FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of bachelor's thesis

Title:	GraphQL API generating service for Django
Student:	Ladislav Louka
Supervisor:	Ing. Jakub Žitný
Study program:	Informatics
Branch / specialization:	Computer Security and Information technology
Department:	Department of Computer Systems
Validity:	until the end of summer semester 2021/2022

Instructions

Django is a web framework where the database is easily accessible via an auto-generated admin interface. GraphQL APIs provide a fast alternative to REST, but with insufficient support in Django.
- Analyze missing functionality in Django. Based on a survey of existing open-source solutions, create a service that automatically generates GraphQL APIs from Django model definitions. The service should generate frontend clients from information contained in the API responses.
- Implement better interoperability between frontend and backend, focus on access rights (limits and indications).
- Generate a client where you will demonstrate the capabilities of your service with pre-generated queries.
- Perform, describe, and evaluate appropriate tests and write documentation. Implement the service with unit tests.
- Publish the service on PyPI and GitHub.

Electronically approved by prof. Ing. Pavel Tvrdík, CSc. on 8 January 2021 in Prague.

Bachelor's Thesis

GRAPHQL API GENERATING SERVICE FOR DJANGO

Ladislav Louka

Faculty of Information Technology - CTU in Prague
Department of Computer Systems
Supervisor: Ing. Jakub Žitný
11th May 2021

Citation of this thesis: Ladislav Louka. *GraphQL API generating service for Django.* Bachelor's Thesis. Czech Technical University in Prague, Faculty of Information Technology, 2021.

Contents

List of Figures

List of Listings

First I would like to thank my supervisor, Ing. Jakub Žitný, for consultations and advice throughout writing this thesis. I would also like to thank the main author of Simple API, Karel Jílek for consulting its design, implementation and usage with me. Finally, I would like to thank my family, friends, and colleagues from FIT CTU for all other support I've received.

Declaration

Abstrakt

Tato bakalářská práce se zabývá vývojem modulu pro generování GraphQL API ve webovém frameworku Django. Hlavní část práce řeší rozvoj modulu Simple API, který se zaměřuje na rychlou tvorbu API vycházející z existujících definicí v Django. Práce obsahuje analýzu existujících modulů na integraci API do frameworku Django, nejen ty založené na GraphQL, ale také na architektuře REST. Práce také obsahuje analýzu aktuálního stavu vývoje Simple API a na základě této analýzy je vytvořen návrh na doplnění chybějících funkcí, následovaný jejich implementací. Rozšíření se zabývá nejen prvky důležitými pro funkčnost modulu, ale také integrováním zabezpečení GraphQL serveru proti Denial of Service (DoS) a batching útokům prostřednictvím komplexních dotazů. Práce obsahuje popis vývoje dynamické webové aplikace pro testování výstupního API. Výsledkem práce je rozšíření Simple API a webové prostředí Simple API Admin pro prozkoumání a testování vygenerovaného koncového bodu.

Klíčová slova generování API, bezpečnost API, Python, GraphQL, Django, JavaScript, React.js

Abstract

This bachelor's thesis describes the development of a module for generating GraphQL API for the Django web framework. The main part of this thesis is concerned with furthering the development of Simple API module, which aims to facilitate fast creation of API endpoints from existing Django definitions. The thesis contains an analysis of existing modules for API creation in Django, not just GraphQL, but also of those generating REST-style API. From the analysis of the current state of Simple API development, new additions to Simple API are designed, then implemented. Other than functional additions, the development also concerns the design and implementation of security features necessary for prevention of Denial of Service (DoS) and batching attacks through complex queries. Furthermore, a web application for testing of the API is also developed in the thesis. The result of this thesis is the expansion of the module Simple API and web application Simple API admin for exploration and testing of the generated endpoint.

Keywords API generation, API security, Python, GraphQL, Django, JavaScript, React.js

List of Abbreviations

AJAX	Asynchronous JavaScript and XML
API	Application programming interface
CSS	Cascading Style Sheets
DRF	Django REST Framework
FQL	Facebook Query Language
GraphQL SDL	GraphQL Schema Definition Language
HATEOAS	Hypermedia as the Engine of Application State
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
IDE	Integrated Development Environment
JSON	JavaScript Object Notation
ORM	Object-Relational Mapping
REST	Representational state transfer
RFC	Request for Comments
SOAP	Simple Object Access Protocol
SSO	Single Sign-on
UI	User Interface
URI	Uniform Resource Locator
URL	Uniform Resource Identifier
XML	Extensible Markup Language

Chapter 1
Introduction

Evolution of the web from static to dynamic pages, in what is often called Web 2.0 evolution, brought along a wide adoption of web APIs (Application programming interfaces). These allow for asynchronous request-reply communication - eliminating the need to reload the whole website for a change. This allowed users to interact with websites in a much more intuitive way. Many conventions and technologies exist to facilitate such communication between user space and the server, with the most famous being REST (REpresentational State Transfer). With the increasing number of API requests on unreliable connections, some compromises of RESTful API style become problematic, and further technologies have been developed to mitigate them. One such solution is GraphQL, the usage of which is part of this thesis. Currently, support for GraphQL in one of the most popular Python web frameworks - Django - is lacking.

While there are multiple options, none fully support Django's philosophy of minimizing code on the developer side, requiring a definition of each action and defining its resolver as opposed to just building commonly used actions based on already defined models in Django. This deficiency has been the main motivation behind the project worked on for this thesis and the assignment itself. GraphQL is a promising technology and the integration with Django, or rather the lack of, creates a significant roadblock for any developer intending to use one with the other.

Practical part of this thesis is furthering the development of "Simple API". Simple API was co-developed with Karel Jílek, the full list of additions made as part of this thesis are in the Appendix A. The goals of this thesis are to expand the project, which aims to facilitate a fast creation of API endpoints in Django, based on model definitions. The expansion consists of additions to the main project, integrating security features to prevent denial of service attacks against GraphQL endpoints, and to create a web interface for testing and exploration of the API. While there is an existing implementation of GraphQL for Python and Django called graphene-django, it rarely uses existing components offered by Django, such as models (Django implementation of Object–relational mapping) or authentication. It's also missing multiple features competitive frameworks in other programming languages offer, many of them concerning the endpoint security. When using the foundation built by Django, APIs can be built faster and developers can have a basic endpoint faster, allowing them to focus their work on more important parts.

This work consists of six chapters each concerning different stage of development.

Analysis with an overview of how supported REST and GraphQL are in Django, with focus on the most popular frameworks, is inside chapter 2. The chapter also summarizes the current development stage of Simple API and from this analysis it lays down the requirements necessary to implement to satisfy the assignment expectations. In chapter 3, we talk about the choices in designing the needed expansions for Simple API, which have been done for this thesis, and what considerations have been made for each of them. The design is followed by details about the development and implementation done on Simple API as part of this thesis in chapter 4. Testing and distribution of the final product is described in chapter 5. It examines the current test coverage and its possible expansion in the future. Finally, chapter 6 sums up this thesis, what has been implemented as part of it, what further development could be done, and what other outcomes came from it.

Chapter 2

Analysis

2.1 Web APIs

The internet has been created as a system of connected machines that exchange information with each other. However, until the invention of the World Wide Web (WWW) and other standard technologies such as electronic mailing or file transfer, its capabilities have been largely limited. At first the needs of clients were mostly served by basic websites, using plain HyperText Markup Language (HTML), but as the number of users and service providers grew, the need to connect service providers together became obvious [1].

The growth in the commercial use of the internet at the beginning of the millennium also started the rise of web APIs. The ability to use connected structures of the internet allowed several companies such as Salesforce or Amazon to extend their reach and the reach of their partners. The ability to syndicate products across different e-commerce websites and connect customers with multiple service providers at once was best solved with the usage of web APIs and lead to incredible growth for these companies. While at the beginning most web APIs were based either on XML-RPC (XML remote procedure call), SOAP (Simple Object Access Protocol) or proprietary protocol [2], as those were the most common HTTP API technologies at the time, they are currently overshadowed by the widespread adoption of REST (Representational state transfer), which turned out to be a better fit for the job. During the spread of social networks, web APIs started to play even bigger role, allowing developers to build new applications and services with the use of existing platforms [3].

With the use of technologies such as AJAX (Asynchronous JavaScript and XML), web APIs made their way into client-side applications (such as web browsers) rather than just server-to-server communication, creating an entirely new model of how web applications are created. Rather than the server managing and sending entirely new pages to the user with every change, it simply sends Ajax engine to request and parse data from the API and sends data for the engine to use, in order to change web pages on the browser side, reducing the server load [4]. One of the biggest pioneers in this space was Google, first releasing Gmail and later Google Maps, both of which leveraged the flexibility of asynchronous access.

Boom of mobile applications pushed web APIs even further to the forefront, as they became the primary way to include dynamic content inside such applications without the need to update the application itself. Furthermore, a single API could service both a web application and a mobile application, with each interpreting the data in its own way, reducing the amount of systems needed to maintain.

2.1.1 REST architecture

Representational State Transfer (REST) is an architectural style for creating distributed hypermedia systems defined by Roy T. Fielding in his doctoral dissertation [5]. As it's a style, rather than a strict protocol, the doctoral thesis doesn't contain a strict definition for communication between devices, but it's instead a discribed through six constraints (one of which is considered optional), that make API be "REST" and give it the ability to use its strengths. Which are deemed particulary desirable in comparison to the compromises it makes. If an API follows all five mandatory constraints it's often described as RESTful, while those APIs who don't or are just vaguely inspired by the aims and goals of the architecture are being described as REST-like [6].

The 6 constraints of REST architecture as defined in Fielding's dissertation [5, p. 5.1][6]:

- **Client-Server**

  This constraint is REST's designation of a generally used term "separation of concerns" – client and server are to be independent of each other with neither interested in how operations are implemented on the other end. The only requirement for their communication is the mutual understanding of the rules under which the API communication is directed. This is to allow independent development on each side and to allow APIs to be generally independent on specific implementations of specific operations, making the API significantly more portable.

- **Stateless**

  Communication must be stateless in nature, requiring that every request from a client must contain all information for the server to understand and resolve the request. Communication cannot take advantage of any context stored on the server, therefore requiring the session state to be kept and maintained on the client. The main advantages of this choice are clarity of communication and scalability on the server side. Scalability thanks to letting go of the need to store the state of each connection and passing such responsibility to the client. Clarity results from the need to only examine a single request to determine its full content and context. This constraint is a significant trade-off, as for the previously mentioned advantages we also can get problems with incorrectly implemented client-side state management that causes unwanted behaviour by repeating requests or sending them out of order.

- **Cache**

  To improve performance, the client (and optional intermediaries) should have the option to cache server responses for a specified time to decrease the need for repeated requests and therefore reduce latency. This does require the server and the API developer to carefully select how long or if a request can be cached at all, so that the client will not use invalidated data with the conviction that they are up to date. However caching can produce such speed and bandwidth benefits that even basic usage is almost always useful.

- **Uniform Interface**

  Every component within the system should have one singular style of interaction and such interaction should be self-descriptive. Server should report the current state of a resource and the client should send representation of the state in which it wants the resource to be. When used over HTTP that should result in explicit and consistent use of HTTP methods to initiate operations (GET, DELETE, POST, PUT, and more) and the server should use expressive HTTP response codes to explain its behaviour. Finally, REST APIs should be hypermedia driven – described as HATEOAS (Hypermedia as the Engine of Application State), that means that the client should be able to navigate and use the API just by the data it receives in response from the server. There is a number of ways of implementing such functionality, from simply including connected nodes in a special field inside the response, to formalized ways such as *Links* header defined in RFC (Request for comments) 5988 [7] (which was surpassed by RFC 8288 [8]) or formal response syntax such as JSON HAL [9] (JSON Hypermedia API Language).

- **Layered system**

  RESTful API should be capable of being composed from multiple layers that communicate only with those that are only above or below the current layer. System should therefore be able to allow structured infrastructure to be hidden behind the API, allowing for separation between logic, data storage and other parts of the system. Intermediaries can be added into the system to improve not only functionality but also scalability.

- **Code on demand**

  The last constraint is also the least important, which is why it is also named as only optional. REST should be able to allow clients to better leverage its own functionality by allowing download of code, applet, or another executable to be given to the client. While it could allow clients to better leverage things such as better navigation and usability of the API, it comes with some significant trade-offs, and as clients are usually distributed in different ways, the need for such expandability is often unnecessary.

REST doesn't require API to use a specific data format, but it requires abstraction over "Resources". As explained in Fielding's work [5, p. 5.2.1.1], resource can be a: *"… document or image, a temporal service (e.g. "today's weather in Los Angeles"), a collection of other resources, a non-virtual object (e.g. a person), and so on".* While any information that can be named can be made into a resource, it has to be uniquely identifiable throughout the API using URI (Uniform Resource Identifier), which usually corresponds to URL (Uniform Resource Locator). While the resource itself may be constant or changing based on any criteria, the only thing that always must be static is what mapping it uses. Today, many RESTful APIs today use `<Resource> / <identifier>` mapping, with optional further access to related resources by chaining `<related resources>/<identifier of related resource>(/<further relation>…)`.

2.1.2 GraphQL

GraphQL is a query language internally developed by Facebook in 2012 and released in 2015 to the public. It was developed as part of Facebook's shift from mobile web to native applications for iOS and Android. The shift was necessary, as first applications were just thin wrappers around the mobile web and the resulting products were lacking at best [10]. Facebook even stated that failure of their mobile expansion was one of its biggest risk factors is its initial public offering of shares [11]. While those applications were developed, RESTful API were initially used for all their operations, but it soon became obvious that such approach wasn't working. One of the creators of GraphQL, Lee Byron, explained the problems that were encountered to website ProgrammableWeb [12]:

> *"REST really wants to have one model per URL. Recursion is just really difficult to correctly model in that framing, especially when, in order to resolve any of the links between these things, you need to go back to the network. And here we're talking about relatively early days of the smartphone world where the vast majority of people are still on 3G, 4G isn't even a thing yet, let alone LTE. So network is absolutely the bottleneck."*

Problems with the network bottleneck were further reinforced by the type of information that the applications such as Facebook often request, which is recursive data. While browsing your newsfeed, you might want to look at the comments under a post with the interface that Facebook was used to offer, that means first loading the list of comments, then details about their respective authors, their profile picture, the amount of reactions on each of those comments, perhaps even

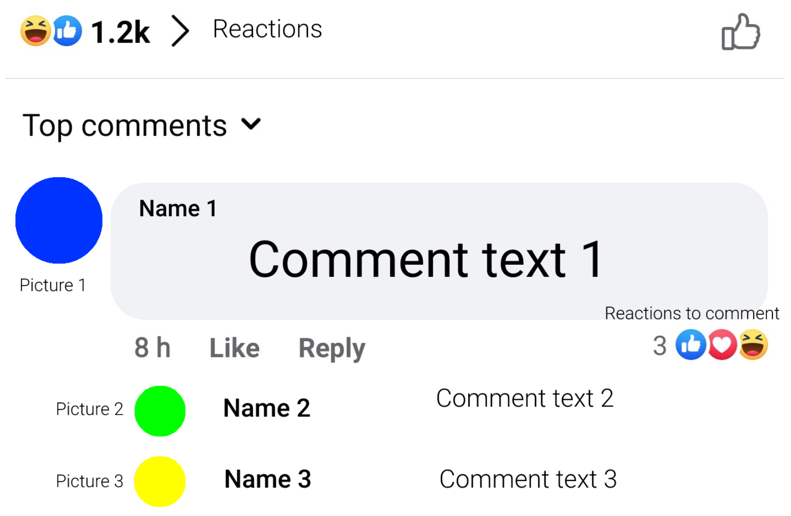


Figure 2.1 Snippet of Facebook mobile application with components highlighted

a list of authors of those reactions. While some of these can be summarized into a single resource and reduce the amount of requests, for it to make any significant impact, one has to intentionally break REST principles.

While internally other solutions were developed sooner, for example, the Facebook Query Language (FQL), which despite its similar name is very different from GraphQL, as it's a language for creating optimized SQL queries, they were hard to use and didn't fully fix the problem. FQL did make querying data necessary to compile different UI components possible in a reasonable time, it was difficult to make it perform effectively, and generally developers at Facebook felt as if it wasn't the right tool for the job [12]. In such background GraphQL was developed, and it was built specifically for the abstractions that were in use at Facebook at the time, and to allow for deep recursion and full graph-based access to data. Developed mainly by Facebook engineers Lee Byron, Dan Schafer, and Nick Schrock [12], it first powered the iOS 5.0 Facebook app release, later spreading through the rest of the Facebook ecosystem. The design principles behind GraphQL is available within the chapter Overview of its specification [13]:

- **Hierarchical**

  "Most product development today involves the creation and manipulation of view hierarchies. To achieve congruence with the structure of these applications, a GraphQL query itself is structured hierarchically. The query is shaped just like the data it returns. It is a natural way for clients to describe data requirements."

