

Markdown Generation from a GraphQL Schema

Author:

Alessandro BUONERBA

Supervisor:

Dr Konstantin KAPINCHEV

A dissertation submitted in fulfillment
of the requirements for the degree of
BSc (Hons) Computer Science (Cybersecurity)

Department of Computing & Mathematical Sciences
Liberal Arts & Sciences



University of Greenwich
London, United Kingdom

March 2022

Markdown Generation from a GraphQL Schema, © March 2022

Author:

Alessandro BUONERBA

Supervisor:

Dr Konstantin KAPINCHEV

Institute:

University of Greenwich, London, United Kingdom

CONTENTS

List of Figures	v
Abstract	vi
Acknowledgments	vii
1 INTRODUCTION	1
1.1 Aims	1
1.2 Objectives	1
2 ETHICAL CONSIDERATIONS	2
3 LITERATURE REVIEW	3
3.1 Introduction	3
3.2 GraphQL Security	3
3.3 GraphQL vs REST	3
3.3.1 Introspection	4
3.3.2 N+1 Problem	5
3.4 Exploring GraphQL API	6
3.5 Framework Agnostic	6
3.6 GraphQL Schema	6
3.7 MD and MDX	7
3.8 Conclusion	8
4 UX DESIGN AND AGILE METHODOLOGIES	9
4.1 Ways of Working	9
4.2 Kanban vs Scrum	9
4.3 Agile Project Management	10
4.3.1 Epics	10
4.3.2 Spikes	10
5 IMPLEMENTATION	11
5.1 Design	11
5.1.1 Packet Manager	11
5.2 Proof of Concept	12
5.3 Refactoring	18
5.3.1 TypeScript Module Support	18

5.3.2	TypeScript Class Support	18
5.3.3	TypeScript Type Support	19
5.3.4	TSConfig	20
5.3.5	How to Run	22
5.3.6	MDX Generation	22
5.4	Version Control System	23
6	TESTING	24
6.1	Testing	24
7	CONCLUSIONS	26
7.1	Future Improvements	26
7.1.1	Headless CMS	26
7.1.2	CI/CD	26
7.1.3	Snyk	26
7.1.4	Terraform	27
7.1.5	Docker	27
7.1.6	Software as a Service	27
7.1.7	Integrated Frontend	27
7.1.8	Pure Functional Programming	27
7.2	Tool Used	28
7.3	Reflection	28
	BIBLIOGRAPHY	i
A	APPENDIX	1
A.1	Project Proposal	1
B	APPENDIX	5
B.1	Frontend Screenshots	5
B.2	Code Screenshots	5

LIST OF FIGURES

Figure 3.1	Introspection Query	4
Figure 3.2	GraphQL N+1 Problem	5
Figure 3.3	Object Type	7
Figure 3.4	Wizard Query Result	7
Figure 4.1	Ready for Development Agile Board	10
Figure 5.1	High-Level Architecture	11
Figure 5.2	Handlebars Template Language	13
Figure 5.3	PoC Imports	13
Figure 5.4	PoC Type Guards	14
Figure 5.5	PoC Scalar Path	14
Figure 5.6	PoC Scalar Path Markdown	14
Figure 5.7	PoC Format Type Lambda Function	15
Figure 5.8	PoC Print AST Node	15
Figure 5.9	PoC Empty Structure	16
Figure 5.10	PoC Full Structure	16
Figure 5.11	PoC Nested Loop	17
Figure 5.12	PoC Output Folder	17
Figure 5.13	Constants File	18
Figure 5.14	Union Types	19
Figure 5.15	TSConfig	20
Figure 5.16	OutDir Option Result	20
Figure 5.17	Generated Folder with Markdown Files	22
Figure 5.18	Monorepo	23
Figure 6.1	Test formatItem Mutation Type	24
Figure 6.2	Test Output	25
Figure 6.3	Passing Tests	25
Figure A.1	REST API vs GraphQL API	1
Figure A.2	GraphQL N+1 Problem	2
Figure A.3	JAMstack Workflow	3
Figure A.4	High-Level Architecture	4

ABSTRACT

Since GraphQL was publicly released in 2015, many developers adopted it to create new public faced APIs, but these are often poorly documented. Writing well-structured documentation requires time and manual work, and the tooling currently available at the time of this paper would require technologies and frameworks lock-in. This project aims to generate markdown files that can then be parsed in any framework of choice. Since the output will be in markdown, the user can then use his parser to manipulate the documents and produce static files for their frontend.

ACKNOWLEDGMENTS

This project required a lot of time and the minimum I can do is to thanks the support of my friends and family. *Thank You for everything!*

Also, a huge thank you to supervisor **Dr Konstantin Kapinchev** for support me throughout this year. I have been lucky to have you!

INTRODUCTION

Generating an up-to-date documentation from a specific source of truth such as a schema, is a very important aspect of documenting an Application Programming Interface (API) for a large corporate company and not only. The absence of a tool to generate such documentation, more specifically a tool to generate Markdown files. Even more importantly, one that could generate a file with a support for syntax aimed at custom components such as Markdown for the component era (MDX). The modern world is building the majority of their web in React, and it is very important and relevant that a file supports both Markdown (MD) and JavaScript XML (JSX) and MDX does just that. This would certainly mean and resolve one of the biggest issues developers are finding when working with GraphQL APIs, which is Introspection. Introspection is a GraphQL query that allows developers to navigate and discover the traits of an entire Schema from the external world, which is something we all want to avoid at all cost for a security perspective. This project has been designed and researched and discussed with my supervisor to be the solution at the above problems while keeping it framework agnostic.

1.1 AIMS

This project will not only generate the markdown files but also give examples on how to integrate them in a frontend framework of choice.

1.2 OBJECTIVES

Chapter 2 will explain the legal, social, and ethical aspects of the project.

Chapter 3 will give an overview of the literature review.

Chapter 4 will give an overview of the user experience and agile methodologies.

Chapter 5 will give an overview of the implementation.

Chapter 6 will give an overview of the testings.

Chapter 7 will give an overview of the conclusions.

ETHICAL CONSIDERATIONS

The data that will be used to generate the documentation and used within the software is not connected anyhow to any individual or company. The source of truth will be a schema that is a representation of the data that could be used to auto-generate the documentation and might point bad actors to write malicious queries, hence why it is suggested to have schema and the documentation private. If by any chance there might be data into field descriptions that could be used, it will be processed securely as it will not be stored in any database and directly served to the end user locally without traversing the web, following the Data Protection Act-1998 (Legislation.gov.uk, [2022](#)).

LITERATURE REVIEW

3.1 INTRODUCTION

Documentation is a crucial aspect of the development of software. It is so important that every developer building a GraphQL API should be aware of it and keeping their Schemas documented to enable and facilitate consumers to understand how to use the API (Derks, 2021). A very good way to document such APIs is not to be locked in with any vendor-specific that generates documentation on the fly given a specific schema but building static documentation that automatically updates on the fly when a change is made to it (ibid). The following literature review will point the differences between different APIs, analyse and critically evaluate the options developers have to reach the same goal and why the software implementation of the tool used in the project gives the producers better results, freedom and save them money. Security implications are also another important aspect of the lifecycle and this paper will dive into analysis and confrontation to understand this.

3.2 GRAPHQL SECURITY


3.3 GRAPHQL VS REST

Before diving in, context needs to be added on why the projects build solutions for GraphQL instead of REST. REST has become the common choice to build and design APIs in the past years (Brito and Valente, 2020), but as the developers needed more flexibility on evolution, a sharp decouple between the backend and frontend and a gateway to access all the data they needed through a single point of access. GraphQL came to the rescue of such developers wanting a better way to build API. While REST usually have multiple endpoints to fetch different type of data, GraphQL unifies everything through a single query to the server with a simple JSON return with the data requested by the consumer. One of the biggest problems of REST is overfetching. Designing an API that returns exactly what the consumer wants in REST requests is impossible (Seabra, Nazário, and Pinto, 2019).

GraphQL solved the overfetching problem as the user can write a specific query with the data needed. As previously mentioned, GraphQL - differently from REST - is decoupled from the backend. That means the front end can change its UI without changing the back end for adjustments. Not only for the same reason, but the backend also has instrumental analytics on the data requested by the consumer since everything will be requested through a specific query. That leads to supporting graceful deprecation of specific fields that are not mainly used. That is alluring, but how does a consumer know which query to write to gather his data? This is when Introspection comes in handy, but it has some critical drawbacks and security implications that makes it unusable in production. Contrary to all the advantages previously explained, the team at Kernel (2021) also emphasised many disadvantages. One of that is that every query always returns a state of 200 even if the query was not successful. It is very accurate and changes the way of monitoring and measuring errors from how everyone is used with REST. GraphQL is also very bad at handling caching which is also very complex to implement and manage, but one thing that has not been mentioned from Kernel (2021) is that it is also fairly simple to wrap the whole API in a Content Delivery Network (CDN) that would be the first point of contact between the server and the consumer. There are many very viable techniques that would make this solution not only a patch to what GraphQL does badly, but also to generally improve the API. A CDN could fully cache any document and not only small pieces of it, making it very fast, secure, scalable and on edge, which is always a great addition to services such as an API.

