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

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!

1 INTRODUCTION

Generating an up-to-date documentation from a specific source of truth such as a schema, is a very important aspect of documentating 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 ?? will explain the legal, social, and ethical aspects of the project.

Chapter ?? will give an overview of the literature review.

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

Chapter ?? will give an overview of the implementation.

Chapter ?? will give an overview of the testings.

Chapter ?? will give an overview of the conclusions.

2 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.ukDataProtectionAct2022**).

3 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 an keeping their Schemas documented to enable and facilitate consumers to understand how to use the API (**derksDocumentingGraphQLAPIs2021**). A very a 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 AND PERFORMANCES

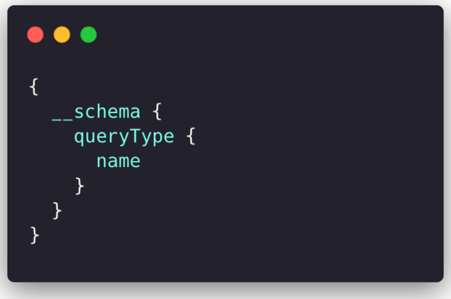
hartigInitialAnalysisFacebook2017 define GraphQL as a new form of web-based connectivity interface that provides alternatives concept of REST approaches. Due to its benefits over REST, the subject of this paper (GraphQL) has gained popularity since its debut. GraphQL has grown in popularity and is being used by many users. GraphQL is a novel technique of interacting using APIs. According to **vogelExperiencesMigratingRESTful2018**, GraphQL is not a database technology, as many people believe. Users may already be aware that most modern applications save their data on a distant server in a database, and here is where GraphQL comes in. Simply providing a means of accessing data required by the application is all required by the API. REST APIs may also be built with GraphQL (**vadlamaniCanGraphQLReplace2021**). It was created internally by Facebook for various purposes before being made open-source and made available to the public. Because of this, it has become one of the most used technology stacks for building online services. GraphQL is a query language that specifies how an application program can ask a distant server for the data it needs. Consequently, the requested client query receives a response from the server app. Client applications may inquire about what they need without relying on their server-side counterparts for guidance. **hartigDefiningSchemasProperty2019**, in their research study, has focused on repurposing schemas for graphs, which was initially intended as a language that clarifies various kinds of things, which could be fetched through queries when accessing a specific endpoint. The perspective of data use GraphQL have a schema to express the shape of the graphs (ibid). Many different viewpoints have been put together by the same author in his paper to define GraphQL schemas in terms of their quantitative definitions (**hartigInitialAnalysisFacebook2017**). GraphQL is sometimes described as an edge-labelled multi-graph with a dictionary of attributes, wherein every node is connected with an object type. **britoMigratingGraphQLPractical2019** have a case study on migrating API customers to this new technology in this article. First, they surveyed the grey literature to acquire a thorough knowledge of the benefits and essential qualities of the technology that practitioners typically connect with. Following that, **britoMigratingGraphQLPractical2019** demonstrate these benefits in reality by moving seven services to GraphQL rather than typical REST-based APIs. As a crucial finding, the researchers demonstrated that it could significantly minimize the volume of JSON records supplied by REST APIs (ibid). In blog, **kristopherUniqueBenefitsUsing2018** outlined the benefits of utilising the graph. He points that since it is so strong, it has been employed by some service providers who require a high level of accessibility and indexing speed. For the most part, the practical applications for GraphQL are those that need fast data flow and easy sorting and representation by its most high-profile customers. An architectural model's specific performance characteristics may be gleaned via studies on response time and average transfer rates between requests, as noted by **seabraRESTGraphQLPerformance2019**. Two-thirds of the evaluated applications found that moving to GraphQL led to a performance boost. After constructing REST and GraphQL API endpoints and testing them in an open-source playground, Seabra et al. performed a PoC not by transitioning to GraphQL but rather by establishing prototype API endpoints for both testings the response time for data extraction across three iterations. In the first iteration there was little or no change when we investigated a single API endpoint for GraphQL and REST with the 1000 data records quantity (**seabraRESTGraphQLPerformance2019**). In the second iteration there was a 35% faster response time utilizing GraphQL, despite a tenfold increase in data volume than Iteration 1 and Iteration 2 were conducted with 10000 data records (ibid). In the third iteration, whenever Seabra et al. raise the quantity of data 100 times the initial quantity and run using one hundred thousand records, they evaluate three endpoints rather than the needed one (ibid). The researchers suggested a roughly 40% reduction in reaction time needed in the GraphQL experiment findings. According to Seabra et al. among software development experts, the subject of which architectural model to adopt is frequently asked: This topic of performance has been answered by examining the performance of three different target apps, each of which was built utilizing two different web services architectural models: REST and GraphQL. Every architectural model's performance indicators may be deduced from the study of reaction time and average transfer rates. Migration to GraphQL led to a dramatic increase in the average demand per second and information transfer rates in various evaluated apps. After the migration research, it was shown that GraphQL-based services performed worse than their REST-based counterparts for loads exceeding three thousands requests, varying 99 to 2160 KB/s. Both REST and GraphQL services performed similarly for simple workloads, with values ranging from six to seven requests each second for loads of one hundred responses between GraphQL and REST services. Researchers at Salt Security's Salt Labs discovered the flaws in the FinTech firm's mobile apps and SaaS platform while doing their study. Problems stemmed from a lack of authorisation and nested queries being prone to many mistakes. Salt Labs discovered that the researchers might submit operations against any user's account or capture any customer's sensitive data if proper authorisation checks were not implemented (**vadlamaniCanGraphQLReplace2021**). A report from Salt Security shows that 62% of firms have no or a minimal API security policy (**vadlamaniCanGraphQLReplace2021**). A basic attack like not verifying or cleaning the API queries may lead to trouble. According to **britoRESTVsGraphQL2020**, GraphQL is becoming increasingly popular as a preferable API management platform in environments with high data volumes, where response speed and the usage of fewer API endpoints are essential performance indicators. Although REST is becoming an enterprise standard and API administration utilising REST endpoints has evolved, GraphQL has a learning curve. With enhanced tooling features over time, future uses of GraphQL will rise manifold. This research spur other researchers to pursue the goal of flexibility and performance with minimal involvement. Several GraphQL implementations allow Introspection by default, which means they may be accessed without authentication (**eizingerAPIDesignDistributed2017**). The fact that Introspection enables the requester to understand all of the available schema and queries is troublesome. API owners may use Introspection to inform users about the API's workings and potential pitfalls. Users prefer to learn about a service via a distinct documentation channel like a Git Readme or reading the