- **Product-centric**

  "GraphQL is unapologetically driven by the requirements of views and the front-end engineers that write them. GraphQL starts with their way of thinking and requirements and builds the language and runtime necessary to enable that."

- **Strong-typing**

  "Every GraphQL service defines an application-specific type system. Queries are executed within the context of that type system. Given a query, tools can ensure that the query is both syntactically correct and valid within the GraphQL type system before execution, i.e. at development time, and the service can make certain guarantees about the shape and nature of the response."

- **Client-specified queries**

  "Through its type system, a GraphQL service publishes the capabilities that its clients are allowed to consume. It is the client that is responsible for specifying exactly how it will consume those published capabilities. These queries are specified at field-level granularity. In the majority of client-server applications written without GraphQL, the service determines the data returned in its various scripted endpoints. A GraphQL query, on the other hand, returns exactly what a client asks for and no more."

- **Introspective**

  "GraphQL is introspective. A GraphQL service's type system must be queryable by the GraphQL language itself, as will be described in this specification. GraphQL introspection serves as a powerful platform for building common tools and client software libraries."

However, while these statements do inform us of the philosophy, with which the language was developed, most of them don't help us explain how its own architecture differs from REST or other API models. Thankfully, during many talks done at conferences, its creators have further expressed its goals in a much clearer way. At React-Europe [14] Lee Byron specified that the problems that GraphQL is made to resolve are:

- **Clients express what they need**

  On limited connections with high latency, over-fetching or under-fetching are major problems. Over-fetching means that the client receives data, that can be thrown away, as there isn't a use for it, unnecessarily using bandwidth that is already sparse on a poor connection. Under-fetching on the other hand, means that you are receiving less data then necessary, requiring additional request or multiple requests to obtain the necessary data. Thanks to GraphQL letting clients specify exactly what they need, it fixes both problems. Another positive attribute of this style is that now iterating the API is much simpler than before, new clients simply request the new fields or types specified, while the old usage can stay parallel.

- **Minimize network request**

  Low bandwidth, higher latency, and unreliability were (and in many places still are) significant problems of mobile networks. That further increases the need to limit over-fetching and under-fetching. Users will be unhappy if an application consumes their full data limit - most mobile networks use "Fair User Policy" which restricts the amount of data user can use - and over-fetching results in some of the limit ending up in vain, as the client receives data it doesn't need. They'll also dislike applications that take too long to load when they need to make multiple HTTP requests when data is fragmented - causing under-fetching. Mobile networks often have significantly higher latency and volatility compared to traditional connections, and lowering the time on the network lowers the chance of fault happening.

- **Colocate data needs to views**

  The ability to decompose the request code into fragments that correspond to individual components (views) inside the application allows for simpler iterations of the client-side code and makes the query significantly more readable. Instead of a single long query it's possible to extract each component and iterate it separately and make reuse extremely simple.

- **Strong types**

  As previously mentioned in the snippet from the specification, GraphQL defines a system-specific type system. That allows it to not only offer advanced tools for query development, offer query checking before execution and linting, but also allow to use features of strongly typed front-end languages such as Swift, Objective-C or Kotlin without the need for conversion and cherry-picking in the response.

2.1.3 Architectural differences between REST and GraphQL

The most important difference between REST and GraphQL is in what they principally are. REST is an architectural style, a set of rules for creating APIs with certain attributes, while GraphQL is a query language for an API to utilize. REST focuses on designing long-living APIs that take advantage of the protocols above which it lays, GraphQL is designed for communication over HTTP and optimizes for performance and flexibility. GraphQL doesn't aim to simply replace REST, rather to fill a space currently unsatisfied by it, exemplified by how GraphQL official FAQ [15] page answers the question "Does GraphQL replace REST?" :

> *"No, not necessarily. They both handle APIs and can serve similar purposes from a business perspective. GraphQL is often considered an alternative to REST, but it's not a definitive replacement."*

There are, however, significant distinctive attributes in GraphQL that make it differ from almost all solutions for REST-like and RESTful APIs. GraphQL has been designed for a specific aims and with those goals it has had specific choices made, that result in diametrically different end product. Significant work has already been dedicated to comparing both styles [16, 17, 18], and that is not an objective of this thesis, it is however, appropriate to highlight some differences that result from the comparisons cited.

- **Single "smart" endpoint / Multiple "dumb" endpoints**

  The single most obvious difference between GraphQL and RESTful APIs is how they present themselves to the client. Generally recognied model of REST is the separation of endpoints by the resource they represent. While following the Uniform Interface constraint, this allows for excellent scalability, however, it also means that if the goal is to receive multiple different resource types that are connected, it will require just as many requests. With GraphQL, while the request throughput of the interface is lower (due to parsing and satisfying the query), it makes up for it by requiring a significantly smaller number of requests as everything is handled on the GraphQL Gateway.

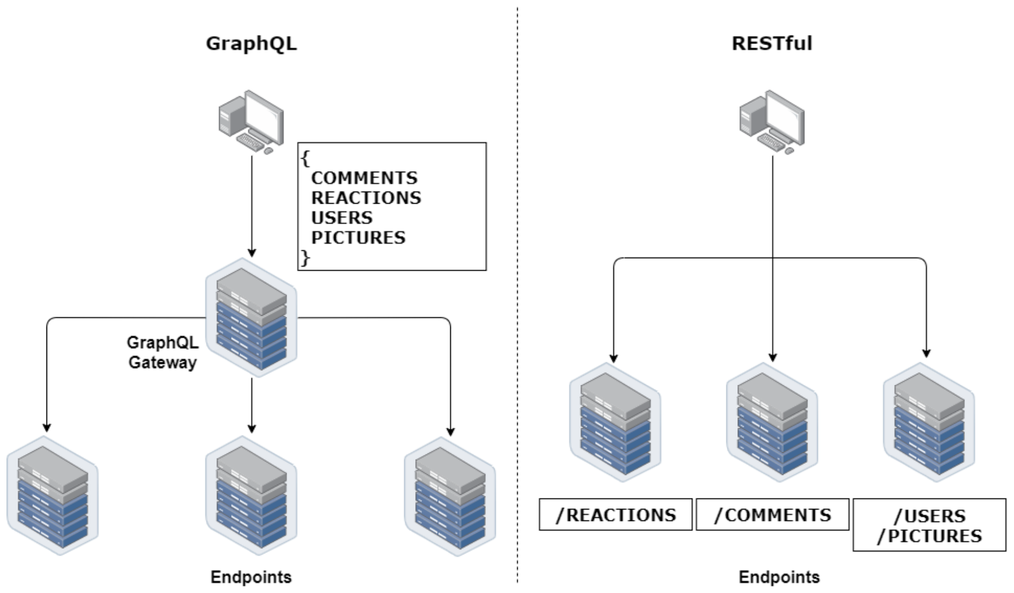


Figure 2.2 Typical architectures for GraphQL and RESTful APIs

- **Introspection / HATEOAS**

  Eight years after his dissertation, Roy Fielding wrote a blog post [19] complaining about APIs calling themselves REST, yet largely disregard many of the core tenets of what REST should look and work like. In the article, he called out hypertext as the most problematic out of all of the constraints. HATEOAS is meant to be used in the API so that the user or machine can obtain choices and select actions. This way the API becomes extremely accessible as all that is necessary for its usage is the knowledge of the URI of the API and standard media types. However, there are many APIs that claim to be REST but don't follow Fielding's constraints. These often try to document the choices and actions in other ways, however, separating it from the API makes automatic access to the API difficult, requiring hard-coded parts and breaking the Uniform Interface and partially Client-Server constraints.

  In place of HATEOAS, GraphQL presents introspection, the ability to access API definitions – called schema in GraphQL – from the client. Through introspection, the client receives system types, operations allowed on the API, actions with arguments and return types, and all additional data for creating a query such as enums (enumerable types), directives or schema documentation. It contains only static information and therefore doesn't pass dynamic information that HATEOAS can, such as if an action is possible on specific object instance. On the other hand, with the GraphQL schema information from the introspection, clients can validate queries and implement a wide range of assistance in writing correct queries with features such as suggestions and linting. In that sense, GraphQL is more of a typical non-RESTful API, in which queries are always written by the developer and at most pieced together by the client application. In summary - GraphQL is made to provide tools to the developer who creates queries with the introspection schema and possible documentation in it, REST with HATEOAS is made to be able to be navigated without documentation just from responses.

- **Caching**

  Design principle and one of the biggest advantages of RESTful APIs is the ability to fully leverage the architecture on top of which they are constructed. While any system can add caching to its functionality, RESTful APIs - when using HTTP - can make use of an existing HTTP cache with close to zero work, and since the caching methodology is very well understood, the use of services such as those that provide DDoS protection, Load Balancing, proxying or caching itself can be incredibly simple. In contrast, GraphQL doesn't rely on many existing standards - and defines its own - therefore it cannot use any of these services directly. Over HTTP protocol, it sends almost all communication with POST requests and includes the request query inside its body, therefore making it almost impossible to reuse the existing HTTP caching infrastructure. There are methods for implementing caching of GraphQL on multiple layers, and many GraphQL clients implement them by default, it still means more work and thought is necessary from the developer of the client application.

2.2 Django

Being one of the most popular programming languages [20], Python has multiple options for developing web servers. Two of the most popular are Django and Flask [20], with the focus of this thesis being on the first. Django is a comprehensive high-level web framework for Python that has been in development since 2005 [21] and it's currently on version 3.1.7 as of writing. While possible to write APIs completely from scratch in Django, many extensions exist to help in quick API endpoint creation. The three most popular on the RESTful side are Django REST Framework (DRF), Django-Tastypie and Django Ninja. As Tastypie is no longer in active development (Last update was on January 6th, 2020 – over a year ago [22] with multiple Django versions arriving in the meantime) further only the other two are compared. For GraphQL there's

```python
class BookSerializer(serializers.ModelSerializer):
    class Meta:
        model = Book
        fields = ['author', 'title', 'ISBN', 'shelf']
```

Listing 2.1 Django REST Framework serialization of Django Model

Graphene-Django, which itself is an extension of Graphene (Python implementation of GraphQL) adding handling of Django requests and conversion from Django to GraphQL types. Before implementing any new functionality, it's important to analyse what these existing solutions offer.

2.2.1 Django REST framework

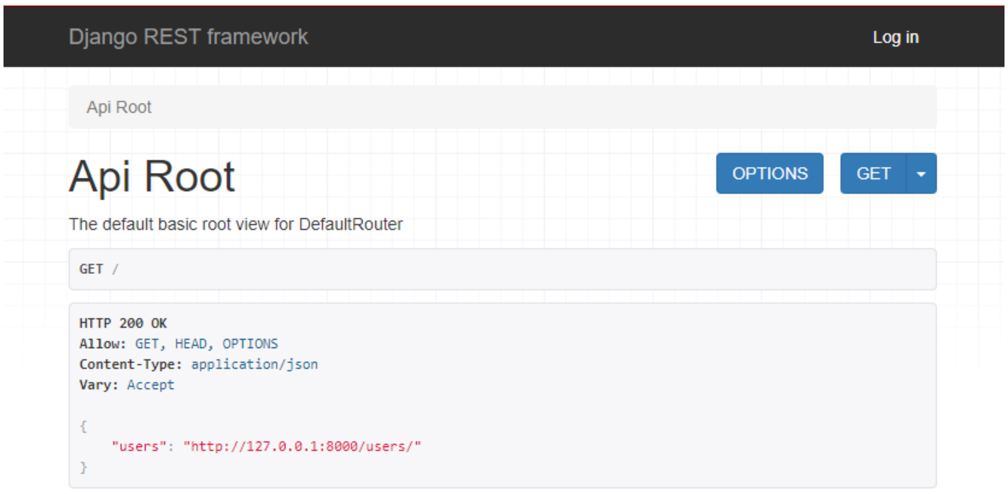
Django REST framework (DRF) is the most popular Django API framework by a significant margin [23]. On its official website [24], it prides itself on these points:

- *"The Web browsable API is a huge usability win for your developers."*
- *"Authentication policies including packages for OAuth1a and OAuth2."*
- *"Serialization that supports both ORM and non-ORM data sources."*
- *"Customizable all the way down - just use regular function-based views if you don't need the more powerful features."*
- *"Extensive documentation, and great community support."*
- *"Used and trusted by internationally recognised companies including Mozilla, Red Hat, Heroku, and Eventbrite."*

Django REST framework is one of the most popular Django modules [25], and is usually the default choice for a developer trying to add REST API to their application. It's closely intertwined with Django, heavily utilizing its architecture. The module is without dependencies beyond Django and its dependencies. It's fast to set up, with a simple way to create endpoints from models - shown in Listing 2.1. As it's the most popular framework, there are many example usages and extensive official and unofficial documentation. There are also several modules further expanding it – previously mentioned packages for OAuth or generation of Swagger or OpenAPI schemas. It does support multiple ways of authentication and multiple permission controls with further possible expansion from the developer or extensions such as django-guardian for object level control.

2.2.2 Django Ninja

Django-Ninja is a relatively new framework for Django, being released in 2020. It's not as strongly built on Django principles as DRF; however, in exchange it promises faster speeds [26]. In terms of functionality with Django, it can create a schema from Django model which can be used for validation or stylistics, but fully customizable custom object classes are currently not implemented – such functionality is only in the proposal stage [27]. Compared to DRF, it uses standardized OpenAPI and JSON Schema generation for machine readability of the API. As a much younger and lighter framework, it however lacks some functionality that DRF prides itself on – such as advanced permissions and authentication with just Django authentication built



Api Root

The default basic root view for DefaultRouter

```
GET /
```

```
HTTP 200 OK
Allow: GET, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

{
    "users": "http://127.0.0.1:8000/users/"
}
```

Figure 2.3 Web UI for documentation created by Django REST Framework

in. It does however have the advantage in its usage of a standardized specifications, instead of creating its own documentation web UI, it uses an independent and already well-tested Swagger UI, that utilizes the OpenAPI standard schema to understand the endpoint and doesn't need additional expansions for such features like DRF does.

2.2.3 Graphene Django

Graphene is a Python library implementing GraphQL specification, Graphene-Django is its extension for better interoperability with specifically Django. In its integration with Django, it allows importing Django models into GraphQL types, however, it expects all actions in the API to be manually defined by the user. For API definition, it uses a programmatical approach instead of writing the schema in GraphQL Schema Definition Language (SDL) as is standard, although the programming syntax is still heavily influenced by the SDL. It provides Views to serve GraphiQL (web IDE for exploring and testing the API) and handle the API queries. It does support basic authorization, although its implementation is rather basic [28], requiring additional code to truly be useful, as it, by default, can only distinguish between logged in and anonymous users. It supports multiple Django modules such as `django-filter` and has some of its own, for example, `django-graphene-extras` that provides pagination for lists, filtering, and more. One significant bonus for deployment is the ability to reuse Django REST Framework serializers to define actions, therefore giving the ability to quickly add GraphQL endpoint to an existing DRF application. However, the package is still a rather thin interconnect between Django and Graphene assuming the knowledge of the latter rather than the first. It does require a significant amount of code to set up even a basic API with multiple models and actions, with most of the code being rather unnecessary – breaking one of Django's principles – require as little code as possible.