3.3.1 Introspection

Introspection is a GraphQL feature that, if enabled, grant access to a query that returns any possible query and updates as the schema evolves overtime. A bad actor could easily access sensitive information, types, and operations supported with such a tool.



```
{
  __schema {
    queryType {
      name
    }
  }
}
```

Figure 3.1: Introspection Query

Introspection enables users to query a GraphQL API and discover its schema structure, giving bad actors a chance to find potentially malicious operations (Khalil, 2021) quickly and disrupt the availability of the API. However, it is also a requirement for tools such as *GraphiQL* and *Playground*. This is a severe dilemma for producers who want to keep their APIs as secure as possible, away from indiscreet eyes, and closed to potential threats but still have documentation tooling. If the attackers have access to the whole schema through introspection, it will be effortless to find and exploit API calls meant for internal use and debugging purposes (Rizwan, 2021). Through the same technique, the attackers could also get access to mutations and API calls intended to add, edit or delete specific data on the database, making it a real threat. Many other security issues are linked to the activation of the introspection and misconfiguration; some are information disclosure, insecure direct object references, and inexistent Access Control List (ACL) (YesWeHack, 2021).

3.3.2 *N+1 Problem*

By design, GraphQL has a fetching inefficiency known as *N+1 Problem* where the number of queries executed against the database (or other upstream services) can be as large as the number of nodes in the resulting graph (GraphQL by PoP, 2020).



```
query {  
  students(first: N) {  
    name  
    friends (first: M) {  
      name  
    }  
  }  
}
```

Figure 3.2: GraphQL N+1 Problem

In the example above, the query against the schema would make a single call to the database to retrieve the first N students, and then for each of these N s students it would make a separate query to the same database to fetch M friends details (N calls), hence $N+1$. Having introspection disabled is the right choice looking at a security perspective, and this project will help solve the downside of not having tools to help document the API and more.

3.4 EXPLORING GRAPHQL API

There are different tools which lock-in both producers and consumers that aims to explore a specific GraphQL API. The problem with those tools is that the producer does not ever have the freedom and the flexibility to have a static documentation running on a machine. Not only, most of the time the GraphQL server needs to be communicating with the service, with introspection enabled. Apollo Studio is a great tool to explore GraphQL APIs with just registering a schema but it is a paid tool and does not come cheap. Also, in order to use most of the tooling offered from Apollo, the server must be built with Apollo server and this hugely lock the company or developer of the project to a single vendor with huge implications in terms of flexibility and ownership of the whole ecosystem that will be built around it. In practical terms, if a producer do want to create a GraphQL API in Elixir using Absinthe (Hexdocs, [2022](#)), it will be impossible for the developer to utilise any of the Apollo tooling.


3.5 FRAMEWORK AGNOSTIC

As previously mentioned, the end goal is to build a framework agnostic tool, meaning that using a single GraphQL schema as a source of truth, the tool will generate a well structured set of markdown files to document the API and utilise those files in any way the developer wants. This would boost productivity and flexibility as the developers can utilise the deep knowledge they already have on a framework of their choice, with the language of their choice (Stefan, [2018](#)).

3.6 GRAPHQL SCHEMA

A GraphQL schema is a set of structured rules that express requirements for an application. It might be easy with a simple UI but it can easily become difficult as the interface of the application grows and evolves. It is to keep in mind that most of the benefits that GraphQL brings to the table are attributed to the schema itself as it enabled most of the features that GraphQL provides such as code generation, parsing, validation and type checking. For this exact reason, many developers couple GraphQL with a strong typed language such as TypeScript to have an ecosystem of tools at their disposal that can help them catch errors at runtime through the IDE or editor itself, rather than at compile time. The GraphQL schema is also a graph because not only is a representation of data graph but also a definition of relationships between entities (Karthic, [2020](#)). That are few in-built types that are covered in GraphQL which are Object, Scalar, Query and Mutation. The Scalar types that GraphQL supports are Int, Float, String, Boolean and ID. All these specs

will be covered and supported in the project as it is very important to support all the types that are built-in GraphQL or the documentation would fail to be procued as the schema could have types not supported. An example of object type below.



```
type Wizard {  
  name: String!  
  age: Int!  
}
```

Figure 3.3: Object Type

In this object type there are also two scalar type as field which represent the decoration of their entities. In this case both fields are both non-nullable fields, which means that the field cannot be null and is mandatory. Meaning that if a consumer performs a query on a non-nullable field, he will never be able to receive a response like the one below.



```
{  
  wizard: {  
    name: null,  
    age: null  
  }  
}
```

Figure 3.4: Wizard Query Result

3.7 MD AND MDX

As Gruber (2020) explains in his article, Markdown is intended to be simple to read and simple to write. It is important to have a an output text that is easy to read so that the end consumer does not have to know anything related to coding to make things as simple as possible. In this case, though, we also want to embed and integrate components in our markdown files, hence the use of the MDX. It is fair to say that MDX is a superset of MD that blends both markdown and JSX syntax to build an hybrid to power the most modern frameworks such as React, Vue and Angular, but still leave the choice to the developers building the project.

3.8 CONCLUSION

Developers are not only interested in building an API but also in continuously document it and provide a way for their consumers to have a better understanding of the API. Even though GraphQL could have some downsides, the advantages of using it are just far too greater to ignore. A tool to create documentation while keeping the schema and the structure of the API secure and keeping everything framework agnostic is something that this project will try to achieve. After some research it is easy to say that it will be unique and could be interesting to see how it could expand after making it open-source.

4

UX DESIGN AND AGILE METHODOLOGIES

4.1 WAYS OF WORKING

During the early stage of the project, many things needed to be decided, such as the language, the agile framework, testing, and how to document everything. The best agile frameworks that would be fit for purpose during the development process were Kanban and Scrum. It is essential to have a good understanding of how to work Agile. Agile is an approach very well known in the industry that facilitates the management of a project by an individual or a group of developers, designers and managers of the same team. It provides a rigorous methodology, depending on which framework, to self-organise work and not deviate from the end goal the team did impose themselves (or their company). While working on this project, an Agile framework will be chosen and used as one of the main requirements to simulate the working of a team of one or more, working for a company or themselves. The goal is to have a simulated experience, create a habit, and document the process, the decisions, and the steps taken from the initial thoughts to the end product. Two main two frameworks will be discussed and evaluated: Kanban and Scrum.

4.2 KANBAN VS SCRUM

Scrum is fast and has ceremonies split into sprint planning, review, retrospective, and daily standups. With Scrum, the team wants to ship a piece of functionality by the end of each sprint and each sprint usually last two weeks. Kanban on the other hand is much more flexible and based on continuous delivery on the idea of having a backlog where the team can park their future work and then move it next as soon as it is unblocked from other tickets. In Kanban new feature and pieces of code are released when ready and one do not need to be too worried of the deadlines given by the sprint such as in the Scrum framework.

4.3 AGILE PROJECT MANAGEMENT

As previously mentioned, there must always be a tool and techniques to keep track of the work must has been done and the progress in general. For this specific project, a Kanban board has been used to keep track of the work, while still retaining some aspects of Scrum, which gives the framework a more flexible approach that is more recognised as a Scrumban approach. Below an example of tickets created in a Kanban board that makes it so visually appealing and easy to work with.

Ready for Development		Reporter	Problem/Opportunity	Priority
[Spike]	Which packet manager?	A		High
	Import GraphQL utilities	A		
+ Add Feature				

Figure 4.1: Ready for Development Agile Board

The Ready for Development board is used for all the tickets that have been through the whole initial lifecycle that comprehend creation, refinement, estimation, and ready to start. This approach is used throughout the project ceremonies but for the sake of the speed have been reduced to a single evaluation in the backlog as in this project there is only one working developer.

4.3.1 Epics

When a huge pieces of functionality needs more than a ticket, a new epic is created where all the moving parts of the new functionality are put together. The epic usually have a description that summarise the whole functionality, what needs to be done to reach the end goal and the members involved.

4.3.2 Spikes

When an investigation is required to find a solution which is unknown to the team, a spike is created. A spike is a ticket that is used to gather information and answering questions instead of producing a piece of functionality or solution. The outcome of a spike is usually documented in the ticket itself and it usually blocks other tickets from being moved to Ready to Development. This project has a spike for starting points such as language to use, tools, and frameworks.