manuals. The safest and most convenient option is deactivating Introspection and the GraphiQL scheme (ibid). If disabling Introspection is not supported natively by the implementation, or if users want to provide this access to certain users or roles, therefore may create a filter in the service that restricts access to the introspection system to those users who have been allowed by the service's administrators. Even if Introspection is turned off, an attacker might still brute force fields to estimate their values. As a bonus, GraphQL comes with a built-in hinting function suggests a working query based on whenever a requester's field title is close (but wrong) to an already-existing field. Not all installations of GraphQL enable this functionality; therefore, users should consider deactivating it if they want to reduce their exposure (**kouraiSecureOffloadingLegacy2016**). A tool like Shapeshifter might be able to help with this. Disabling Introspection is a sort of security via obscurity since bad actors still may learn ways to construct harmful queries by reverse-engineering the GraphQL API using several trials and errors (ibid). When used with additional security measures such as volumetric quotas and operation safe listing, it is not the most excellent protection method. It is possible to examine all schema data using GraphQL Introspection. When Introspection is enabled in production, it makes it easier for hostile actors to find graph weaknesses, such as disclosing sensitive information. Using a schema registry to record the graph and metadata is safer and more secure. SQL injection, OS Command Injection (SSRF/CRLF), NoSQL injection, and DoS attacks may all be prevented by disabling Introspection (Denial of Service) (**eizingerAPIDesignDistributed2017**). Attacking the API's accessibility and stability is a form of DoS, which can cause the API to be sluggish, unresponsive, or inaccessible (**gozneliIdentificationEvaluationProcess2020**). There is a fair chance the target has Introspection turned off, so take advantage of this. GraphQL backends come pre-loaded with proposals for fields and functions. GraphQL may offer fields equivalent to the original query if users try to query the field but make a typo. Hackers can use the Field Suggestion functionality to learn more about GraphQL's schema, even though it is not vulnerable (ibid). An investigation on timeouts in GraphQL was conducted by **witternEmpiricalStudyGraphQL2019**. Adding timeouts is a straightforward approach to limit the number of resources a single request may use. The problem is that timeouts do not always work since they do not kick till a malicious query has used a large amount of computing power. There is not a single timeout number that will work for all APIs and data fetching mechanisms. If desired, queries and resolver functions can set timeouts at the application level. Because the timeout has been reached, this technique is typically more effective. Request timeouts cannot be implemented natively in GraphQL due to technical limitations. The proliferation of APIs has resulted in a rise in attempted assaults with automated apps executing harmful operations via the Application Programming Interfaces (**hartigInitialAnalysisFacebook2017**). As per research from wallarm.com, harmful bots now account for up to 20% of the internet. As a result, protecting APIs is a vital component of application security. GraphQL and REST API development lifecycle techniques are helpful in certain situations, and each has its own set of perks and downsides. GraphQL is growing in popularity at a breakneck pace, owing primarily to its "no over and under-fetching" capability (ibid). It enables more effective client-side communication and is an instrumental and powerful technology, particularly as the software sector transitions to an agile framework. GraphQL is a mechanism for accomplishing specific query-oriented goals; nevertheless, it is not a panacea for all API-related difficulties and is not a substitute for REST. When frontend development teams consume REST APIs, they must wait for the backend team to complete creating the APIs necessary for the client application to collect and send data. The complete development process is entirely dependent on the creation and delivery of the REST API. The GraphQL lifecycle method is significantly different and more efficient, allowing frontend and backend programmers to operate concurrently without impeding the whole development process. GraphQL's essential quality is that it is less verbose than REST APIs. GraphQL prioritises speed above everything else, whereas REST prioritises service dependability. It does not matter whether the REST API provides only a simple partial because it is still sending more data than is necessary (**vadlamaniCanGraphQLReplace2021**). The API can add new fields to the GraphQL query, but clients do not get them until they explicitly request them. REST is slower since users cannot select which fields to query; therefore, the request will be as minimal as possible, making it quicker with GraphQL. Since GraphQL transfers fewer bits over the wire, the applications will run quicker than when using REST. Furthermore, developers may query several entities at once using GraphQL (**lawiEvaluatingGraphQLREST2021**). Unpredictable APIs are of no utility. As a result, every program that uses an API must be aware of what it may call and how it can anticipate and act on the API's output appropriately. In other words, an API's explanation of what may be accessed is critical. GraphQL introspection and REST API schema frameworks such as Swagger allow developers to explore programmatically the information they gain from the API documentation. Concerning security, REST appears to have the upper hand when comparing GraphQL to REST. API security may be enforced in several ways using REST (**vadlamaniCanGraphQLReplace2021**). For example, one may secure REST APIs by supporting several API authentication methods, such as the HTTP authentication mechanism. JSON Web Tokens, HTTP headers, or OAuth 2.0 protocols can all transmit private information, such as passwords and credit card numbers. In addition to REST, GraphQL has certain security features; however, they are not as developed as REST's GraphQL (**lawiEvaluatingGraphQLREST2021**). For example, it helps integrate data validation, but users must figure out how to implement permission and authentication procedures on top. In his research, **vadlamaniCanGraphQLReplace2021** investigated whether REST can replace GraphQL. In API development, REST has long been the go-to architectural approach. In 2015, Facebook announced GraphQL, which has since challenged GraphQL's popularity. GitHub, Airbnb, Shopify, Twitter, and many other internet platforms have already implemented it (ibid). Many questions remain unanswered about the efficiency and viability of GraphQL's application, even if it offers a significant improvement over REST. This article evaluates the quantity and quality viability of adopting GraphQL over REST for API design. When it comes to API response speeds, GitHub has built a unique API client that can be used to measure the change between REST and GraphQL. After that, a GitHub workers poll was conducted to understand better what REST and GraphQL mean in software engineers' experience with APIs. Both API paradigms have advantages and disadvantages, and none can substitute the other. There is a clear preference in the GraphQL versus REST argument for REST APIs. A State of API 2020 report found that 82% of API customers and practitioners utilise REST-based OpenAPI authentication, whereas just 18% use GraphQL (ibid). There has also been a steady rise in GraphQL's popularity. JavaScript use has increased from 5% in 2016 to 38% in 2019, as per the State of JavaScript 2019 Report (ibid). Today, most browsers support the HTTP cache so that resources are not re-fetched. It can also be used to tell if two factors are identical in any other way. In GraphQL, users cannot use the same URL for all requests to receive a globally unique id for an item. For this, one will need to configure the GraphQL cache.