2.3 Requirements

From the analysis we can now determine requirements that the Simple API package has to fulfil. Its focus should be to primarily facilitate a quick setup of basic operations over the models selected by the developer, rather than require writing of all basic *CRUD* (Create, Read, Update,

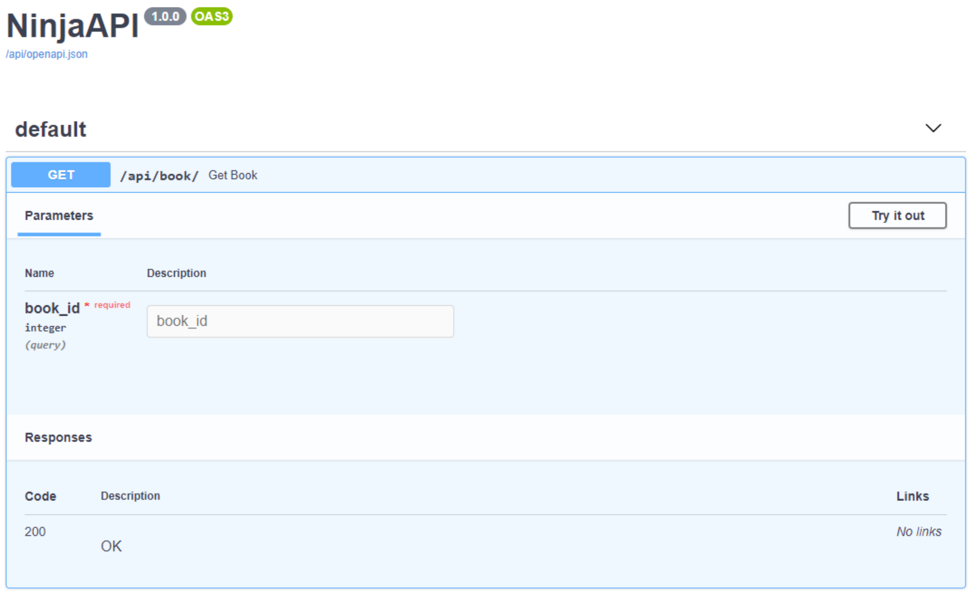
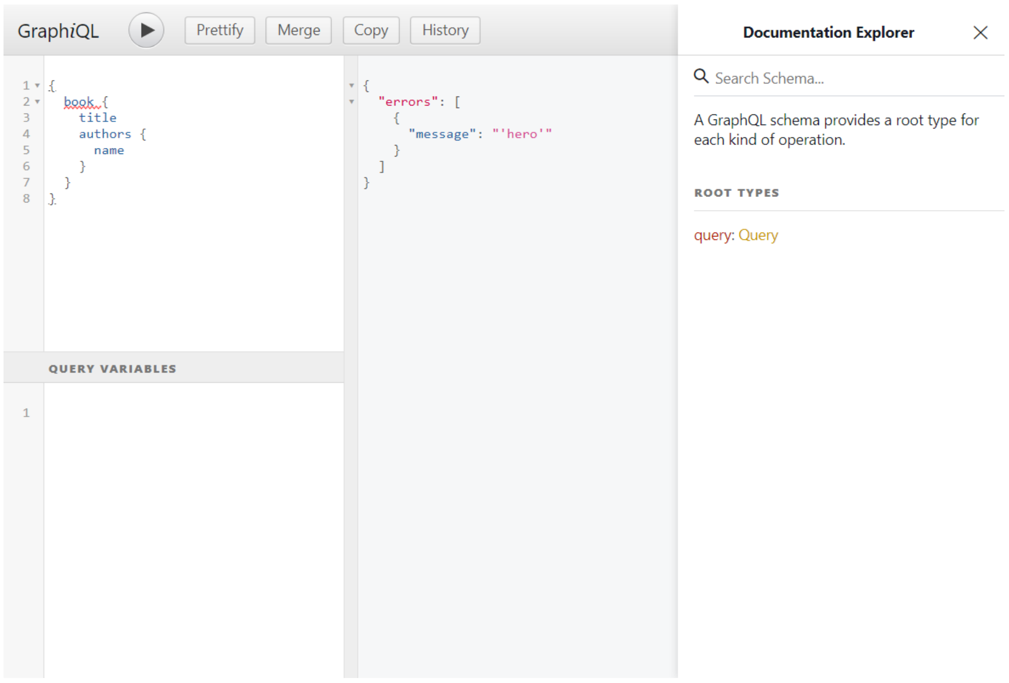


Figure 2.4 Web UI for documentation created by Django Ninja



```
{
  book {
    title
    authors {
      name
    }
  }
}
```

```
{
  "errors": [
    {
      "message": "'hero'"
    }
  ]
}
```

A GraphQL schema provides a root type for each kind of operation.

Figure 2.5 GraphiQL - IDE for testing GraphQL API, provided by graphene-django

Delete) operations from scratch. Besides GraphiQL, custom web interface should be available for exploration of the API, to showcase the result of the generation. While GraphiQL is excellent for writing and testing queries, its schema browser doesn't reveal every detail of the API unless it's very well documented, and documentation shouldn't have to be the focus for simply allowing development of the client. To allow the usage of the schema for automatic frontend generation, there has to be a separate schema to inform the client whether or not the action can be done, why it's denied, and have the option to hide an action, based on the current authorization. It should add functionality such as pagination and filtering for list type fields. For security it should implement at least passive query analysis of depth and complexity and enforce reasonable list limits so that pagination cannot be overridden, therefore preventing most denial of service attacks through overcomplicated queries.

2.4 Simple API

Now that we have outlined the requirements we have for Simple API we should look at what it currently is and if any of them are already fulfilled in the current implementation. Simple API is an extension to Django and Graphene-Django, aiming to provide a *fast-to-setup* API endpoint for already defined Django models with as little code as possible. It provides the ability to include a model in an API with basic operations on it with less than 10 lines of code. While only GraphQL endpoint is currently supported, much of the code is written to support possible expansion to other architectures or query languages. Other than just generating an endpoint from the model, Simple API provides additional functionality that Graphene only supports with other extensions, such as paginated lists, filtering, datetime types, or permission management. When Simple API extracts information from the model, by default, it generates actions for:

- Creation of a new object (Create)
- Deletion of an existing object (Delete)
- Requesting a specific instance of an object (Detail)
- Getting a Paginated list of objects (List)
- Changing values in the fields of an existing object (Update)

As part of the generation, it also adds the necessary support in schema type definitions to allow the functionality of these actions. For direct types corresponding to models, that means the type containing all fields and relations (unless specified otherwise). Also added are methods marked by Python decorator `@property`. Additional types are also defined to support paginated lists and filters. Simple API defines an additional layer over Graphene Django, adding its own definition of types and objects which are converted to be understood by Graphene.

As we can see, while some requirements for Simple API are already fulfilled, there is still a significant amount of work that needs to be done. Expanding Simple API is important, but work must also go towards the creation of a new web UI that will serve as part of the documentation of what Simple API does. The web application will provide a showcase for its functionality and create an example of its usage. These are the requirements for the expansion that's part of this thesis:

```
# Imports ommited
class Book(DjangoObject):
    model = BookModel

schema = generate(GraphQLAdapter)
patterns = build_patterns(schema)
```

Listing 2.2 Simple API code generating operations from a Django Model.

2.4.1 Requirements for additions to Simple API

- **Implement Validation**

  Simple API currently lacks any option to add validators to actions, other than using either model validators from Django, or implementing validation in the resolver functions. The first option is not usable, if the validation is to be applied over more than a single field or only wanted to be utilized for the API, not for other forms of ORM access. The second could lead to nonstandard behaviour and Django validation of model fields would still be done and raise different errors. Therefore, Simple API must implement its own validation that would, in addition to user-defined validators, consume Django model validators and stop the action before they would be triggered.

- **Expand custom schema**

  Simple API currently implements a basic custom schema to provide information not contained in the standard GraphQL introspection schema. It offers information about the primary keys of objects, and what actions can the current client make use of. To function as the sole source of data for UI creation, it needs to contain more information about the types – all fields, their types and if they have a default value – and about actions – what arguments they expect, whether or not they mutate the data, and need a mutation call to be executed and what type do they return.

- **Query DoS scanning**

  GraphQL generally suffers from similar security problems as other APIs [29] – possibility of SQL injection, flawed authentication or validation, and Denial of Service (DoS) by multiple expensive requests or extreme amount of requests in general. However, GraphQL can also be vulnerable to a problem when a single but highly intensive request arrives to the endpoint. The endpoint can also be vulnerable to a type of batching attack [29]. Batching attacks allow for brute-force multiplication through allowing the attacker to include multiple queries in a request, therefore giving them the ability to, for example, try multiple possibilities when trying to guess a password with just one request. It also allows for object enumeration even when access to a list of objects shouldn't be available. Multiple methods can be used for mitigating such problems – but not all can be used without knowledge of the full API schema. Passive query analysis with multiple detection vectors, such as limiting depth, number of actions, or selections, is a method that allows for high flexibility and can catch most malicious queries.

2.4.2 Requirements for web UI

- **Present all actions grouped by the object, which they are children of**

  Simple API allows actions to be defined as either connected to an object or separately. While it does have a multiple uses in internal operations (for example, giving the ability to define return type of an action with string "`self`"), it doesn't directly reflect into the generated GraphQL schema (other than automatically generated names for the created functions). If shown correctly, it can be beneficial for exploration of the generated API, as it allows actions that only affect one object to be explicitly separated.

- **Allow testing of filtering and sorting on object instance list**

  Through the web UI, the developer should be able to access a list of existing types and after selecting one, the UI would present a list of instances, that can be limited by all filters defined and be able to be sorted by any single field or multiple at once.

- **Present permission status and allow for quick changing of accounts**

  Each action should be marked depending on the current permission status. Disallowed actions should be visibly different to the allowed ones and present the denial reason to the user. For quick testing of the permissions, the web UI has to allow for quick user change without full login every time the tester wants to switch accounts.

- **Present functions on instances of objects**

  Some actions can require an object instance to be called upon – usually those directly interacting with existing instances and often expecting a primary key value of the referenced object. While these actions can be shown just like the general functions – differences in permissions might don't allow a specific user to call them – for example, a user editing another user's data. The UI should therefore show the actions that can be called on each instance and if they aren't allowed to call it on a certain object, show the user reason, as specified in the previous item.

2.4.3 Validators

We already explained why implementing validators on the API would be useful compared to other currently available solutions, in this section we'll examine how such functionality has to work to be beneficial. Simple API already has implemented permissions and as the feature is similar in its basic principles – deny action execution if the condition is not met – we should examine the implementation and see whether or not it can be reused or modified to work in this role.

Permissions in Simple API are a separate class of objects that implement a method named `permission_statement` and inherit from class `BasePermission` its `has_permission` method. While the logic of a specific permission sits in the statement method, the second one controls the resolution with respect to inheritance, trying to execute the method `permission_statement` on all of its parent classes. Inheritance of permissions is useful, as we can easily implement multiple security levels, each inheriting the conditions of lower levels. While the separation of execution and logical statements are useful even for validators, there isn't equivalent use of several levels of validation, and any such functionality can be achieved with logical operators (and, or, not/negation). However, there's no significant cost in implementing such resolution and there might be a specific usage that is enabled by it, therefore implementing it might be beneficial anyway. One necessary functionality for validators that permissions don't have is the ability to be bound to an argument, only the action itself. While permissions don't have a use for such functionality, and it can be implemented on the action level once arguments are passed, it does result in better readability. Especially with simple validators that can be assigned directly to arguments, and receive the values directly in arguments, both creating and understanding such validators becomes much simpler.

Another existing implementation we should examine is how Django handles validation on its model fields [30]. In Django, validators are functions (or callable classes) with a single argument – value to be checked – and either return `ValidationError` or nothing. As we want to check our input before the model would see it's not correct, we have to also have the ability to import Django validators and add them together with API validators. The function should also be exposed to the developer, in case they would like to reuse the validator they already created for a model field.

There can therefore be two different types of validators – action validators and field validators. While all are to be executed at the same time, the difference is in what data is passed into the validation statement to allow for simple reuse on multiple fields.

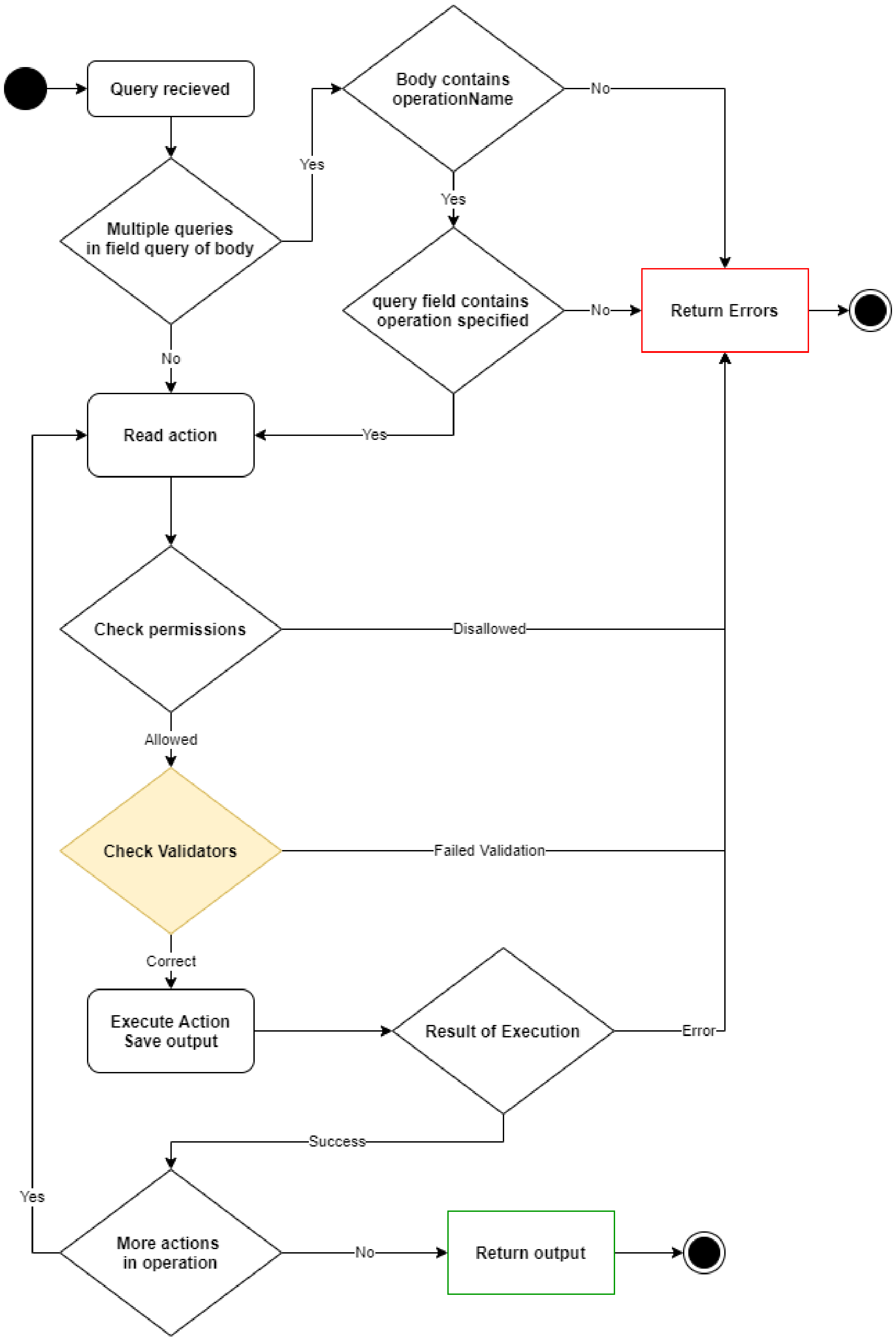
Figure 2.6 shows the expected placement of validation in the execution of a request. It must be evaluated before the action is executed and only after the permission check has finished to prevent possible testing for attributes of objects that we shouldn't have access to. Compared to some other implementations of GraphQL, Graphene explicitly returns only the first error it comes upon, ignoring all other valid content. Thanks to this behaviour, if the request contains invalid or otherwise faulty queries, we don't have to make sure the rest of the query is correct after the validation fails.

2.4.4 Metaschema

One of the important advantages of GraphQL is its introspection, which gives the ability for clients to explore the schema. However, while it's made to be understood by machines, it wasn't designed for the generation of UI elements by itself, thanks to its priority for assisting in development, not in production. For such use it has several disadvantages. Primarily, it cannot be easily modified in Graphene, that means that information such as permissions or hiding types cannot be implemented without significant modifications of Graphene. Additionally, its hierarchy isn't designed with separated types as Django models in mind, actions are grouped in either "Query" or "Mutation" types rather than by the objects which they are connected to. Simple API does separate the actions around types, but it can't be fully reflected into the introspection schema.

GraphQL introspection schema also isn't meant to contain strictly runtime information such as current permission status, it's purpose is to inform about the constant rules to parse queries with. For the reasons of permissions, Simple API already contains so-called "Metaschema" which returns basic runtime information – action list and their current permission status and information about objects - their primary key and actions over them. This layout groups actions by the object they are connected to, which does allow better discoverability for the actions Simple API makes, but to be fully sufficient for automatic generation of interfaces, it needs to be expanded. Grouping actions by objects rather than operation type means that the information must be included in the response as a field adding the need for boolean "mutation" field. To allow full separation from the default introspection, it must contain information about parameters and return types. When specifying a type, GraphQL conventions can be used for simple understanding. With such information, the client can dynamically create queries and understand its return value.

Proposed schema for the new layout is in Figure 2.7. Values marked with green background are already implemented in the current Metaschema. Two new objects are added to fully replace introspection, Type information provides information about new types added to GraphQL by Simple API, and Field information provides information about object fields or action parameters. Object information is already implemented and provides data about individual instances of objects and it's implemented as a field of the instance. Action information currently provides just permission resolution and has to be expanded with information about action parameters, type returned, and whether the action needs to be inside of a mutation to execute. Metaschema objects are linked to provide child information with just a GraphQL selection, the only exception being "Type name" inside of Field information. While it does provide a connected to type information, it could also connect to what GraphQL calls Scalars – types specified by their direct value – which are not always included in Type information.