IMPLEMENTATION

5.1 DESIGN

Before diving into implementing the code, it is always imperative to have a whiteboard and design the architecture of the tool being built. The tool that will be made will have a single source of truth for a schema that will then be taken and processed through the logic of the application and transformed into structured folders containing files that represent each living part of the source. The mark-down will support framework-agnostic plug-in and integration, meaning that any JSX framework can parse it and build HTML static files that any browser can read. It is vital to support as many types and Scalars as possible to consistently generate the files from any schema, regarding their complex tree structure and the scalars used by the producer. The diagram below visually explores the solution that will be chased for the end goal.

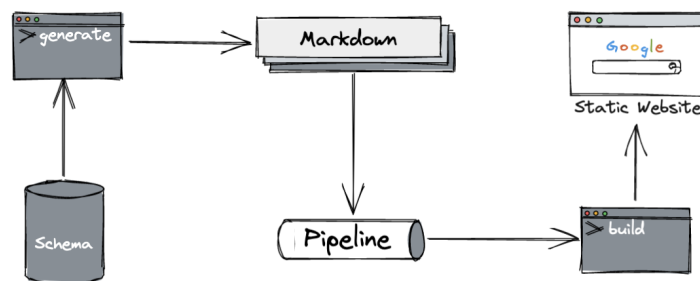


Figure 5.1: High-Level Architecture

5.1.1 *Packet Manager*

A package manager is used to manage any project dependencies in different ways. It allows the developers to execute operations such as deleting, updating or installing packages or running scripts. NPM stands for Node Package Manager, and it was created in 2010. NPM is composed of three main parts: a website for user experience regarding npm aspects, a registry where all the packages are installed

and released for the user, and a CLI to perform an operation from the terminal. Yarn was created in 2016, and it immediately showed more extraordinary performance than npm. It allowed the creation of a `.lock` file that contained the list of the version installed for each dependency, allowing repository sharing improvements. The main difference between these two package managers is how they run operations. In fact, Yarn uses a parallel way by installing dependencies at the same time while NPM installs one package at a time. This increases performance and speed on any kind of operation for the dependencies and is the main reason why Yarn is the preferred package manager in this project, but also in the rest of the world.

5.2 PROOF OF CONCEPT

This section will discuss the first iteration of the project as Proof of Concept (PoC). This will be the initial implementation stage of the project used for discovery and testing. The PoC is a software development methodology that helps the project developers implement the initial logic and validate their hypothesis on the project's feasibility. This specific PoC will be coded in JavaScript using Node as a backend runtime. In a perfect world, the tool should be written using a more robust programming language and ecosystem such as Scala to have the advantage of an effective, high-performance in pure functional programming. This would massively improve concurrency with the use of IO, which facilitates Fibers as a replacement for native OS threads and allows for the benefit of millions of concurrent processes without the need for a thread manager. In this use case, the PoC is using JavaScript to keep things simple, without looking too much at the performances that it would have in a production environment as it will be only used to demonstrate how feasible the project is. Since JavaScript does not support the concept of template literals, a templating language named Handlebars will be used to generate the inside structure of the markdown files. Handlebars use expressions that are evaluated and replaced with the evaluation results. An example of this can be seen below.

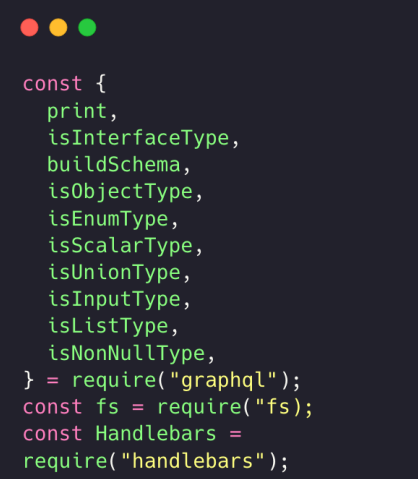


```
---
id: {{name}}
title: {{name}}
hide_title: true
---
#{{name}}

{{#if description}}
{{description}}
{{/if}}
``
{{{printed}}}
``
```

Figure 5.2: Handlebars Template Language

The imports for the PoC will be kept at the very minimal. Below is shown the start of the index file with the imported tools to complete and reach the end goal.



```
const {
  print,
  isInterfaceType,
  buildSchema,
  isObjectType,
  isEnumType,
  isScalarType,
  isUnionType,
  isInputType,
  isListType,
  isNonNullType,
} = require("graphql");
const fs = require("fs");
const Handlebars =
  require("handlebars");
```

Figure 5.3: PoC Imports

In order for it to work, we need to install the GraphQL library and import all the type guards that are implemented in it to be able to check the types while navigating in the complex nested structure of the Abstract Syntax Tree (AST) generated by the reading the schema given in the folder. An example on how the type guards are implemented is shown in the image below.



```
const createPath = (type) => {
  if (isListType(type) || isNonNullType(type)) {
    return createPath(type.ofType);
  }
  if (isScalarType(type)) {
    return `/scalars/${typename}`;
  }
}
```

Figure 5.4: PoC Type Guards

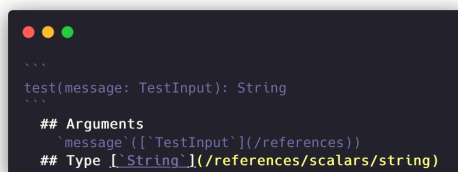
Since in the markdown file, there are places where it would be nice to have hyperlinks that point the user to the right page when navigating the relationship between the fields, there is a need of the type guards to be able to place the right path into the object of a specific formatted data structure used to group the data. An example of the object related to the image above is shown below.



```
mutations: [
  {
    name: 'test',
    description: undefined,
    type: String,
    printed: 'test(message: TestInput): String',
    arguments: [Array],
    fields: [],
    path: '/scalars/string',
    isRootType: true
  }
]
```

Figure 5.5: PoC Scalar Path

As shown in the image above, the field has a type of String and the path to the scalar is /scalar/string. The type guard is implemented as a way to recursively navigate the AST and find the right path to the each specific hyperlink case.



```
test(message: TestInput): String
## Arguments
message ([`TestInput`](/references))
## Type [String](/references/scalars/string)
```

Figure 5.6: PoC Scalar Path Markdown

The image above shows the exact same mutation but in a generated markdown format, completely done by the tool.

In order to have a great structure of the data we are going to manipulate for the generation, a special lambda function has been created which return an object with

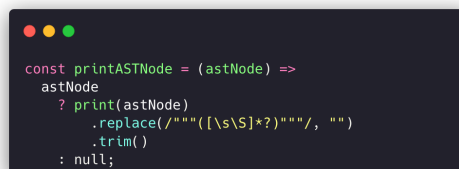
all the properties each type has to offer, it goes and traverse the AST and places the information in the right properties available for later use.



```
const formatType = (type) => {
  return {
    name: type.name,
    description: type.description,
    type: type.type,
    printed: printASTNode(type.astNode),
    arguments: type.args,
    fields: type.getFields
    ?
    Object.values(type.getFields()).map(formatType)
      : [],
    path: createPath(type.type) ?
    createPath(type.type).toLowerCase() : null,
  }
};
```

Figure 5.7: PoC Format Type Lambda Function

It is important to note there is another function operating in the body format-Type which is used to pull out the printed version taken from the schema. This is useful to have so that the documentation can have an original representation of the piece of schema that is being discussed. The image below represent the function written to achieve the printed version of the schema in the formatted structure.



```
const printASTNode = (astNode) =>
  astNode
    ? print(astNode)
      .replace(/""([\s\S]*)""/, "")
      .trim()
    : null;
```

Figure 5.8: PoC Print AST Node

It is again a lambda function that is used to traverse the AST till it reaches a specific node and then surgically extract the printed version of it, making sure it doesn't have any unwanted literal strings, characters or whitespace in it, so that it can directly be used without any further manipulation in the templating system previously described.

One of the last pieces of the puzzle is a wrapper that groups all types in a last and single object, done used the previously described functions recursively for any type in the AST. This prints out a massive object that is then used in a nested loop to again recursively generate the markdown files for each type, while accessing the Handlebars templating for valuating the expressions.