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 (**britoRESTVsGraphQL2020**), 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 (**seabraRESTGraphQLPerformance2019**). 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 **stablekernelAdvantagesDisadvantagesGraphQL2021** 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 **stablekernelAdvantagesDisadvantagesGraphQL2021** 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.



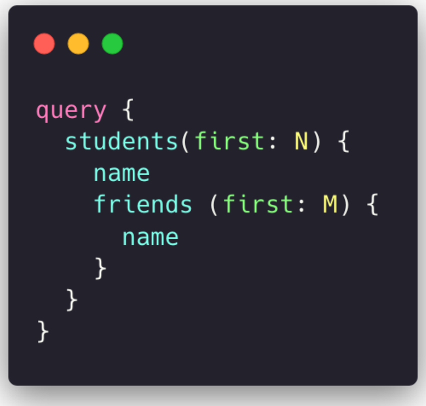
```
{
  __schema {
    queryType {
      name
    }
  }
}
```

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 (**khalilWhyYouShould2021**) 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 (**rizwanGraphQLCommonVulnerabilities2021**). 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) (**yeswehackHowExploitGraphQL2021**).

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 (**graphqlbypopSuppressingProblemGraphQL2020**).



```
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 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.

3.4 EXPLORING GRAPHQL API

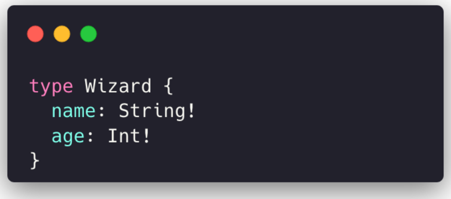
There are different tools which lock-in both producers and consumers that aims to explore a specific GraphQL API. The problem with those tools is that the producer does not ever have the freedom and the flexibility to have a static documentation running on a machine. Not only, most of the time the GraphQL 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 (**hexdocsOverviewAbsintheV12022**), 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 (**stefanReasonsWhyWent2018**).

3.6 GRAPHQL SCHEMA

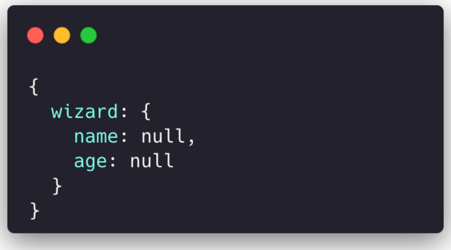
A GraphQL schema is a set of structured rules that express requirements for an application. It might be easy with a simple UI but it can easily become difficult as the interface of the application grows and evolves. It is to keep in mind that most of the benefits that GraphQL brings to the table are 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 (**karthicDesigningGraphQLSchemas2020**). 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 **gruber38MarkdownSyntax2020** 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 continously 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

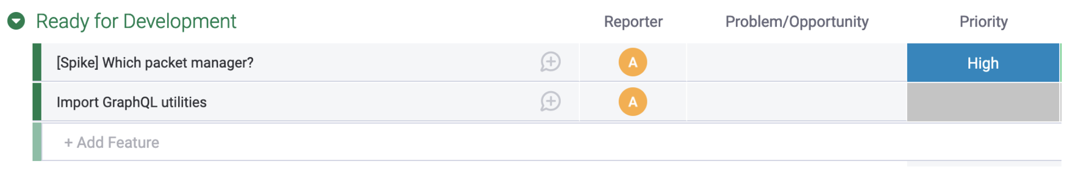