■ **Figure 2.6** Workflow for queries in Simple API after validator implementation

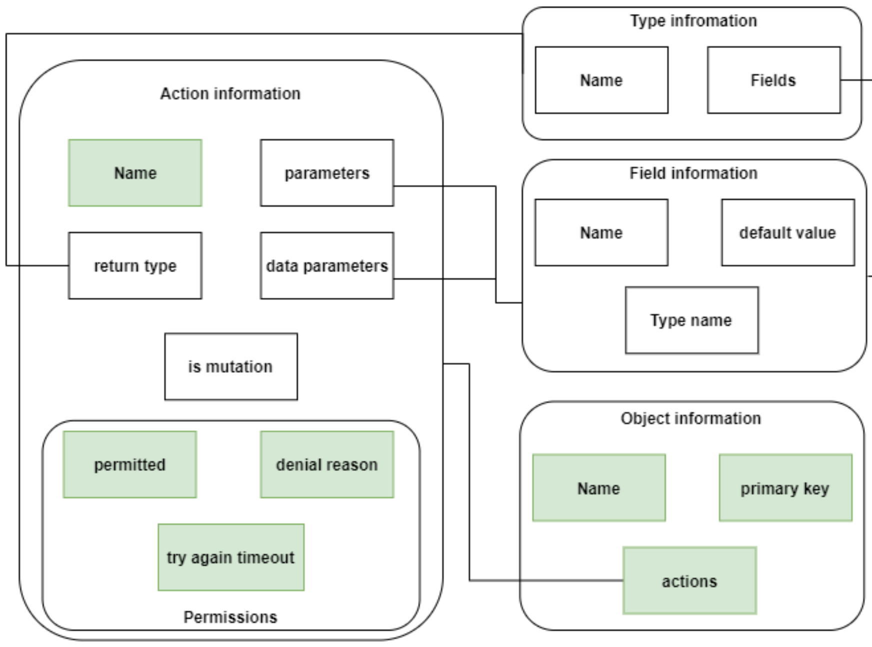


Figure 2.7 Proposed Metaschema layout

To minimize the possibility of naming collision with user-defined actions, Metaschema should use the reserved naming space of "`__`" prefix (`U+005F:LOW LINE` - called simply underscore in the specification to be encoding independent) [13, p. 4.1]. While this could cause problems when using clients that rely on introspection and strictly require the namespace to be empty for all but basic introspection types and actions, in production introspection is recommended to be disabled [29], so such problem shouldn't matter when just Metaschema is being used.

2.4.5 Preventing malicious queries

Denial of service is an attack which aims to disrupt the stability and availability of the API. While the server hosting our Django Application with GraphQL endpoint can be attacked in several different ways, the security of other components is not of concern here. Common way to prevent the overwhelming of a generic API is to limit the number of requests that can come in from a single client in a short time frame. While this is useful for a GraphQL API, it should be implemented on a higher level (for example, the proxy which the Django application would sit behind) not inside our module, because checking on a higher level means less resources are consumed. Rate limiting on request level is useful, but as many actions can be fit into a single request with GraphQL, it can only help so much. For true prevention, GraphQL endpoint must check the incoming query before it starts to evaluate it.

Open Web Application Security Project (OWASP) recommends multiple different ways how such vulnerabilities can be tackled, mentioning 5 different ways to limit vulnerability to these attacks [29]:

- **Add depth limiting to incoming queries**

  Thanks to the ability to nest recursive selections within GraphQL queries, multiple separate resolutions might need to be made to fully satisfy a single request. If there are any types that can create cycles (which most APIs will have), the attacker can just repeatedly move through it, requiring that many object resolutions. Limits must be therefore made to enforce the query depth, preventing multiple very nested queries from overwhelming the application.

- **Add amount limiting to incoming queries** – and – **Add pagination to limit the amount of data that can be returned in a single response**

  Not just high amount of nested selections can mean too many resolutions - requesting a large number of items from a list, or simply making an extremely large query can also overload the API, if fetching of the information is taxing. Simple API already expects that no field that would return an unregulated number of items would be exposed to the client thanks to its PaginatedList type. Paginated list by default returns just a set number of items (with the specific number requestable in an argument), but nothing stops the client from simply requesting 99999 items or however many is needed to overload the API. Such misuage must be checked and prevented when implementing limits on queries.

- **Add reasonable timeouts at the application layer, infrastructure layer, or both**

  OWASP suggests implementing a timeout to actions, so that even when a query gets through the check before execution, it's stopped before it fully executes. While these limits are usually easy to implement, they are reactive, and often don't stop the query before it has already consumed significant resources. Sadly, thanks to the way Graphene executes queries, without significant modification to the resolver, this is not possible to implement without significantly complicating the execution with multiple threads. Timeouts can still be implemented on the infrastructure level if necessary, or inside the resolver function if they are worth it for the specific operation.

- **Consider performing query cost analysis and enforcing a maximum allowed cost per query**

  Considered the most effective, but also usually the hardest to implement [31], the last suggested method is assigning costs to fields or types, and validating the query before execution by filtering those, whose total weight exceeds the specified limit. In addition to assigning cost to individual fields of objects, assigning costs to actions might be even more important, as selecting an object based on id, which can be quickly fetched from the database might be easy, but attempting to find an object based on a complex condition might mean a significant load on the server. Query cost analysis is generally not recommended for implementation as it might be unnecessary with depth and amount limiting [31]. However, as the implementation of timeout has been deemed difficult to achieve, any passive analysis will be useful for preventing overwhelming of the API.

Another important question when designing security features is where in the execution order can they be implemented. Graphene-django has built in support for two positions where additional functionality can be added to GraphQLView – called `backend` and `middleware`. Backend is called when a query is forwarded for resolution, the default backend is provided by Graphene itself. This is the position, which two existing implementations of query scanning for Graphene already utilize. They are named "graphene-protector" [32] and "secure-graphene" [33], both are fairly simple, implementing depth scanning with protector also counting the number of selections in a query and adding a Django integration with the option to save limits in Django `settings.py` file. Ability to analyze a query before its resolution is what we want to achieve, but when we resolve the cost at the backed, we don't have access to the context of the request, such as the currently logged in user or the request origin. We couldn't therefore custom limits, which would differ with based on the currently logged in user, nor implement IP banning or rate limiting. `Backend` therefore isn't the best place to do the query classification. `Middleware` has access to the full request context but has one fatal flaw – it's called every time part of the query is executed, and only after it's resolved. Therefore, it's not useful for us if we want to stop the query before it can do its damage. Neither place seems well fit for such functionality then and we will need to modify some part of Graphene to fit the query analysis in. Disabling introspection is generally recommended on the Middleware level [34], but other precautions must be put elsewhere.

```python
# urls.py
from django.contrib import admin
from django.urls import path

urlpatterns = [
    path('admin/', admin.site.urls),
]
```

Listing 2.3 Adding Django admin to URL configuration

That place is `GraphQLView` method `execute_graphql_request`, which is called whenever a response to GraphQL request is to be made. As both the middleware and backend are invoked from it, it has access to all information that is known and it's called before execution, therefore allowing us to stop the query before we start to resolve it. Additionally, as `GraphQLView` is created by Simple API, we can pass arguments such as a weight schema or limit values we would like to use.

2.4.6 Web User Interface

Significant work must also go towards the implementation of a web application using the Metaschema. Ideal candidate is a web UI for testing and exploration like Django REST Framework or Django Ninja provide. Such application would also spotlight the separation of actions by objects. As we are developing an extension to Django, it must be served by it and therefore it has to be implemented as a View. To make it as easy as possible to install, it must be implemented as a single view so that the installation takes just install with `pip` (package manager for Python), adding the application to `INSTALLED_APPS` setting and setting up the URL, just like the Django admin extension works.

Sadly, building the UI with Django templates is out of the question, as the UI needs to work based on the responses received from the API, which are not known to our application. We could generate them and pass them to the template, but then they wouldn't showcase the ability of automatic generation based on the Metaschema. We can therefore only serve the foundation and the UI has to work through asynchronous requests (AJAX).

While we could work with plain JavaScript as our scripting language, more suitable would be the usage of a frontend framework which supports components. Components are chunks of code that help in separating and reusing different elements over a webpage. There are several JavaScript web frameworks supporting such functionality, the most popular among them React.js, Angular and Vue.js [20].

Important part for the functionality is the ability to quickly log in as a user or change accounts for testing permissions. As logging in multiple times to just switch the user would be unnecessarily slow, we can support password-less login. The web UI is meant to be used before production in the development stage, so insecure by design shall be an acceptable compromise, and if the developer forgets to disable the web UI before deployment, we can check if Django is in Debug/Development mode, and disable the UI.

While some queries can and have to be hardcoded, such as the Metaschema query, the interface should be able to generate selections for action returns by itself. To start it's enough if the API generates just direct fields with minimal depth, because it should represent the minimal viable query, additional depth, or further expansion such as fragments can be added manually by the user.

Chapter 3
Design

3.1 Validators

In the previous chapter we have mapped out the needs and requirements for the implementation of validation. Developers can add it to an action or argument of an action and every time the action is invoked, it checks if the validator's condition is satisfied. To give the developer ability to see more than just the value of the field, validators have to accept not only an argument for value, but also one for the request context. As we want to forward our validation to Graphene, we must integrate validation into all resolution functions. The implementation can therefore be separated into four parts – creating a validator class, allowing attaching validators to types and actions, creation of the validation function, and adding the validation function to the function which resolves the action.

3.1.1 Generic validators

Base class for the validator should be fairly simple – just three methods, one for the validation statement which the developer creates, one for checking the statement and lastly the definition of error message. For logical operations, additional classes must be created, as they have to accept other validators as arguments and define their own validation functions.

To allow the resolver function to test for the validity of the input, the validators must be forwarded to the `TemplateFunction` class, which is then converted into Graphene action call. While passing a list of validators would work, a much simpler and cleaner way would be to contain all logic into a single object. We can even make that object directly callable, so once we know what validators should be applied on each action resolution, we can simply add one line into the resolving function that stops it if one of the validation steps fails. Such implementation is clearer, with validation function logic kept close to the validators.

Thanks to the fact that we're working in Python, we can simplify it even further, and instead of creating a custom object, we may simply dynamically create a function and use it. Python will make sure that the inner function keeps the context of the validators for us [35]. We must have the Validator assigned to its respective field and then correctly pass it the value for validation, therefore keeping a dictionary for the field and a list of its validators. Dictionary is a collection datatype that stores its values in key-value pairs, in this case the key is the field, and the value will be an array of all validators that are assigned to it. Since action validators don't have a specific field connected to them, we need just an array (list) of them.

To collect the functions, we add a collector in the action initializer, that gathers all validators from the parameters in its arguments and saves them into an attribute, then passes it into our

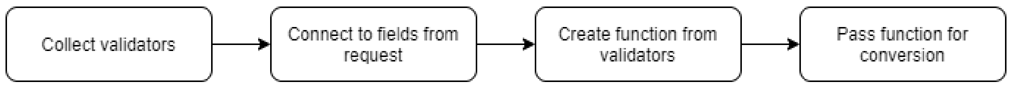

Collect validators
Connect to fields from request
Create function from validators
Pass function for conversion

Figure 3.1 Checkpoints in creating validation function

function generator, adding the generated result in the TemplateFunction to be checked with respect to Figure 2.6.

Simple API sorts its object and action handling into two separate subfolders/submodules, one for generic types named `object`, and the other for the ones that originate from the connection to Django, named creatively `django_object`. Since the basic validation functionality is in no way directly connected to Django, all components described so far go into the generic type.

3.1.2 Django specific validators

In the second group, we have a validator implementation that's exclusively connected to Django. In the analysis, we've determined that one of the reasons to implement validators is the separation from default Django model field validators. On the other hand, thanks to the way Graphene currently handles exceptions, we don't want our actions to be stopped by field validators as it can expose the error stack and make for too explicit error messages. Since we don't want that to happen, we must create a wrapper around the Django validator which checks it before Django itself does and if `ValidationError` would be raised, consumes the error, and returns a generic error message instead. We shall do that for all validators in the field when Simple API converts them to its own equivalencies.

Another validator needed for Simple API to correctly work with Django is the validation of foreign keys. Foreign key field in Django is a representation of many-to-one relationship between models [36]. In the database, it results in a field representing id of the target object from the referenced model and a foreign key or other equivalent constraint [37]. If the constraint would be violated, Django raises IntegrityError. However, the check is not done as a validator but raised later when the database is accessed. To avoid returning this error, we can use our new implementation and create a foreign key validator, which will check if a given value exists in the referenced model. Then we simply add this validator to all fields that end up using such relation. If the fields have the validator attached, the action will be stopped before any resolution is attempted, and we can return a better error message that's easier to understand.

3.2 Metaschema

First step in building additional schema is finding all information we're intending to distribute about the actions and objects. Schema about object instances and actions is already implemented, so we can get inspiration from its current implementation and then expand it to fit our need. Its type information for GraphQL is defined with Simple API objects in `meta_types.py`, to fulfil our requirements from the analysis simply expanding them is straightforward. We do need to find where we can obtain the data to populate our structures with. Current implementation for generation collects all objects from a class `object_storage`, which is where Simple API stores all information that is then converted into Graphene schema. Information about types is simply collected by iterating over its fields attribute.

For fulfilment of the requirements, we must expand the current implementation of action info with new information about argument fields, whether it's a mutation operation and its return type. Each of these can be simply extracted from the Simple API object in `object_storage`. Object information can stay unchanged, and we must define new type information that contains information about the types and their fields. Its implementation should, however, be quite similiar to how object information currently works. Once all resolvers are defined, we assign them to a Simple API action and then add them behind the scenes before we generate the GraphQL schema for Graphene.

3.3 Query scanning

3.3.1 Depth and width (amount) scanning

Depth and amount of selections are simple to scan for in a GraphQL query once it's parsed. In `GraphQLView`, where we decided the security features should be located, it means using a backend method called `document_from_string`, which parses the query for us. From then on, it's a simple iteration over nested objects and their selection. While counting individual nested elements is quite easy with just recursive calls, we run into problems when we try to count the number of fields within the query. Primitive iterations would count the number of fields written in the query, but not the number of fields accessed and returned. For example, the query in Listing 3.1 has only 5 fields in the query and just 2 final selections (`count` and `__str__`), but could return as many as 101 fields back to the user - up to 20 book objects each with a maximum of 5 bookmarks. We therefore must calculate not only the basic sum of selections, but also multiply them if they are requested in a list. For that, we must first be able to extract the limit set on the list field, if there is one and then pass it further on. Seems easy enough, the limit is simply an argument for the field, which Graphene has conveniently already parsed for us. However, how do we recognize that it's a list if we don't have a limit set, and how do we know it's really a list and not just a different field with an argument called "`limit`"? We have no ability to know from the information we we recieve the query and we must therefore get the information beforehand for such recognition. We can collect the information when Simple API converts its own objects into Graphene schema and then pass it into our view. However, when we're already collecting information about fields, why not take it further and let the developer assign individual weights to the fields instead of forcing it to equal one, as a simple amount count would. That's where the query cost analysis we mentioned previously comes in.

3.3.2 Query cost analysis

Since we've decided to add a full field cost analysis, we can also assign weights to actions, if, for example, one way to access an object is more difficult than another. Adding the option for

```
query {
  BookList{
    count
    data (limit: 20)  {
      bookmark_set {
        data (limit:5) {
          __str__
        }
      }
    }
  }
}
```

Listing 3.1 Sample GraphQL query with paginated lists

assigning cost to an action is simple, just adding another argument that is then assigned to the object attribute. For fields, the assignment is much less intuitive to implement, and the best option lacks in discoverability. If we were only working with simple objects manually created by the developer, it would be just as simple as the assignment in action, but for objects we have to add the setting to an attribute, as they are defined, not called through their constructor.

Once we have assigned the values, we can collect them, and also save the fields that would forward us to another type or the list type. Since this style of query cost analysis is made specifically for GraphQL, it should be connected to its Simple API adapter, so that if there is ever a different analysis implemented for other API styles or query languages, it can generate its own weight schema.

Now that we have a way to collect the weight values, we can start thinking about what's the best way to calculate the query cost from them. The first and simplest idea we can come up with is just adding up all actions and selections together with respect to the list size, so that selections inside a list get multiplied by the number of items requested. Let's say, we assign costs to the query in Listing 3.1, 10 for the action BookList, 1 for the count as that's easy to obtain from the database, 2 for the `bookmark_set` in Book and 3 for the string representation of the bookmark – we would have a query cost of 351. We could increase the importance of the action by not just adding it but multiplying by its cost, and come to a quantified difficulty of 3410, but such change only highlights the major problem of query cost analysis – it's hard to quantify the difficulties and not having them be too abstract to understand and use. Simpler algorithm is easier to imagine and test, and generally easier to use while providing the same protection. Another reason why pure addition is preferable over multiplication, is that once the action is executed, the selection will almost always be easier than the action itself, multiplication would therefore be adding priority to the less demanding part.