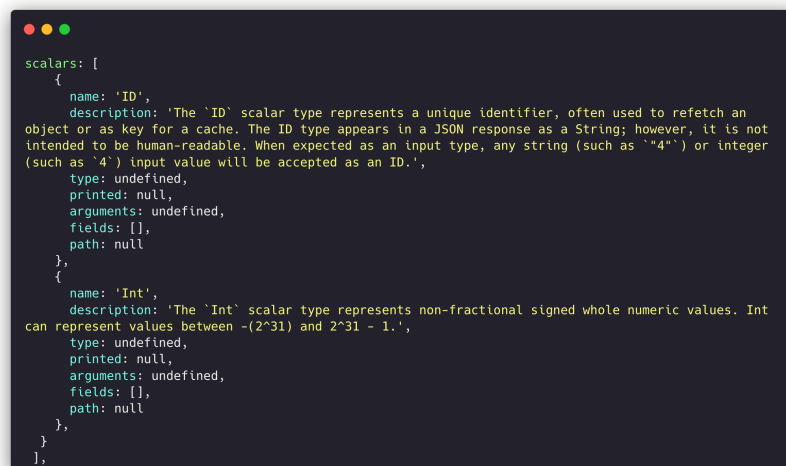
The structure of the final, but empty, data structure is as shown below.



```
{
  queries: [],
  mutations: [],
  subscriptions: [],
  objects: [],
  enums: [],
  scalars: [],
  interfaces: [],
  unions: [],
  inputs: [],
  directives: [],
}
```

Figure 5.9: PoC Empty Structure

The structure is being shown as empty because it would be too much to show in a single image. In order to facilitate the understanding, below also an image representing a single type with its full array structure.



```
scalars: [
  {
    name: 'ID',
    description: 'The `ID` scalar type represents a unique identifier, often used to refetch an object or as key for a cache. The ID type appears in a JSON response as a String; however, it is not intended to be human-readable. When expected as an input type, any string (such as `"4"`) or integer (such as `4`) input value will be accepted as an ID.',
    type: undefined,
    printed: null,
    arguments: undefined,
    fields: [],
    path: null
  },
  {
    name: 'Int',
    description: 'The `Int` scalar type represents non-fractional signed whole numeric values. Int can represent values between  $-(2^{31})$  and  $2^{31} - 1$ .',
    type: undefined,
    printed: null,
    arguments: undefined,
    fields: [],
    path: null
  },
]
```

Figure 5.10: PoC Full Structure

And as previously mentioned, the final structure is then used in a nested function shown below.



```
Object.keys(groupedTypes).forEach((group) => {  
  fs.mkdirSync(`output/${group}`, { recursive: true });  
  
  groupedTypes[group].forEach((type) => {  
    fs.writeFileSync(`output/${group}/${type.name}.md`, queryTemplate(type));  
  });  
});
```

Figure 5.11: PoC Nested Loop

The nested loop makes sure that all the types are accounted for and uses the templating system to generate the markdown files for each type.

The final output is an output folder with all the markdown files generated, and the goal of the PoC has been validated and the code is ready to be used and refactored for better improvements and use cases.

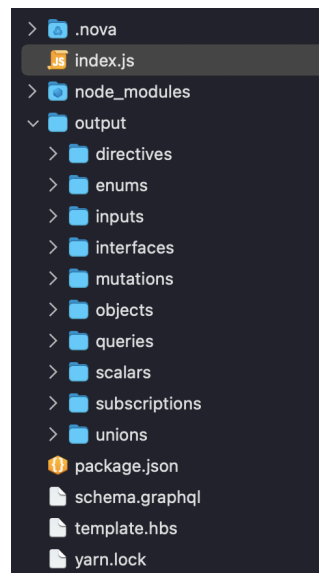


Figure 5.12: PoC Output Folder

Each folder has multiple md files that can be used by any framework that can parse the markdown files and build static generated pages in HTML.

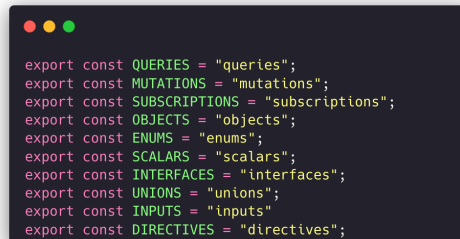
Now that the proof of concept has been completed and resulted in a successful experiment, the next step is to refactor the code and removing unnecessary steps such as Handlebars templating, which will be redundant with TypeScript. It is always good to keep the dependencies as minimum as possible in order to keep the code clean and easy to maintain. Easy to maintain because deprecation of functionalities are always behind the corner and it means that the code needs to be migrated to a new version of the library which would require time, effort and money.

5.3 REFACTORING

For the same reasons explained before, the code will be then refactored in TypeScript removes the templating system and utilises the one in-built with the language. The template literals types will produce the same result shown and achieved in the JavaScript PoC, making it much more succinct, readable and powerful as the code will not be decoupled in different files and folders. It will also keep the code much more DRY with a single source of truth, which the language itself tries to doctrine. The main reasons to migrate any project from JavaScript to TypeScript are listed below.

5.3.1 *TypeScript Module Support*

TypeScript supports modules, meaning that any classes, variables and functions can be exported and imported in other files using the keywords provided by the default API similarly to what is done in ECMAScript 2015. Any file containing import and export keywords will be considered as a module, and those without top-level declarations are considered scripts available globally. A perfect example can be seen in one of the types that are being exported in the refactored tool, in the image below.



```
export const QUERYES = "queries";
export const MUTATIONS = "mutations";
export const SUBSCRIPTIONS = "subscriptions";
export const OBJECTS = "objects";
export const ENUMS = "enums";
export const SCALARS = "scalars";
export const INTERFACES = "interfaces";
export const UNIONS = "unions";
export const INPUTS = "inputs";
export const DIRECTIVES = "directives";
```

Figure 5.13: Constants File

This file contains a set of strings used as keys in the larger object containing all the types and their formatted data. It is very convenient to have a bunch of constants and not have the names of the keys hardcoded directly in the object for both readability and convenience in case of later refactoring.

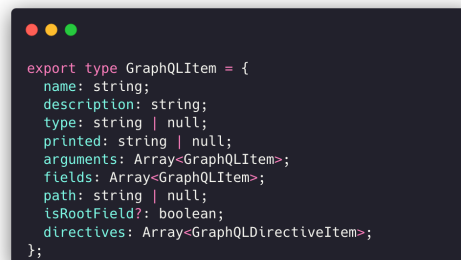
5.3.2 *TypeScript Class Support*

TypeScript is a superset of JavaScript, or even better an object oriented version of JavaScript. Even though TypeScript is considered object oriented programming, it also has and shares many common features with function programming languages.

A class in TypeScript is a recipe to create objects, very similarly to any other object oriented programming languages that have also been covered at the University. This feature could be a very huge selling point to any developer used to Java as main backend language, so that it will be easy to switch from one language to another in case the company has two different languages in different codebases. In this project, there is a slightly different approach as a class has not been used and the methodology leans more towards a functional programming approach without the use of any functional programming libraries such as Lodash or fp-ts, to keep everything as simple as possible.

5.3.3 *TypeScript Type Support*

This is the main reason why TypeScript is a better choice than JavaScript, the type system. TypeScript is statically typed, meaning that all checks are done at compile time, and not at runtime. The compiler will throw an error if variables and expressions do not match the type that is expected. This is all out of the box, for free, and right into any IDE of choice. The primitive types that TypeScript supports are: strings, boolean and numbers. On top of that, it also support a very special type called Union, which is a way to combined and build new types out of already defined or in-built types.



```
export type GraphQLItem = {
  name: string;
  description: string;
  type: string | null;
  printed: string | null;
  arguments: Array<GraphQLItem>;
  fields: Array<GraphQLItem>;
  path: string | null;
  isRootField?: boolean;
  directives: Array<GraphQLDirectiveItem>;
};
```

Figure 5.14: Union Types

In the image above, the Union type is used to combine some primitives such as string with a nullable type which is a special type which make the whole type nullable. Nulls and undefined are the two special types of TypeScript and are very annoying to deal with. Even their creator, Tony Hoare, realised it and admitted it was one of his biggest professional mistakes, or more specifically, his “billion dollar mistake”. The GraphQLItem type shown in the image above, is a representation of the formatted object we previously used to represent data in the proof of concept section. Creating such a type is very simple and effective, meaning any value that the object takes, will need to adhere with the strict type rules provided, and nothing less. If the manipulation of data would result in a different kind of

primitives or newly created type to be used and filled in the object, it will instantly be shown as an error in the IDE and will not compile.