During the early stage of the project, many things needed to be decided, such as the language, the agile framework, testing, and how to document everything. The best agile frameworks that would be fit for purpose during the development process were Kanban and Scrum. It is essential to have a good understanding 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 comphrehend creation, refinement, estimation, and ready to start. This approach is used throughout the project cerimonies 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 Developement. This project has a spike for starting points such as language to use, tools, and frameworks.

5 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 markdown 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.

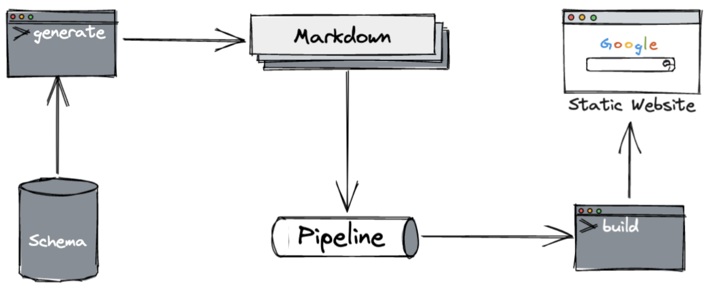


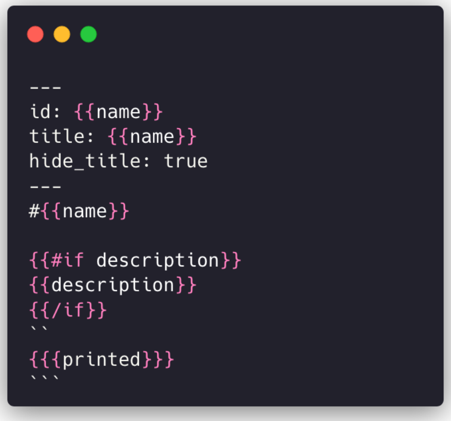
Figure 5.1: High-Level Architecture

5.1.1 *Packet Manager*

A package manager js 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 decencies 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

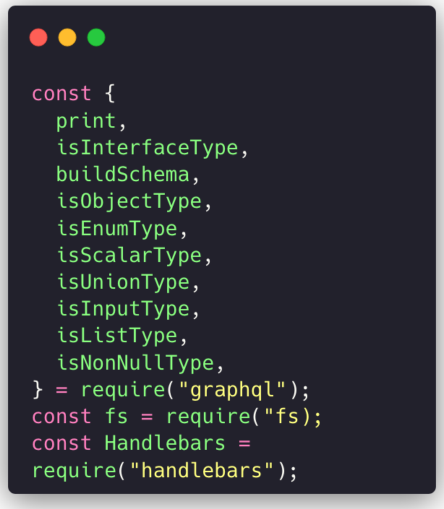
This section will discuss the first iteration of the project as Proof of Concept (PoC). This will be the initial implementation stage of the project used for discovery and testing. The PoC is a software development methodology that helps the project developers implement the initial logic and validate 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 effectful, 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

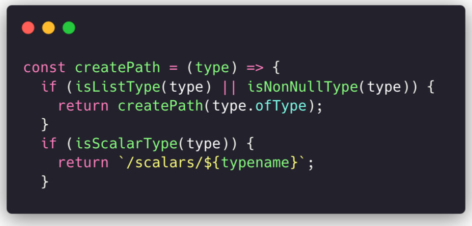
The imports for the PoC will be kept at the very minimal. Below is shown the start of the index file with the imported tools to complete and reach the end goal.



```
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

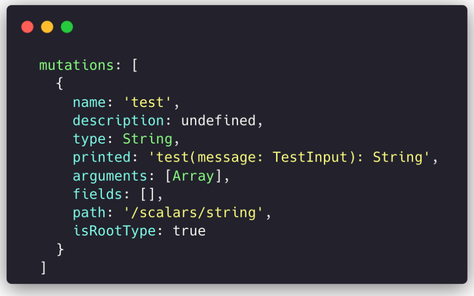
In order for it to work, we need to install the GraphQL library and import all the type guards that are implemented in it to be able to check the types while navigating in the complex nested structure of the Abstract Syntax Tree (AST) generated by the reading the schema given in the folder. An example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

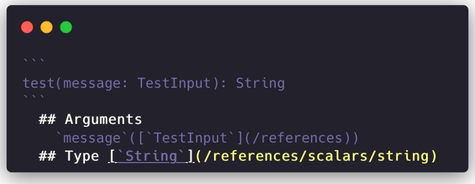
Since in the markdown file, there are places where it would be nice to have hyperlinks that point the user to the right page when navigating the relationship between the fields, there is a need of the type guards to be able to place the right path into the object of a specific formatted data structure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

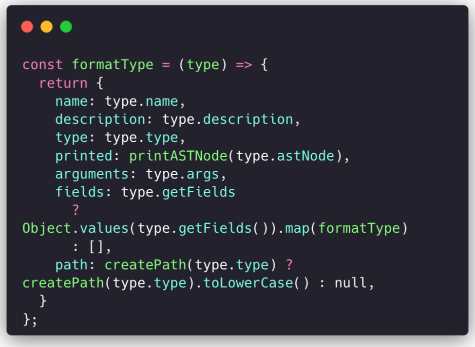


~~~~
```
test(message: TestInput): String
```
  ## Arguments
    `message`([`TestInput`](/references))
  ## Type [`String`](/references/scalars/string)
~~~~

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.