Because the query analysis is so complicated to get right and requires a significant amount of experimentation, it's important that other limiters will also be available to the developer. Since we can now understand what fields are paginated lists, we can enforce that only queries with reasonable limits on such fields get executed and the depth limit is still there for extreme nesting. Finally, after all these query limits are implemented, the settings should be concentrated into `settings.py` so that the configuration is concentrated on each field.

3.3.3 Other security features

In the previous chapter, we talked about multiple different preventive tools that can be implemented, and we already designed query depth analysis, width analysis and query cost analysis, so what else is there to do? We would overstep the role of the module if we tried to scan for vulnerable code. The module can still implement one additional security features other servers provide, we can close the information pipeline that's the GraphQL introspection. Since the generation of introspection fields and actions is out of our control, we have to filter the query that either contains the queries containing requests for introspection or filter the returned values. The option chosen for this thesis, as it is already proven to work [34], was to filter out the results. That means we must do to close it is recognize when the query from introspection is called, and instead of returning the schema we return None, error, or anything else, just not the schema.

First instinct for filtering the queries might be to search for everything starting with the previously mentioned reserved substring "`__`". However, we would instantly run into multiple problems. One that we might not care about is the inability to use the Metaschema, which also uses the reserved prefix and is large part for the result of this thesis, so it's not viable. However, if the developer doesn't use it, they might still be interested in it, as the namespace also contains a special field called `__typename` which is an introspection field that's present in all types and returns the GraphQL type name. Its importance might not be immediately obvious, but the reason disabling it is not wanted and can cause significant problems, is that it's the basis of multiple available caching solutions for GraphQL, such as the one provided by Apollo Client [38]. While listing all other fields is possible, it only really makes sense to filter those that could expose something usable, mainly introspection objects `__schema` and `__introspection`. For simple filtering, we can implement solutions similiar to existing ones as [39] or those mentioned in from [40].

3.4 Web UI

Out of the multiple frameworks presented in the analysis in 2.4.6, React.js was selected, partially because it has the same roots as GraphQL (it was also developed by Facebook) and partially thanks to the recommendation from my supervisor. However, as we're using it with Django, we must respect its Model-Template-View architecture. With the usage of `INSTALLED_APPS` loader to install our extension, we can use this architecture, as Django treats our application as it would the one the developer creates. Model part is very simple – we don't have any - the interface itself doesn't store any data, any interaction with the model is done through the API. The whole reason for creating the user interface as an example is that it uses just the Metaschema as its data source, and the schema is already served by another View (`GraphQLView`). We do need to create one to host our web application and to facilitate logging in and out of Django authentification to fulfil the requirement from 2.4.2. Just like Django Admin is modifiable [41], if there's a possibility to allow the developer to customize the functionality, we should allow it, but the focus has to be on making it easy to install and use.

Web application itself can be presented as a single webpage, with all logic facilitated by React.js and state kept as an internal variable. This approach largely bypasses most of the strengths of Django templates, but as we're generating the UI based on the API responses, not variables known to Django templates cannot be of much use for us anyway. Thanks to a lack of proficiency in regards to website design, user interface framework should make the application both better looking and more useful. Bootstrap framework is a collection of JavaScript and CSS, that offers simple HTML classes that are preconfigured for use. Because Bootstrap is not made primarily for React.js, it doesn't follow its structures, there isn't the concept of components as JavaScript entities in Bootstrap, whilst the whole React architecture builds on it. Rather than having to declare separate components each for different Bootstrap component, the extension React-Bootstrap was selected. The module takes the original JavaScript from Bootstrap and rewrites it with React components and conventions, and was used to simplify the work on the interface.

3.5 Documentation and distribution

While not mentioned in the analysis, both documentation of the project and its distribution are important parts of the thesis. It should not only provide reference to different options and arguments, but also contain information on how to set up both Simple API and the Web UI to test it with. Large amount of information and documentation has been written and is in `README.md` of the project, however, it doesn't cover all arguments and attributes, that exist, and the style it's written in makes to hard to read. Many projects in Python have their documentation made with Sphinx framework, but Simple API doesn't have a use for automatic documentation generation and other advanced features that it offers. Documentation for this project should prioritize the ease to edit the API over being comprehensive, because the project will probably still go through significant changes. For the simplicity of writing and editing the documentation, instead of Sphinx, another popular framework MkDocs is therefore preferable. It uses Markdown syntax to simplify and accelerate creation, which is perfect as the current README file containing the information is already in Markdown.

As a distribution channel, PyPI (Python Package Index) is not only required by the assignment of this thesis, but is also the obvious choice, as all packages previously mentioned are already distributed through it. However, the question is whether the testing web interface should be distributed together with the Simple API or separately. Benefit of packaging both together is that immediately after adding Simple API, the developer can just add the path and import into `urls.py` file and immediately start testing out the endpoint. However, it also means that the UI and original module become interconnected and each change in one has be reflected in the other, in the worst case forcing the removal from the combined package, if it becomes independent or incompatible. Another reason separation is useful, is that compared to other web interfaces, such as those provided by Django REST Framework or Django Ninja, our interface is designed as insecure and only wanted to function during development, not to perform the role of documentation after development is finished. If we separate them, we can allow the developer to make the choice about when they want to include it in the application explicitly.

Chapter 4 Implementation

This chapter is concerned with the implementation of previously designed expansions to Simple API. It details not only how they ended up being implemented in the final product, but also what dead-ends were explored, and what problems were encountered during the implementation. Metaschema development is not described in this chapter, as it was implemented by Karel Jílek, only small adjustments were added.

4.1 Used tools

During the development several tools and libraries were used to support the writing, testing and deploying the application. As any development cycle is largely influenced by the technology choices, they are listed in this thesis.

4.1.1 Versioning

Versioning tools were used during development to support tracking changes in code. As Simple API is already hosted on the web service GitHub, the Git technology was an immediate choice. In addition to hosting the repository, GitHub offers multiple cloud based services, for example a ticketing system, dependency checks, or difference assessment for commits, many of which were used in the development. The service GitHub Actions, which allow developers to run workflows in the repository when various events occur, has been used to automatically run tests over Simple API and for packaging and deploying the web interface repository. The ability to assess changes before integrating them into the main branch has been vital in developing extensions to the main project.

Original repository of Simple API has been forked (copied) and changes have then been added in parts to a custom branch of the original project through pull requests. As previously mentioned, separation of the web interface from the original project was decided on, and the project was therefore developed in a separate repository also hosted on GitHub.

4.1.2 Other development tools

During the development, two editors were used, PyCharm and WebStorm, both developed by the company JetBrains. Both are extensive and robust development environments meant for Python and JavaScript development respectively. Both include smart code completion and debugging tools. Development in Python has been done using the integrated virtual environments that Python provides with the utility `venv`. That way, only packages for the project are accessible and no dependencies can be included by accident and show up later in the development cycle as a problem.

For the development of the web application, as previously mentioned, React.js was used. Build tools provided by “Create React App” running on Node.js were utilized to create the base structure for the project and for the development server and the ability to compile the app into a minified build for production. React Create App is an officially supported and recommended way to create and set up single-page React applications [42]. In the design chapter, Bootstrap and React-Bootstrap libraries were mentioned, and during implementation they were used in the web interface for their predefined layout options and built-in components with CSS styles applied. To provide for a more comfortable experience when writing and viewing queries in the UI JavaScript text editor CodeMirror with the official GraphQL lexer for highlighting was used. CodeMirror provides not only better-looking text, but also functions such as smart indentation, key bindings, and large customization. To keep the application fully in the component layout preferred by React, the wrapper React-CodeMirror2 was used to package the CodeMirror editor into a React component.

4.2 Additions to Simple API

4.2.1 Validation

Validators were seen as an important addition to Simple API, and their development didn’t depend on any prerequisites, so it was selected as the first part to implement. Before the development started, permissions module was refactored, which removed the option to add them to fields – functionality which wasn’t implemented, but the field class was set up to accept them. Implementation of validation soon turned out to be very similar to the current implementation of permissions. All added validators are first collected within Action instance constructor `__init__`, parameters, data, and general action validators separately. Just like permissions, validation is handled by creating a custom dynamic function, which is added before the execution to the resolver function. There’s, however, a significant difference between the two, in the complexity of the function which creates the hook. While the permission hook is just an iteration over the full list of permissions assigned to the action, its validation equivalent must keep information about which field is assigned to each validator and then forward the corresponding value from the query to the checking method. While the suggested validation method uses primarily information from named arguments, request and value – request containing metadata, and the value argument contains data used in the specified field of the GraphQL query - additional information known within the scope is contained inside kwargs (additional keyed arguments), so that all available data is accessible by the developer. The validation class itself has been previously laid out, and the only significant change is the functionality of the `error_message` method, which allows the developer to customize the error returned to the client when validation fails.

During implementation of the validation hook builder, the first problem was encountered, as Graphene turned out to be incredibly unclear about the exceptions raised during the request, but also passed a part of the error message back to the user. Debugging an error without the option to view the actual error or traceback is unsurprisingly significantly harder than with it, and not even debugging tools such as PyDev Debugger were helpful. What was required was a modification to Graphene, which helped find the problem. The problem with debugging was raised in the ticketing system of graphene-django [43] and an enhancement/fix was proposed by another developer the same day and merged a week later.

During prototyping, either a single validator or a list could be passed as an argument, and if multiple validators were passed in a list or a tuple, they would be checked in order. While this turned out to be useful as an intuitive way to use the validation, it also proved to be insufficient. As mentioned in the design consideration for validators, full logical operators would

```
class Validator:
    def is_valid(self, request, value=None, exclude_classes=(), **kwargs):
        """Method to provide validation over inheritance"""

    def validation_statement(self, request, value=None, **kwargs):
        """Method to validate input value,
           True -> input is valid, False -> invalid"""

    def error_message(self, error_field=None, **kwargs):
        """Message to return in API when validation fails"""
```

Listing 4.1 Validator class signature

need to be implemented. While overloading the integrated logical operators on classes might be the first instinct, Python does not allow overloading of Boolean operators [44] and even then we couldn't easily pass arguments such as the value to be validated. Python would also try to evaluate the statement before the value is known, making the option completely useless, and a different method had to be implemented. The solution settled on to be implemented was the creation of LogicalConnector classes, which are similar to validators, but don't implement the `validation_statement` method, just contain an additional constructor to accept other validators, and reimplement `is_valid` function to result in either one of `and`, `or` and `not` logical operations. While `and` might seem redundant in conjunction with how validation acts when a simple list of validators is used as an argument, it's necessary to allow the nesting of logical operators.

After implementing generic validators, the next step was adding support for Django validators and validators specifically needed for integration with Django. In the second group, only one type has so far been implemented - the wrapper around Django field validators - to fix the problem of information leakage introduced in the previous discussion about validators. Django itself doesn't validate foreign keys and waits for the database to validate the data to raise `IntegrityError`. As Simple API already knows the type of the field it's working with when it converts the model, all we have to do is create a validator which checks if the foreign key will be valid, and then add it when the field types are converted. Django with its ORM provides a convenient method `exists()`, so all we must do, is to remember the referenced model and find if the referenced object exists.

Django validators are either functions or classes that have `__call__` override and can be called as a function would be. That significantly simplifies the implementation of the wrapper, as all it needs to do is save a reference to the validation function or class and call it when a request for validation is made. As we already iterate over all fields when the conversion is done, and the validators assigned in the model are stored in an attribute of the field, we can simply collect them when the conversion is done and immediately add them to the field. We do lose the error message raised in the `ValidationError`, but as not returning it through the API was the goal, it's acceptable. In the future, the inclusion of the validator conversion could be controlled by a setting, to allow for better debugging of the interoperability of Simple API with Django and its validators.

4.2.2 Query Analysis

Implementation of query analysis began with integrating the checks from in graphene-protector into Simple API. That meant creating a Graphene backend, which in addition to parsing the query from the string in the method `document_from_string` also passes the query to a scanning function. The scoring function of graphene protector scores the query by 3 metrics.

- **Depth** - how deep nested selections are in the query
- **Selections** - number of values in each selection from the query
- **Complexity** – combination of the previous two - number of values selected multiplied by the depth in which they are

Default GraphQL parser is provided by Graphene, and is also used for parsing here, returns the result as a number of query definitions. Each definition is either an operation or a fragment. For proper depth analysis, fragments must be correctly placed, therefore before scanning starts, they are collected and then used instead of a simple field. Then each operation is scanned to check whether it's not excessively complicated based on the three criteria. Depth is measured by recursively traversing over the property `selection_set` which is of type Field. If an attribute of the Field is set to None, it's either a usage of a fragment or a primitive field (scalar in GraphQL), otherwise further selection is nested inside. Selections are measured by the number of all fields in the parsed query, counted during the recursive traversal.

This solution for scanning works reasonably well, but it has two major problems. Immediately obvious is the problem of lists. This method doesn't understand either simple or paginated lists. Query requesting a one field from 500 objects is no different to a query requesting one field from one object. We have already estabilished that blindly using the argument `limit` wouldn't work because of false positives, and the fact that we would miss lists that are not explicitly limited in the query received. To find paginated lists, we must go further up - before the conversion from Simple API types to Graphene is done. To collect the fields, an additional class property is defined inside the Object class – `connected_fields`, which returns all fields referencing other objects, and their resulting type. Such information must be then passed to the scanning functions, for which an alternative generation function `generate_w` has been created. Once we know what fields are lists, we can now not only multiply the selections by the maximum number of times they can occur, but also require the limits to be set and be reasonable as well. Inspiration for such node limiting has been taken from GitHub GraphQL API limits [45]. The other problem, which isn't fixed by this limit, is that the combination of both autogenerated actions and objects with custom ones means that setting a good limit for one is usually going to be insufficient or overly restrictive for the other. That problem would only be solved by allowing the developer to add a custom variable weight to each action and field.

Since we are already traversing the objects before generation to find lists, adding weights to the consideration is simple. Additional class property, which collects the set field difficulties from the attribute and combines them with all fields that exist but do not have their cost explicitly set, needs to be added. When we collect fields that are references, we now also collect the scores and pass them inside the same object to the scoring function.

There's still one more problem – when using the backend functionality, we've pretty much hit the limit in terms of what can be implemented, because we don't have the context of the request. While no functionality using it has been implemented in this thesis, we want the possibility to expand the functionality in the future. When we have context of the request, we can further expand the functionality of the query filtering, limits can differ whether the user is logged in or not, or we can start banning clients who repeatedly try to request too expensive queries. For that reason, query scanning had to be moved one level higher to GraphQLView differing from other existing solutions.

The part we want to edit is just the handling of incoming queries, so all we need to do is to inherit the original View and redefine the method. Sadly, as the query scanning has to occur after the query has been parsed, but before execution, we can't call the parent method through `super()` function and add our code before or after it, unless we don't mind parsing the query twice. To increase readability and reduce the number of arguments each function takes; each is refactored into a method of the View and limits are saved as attributes so they can be simply invoked anywhere instead of being passed around repeatedly.

For the previously mentioned reasons, the complexity and selection limits were deemed unnecessary after the full weight/difficulty scoring was implemented. If the developer deems scoring every action and field too difficult or tedious - figuring out the correct weights certainly fits in both categories – they can use the query scoring with a field weight equal to 1 (which is also the default value) and then the query scoring limit is equivalent to the selection limit previously implemented. Finally, we end up with 3 different limits that can now be used in Simple API.

- **Depth** – The same limit as mentioned previously and implemented in other solutions. Limits purely the number of selections that can be nested inside each other.
- **List** – If set, requires a specific limit to be set for each selection that returns a paginated list (non-paginated lists shouldn't be used with Simple API).
- **Weight** – Applicable as both a weight limit and - when simplified - as a selection limit. When no fields or actions have a custom weight set, it simply limits the number of selections as each field is worth 1 difficulty score and each action is 10 score, but allows the developer to customize values for each field and action individually.

