjs.com/package/protobufjs.

36. MOZILLA CORPORATION. *WebAssembly Concepts* [online]. 2024. [visited on 2024-04-08]. Available from: https://developer.mozilla.org/en-US/docs/WebAssembly/Concepts.

37. MICROSOFT. *What is JavaScript?* [online]. 2020. [visited on 2024-04-14]. Available from: https://www.typescriptlang.org/why-create-typescript.

38. GREIF, Sacha. *Other Tools* [online]. 2023. [visited on 2024-04-14]. Available from: https://2022.stateofjs.com/en-US/other-tools/.

39. GREIF, Sacha. *Front-end Frameworks* [online]. 2023. [visited on 2024-04-08]. Available from: https://2022.stateofjs.com/en-US/libraries/front-end-frameworks/.

40. META OPEN SOURCE. *React* [online]. 2024. [visited on 2024-04-08]. Available from: https://react.dev/.

41. VERCEL, INC. *The React Framework for the Web* [online]. 2024. [visited on 2024-04-08]. Available from: https://nextjs.org/.

42. META OPEN SOURCE. *Start a New React Project* [online]. 2024. [visited on 2024-04-08]. Available from: https://react.dev/learn/start-a-new-react-project.

43. GREIF, Sacha. *CSS Frameworks* [online]. 2024. [visited on 2024-04-08]. Available from: https://2023.stateofcss.com/en-US/css-frameworks/.

44. WIRTZ, Daniel. *protobufjs-cli* [online]. 2021. [visited on 2024-04-09]. Available from: https://www.npmjs.com/package/protobufjs-cli.

45. NX. *The Original Tool for JavaScript Monorepos* [online]. 2024. [visited on 2024-04-10]. Available from: https://lerna.js.org/.

46. KOCHAN, Zoltan et al. *PNPm* [online]. 2024. [visited on 2024-04-10]. Available from: https://pnpm.io/.

Attached Media Contents

- `readme.txt` — brief media contents description
- `website` — exported static website
- `proto-to-json` — proto files to json converter
- `reflection-to-json` — reflection to json converter
- `src`
  - `impl` — implementation source code
  - `thesis` — thesis source code in the LaTeX format
- `text` — thesis text
  - `thesis.pdf` — thesis text in the PDF format