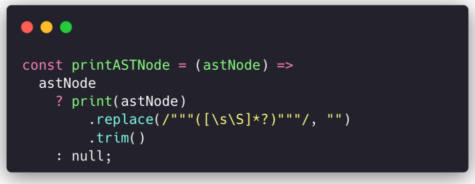
In order to have a great structure of the data we are going to manipulate for the generation, a special lambda function has been created which return an object with all the properties each type has to offer, it goes and traverse the AST and places the information in the right properties available for 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.



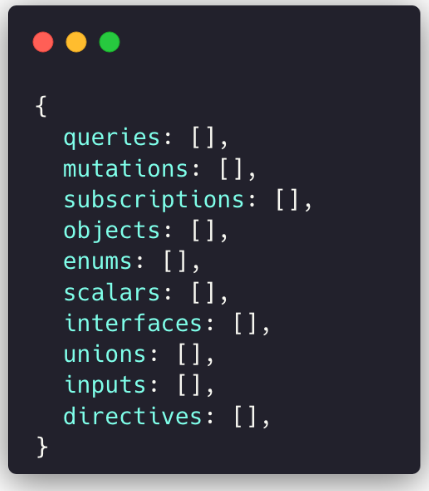
```
const printASTNode = (astNode) =>
  astNode
    ? print(astNode)
        .replace(/"""([\s\S]*?)"""/, "")
        .trim()
    : null;
```

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.

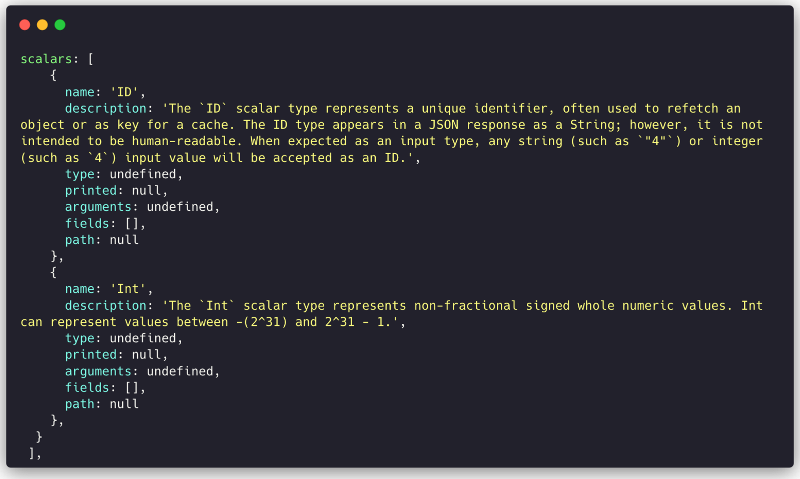
The structure of the final, but empty, data structure is as shown below.



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

Figure 5.9: PoC Empty Structure

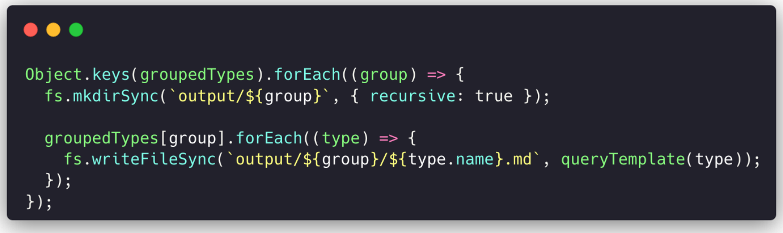
The structure is being shown as empty because it would be too much to show in a single image. In order to facilitate the understanding, below also an image representing a single type with its full array structure.



```
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 previosuly mentioned, the final structure is then used in a nested function shown below.

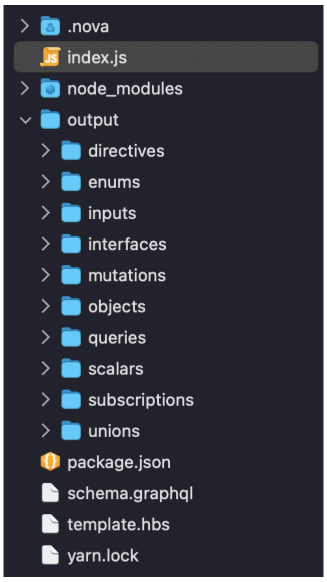


```
Object.keys(groupedTypes).forEach((group) => {
  fs.mkdirSync(`output/${group}`, { recursive: true });

  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.



```
> .nova
  index.js
> node_modules
v output
  > directives
  > enums
  > inputs
  > interfaces
  > mutations
  > objects
  > queries
  > scalars
  > subscriptions
  > unions
  package.json
  schema.graphql
  template.hbs
  yarn.lock
```

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.