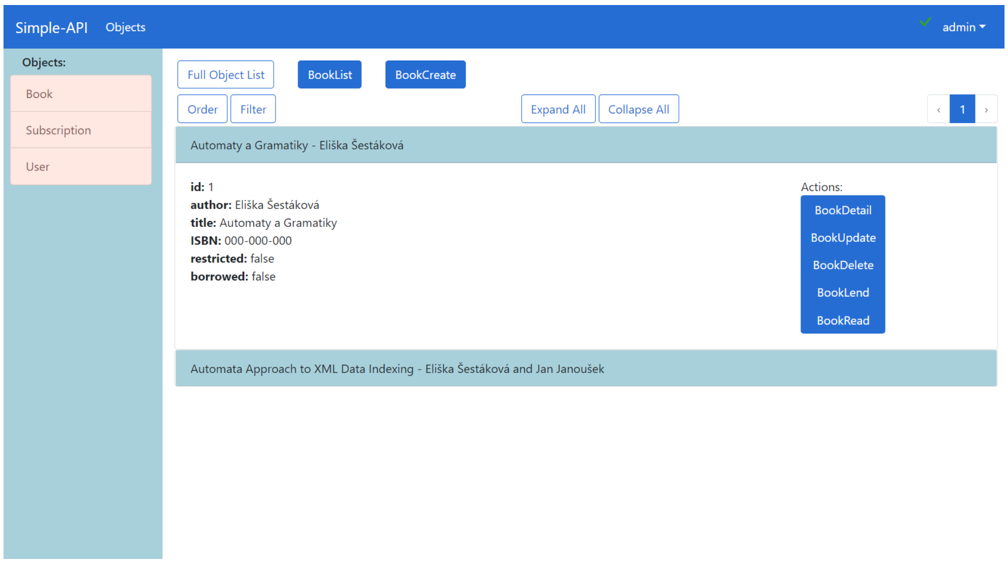
Meanwhile, the scanning has grown from one cycle over actions to three different methods depending on whether they're scoring an action, field, or a list. Scoring still begins in a cycle over all operations in the query, but as only one query can be executed in Graphene (specified in other fields of the request), only the one selected to run is scanned. That way, the scanning is faster, but primarily the results when testing with IDE such as GraphiQL are now accurate. Before, when a different query was to be run, the API would return an error even if a query other than the one called was too expensive. The query scanning is not done, even when using the extended View, when no limits are set. The settings for the limits are placed inside settings.py file of the Django server, as that's the standard place for all applications to store their configuration data.

4.3 Web application

The client to demonstrate the Simple API and especially Metaschema functionality is the primary product of this thesis. Developed with React.js, the goal was to keep the application in as many components as would be reasonable, to allow for straightforward reuse. The application is inspired by the Django Admin interface – top bar shows the "Simple API" name and the login status and the main application window is split into two separate columns. One showing the types that Simple API works with – most often equal to the models in use. The larger column on the right presents actions on the currently selected type and either the list of object instances or an action submenu. The web UI first requests the Metaschema, then constructs the layout based on the response.

4.3.1 Permissions, authentication, and Django integration

Permissions and their indications were a large focus for the application. If an object is hidden when the current user's lacking permission, the Metaschema doesn't contain the type and therefore the object is not shown. The same process applies for actions. If the action is not hidden but the user is lacking permission, the button is rendered, but it's not functional and presents the user with the error (denial) message it received as a tooltip on hover. Since a hover trigger is not raised on disabled components, the disabled button is wrapped with a div, which triggers the event. As then the component lacks the correct Bootstrap styling and for consistency must be added manually. While the package react-bootstrap-tooltip-button used to be able to add it,



Automaty a Gramatiky - Eliška Šestáková

id: 1
author: Eliška Šestáková
title: Automaty a Gramatiky
ISBN: 000-000-000
restricted: false
borrowed: false

Automata Approach to XML Data Indexing - Eliška Šestáková and Jan Janoušek

Figure 4.1 Simple API web interface design

it's no longer compatible with new versions of React due to its dependencies [46]. The styles were scavenged and expanded, as they didn't cover all cases where buttons can appear in Bootstrap (vertical button groups styling was not correctly applied), and a new component was written in contemporary React style. The component with the styles has been published separately on NPM (node package manager) as `react-bootstrap-tooltip-button2`.

Testing permissions is highly dependent on the ability to quickly switch users. For such use case, the application allows the developer to switch the user with just the username of the user, which we want to test the API as. In the web application, currently logged in user is shown, and previously logged in usernames can be quickly switched to. The application saves all users that have been used and not logged out. As the application doesn't use any storage on the server, it saves the list of usernames in a browser cookie instead.

The login procedure and serving of the application are controlled by a Django view. The logging in is facilitated by a basic POST endpoint `/login`, which expects a JSON object as its instruction, and tries to log in the user with standard Django authentication. The endpoint allows for logging in, logging out, and getting the currently logged in account, as the React application has no way to know if a user is logged in when it's loaded. Both the admin application and the login endpoint are automatically disabled and return HTTP Error 404 when the Django setting `DEBUG` is set to False. The view uses Django `LazyObject` to load, as its dependencies would not be ready when the Django server is first loaded. For sending requests, a csrf value is required in Django to prevent Cross-site request forgery. To receive the value in a cookie, `the ensure_csrf_cookie` decorator is set, however, Django documentation advises that the cookie might not be correctly set even with the decorator [47]. The only sure way is to have the `csrf_token` tag in the template. That guarantees the cookie will se set, and a csrf token will also be added to the body of the page as well. Django prefers the cookie [47] to the token in the document, so it is what the application uses.

Other than inserting the `csrf_token` template tag, we also need to modify the links to the JavaScript and CSS static files that Django is supposed to serve. For that reason, a simple build script was written in Python. It inserts the token tag and then substitutes the static paths with Django template style static file link using regular expressions with Python module `re`. As we use static files to serve the JavaScript and CSS, we do require `'django.contrib.staticfiles'` as a dependency, but so does Django Admin and almost any other Django application.

4.3.2 Generating queries

Some queries, such as the one for Metaschema, are distributed with the application, but not all queries can be integrated that way. When presenting actions to the developer, simply showing the action name isn't helpful enough, same information can be got from the standard schema explorer in GraphiQL. To be as helpful as we could, we have to show how the query can be used. To showcase how the API works, we generate the query, possible arguments, and if it returns a selectable object, automatically make a generic selection. Generation of the selection is simple, we find the type, which is returned by the action, if it's selectable we iterate over all its fields. Current implementation leaves only the first level and adds no nesting to simplify the resulting query, but the generation function was written with deeper generation in mind. It does insert the reference to the field being there, but leaves the line commented, so it's not executed by the API unless the user specifically edits the selection.

Important part of creating a query is ensuring the correct types, as the most basic type of validation for the input fields. For that reason, the web UI generates a form to allow the developer to insert arguments into the query, without the need to edit it manually. When the arguments are inserted, they are checked for the basic types and the match or inconsistency is indicated with either a green check or a red warning. Advanced forms of validation are left to the API. The default error message on the field validator returns the field on which the validation failed. The error message is parsed when returned to the client, and if the information about the problematic field is included, the error is indicated the same way as if the argument was of a wrong type. The message is also shown in the response view, just as a successful return would be. As the query view shares the space with the response, they are switched through tab functionality of Bootstrap. When a response to the query is received, its tab is highlighted to indicate a change.

Not all arguments are just simple values, some expect a selection of options – for example Boolean fields or enumerable types - for which a dropdown form is created instead of the traditional input field. Others expect an object with a selection from a large number of possible arguments – filters for instance. To simplify the creation of queries with such types, the form loads the type attributes and offers an option to add and remove these attributes, adjusting the query for the currently included.

To present the query, a familiar interface is preferable, and since GraphiQL is based on CodeMirror, the same is used to display queries in the custom interface. GraphQL does have an official extension to add linting and highlighting to CodeMirror, but the linting and highlighting inside the query is dependent on a standard GraphQL schema, so only the basic highlighting is used. Once the reply is received, it's also shown in a CodeMirror widget, this time with JSON highlighting, making the result is easy to read.

4.3.3 Object list and filtering

The most significant difference in behaviour, that Simple API offers over plain GraphQL, is the separation between generic object actions and actions over a specific object instance. The standard GraphQL schema has no distinction between action connected to an instance and other actions. In Metaschema actions are not grouped by their query type, instead connected to the object type or object instance they're concerning. Instance actions are queried over the instance itself with the field `__actions`, which allows us to separately render possible actions while showing the object itself.

Showing the full list of objects is essential, if we want to allow the developer to test actions and their connections to instances, because if you create an object, you should be able to inspect it and see how you can interact with it. Switching over to another view is not suitable, because it would disrupt the ability to test the actions over multiple instances, so a modal (widget imitating a window over the usual content) is shown instead. In the form, all necessary details are already entered, so as little editing as possible is needed to use it. Permissions are, of course, shown the same way as they are elsewhere in the application. As permissions can differ from one object instance to another, the instance actions are shown directly on the existing objects.

To make the list of objects readable, pagination is employed, using the Simple API limit and offset arguments of the PaginatedList type. For user interaction with the list, it uses Pagination widget from Bootstrap and the the data is fetched dynamically per page. To allow for searching and sorting, an important part of the listing was the addition of filters and making reordering the list possible. Both are added as a modal invoked with a button above the list and persist through switching different pages of the result. The functionality allows for adding and removing of all filters set up on the List action and setting a different order by which the items are sorted.

Initially implemented as a table grid, it was reworked into an expandable accordion style item list. The change was made because even with a reasonable number of columns, the table would quickly run out of space and either overflow requiring a slider or have too thin columns so that the list was unreadable. The accordion uses the method `__str__` that can be overloaded on a model and is a string representation of the object. Simple API explicitely converts the method because of its ubiquity and usefulness. The usage of the method is also helpful for consistency as it's also used for describing objects in Django Admin web application. Additionally, if large exploration is wanted, buttons for expanding and collapsing all tabs were added above the list and the list will stay expanded even through page changes.

4.3.4 Customizing the UI

As the application uses static JavaScript and minimal templating, the UI itself cannot be edited directly from Django interface the same way Django Admin can be. What can be customized in the view is the login procedure. Overloading the `simple_login` method of the view allows for any other form of one input login. Such customization requires the method to take the JSON in the request and implement three operations that can be specified in it. The response from the server is also expected to be in the form of a JSON object. Contained in it are four attributes - `username`, `success`, `operation`, and `reason`, with only the first two being required. Username is what name should be shown in the web application as the currently logged in user, made for the event when the login token isn't identical to the username. Success is used for specifying if the login or logout was successful. Operation returns the same value as was sent the the endpoint, and reason represents the error message to show if the action fails. The three operations that the web application expects to be served are:

- **status** – return the currently logged in user, if no user is logged in return False in the success field
- **login** – log in or switch currently logged in user, return user logged in after a success in the username field. If the operation fails, the returned object should have the field success set to False - then the UI assumes the same user as before (or no user) is still logged in.
- **logout** – log out the currently logged in user and return to AnonymousUser state with Django authentication or equivalent.

Deploying the customized API instead of the default, is then just as simple as just assigning the customized view to the `site` variable in `simple_api_admin.views`.

Chapter 5

Testing and distribution

5.1 Testing

Testing is an integral part of any development and Simple API is no exception. Unit testing has been implemented just before the start of this thesis and all pre-existing tests are described in Appendix B or in the documentation available in the package. These tests cover most of the possible edge cases the API can encounter. The tests are implemented using Django's built-in testing environment [48] and called using `python manage.py test` with an additional shell script. The script copies the model and object definitions to the project, as the testing environment is made to test generic applications with their own models and components, not to test how application performs with multiple different application design.

Graphene provides a helper class `GraphQLTestCase` derived from the original Django `TestCase`, which allows the developer to test the endpoint by implementing methods for querying and checking the responses for errors. Simple API further expands this class with methods for testing the generated schema with a reference provided before the test is run, the same check for Metaschema, and a method for comparing shuffled JSON Arrays, a necessary feature for testing the Metaschema.

The modifications to Simple API changed multiple parts of the framework and some of the changes had to be reflected in all already existing tests. One change was necessary when the schema generation function was moved; however, a significant change was needed after the Metaschema was expanded. All tests had to have their reference Metaschema adjusted as the content has changed, but as the actions to retrieve the schema were changed or completely new, even the standard GraphQL schema (represented by GraphQL SDL) had to be updated. At the end of the development, all tests have been manually checked and edited to correctly indicate the expected outcome.

Tests have not only been updated, but completely new were also added. Initially, additional tests for the basic functionality of Django integration and the integration of permissions were added, as part of familiarizing with the project, and as implementations of validation and query security were added, tests for them were created as well. For validation, checks for validator usage, validator logical connectors, wrappers for Django validators and error messages are done. Tests for query security try the functionality of depth limits, action and field weight assignment, and the query cost limits.

The Simple API tests are comprehensive, but Simple API isn't the only product of this thesis. The web interface also needs some testing, both about its usability and functionality. First one has been tested – the interface has been shown and tried by 3 other students of FIT CTU and their complaints were integrated into the final product. The same has been done with the feedback from Karel Jílek, who will have the most contact with it while further developing Simple API. Testing the functionality is harder – while the query generation and component rendering has been tested with all currently written Simple API schemas taken from the previously mentioned tests, no automatic testing using libraries such as jest.js has been implemented. Further expansion of tests should focus on testing the integration of both parts together or automating the testing by either creating Metaschema generator for random testing.

5.2 Distribution

Both Simple API and its web interface are shared under the highly permissive MIT License. Distribution is prepared to be done through PyPI, which is the most commonly used Python package index, mainly used by the pip installer. As Simple API is not yet finished, and an analysis of what features implemented inthis thesis will be integrated into the main project is yet to be done, the main project after the inclusions is currently only published through GitHub. Since the web interface is dependent on some of the changes done in this project (especially the extension to the Metaschema), it has also not been published on PyPI. It has however to prove its functionality it has been published on the test instance of the index, available at *test.pypi.org*. While that instance is not permanent, it serves to show that the package is ready to be published, once all changes are validated. The test index can still be used for installation with an argument passed to pip.

The web interface final module is created dynamically in the build routine, combining the prepared Python base with newly compiled React application. As part of the build process, which is run on GitHub Actions, the package is also published as a release there for download. That makes the package simple to download without the need to have build tools installed on the current machine, as the latest tagged push is packaged in a compressed ZIP file and available for download. While the package itself is only dependent on Django, the build routine requires both Python and Node.js with multiple libraries. Unless the developer is planning to modify the JavaScript interface itself, the precompiled package is a preferred method of distribution.

Once all of the changes to Simple API are reviewed, the plan is for the main repository of Simple API to be published to PyPI and have the web interface - as a separate package - published there as well.

5.3 Documentation

Together with the project its developer and user documentation were created. While a large amount of information was already written in `README.md` by Karel Jílek, its layout is not ideal for usage, so its contents were taken and expanded into a fully fledged web documentation. As previously mentioned, MkDocs was selected as to engine to power the documentation, therefore the documentation sources are written in Markdown and compiled to a static HTML site. The final look is provided by the extension Material UI for MkDocs and examples of the look and content of the documentation are available in Appendix D. Search, navigation bar and bookmarks are added for better experience, and the documentation is separated into chapters based on their topic. The documentation is comprised of development information, installation instructions, descriptions of features Simple API offers, and the same info for the Simple API Admin. The documentation is stored in a git submodule of the Simple API repository, and it's prepared to be published to a hosting service such as GitHub Pages or ReadTheDocs.

Chapter 6

Conclusion

The goal of this thesis was to further the development of Simple API, an extension to a popular Python web framework Django. Specific aim of this expansion has been to add functionality for creating dynamic web pages powered purely by the responses of the API. It has been achieved by the creation of the Metaschema and tested by the work on the testing web interface, both created as part of this thesis.

Furthermore, the Simple API package has been enhanced by the implementation of validators and multiple security features for preventing information exposure and denial of service attacks. The enhancements not only push Simple API closer to a fully functional framework, but they also accelerate further development by allowing easier testing, and provide proof of concept for automatic generation of clients from the Metaschema. This thesis has also not only examined, but also added precautions for stopping several problems GraphQL endpoints encounter. Improvement or expansion of the existing functionality of authorization and input validation has also been successfully implemented. Another goal was to significantly increase the number and coverage of tests in Simple API. However, since the package is still very much not finished and many things change before full release, this goal ended up being mostly unfinished.

Graphene, through which Simple API creates the GraphQL endpoint, turned out to be less production ready than was expected and working with it turned out to be significantly more complicated than anticipated. However with how fast the community is responding to issues, such as the one made during this thesis, it shouldn't take long before it catches up to other alternatives, such as the Apollo server. One example of the missing functionality is partial query resolution, which is supported on many other servers and recommended as a solution to multiple problems GraphQL encounters, and isn't supported in Graphene. Simple API, on the other hand, is further in development than expected and working with it has been enjoyable and efficient. React.js turned out to be an excellent choice for the web interface, as working with it was simple, efficient and its ubiquity makes it especially useful for future development.