5.3.4 *TSConfig*

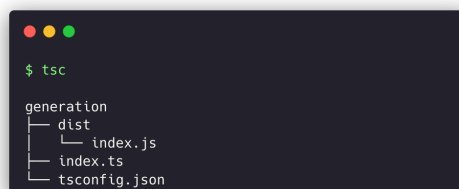
To be able to use TypeScript in the project, it needs to be installed with `yarn install -d typescript`. It will add TypeScript to the `package.json`. Other packages like `ts-node`, `jest`, `eslint` needs to be installed for the project to compile and run. It will also add support for tests which will be shown in later chapters. On top of that, a new file named `tsconfig.json` needs to be created at the root of the TypeScript project which specified the root files and the options given to the compiler for when its time to compile. The options given in this project are the following listed in the image below, which is taken from the project itself.



```
{
  "compilerOptions": {
    "outDir": "./dist",
    "target": "ES2021",
    "module": "commonjs",
    "strict": true,
    "moduleResolution": "node",
    "esModuleInterop": true,
    "skipLibCheck": true,
    "forceConsistentCasingInFileNames": true
  },
  "include": ["src"]
}
```

Figure 5.15: TSConfig

The root object starts with `compilerOptions` which is an object itself that contains the options the compiler will read to compile and build the project. The `outDir` option is used to direct the files on a specific folder when the command `tsc` is given to build the project. It will take the TypeScript files, and convert them in JavaScript files in the destination folder which will look something like the folder structure shown in the picture below.



```
$ tsc
generation
├── dist
│   └── index.js
├── index.ts
└── tsconfig.json
```

Figure 5.16: OutDir Option Result

The `module` option is to specify the module system that will be used in the project. The `target` option is used to specify the target version of JavaScript will be produced from a given piece of TypeScript code. In this case the project will use the most up to date version of ECMAScript (ES). The option `strict` is the most important and also interesting one as it is a little shortcut to enable many strict rules options, to be more specific seven, which they all make the code to have much more strict rules on type checking, which will be very good as stricter is synonym of better and safer. The seven options included in the `strict` shortcut are the following: `noImplicitAny`, `noImplicitThis`, `alwaysStrict`, `strictBindCallApply`, `strictNullChecks`, `strictFunctionTypes`, and `strictPropertyInitialization`.

Below a deep explanation of the options being used in this project:

`noImplicitAny`: is probably the most important as it basically forces the developers to the usage of TypeScript and won't allow JavaScript at all costs. Any code written in JavaScript, for example omitting the type in a parameter of a function, will return an error in the compiler and the IDE itself. This is because, if no type is assigned, it will implicitly be assigned to a type called `any`, hence the name of the option. It is not good practice, but it is still possible to explicitly assign the `any` type to any variable or parameters and make the compiler not complain about it.

`noImplicitThis`: is another option that is used to prevent implicit `this` usage in TypeScript. This is a very good option to use as it will prevent implicit `this` usage in TypeScript and will return an error in the compiler and the IDE itself.

`alwaysStrict`: is used to imaginary place a `"use strict"` statement on top of any and every compiled JavaScript file that will be generated from TypeScript.

`strictBindCallApply`: is used to prevent a common loophole that would make the type check not working in certain situations. This certainly is not a good thing, so keeping this option enabled is recommended in any project.

`strictNullChecks`: is a good option to prevent variables with inferred or explicitly types to be assigned as `null` or `undefined`.

`strictFunctionTypes`: is an option that could be a little intimidating to explore and understand as it cover the world covariance and contravariance, which are topics that are covered more in programming languages such as Scala and are mostly used to build libraries as they are quite advanced. For the purpose of this explanation, this option force the function type parameter checks to be contravariant instead of bivariate and it applies to all function types with the exception of the those that comes from constructors and methods declaration.

strictPropertyInitialization: this is related to classes and a member of a class needs to be initialised with a value in the declaration or constructor, otherwise an error will be thrown.

5.3.5 How to Run

The finalised product is a reference generator that takes a GraphQL schema and spits out a directory of interlinked markdown files for traversing. The intention is to create an exhaustive piece of documentation that can be used by those looking for a quick reference to a specific API.

Requirements: to run the tool you will need a valid GraphQL schema file

Running it: install the dependencies with `yarn install` and run the tool with `yarn generate <path/to/schema> <path/to/output> <linksPrefix>`.

If a path to the schema is not given, the tool will look for a file named `./schema.graphql`. Markdown will be generated into the a specified path, otherwise it will output to a `_generated_` directory. Adding a `linksPrefix` is optional, if one it is not provided, the behaviour will be that all links to other markdown pages will be relative. If one is provided the links will be prefixed with the path provided.

5.3.6 MDX Generation

From previous section, we are at the point where the tool is working and the command to generate the markdown files from a schema is given. The output will be a directory as shown in the figure below, with all the markdown files being generated from the schema.

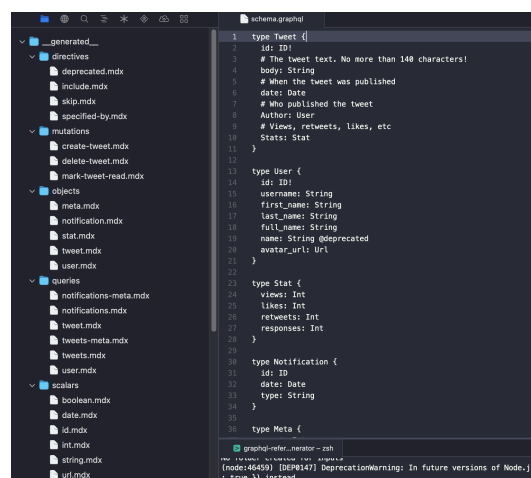


Figure 5.17: Generated Folder with Markdown Files

5.4 VERSION CONTROL SYSTEM

To track changes to the code and have a record of each commit and the work that has been done during the development of the project, a version control system (VCS) has been used. In this project, Git is the chosen VCS and it is used with the integration of GitHub as companion to have a nice and clean interface and quality of life. Below a screenshot of the monorepo created and continuously updated throughout the project.

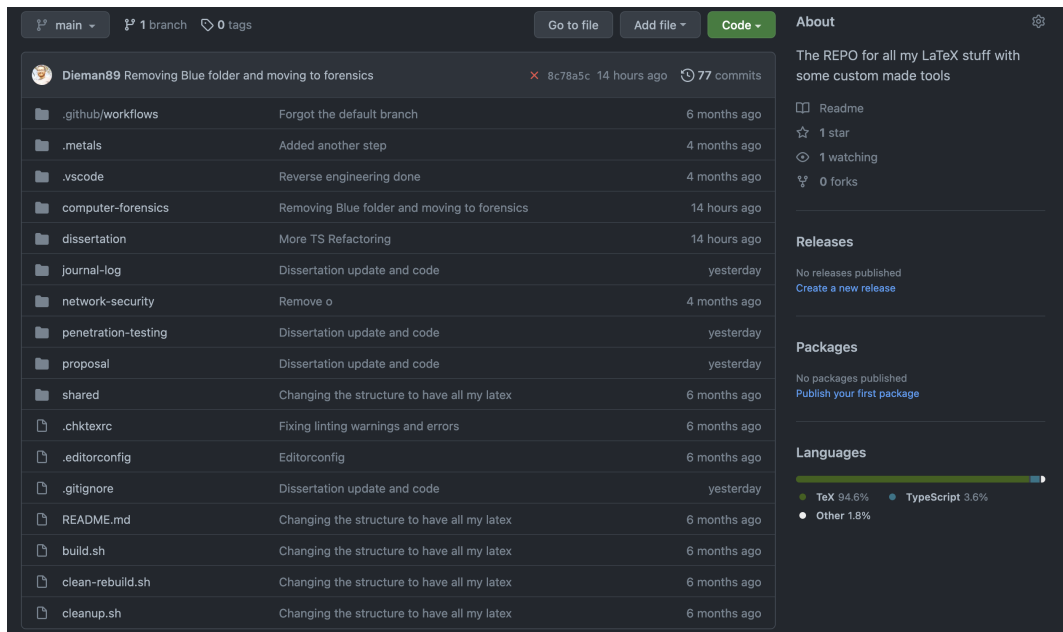


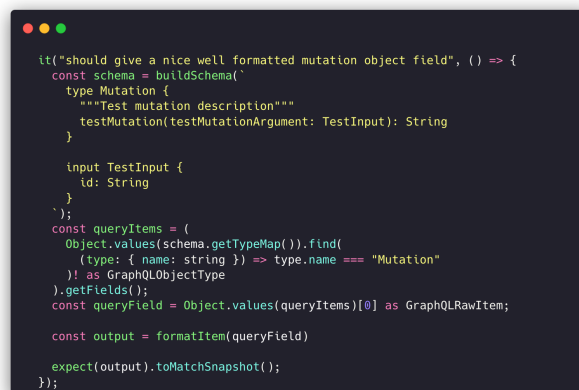
Figure 5.18: Monorepo

TESTING

6.1 TESTING

Testing is a crucial aspect in Software Engineering, so much important that many advocate the use of Test-Driven Development, especially in the Agile world. Test-Driven Development is a process of having tests before anything else, developing them before the code of the software has been written yet, and using such tests to verify the correct behaviour of the business logic. It is a very important methodology, but it was tough to work with it during this project because of the highly nested data structure that the abstract syntax tree would return on the manipulation. For this reason, tests have been written later to ensure vast use-cases and find bugs using large schemas as the starting point for a mock generation. This project uses Snapshot testing to guarantee consistent behaviour in the output of generated objects. That can be very useful because of changes to the logic that can cause everything to break. Such tests would instantly recognise that and fail.