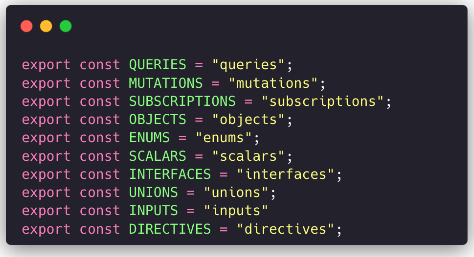
Now that the proof of concept has been completed and resulted in a successfull experiment, the next step is to refactor the code and removing unnecessary steps such as Handlebars templating, which will be redundant with TypeScript. It is always good to keep the dependencies as minimum as possible in 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 QUERIES = "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.

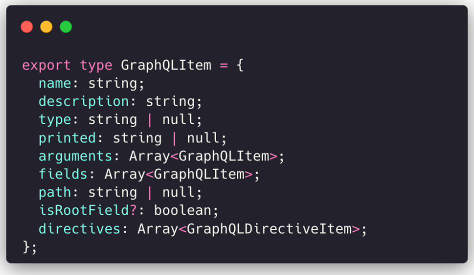
5.3.2 *TypeScript Class Support*

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, maening 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.



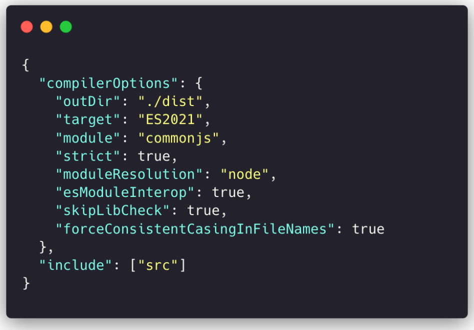
```
export type GraphQLItem = {
  name: string;
  description: string;
  type: string | null;
  printed: string | null;
  arguments: Array<GraphQLItem>;
  fields: Array<GraphQLItem>;
  path: string | null;
  isRootField?: boolean;
  directives: Array<GraphQLDirectiveItem>;
};
```

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 differenty 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*

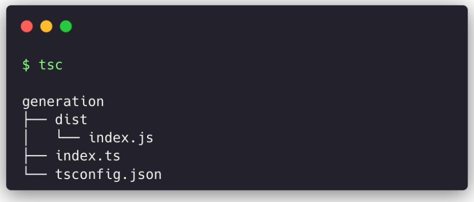
To be able to use TypeScript in the project, it needs to be installed with `yarn install -d typescript`. It will add TypeScript to the `package.json`. Other packages like `ts-node, jest, eslint` needs to be installed for the project to compile and run. It will also add support for tests which will be 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 explaination 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 reccomended 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 explaination, this option force the function type parameter checks to be contravariant instead of bivariant 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.

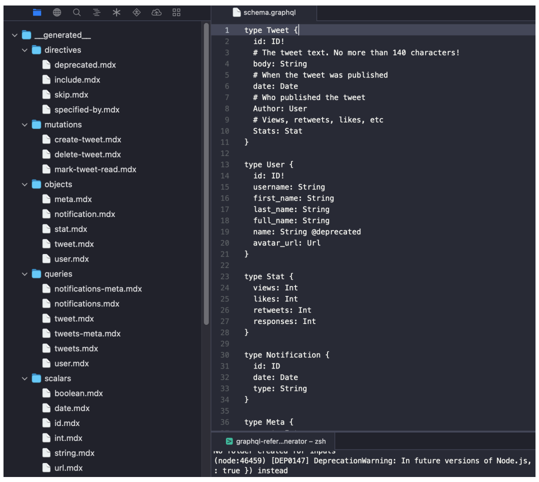
Requirements: to run the tool you will need a valid GraphQL schema file

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.



```
__generated__
  directives
    deprecated.mdx
    include.mdx
    skip.mdx
    specified-by.mdx
  mutations
    create-tweet.mdx
    delete-tweet.mdx
    mark-tweet-read.mdx
  objects
    meta.mdx
    notification.mdx
    stat.mdx
    tweet.mdx
    user.mdx
  queries
    notifications-meta.mdx
    notifications.mdx
    tweet.mdx
    tweets-meta.mdx
    tweets.mdx
    user.mdx
  scalars
    boolean.mdx
    date.mdx
    id.mdx
    int.mdx
    string.mdx
    url.mdx
```

schema.graphql

```
type Tweet {
  id: ID!
  # The tweet text. No more than 140 characters!
  body: String
  # When the tweet was published
  date: Date
  # Who published the tweet
  Author: User
  # Views, retweets, likes, etc
  Stats: Stat
}

type User {
  id: ID!
  username: String
  first_name: String
  last_name: String
  full_name: String
  name: String @deprecated
  avatar_url: Url
}

type Stat {
  views: Int
  likes: Int
  retweets: Int
  responses: Int
}

type Notification {
  id: ID
  date: Date
  type: String
}

type Meta {
```

