Documentation from GraphQL Schema

Author:
Alessandro Buonerba

Supervisors:
Dr Konstantin Kapinchev

A proposal 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

October 2021

PROJECT PROPOSAL

INTRODUCTION

Application Programming Interface (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, solving the overfetching and underfetching problem related to the traditional REST APIs (Wittern, Cha, and Laredo, 2018).

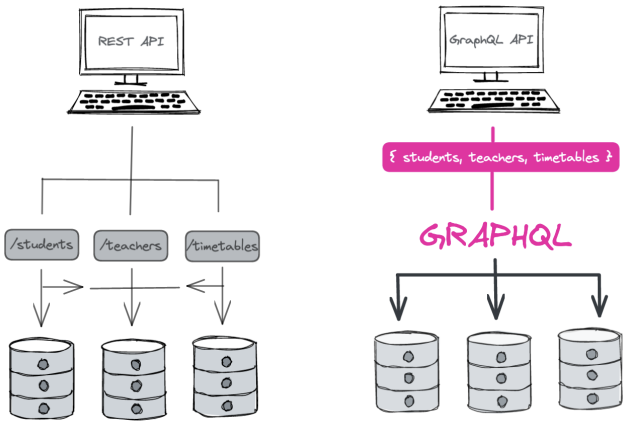

Figure 1.1: REST API vs GraphQL API

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).

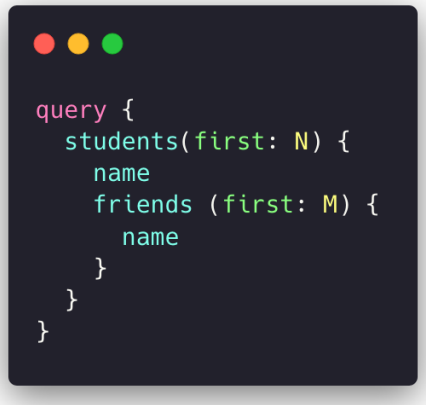

Figure 1.2: GraphQL N+1 Problem

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).

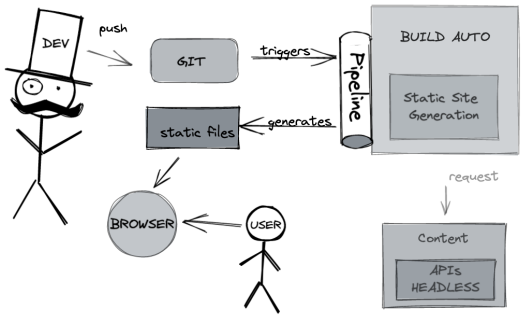

Figure 1.3: JAMstack Workflow

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.

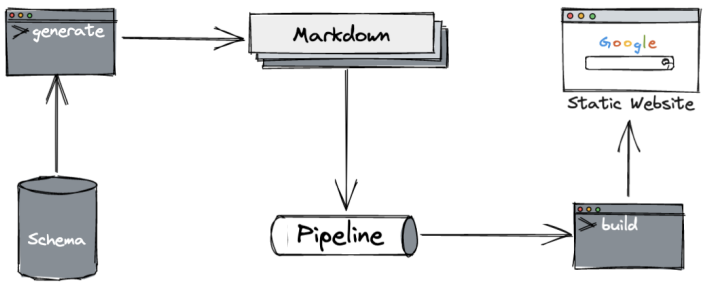

Figure 1.4: High Level Architecture

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.

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. 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. 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. 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).

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. URL: https://doi.org/10.1007/978-3-030-46981-8_12 (visited on 10/09/2021).

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. URL: https://doi.org/10.1145/3184558.3188740.

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. 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).

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/yeswerhackers/how-exploit-graphql-endpoint-bug-bounty/ (visited on 10/09/2021).

Yu, Liang, Emil Alégroth, Panagiota Chatzipetrou, and Tony Gorschek (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. 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. 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,

p. 2030002. ISSN: 0219-8770. DOI: 10.1142/S0219877020300025. URL: https://www.worldscientific.com/doi/epdf/10.1142/S0219877020300025 (visited on 10/06/2021).