The test files sit in the `src/__tests__` folder and test the behaviour of two main functions, which are the core of the whole application: `formatItem` and `formatSchema`. A screenshot of one of a test below:

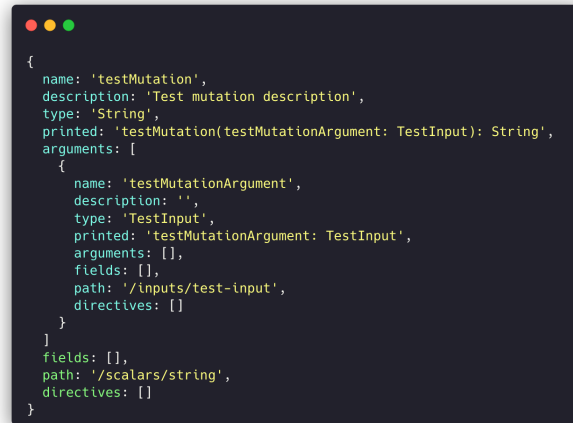


```
it("should give a nice well formatted mutation object field", () => {
  const schema = buildSchema(`
    type Mutation {
      """Test mutation description"""
      testMutation(testMutationArgument: TestInput): String
    }

    input TestInput {
      id: String!
    }
  `);
  const queryItems = (
    Object.values(schema.getTypeMap()).find(
      (type: { name: string }) => type.name === "Mutation"
    )! as GraphQLObjectType
  ).getFields();
  const queryField = Object.values(queryItems)[0] as GraphQLField;
  const output = formatItem(queryField);
  expect(output).toMatchSnapshot();
});
```

Figure 6.1: Test `formatItem` Mutation Type

In the picture above, the mutation type is being tested. The will build a schema, get the field from the AST and then format it.



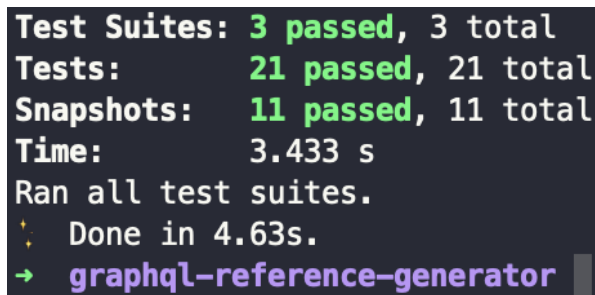
```

{
  name: 'testMutation',
  description: 'Test mutation description',
  type: 'String',
  printed: 'testMutation(testMutationArgument: TestInput): String',
  arguments: [
    {
      name: 'testMutationArgument',
      description: '',
      type: 'TestInput',
      printed: 'testMutationArgument: TestInput',
      arguments: [],
      fields: [],
      path: '/inputs/test-input',
      directives: []
    }
  ],
  fields: [],
  path: '/scalars/string',
  directives: []
}

```

Figure 6.2: Test Output

After that will then compare the formatted output with the snapshot, and if they match, the test will pass. All the types are being tested and the screenshot below shows the terminal output after `yarn test` which run all the tests in the project.



```

Test Suites: 3 passed, 3 total
Tests:      21 passed, 21 total
Snapshots: 11 passed, 11 total
Time:       3.433 s
Ran all test suites.
Done in 4.63s.
→ graphql-reference-generator

```

Figure 6.3: Passing Tests

The output above confirms all passing tests, which is a good sign that the functions used in the project are working as expected and covering multiple uses-cases.

CONCLUSIONS

This section will cover various aspects of how I would improve the project coupled with my final reflection at the end of it.

7.1 FUTURE IMPROVEMENTS

When a project seems to finish, in reality is never really finished. In this section I will discuss some of the aspects that I would like to improve in the future.

7.1.1 *Headless CMS*

In order to separate the content from the frontend and backend, an Headless CMS would result in a very ideal choice. This would make the content management much more easier to maintain and to work with in a large scale as the content would be fetched through a third party API rather than having it in the source code.

7.1.2 *CI/CD*

It would be very nice to not have to manually run things locally and have an automated pipeline to run tests in the continuous integration environment and deploy a new version of the service to a non-prod and prod environment. This whole process not only create the base of a service that would run on a server continuously but also reduce manual errors from developers that would then only focus on writing code rather than monitoring the behaviour of the service in production.

7.1.3 *Snyk*

Snyk is a security platform that is used to automatically and quickly detect and fix vulnerabilities in the codebase. It doesn't not only scan the code, but also the dependencies and vulnerabilities that they might have in their relative source code. Meaning that it would secure an entire lifecycle of a project. It would be very much

on point on having such a security tool but it is a paid product that would be pretty expensive to use for a student. The best thing about Snyk is that it also make sure that a containerised application is secure, and the infrastructure as code is not misconfigured in the cloud.

7.1.4 *Terraform*

Terraform is a tool used to manage an infrastructure on AWS, Azure or Google Cloud Platform. It is known as infrastructure as code as it is used to defined the infra in configuration files, run the plan that would review and make changes to the infrastructure through the public API and finally apply the provision and update the state files on the remote servers.

7.1.5 *Docker*

The benefits of containerisation are that it is easier to manage and it is easier to vertically scale the application. A containerised application uses less memory, start much quickly and enabled portability.

7.1.6 *Software as a Service*

It could be a nice idea to have the tool running on a distributed system that could operate on the web on demand. This would enable developers in need of fast documentation to have quick access and a portal up and running in a third party application with just a schema as a requirement. An integration would then be easily supported through GitHub as a registry to update and deliver schema evolutions automatically.

7.1.7 *Integrated Frontend*

This would be part of the previous section, as once the schema is uploaded, it would require a frontend framework that render the files and display it to the end users.

7.1.8 *Pure Functional Programming*

Probably a refactoring in another language that is more scalable and safe such as Scala with libraries such as Cats and Cats Effect would be a better choice and a safe bet for a tool used in production. Cats is a library that delivers abstraction for

pure functional programming for Scala. Cats is a library that delivers abstraction for pure functional programming for Scala. Cats is short for Category, which, as the name would suggest, is short for Category Theory. It supports type classes such as Functors, Monoids, Bifunctors and Kleisli Categories. Even though there are many things to be learned and explored on such a complex library, it provides a lot of abstraction to extend existing libraries with new functionalities directly in the codebase, which is very powerful. Cats Effect on the other hand, it allows developers to write composable applications with performances in mind with powerful tools such as IO monads which make pieces of code effectful and composable.

7.2 TOOL USED

This section will expand on the tool used to create the project and the report. The text editor being used is Visual Studio Code.

To avoid building everything manually on each change I am using a collection of tools that are well-integrated with my text editor: LaTeX Workshop, LaTeX Utilities, Zotero, LaTeX Better BibTeX

LaTeX Workshop is probably the most important of all. It automatically builds and watches the output pdf file to have a nice hot-reload without closing and re-opening the file to see the changes in a very short delay time.

LaTeX Utilities is an addon for LaTeX Workshop that expands some functionalities such as Zotero citation integrations and smart paste formatter for the images.

Zotero does not only come in a form of a plugin, but also as Software for references and citations. This plugin will attach to the Zotero server and pull data from their endpoint to easily access the library.

Better BibTeX automates the export of the library and keeps it automatically updated on each change (addition, removal, or change). This will export a nice .bib file that can then be used in LaTeX.

7.3 REFLECTION

The main goal of having good generated documentation to start with is to save money and time. If a developer does not have to document a GraphQL API manually, each time their schema changes, it is an enormous save of resources and time that could be spent on developing the actual product rather than documenting it. Creating such a tool is very difficult without any GraphQL knowledge as the code goes very deep into every aspect of GraphQL, schema, evolutions and types. Moreover, if this product would have to be converted into a SaaS, it would require a secure transaction system, a fully working landing page with a login system, a database, and a dashboard to fulfil customer expectations on User Experience. It

would require a whole team working together to achieve such a goal, but surely there would be demand, as many companies struggle with documentation. The main reason to write API documentation, in my opinion, is not only to help developers understand it but also mainly to raise the adoption of it. If a developer in a company creates an API that other teams could use, how does the developer raise awareness and let other teams integrate with his API? The solution would be to develop and keep automatically updated excellent documentation that would assure the quality of the work done and how effective it would be to start integrating with it, rather than using different microservices with multiple endpoints, in case of migration from REST to GraphQL, for example. It is very sad and excruciating to learn stories about developers having to deal with documenting their API or even codebase using outdated methods and tools such as Microsoft Word or Google Docs. It is a common mistake to think it would be time-consuming to build a tool that makes documentation easy, mainly because the technical writers and managerial roles do not know and do not want to read or learn how to read markdown syntax or have to deal with Git and GitHub. For this reason, this project has been done with a JAMStack headless solution in mind so that people covering those roles would have their slice of content sitting far away from the codebase, with a friendly interface that could then be fetched from the frontend framework in the project. To use a headless solution, as previously suggested, the content would need to be moved on a third-party storage and use an API to fetch it before build time, and having the CI/CD pipeline triggered on each change of the content! Overall the project has been successful, and many ideas came to my mind while building it. I have learned a lot of things, and I would be happy to see this solution being used in production to document and keep secure and protect more APIs. And to everyone out there working with GraphQL: keep your Introspection disabled!