While this thesis has tried to significantly improve Simple API, there are still long development paths to follow, such as adding support for GraphQL subscriptions, or implementation of API styles/technologies, as is one of the goals for the package. Other than subscriptions, further development could implement authorization functionality differing from the basic Django authorization. Support for OAuth2 could be especially useful, as it would allow Simple API to be used in networks using it as their single sign-on (SSO) protocol, such as the one used by FIT CTU. Next step in working with and testing Simple API should be the development of a fully functional application that has Simple API as its data source. When used in an actual development environment, small problems will be flushed out and it could prove itself as the useful tool we believe it can be. Now that the generated endpoint is supported by the web interface and the expanded and reformatted documentation is available, it's the next logical step.

Appendix A

Work done on Simple API in this thesis

Part of simple API	**Work done**
Web interface	Fully built web interface in React.js, module to allow integration to Django server as an application, authorization interface in the Django module
Permissions	Refactoring of permissions, unfinished implementation reworked into validators
Validators	Validator class, Logical connectors, Addition of validation to the action hook, Validators to consume django model field validation functions, Foreign key validators
Securing the API	Addition of static depth scanning before query evaluation, Query weight analysis with customizable action and field weights, Generation of the API without GraphQL introspection
Metaschema	Testing and bug fixing, Expanded to allow for the Web interface to function
Testing	Revision of all existing tests after the Metaschema change, new tests for permissions, Django integration, validation, and query security scanning (both depth limit and weight limit)

Appendix B
Test coverage

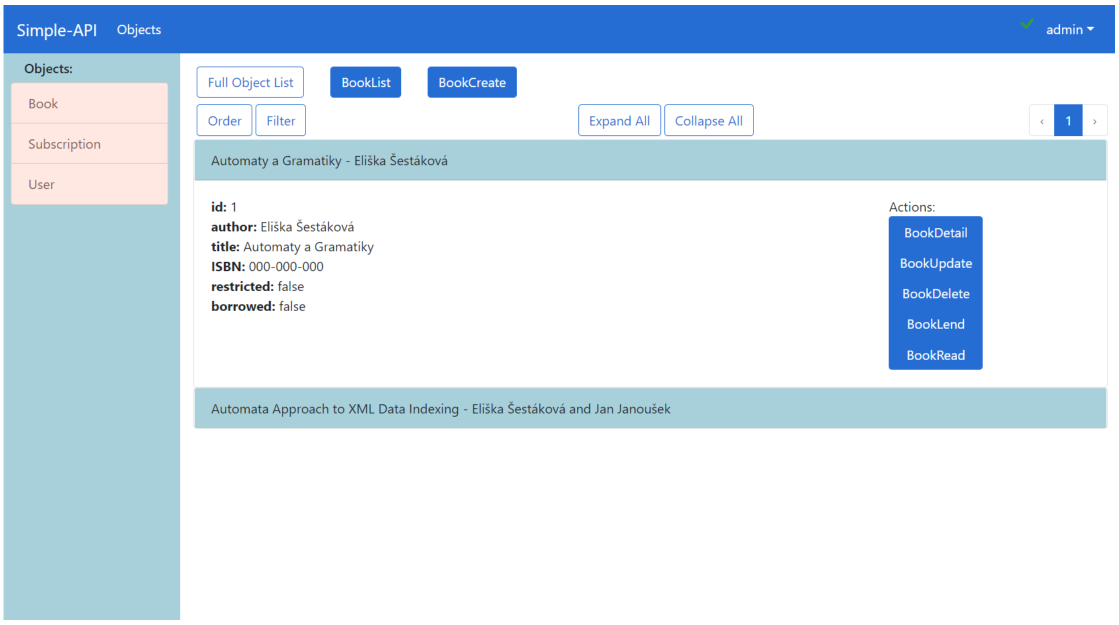
Test name and location	Tested functionality
django_objects	
`autocomplete_user_model`	Generation of object based on existing model
`primitive_model_fields_only_exclude`	exclude_fields and only_fields attributes for DjangoObject
django_objects	
`graphene_python_list_resolve_null_for_all_fields`	Correct resolution of Object list data with Graphene
readme_forum	
`readme_forum_basic`	General functionality based on example usage
`readme_forum_class_for_related`	Multiple objects for one django Model (class_for_related attribute)
`readme_forum_custom_actions`	Custom actions to a generated DjangoObject
`readme_forum_custom_fields`	Custom fields on a generated DjangoObject
`readme_forum_custom_filters`	Custom filters on a list to a generated DjangoObject
`readme_forum_permissions`	Django permissions, logical connectors on permissions
library_system	
`library_system_basic`	General functionality based on example usage
`library_system_permissions`	Django permissions, logical operators on permissions
`library_system_validation`	Simple API validation, consumption of Django validators, logical connectors on validators
`library_system_query_security`	Depth limit, action/field difficulty assignment, query cost limit

Test name and location	Tested functionality
objects	
`action_retry_in`	Permissions and retry_in field of permission part of Metaschema
`and_or_not_permissions`	Permissions and logical connectors for permissions
`car_owner`	Custom defined connected objects
`date_time`	Handling of datetime type from Simple API
`duration_type`	Handling of duration type from Simple API
`hide_action_if_denied`	Hiding action within Metaschema when the client is lacking permission
`infinite_recursion`	Handling of object with infinite recursion (referencing itself)
`int_list_records_pagination`	Manually defined pagination lists
`int_list_records_pagination_ recursive`	Recursion in manually defined pagination lists
`multiple_modules`	Separation of object definition to multiple modules (imported to objects.py)
`nested_object_as_input`	Action with object (with possible nesting) as an argument
`object_list_as_input`	Action with a list as an argument
`object_list_of_objects_field`	Action with list of objects (with possible nesting) as an argument
`object_list_records_pagination`	Action with a list of objects as arguments and pagination over lists
`objects_primitive_fields_ class_ref_by_string`	Ability to reference a class as an object type (in action definition) by string
`objects_simple_list_field`	PlainListType as a return value in action
`object_type_as_input`	Usage of Simple API object as argument definition
`object_type_with_list_as_ input`	Action with defined object of PlainListType as an argument
`object_with_field_parameters`	Arguments on fields of the object
`object_with_input_output_ fields`	Testing of Object attributes input/output fields
`object_with_params_and_ resolver`	Field with a resolver function using parameters

Appendix C

Screenshots of the web interface

All screenshots were taken with 125% site zoom for better readability.



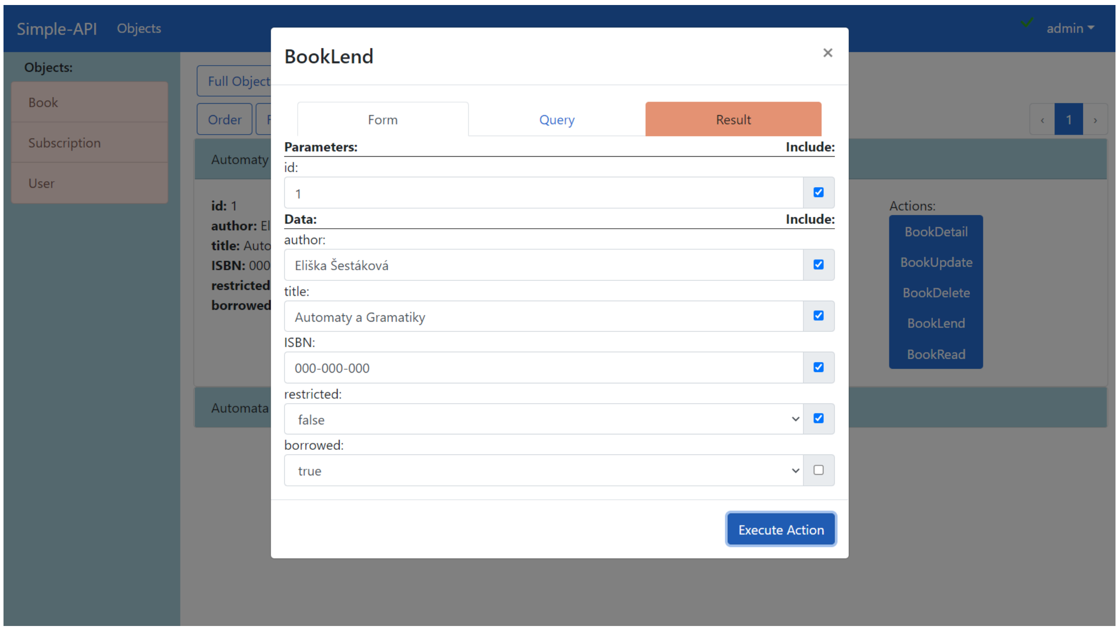
Automaty a Gramatiky - Eliška Šestáková

id: 1
author: Eliška Šestáková
title: Automaty a Gramatiky
ISBN: 000-000-000
restricted: false
borrowed: false

Actions:
BookDetail
BookUpdate
BookDelete
BookLend
BookRead

Automata Approach to XML Data Indexing - Eliška Šestáková and Jan Janoušek

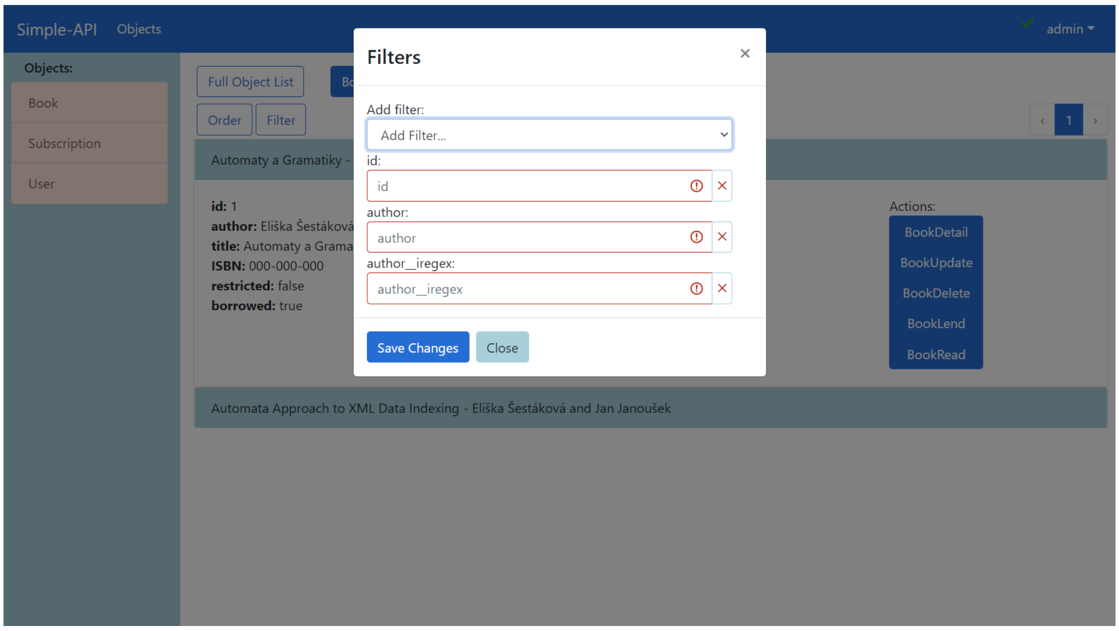
Figure C.1 View of object instances retrieved with BookList action, also see login in top right and type list in the right column



id: 1
author: El
title: Auto
ISBN: 000
restricted
borrowed

Parameters: Include:
id: 1 ✓
Data: Include:
author: Eliška Šestáková ✓
title: Automaty a Gramatiky ✓
ISBN: 000-000-000 ✓
restricted: false ✓
borrowed: true ☐

■ **Figure C.2** Object instance action modal, data is entered automatically when the action is selected over instance



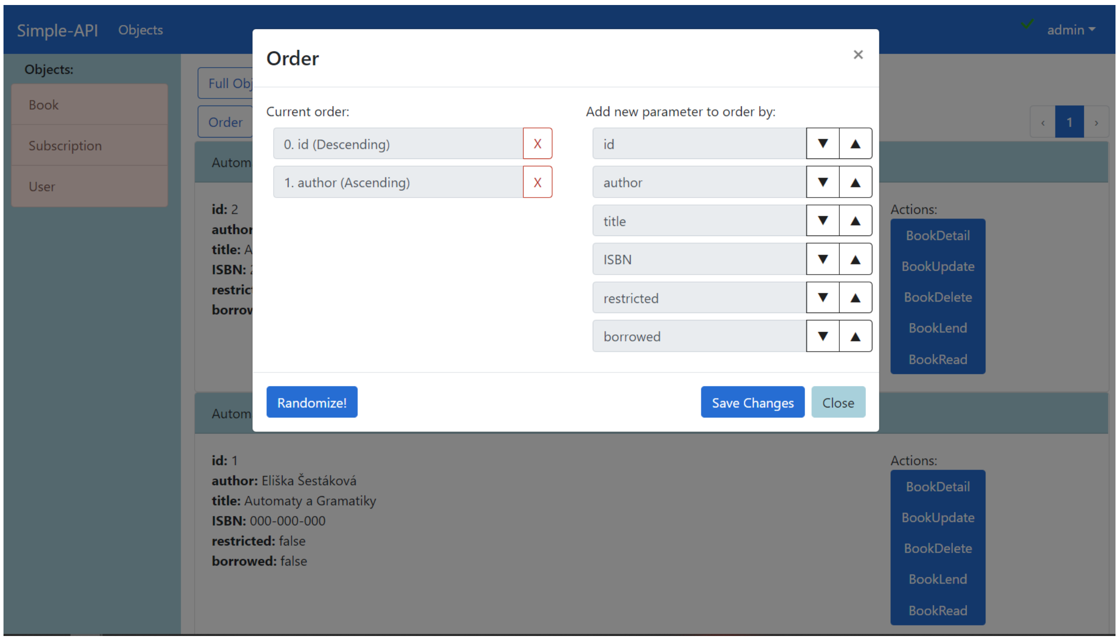
Automaty a Gramatiky -

id: 1
author: Eliška Šestáková
title: Automaty a Grama
ISBN: 000-000-000
restricted: false
borrowed: true

Automata Approach to XML Data Indexing - Eliška Šestáková and Jan Janoušek

Add filter: Add Filter...
id: id
author: author
author__iregex: author__iregex

■ **Figure C.3** Modal for applying filters to the object list



Order

Current order:

0. id (Descending) X

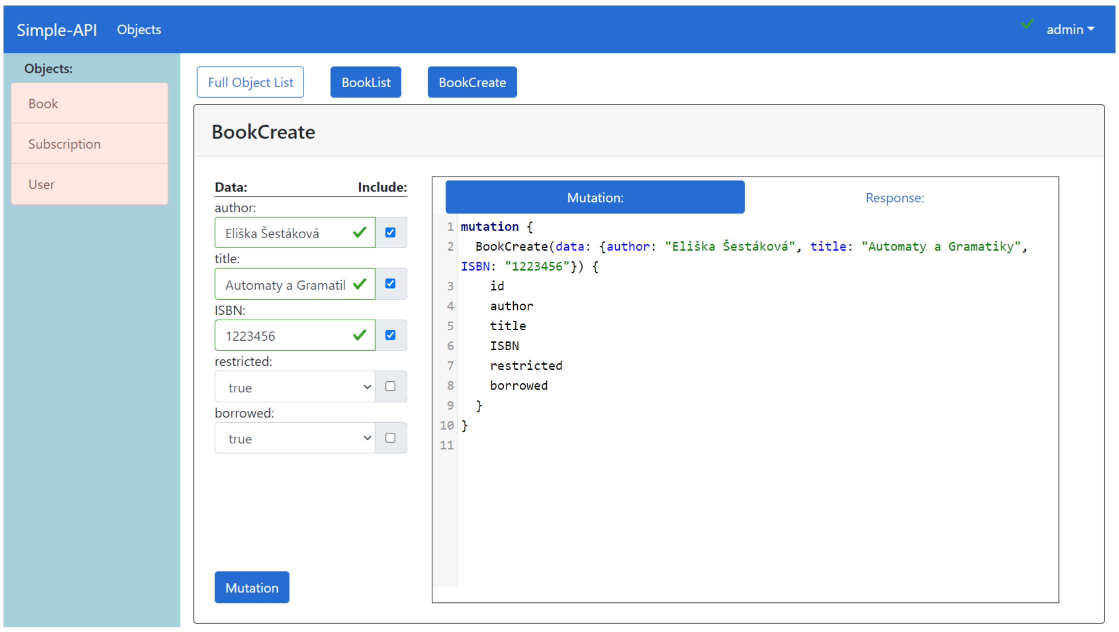
1. author (Ascending) X

Add new parameter to order by:

id	▼	▲
author	▼	▲
title	▼	▲
ISBN	▼	▲
restricted	▼	▲
borrowed	▼	▲

id: 1
author: Eliška Šestáková
title: Automaty a Gramatiky
ISBN: 000-000-000
restricted: false
borrowed: false