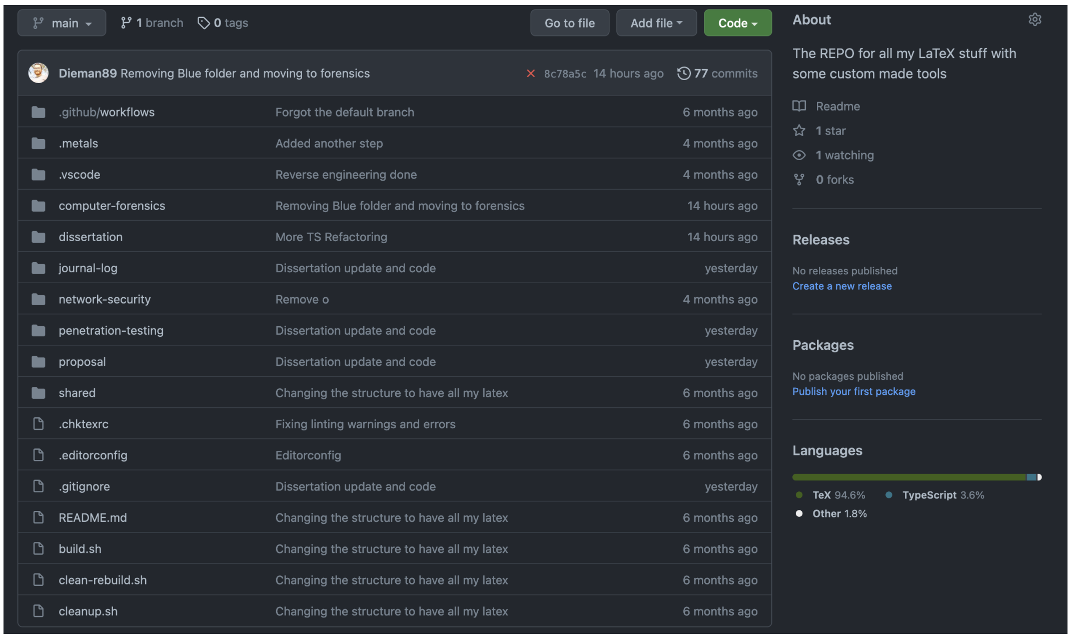
graphql-refer...nerator – zsh

```
(node:46459) [DEP0147] DeprecationWarning: In future versions of Node.js,
: true }) instead
```

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 continously updated throughtout the project.



Dieman89 Removing Blue folder and moving to forensics ✕ 8c78a5c 14 hours ago 77 commits

.github/workflows	Forgot the default branch	6 months ago
.metals	Added another step	4 months ago
.vscode	Reverse engineering done	4 months ago
computer-forensics	Removing Blue folder and moving to forensics	14 hours ago
dissertation	More TS Refactoring	14 hours ago
journal-log	Dissertation update and code	yesterday
network-security	Remove o	4 months ago
penetration-testing	Dissertation update and code	yesterday
proposal	Dissertation update and code	yesterday
shared	Changing the structure to have all my latex	6 months ago
.chktexrc	Fixing linting warnings and errors	6 months ago
.editorconfig	Editorconfig	6 months ago
.gitignore	Dissertation update and code	yesterday
README.md	Changing the structure to have all my latex	6 months ago
build.sh	Changing the structure to have all my latex	6 months ago
clean-rebuild.sh	Changing the structure to have all my latex	6 months ago
cleanup.sh	Changing the structure to have all my latex	6 months ago

About

The REPO for all my LaTeX stuff with some custom made tools

Readme
1 star
1 watching
0 forks

Releases

No releases published
Create a new release

Packages

No packages published
Publish your first package

Languages

● TeX 94.6% ● TypeScript 3.6%
● Other 1.8%

Figure 5.18: Monorepo

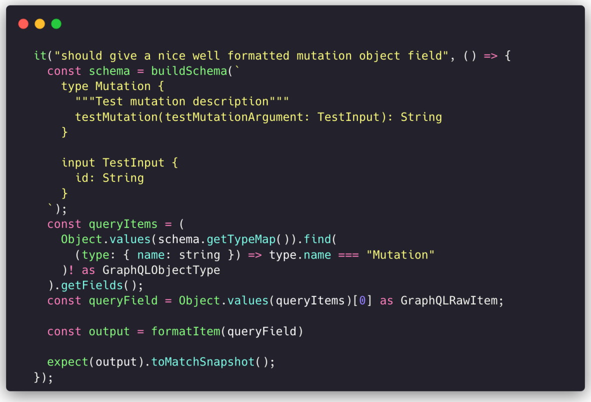
6

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.

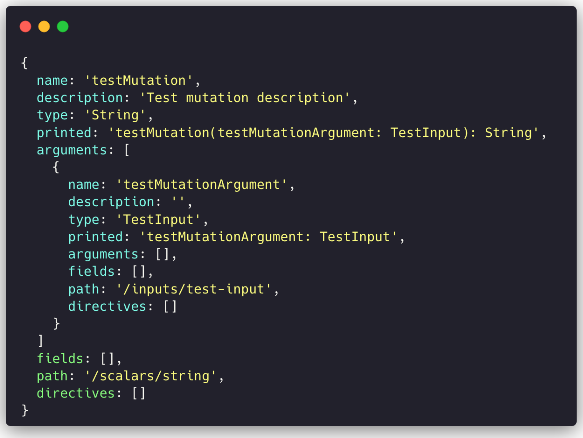
The test files sit in the `src/__tests__folder` and test the behaviour of two main functions, which are the core of the whole application: formatItem and formatSchema. A screenshot of one of a test below:



```
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 GraphQLRawItem;

  const output = formatItem(queryField)

  expect(output).toMatchSnapshot();
});
```