BIBLIOGRAPHY

- Brito, Gleison and Marco Tulio Valente (Mar. 2020). "REST vs GraphQL: A Controlled Experiment". In: *2020 IEEE International Conference on Software Architecture (ICSA)*, pp. 81–91. doi: [10.1109/ICSA47634.2020.00016](https://doi.org/10.1109/ICSA47634.2020.00016).
- Derks, Roy (2021). *Documenting GraphQL APIs* | HackerNoon. URL: <https://hackernoon.com/documenting-graphql-apis> (visited on 03/19/2022).
- Fan, Qiang, Yue Yu, Tao Wang, Gang Yin, and Huaimin Wang (Jan. 2021). "Why API Documentation Is Insufficient for Developers: An Empirical Study". In: *Science China Information Sciences* 64.1, p. 119102. ISSN: 1674-733X, 1869-1919. doi: [10.1007/s11432-019-9880-8](https://doi.org/10.1007/s11432-019-9880-8). URL: <http://link.springer.com/10.1007/s11432-019-9880-8> (visited on 10/09/2021).
- Freeman, Adam (2021). "Understanding TypeScript". In: *Essential TypeScript 4: From Beginner to Pro*. Ed. by Adam Freeman. Berkeley, CA: Apress, pp. 35–41. ISBN: 978-1-4842-7011-0. doi: [10.1007/978-1-4842-7011-0_2](https://doi.org/10.1007/978-1-4842-7011-0_2). URL: https://doi.org/10.1007/978-1-4842-7011-0_2 (visited on 10/09/2021).
- Gagliardi, Valentino (2021). "Django REST Meets Next.js". In: *Decoupled Django : Understand and Build Decoupled Django Architectures for JavaScript Front-ends*. Ed. by Valentino Gagliardi. Berkeley, CA: Apress, pp. 113–132. ISBN: 978-1-4842-7144-5. doi: [10.1007/978-1-4842-7144-5_8](https://doi.org/10.1007/978-1-4842-7144-5_8). URL: https://doi.org/10.1007/978-1-4842-7144-5_8.
- GraphQL by PoP (2020). *Suppressing the N+1 Problem* | GraphQL by PoP. URL: <https://graphql-by-pop.com/docs/architecture/suppressing-n-plus-one-problem.html#what-is-the-n-1-problem> (visited on 10/11/2021).
- Gruber, John (2020). *38 Markdown: Syntax*. URL: http://scholar.googleusercontent.com/scholar?q=cache:VSqALECcGQYJ:scholar.google.com/+markdown&hl=en&as_sdt=0,5 (visited on 03/19/2022).
- Helmold, Marc (2020). "Lean Management KPI and OKR". In: *Lean Management and Kaizen: Fundamentals from Cases and Examples in Operations and Supply Chain Management*. Ed. by Marc Helmold. Management for Professionals. Cham: Springer International Publishing, pp. 113–122. ISBN: 978-3-030-46981-8. doi: [10.1007/978-3-030-46981-8_12](https://doi.org/10.1007/978-3-030-46981-8_12). URL: https://doi.org/10.1007/978-3-030-46981-8_12 (visited on 10/09/2021).
- Hexdocs (2022). *Overview — Absinthe v1.7.0*. URL: <https://hexdocs.pm/absinthe/overview.html> (visited on 03/19/2022).
- Karthic, Rao (2020). *Designing GraphQL Schemas* - Dgraph Blog. URL: <https://dgraph.io/blog/post/designing-graphql-schemas/> (visited on 03/19/2022).
- Kernel, Stable (2021). *Advantages and Disadvantages of GraphQL* | Stable Kernel. URL: <https://stablekernel.com/article/advantages-and-disadvantages-of-graphql/> (visited on 03/20/2022).
- Khalil, Stemmler (2021). *Why You Should Disable GraphQL Introspection In Production* – GraphQL Security. URL: <https://www.apollographql.com/blog/graphql/security/why-you-should-disable-graphql-introspection-in-production/> (visited on 10/09/2021).
- Koren, István and Ralf Klamma (2018). "The Exploitation of OpenAPI Documentation for the Generation of Web Frontends". In: *Companion Proceedings of the the Web Conference 2018*. International World Wide Web Conferences Steering Committee. ISBN: 978-1-4503-5640-4. doi: [10.1145/3184558.3188740](https://doi.org/10.1145/3184558.3188740). URL: <https://doi.org/10.1145/3184558.3188740>.
- Legislation.gov.uk (Jan. 2022). *Data Protection Act 1998*. Text. URL: <https://www.legislation.gov.uk/ukpga/1998/29/contents> (visited on 03/19/2022).
- Pratap Yadav, Mahendra, Nisha Pal, and Dharmendra Kumar Yadav (2021). "A Formal Approach for Docker Container Deployment". In: *Concurrency and Computation: Practice and Experience* 33.20, e6364. ISSN: 1532-0634. doi: [10.1002/cpe.6364](https://doi.org/10.1002/cpe.6364). URL: <https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.6364> (visited on 10/09/2021).
- Rizwan, Bilal (Feb. 2021). *GraphQL — Common Vulnerabilities & How to Exploit Them*. URL: <https://the-bilal-rizwan.medium.com/graphql-common-vulnerabilities-how-to-exploit-them-464f9fdce696> (visited on 10/09/2021).
- Seabra, Matheus, Marcos Felipe Nazário, and Gustavo Pinto (Sept. 2019). "REST or GraphQL? A Performance Comparative Study". In: *Proceedings of the XIII Brazilian Symposium on Software Components, Architectures, and Reuse*. SBCARS '19. New York, NY, USA: Association for Com-

- puting Machinery, pp. 123–132. ISBN: 978-1-4503-7637-2. DOI: [10.1145/3357141.3357149](https://doi.org/10.1145/3357141.3357149). URL: <https://doi.org/10.1145/3357141.3357149> (visited on 03/19/2022).
- Stefan, Nieuwenhuis (2018). *3 Reasons Why I Went Framework Agnostic and Why You Should Do That Too* | by Stefan Nieuwenhuis | Medium. URL: <https://stefannhs.medium.com/3-reasons-why-i-went-framework-agnostic-and-why-you-should-do-that-too-f39ba81c6001> (visited on 03/19/2022).
- Wittern, Erik, Alan Cha, and Jim A. Laredo (2018). “Generating GraphQL-Wrappers for REST(-like) APIs”. In: *Web Engineering*. Ed. by Tommi Mikkonen, Ralf Klamma, and Juan Hernández. Cham: Springer International Publishing, pp. 65–83. ISBN: 978-3-319-91662-0.
- YesWeHack (Mar. 2021). *How to Exploit GraphQL Endpoint: Introspection, Query, Mutations & Tools*. URL: <https://blog.yeswehack.com/yeswehackers/how-exploit-graphql-endpoint-bug-bounty/> (visited on 10/09/2021).
- Yu, Liang, Emil Alégroth, Panagiota Chatzipetrou, and Tony Gorshek (Jan. 2020). “Utilising CI Environment for Efficient and Effective Testing of NFRs”. In: *Information and Software Technology* 117, p. 106199. ISSN: 0950-5849. DOI: [10.1016/j.infsof.2019.106199](https://doi.org/10.1016/j.infsof.2019.106199). URL: <https://www.sciencedirect.com/science/article/pii/S095058491930206X> (visited on 10/09/2021).
- Zammetti, Frank (2020). “What Is JAMstack All About?” In: *Practical JAMstack: Blazing Fast, Simple, and Secure Web Development, the Modern Way*. Ed. by Frank Zammetti. Berkeley, CA: Apress, pp. 1–17. ISBN: 978-1-4842-6177-4. DOI: [10.1007/978-1-4842-6177-4_1](https://doi.org/10.1007/978-1-4842-6177-4_1). URL: https://doi.org/10.1007/978-1-4842-6177-4_1 (visited on 10/09/2021).
- Zayat, Wael and Ozlem Senvar (June 2020). “Framework Study for Agile Software Development Via Scrum and Kanban”. In: *International Journal of Innovation and Technology Management* 17.04. doi: [10.1142/S0219877020300025](https://doi.org/10.1142/S0219877020300025), p. 2030002. ISSN: 0219-8770. DOI: [10.1142/S0219877020300025](https://doi.org/10.1142/S0219877020300025). URL: <https://www.worldscientific.com/doi/epdf/10.1142/S0219877020300025> (visited on 10/06/2021).

APPENDIX

A.1 PROJECT PROPOSAL

INTRODUCTION

This project aims at solving the choice between security and programmatically generated documentation in the world of the GraphQL Application Programming Interface (API). API documentation is an essential part of the software development process as it improves the dev experience — making it easier to integrate — enhancing maintainability and conveniently enables versioning with indication on deprecated fields (Fan et al., 2021). Working with API documentation is not always straightforward, especially when working with gateway and federation, as it would require much effort from all the developers involved in the process. Swagger makes documentation for Representational state transfer (REST) APIs very uncomplicated, as it automatically generates HyperText Markup Language (HTML) based visualisation and interaction out of the box for any consumer of the API (Koren and Klamma, 2018). Since Facebook released GraphQL to the mass in 2015, many individual developers and companies are switching and converting to it to build their APIs, as it enables them to have a much more flexible and efficient way of building their APIs (Brito and Valente, 2020). GraphQL gives to the consumer only the data that he needs, solve the overfetching and underfetching problem related to the traditional REST APIs (Wittern, Cha, and Laredo, 2018).

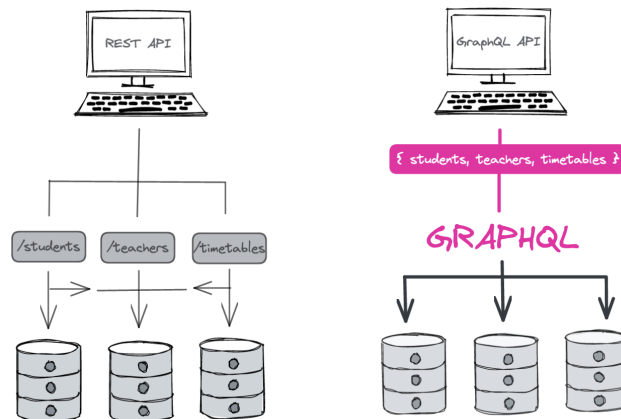


Figure A.1: REST API vs GraphQL API

Querying the API the way users want enables flexibility but can also threaten security if not appropriately handled. This project aims to facilitate API producers

to build secure GraphQL APIs without reachable introspection on the endpoint while still counting on a tool that can generate framework-agnostic structured documentation.

PROBLEM DOMAIN

When working on such a problem, there are many complications in keeping constantly automated and updated documentation for GraphQL APIs. One of them is security. Introspection enables users to query a GraphQL API and discover its schema structure, giving bad actors a chance to find potentially malicious operations (Khalil, 2021) quickly and disrupt the availability of the API. However, it is also a requirement for tools such as *GraphiQL* and *Playground*. This is a severe dilemma for producers who want to keep their APIs as secure as possible, away from indiscreet eyes, and closed to potential threats but still have documentation tooling. If the attackers have access to the whole schema through introspection, it will be effortless to find and exploit API calls meant for internal use and debugging purposes (Rizwan, 2021). Through the same technique, the attackers could also get access to mutations and API calls intended to add, edit or delete specific data on the database, making it a real threat. Many other security issues are linked to the activation of the introspection and misconfiguration; some are information disclosure, insecure direct object references, and inexistent Access Control List (ACL) (YesWeHack, 2021). By design, GraphQL has a fetching inefficiency known as *N+1 Problem* where the number of queries executed against the database (or other upstream services) can be as large as the number of nodes in the resulting graph (GraphQL by PoP, 2020).



```
query {  
  students(first: N) {  
    name  
    friends (first: M) {  
      name  
    }  
  }  
}
```

Figure A.2: GraphQL N+1 Problem

In the example above, the query against the schema would make a single call to the database to retrieve the first N students, and then for each of these Ns students it would make a separate query to the same database to fetch M friends details (N calls), hence N+1. Having introspection disabled is the right choice looking at a security perspective, and this project will help solve the downside of not having tools to help document the API and more.

METHODOLOGY

The project will be divided into two main parts, one being the package to build the structure and generate the documentation from a single source of truth, in this case is a GraphQL schema, and the other is the implementation of a statically generated website that renders the previously generated file in HTML format (Gagliardi, 2021). NodeJS will be used as the backend runtime for the backend to generate the documentation that will then be served to the frontend. The frontend will use NextJS as a React framework to generate the static HTML files parsing previously generated markdowns. A JAMstack (Javascript, API and Markup) with Headless CMS (Content Management System) the approach will be used throughout this project, with additional content outside the generation scope, being decoupled from the whole Versioning Control System (VCS) and being fetched on build time through API queries (Zammetti, 2020).

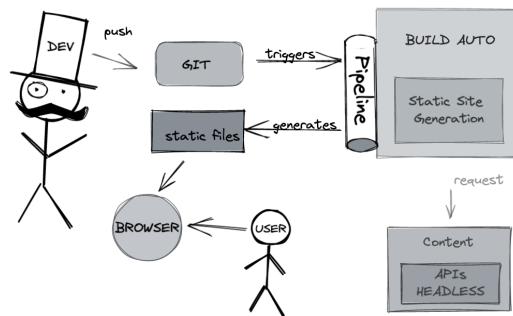


Figure A.3: JAMstack Workflow

A Kanban board will be used to maximise efficiency and visualise the workload and limit the WIP of the project. Kanban has been chosen over Scrum as it is more flexible and does not require running ceremonies, including daily standups. Also, Scrum is overkill for a team of one developer as it would take much time lost on stories, refinements, pointing, spikes and epics while also managing them over different sprints (Zayat and Senvar, 2020). On the other hand, Kanban will focus much more on delivery and is more suited for a small team or individual developers.

The programming language will be JavaScript for the Proof of Concept (PoC) that will then be refactored in TypeScript to make the development process more efficient with its powerful type system saving developers time (Freeman, 2021). The architecture will be managed with popular tools such as GitHub for versioning control and a single source of truth for input files, CircleCI for continuous integration, Vercel for deployment and hosting. To increase package flexibility, the project will also be containerised building an image through Docker that can run on Amazon Web Services (AWS) such as Amazon Elastic Compute (ECS) with a virtual remote machine (EC2) (Pratap Yadav, Pal, and Kumar Yadav, 2021).

EVALUATION

A goal-setting methodology will be used to evaluate the project; the initial goal and key results will be set up at the start of the project with fortnightly updates. To see

if the progress towards the goal is not on the decline, key performance indicators (KPIs) will be set to monitor any gaps and focus the attention on the previously set Objectives and Key Results (OKRs). This will help adjust the goals based on the progress and be able to evaluate if the project reaches a complete state at the end of the deadlines (Helmold, 2020). On the technical side, testing will ensure that the end product is performant, accessible and meets the expectations set at the start of the development process. The project will be tested through a mixture of Unit and End to End (E2E) tests integrated into the previously stated CircleCI pipeline before shipping and deploying the code in production environments. This will ensure that bugged code is only present in non-production environments, and the last deploy step is reached only when the whole project is ready for production (Yu et al., 2020). The development process will be evaluated as successful if the OKRs reaches green metrics and the final project complies with the specifications that have been set (Helmold, 2020). The final product will generate a structured folder with markdown files that document the references of a GraphQL schema and are rendered with a JAMstack philosophy.

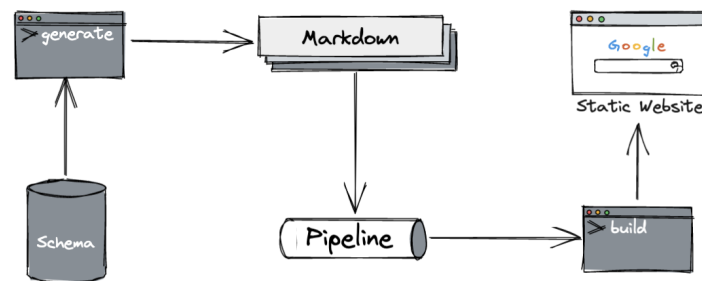


Figure A.4: High-Level Architecture

B

APPENDIX

B.1 FRONTEND SCREENSHOTS

Screenshots Here

B.2 CODE SCREENSHOTS

Screenshots Here