Figure 6.1: Test formatItem Mutation Type

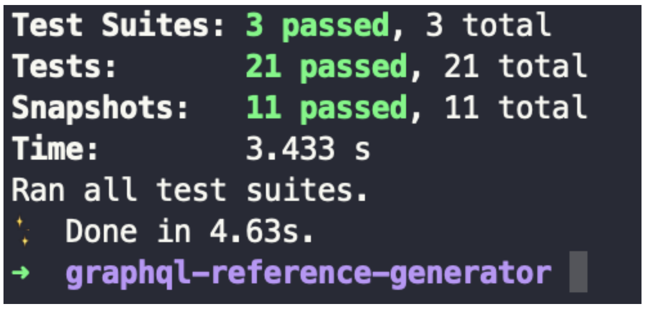
In the picture above, the mutation type is being tested. The will build a schema, get the field from the AST and then format it.



```
{
  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.

7 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 then 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 continously 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 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 misconfugured 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 infrastracture 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 Kleisi 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!

A 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 (**fanWhyAPIDocumentation2021**). 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 (**korenExploitationOpenAPIDocumentation2018**). 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 (**britoRESTVsGraphQL2020**). GraphQL gives to the consumer only the data that he needs, solve the overfetching and underfetching problem related to the traditional REST APIs (**witternGeneratingGraphQLWrappersREST2018**).

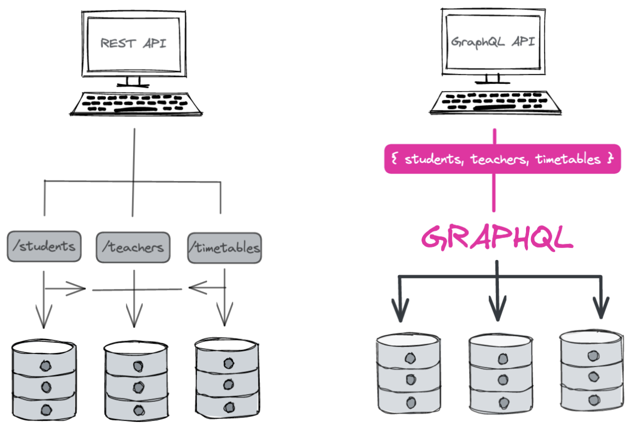
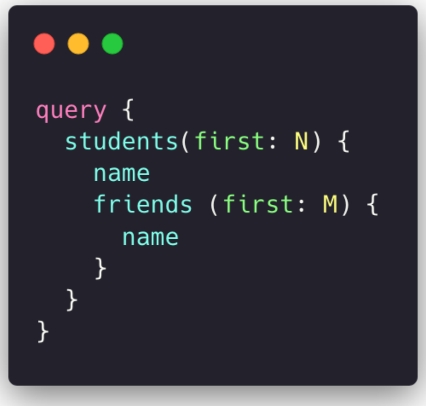


Figure A.1: REST API vs GraphQL API

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 (**khalilWhyYouShould2021**) 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 (**rizwanGraphQLCommonVulnerabilities2021**). 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) (**yeswehackHowExploitGraphQL2021**). 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 (**graphqlbypopSuppressingProblemGraphQL2020**).



```
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 (**gagliardiDjangoRESTMeets2021**). 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 (**zammettiWhatJAMstackAll2020**).

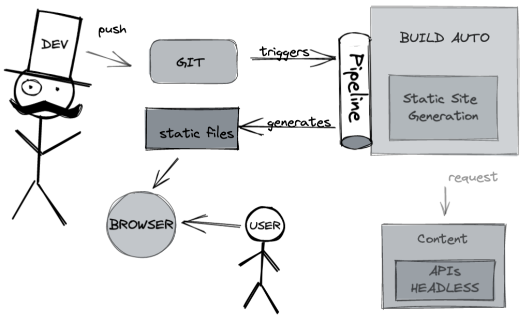


Figure A.3: JAMstack Workflow

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 (**zayatFrameworkStudyAgile2020**). 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 (**freemanUnderstandingTypeScript2021**). 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) (**pratapyadavFormalApproachDocker2021**).

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 (**helmoldLeanManagementKPI2020**). 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 (**yuUtilisingCIEnvironment2020**). 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 (**helmoldLeanManagementKPI2020**). 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.

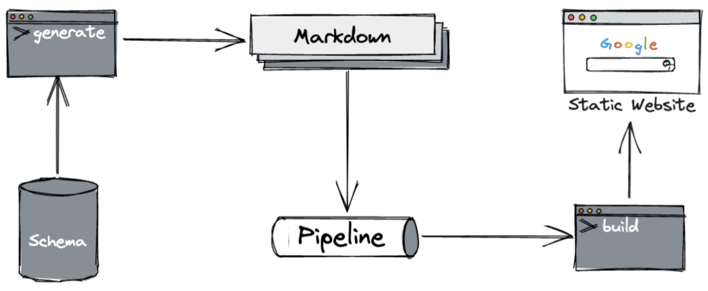


Figure A.4: High-Level Architecture