Figure C.4 Modal to allow sorting of the object list

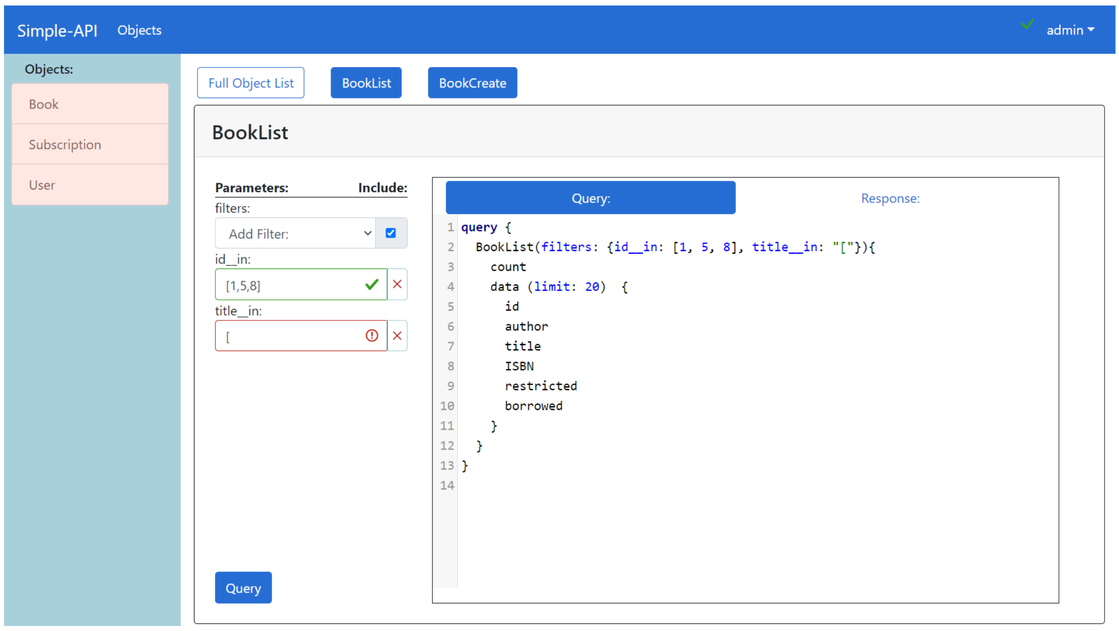


BookCreate

Data:	Include:
author: Eliška Šestáková	☑
title: Automaty a Gramatil	☑
ISBN: 1223456	☑
restricted: true	☐
borrowed: true	☐

```
mutation {
  BookCreate(data: {author: "Eliška Šestáková", title: "Automaty a Gramatiky",
ISBN: "1223456"}) {
    id
    author
    title
    ISBN
    restricted
    borrowed
  }
}
```

Figure C.5 Query creation through a form



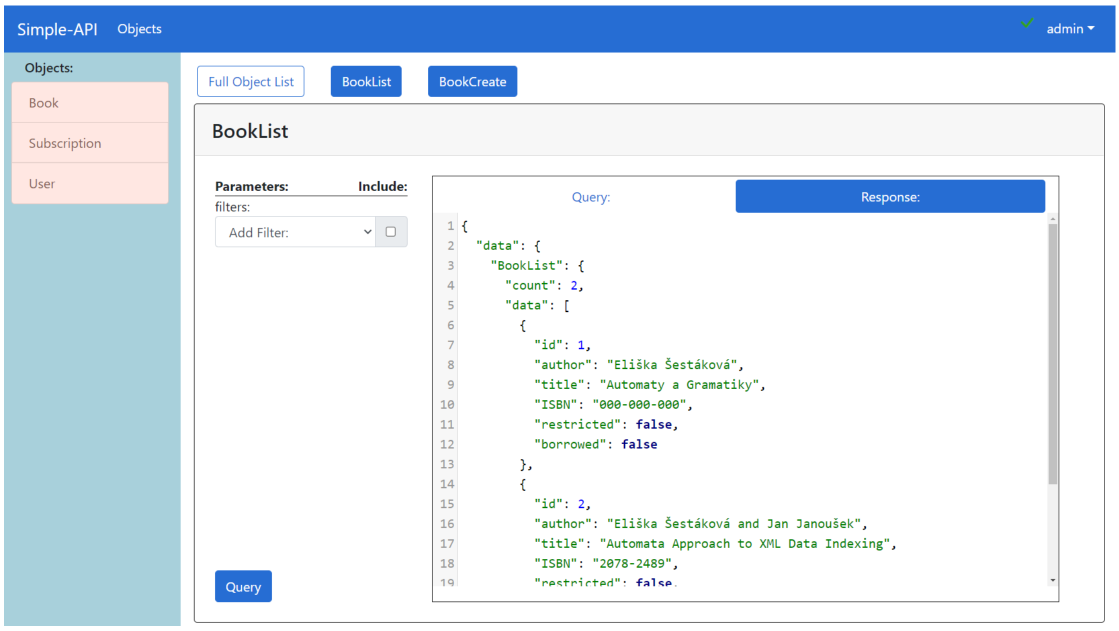
filters:

id__in: [1,5,8]

title__in: [

```
query {
  BookList(filters: {id__in: [1, 5, 8], title__in: "["}){
    count
    data (limit: 20)  {
      id
      author
      title
      ISBN
      restricted
      borrowed
    }
  }
}
```

Figure C.6 Adding filters is done with dropdown menu, and the values are then validated for type

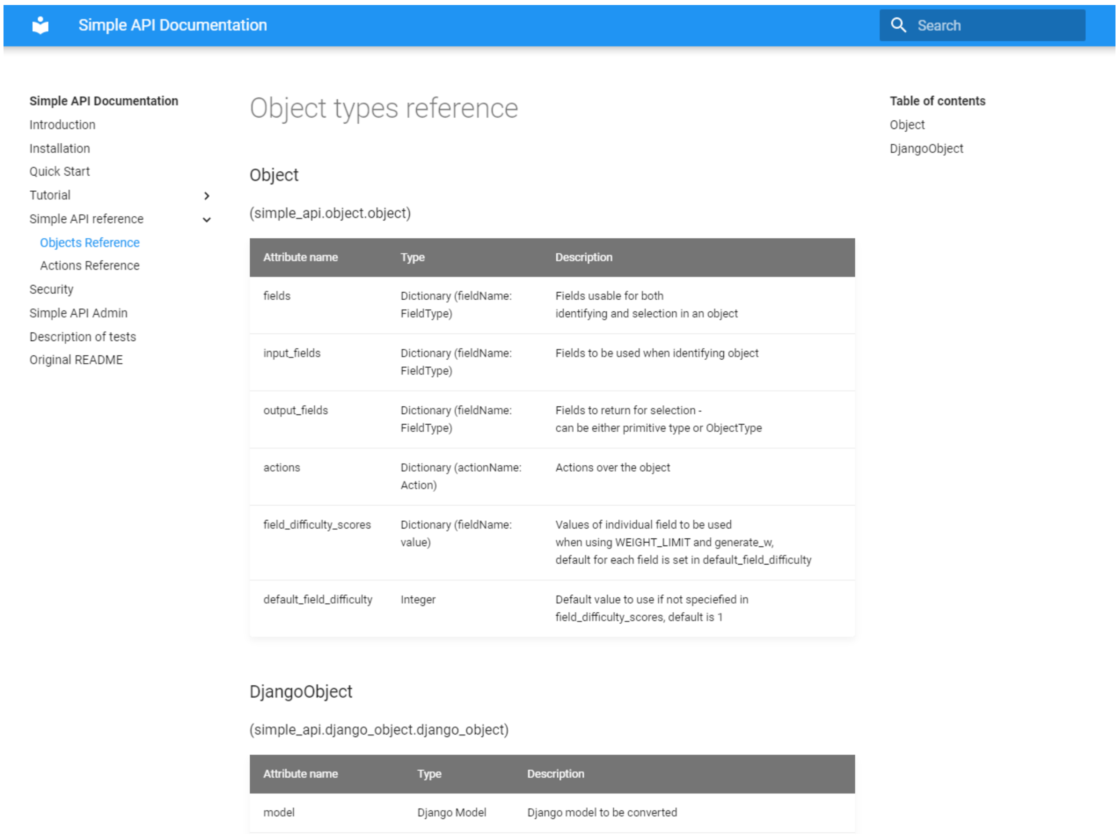


```
{
  "data": {
    "BookList": {
      "count": 2,
      "data": [
        {
          "id": 1,
          "author": "Eliška Šestáková",
          "title": "Automaty a Gramatiky",
          "ISBN": "000-000-000",
          "restricted": false,
          "borrowed": false
        },
        {
          "id": 2,
          "author": "Eliška Šestáková and Jan Janoušek",
          "title": "Automata Approach to XML Data Indexing",
          "ISBN": "2078-2489",
          "restricted": false,
```

Figure C.7 View of the response for a query

Appendix D

Examples of the documentation



Object types reference

Object

(simple_api.object.object)

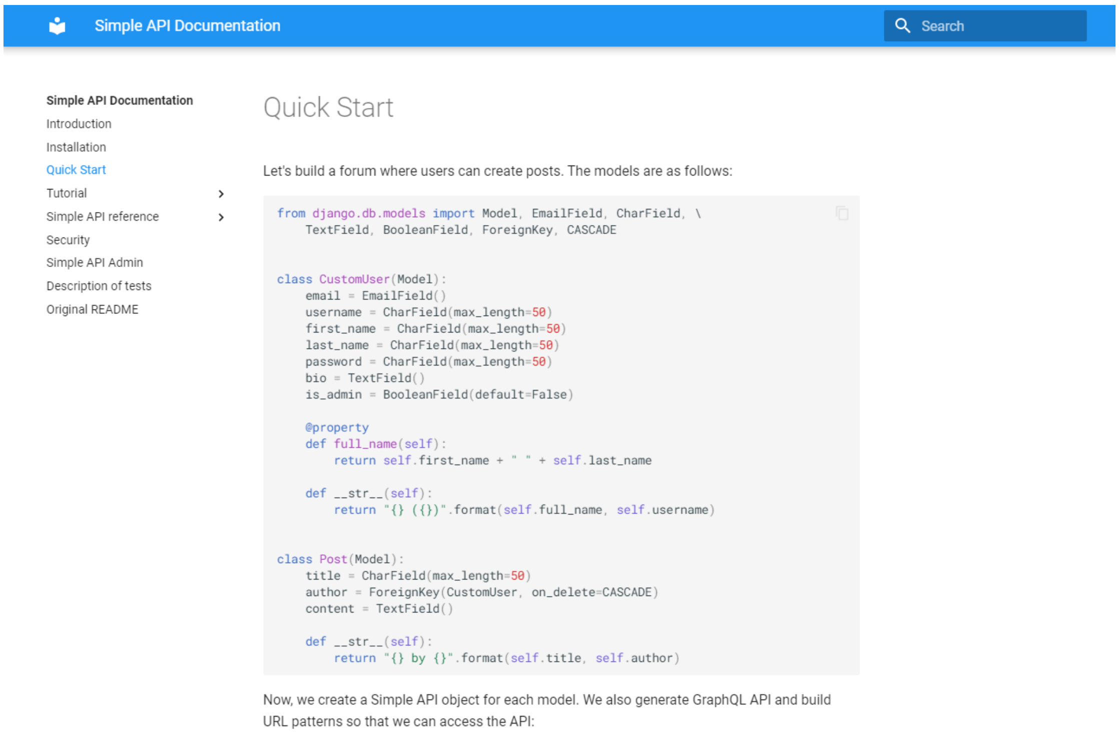
Attribute name	Type	Description
fields	Dictionary (fieldName: FieldType)	Fields usable for both identifying and selection in an object
input_fields	Dictionary (fieldName: FieldType)	Fields to be used when identifying object
output_fields	Dictionary (fieldName: FieldType)	Fields to return for selection - can be either primitive type or ObjectType
actions	Dictionary (actionName: Action)	Actions over the object
field_difficulty_scores	Dictionary (fieldName: value)	Values of individual field to be used when using WEIGHT_LIMIT and generate_w, default for each field is set in default_field_difficulty
default_field_difficulty	Integer	Default value to use if not speciefied in field_difficulty_scores, default is 1

DjangoObject

(simple_api.django_object.django_object)

Attribute name	Type	Description
model	Django Model	Django model to be converted

Figure D.1 Object type attribute references for Simple API usage



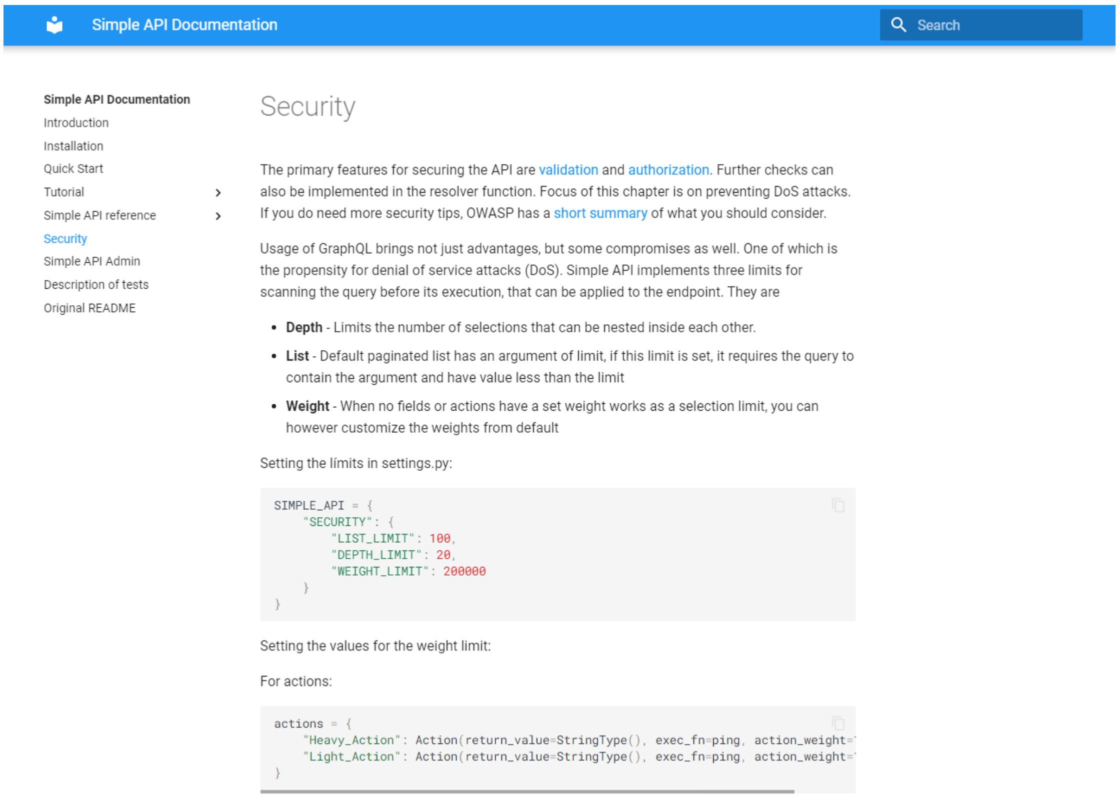
Quick Start

Let's build a forum where users can create posts. The models are as follows:

```python
from django.db.models import Model, EmailField, CharField, \
    TextField, BooleanField, ForeignKey, CASCADE


class CustomUser(Model):
    email = EmailField()
    username = CharField(max_length=50)
    first_name = CharField(max_length=50)
    last_name = CharField(max_length=50)
    password = CharField(max_length=50)
    bio = TextField()
    is_admin = BooleanField(default=False)

    @property
    def full_name(self):
        return self.first_name + " " + self.last_name

    def __str__(self):
        return "{} ({})".format(self.full_name, self.username)


class Post(Model):
    title = CharField(max_length=50)
    author = ForeignKey(CustomUser, on_delete=CASCADE)
    content = TextField()

    def __str__(self):
        return "{} by {}".format(self.title, self.author)
```

Now, we create a Simple API object for each model. We also generate GraphQL API and build URL patterns so that we can access the API:

Figure D.2 Instructions for quick setup of Simple API



Security

The primary features for securing the API are validation and authorization. Further checks can also be implemented in the resolver function. Focus of this chapter is on preventing DoS attacks. If you do need more security tips, OWASP has a short summary of what you should consider.

Usage of GraphQL brings not just advantages, but some compromises as well. One of which is the propensity for denial of service attacks (DoS). Simple API implements three limits for scanning the query before its execution, that can be applied to the endpoint. They are

- **Depth** - Limits the number of selections that can be nested inside each other.
- **List** - Default paginated list has an argument of limit, if this limit is set, it requires the query to contain the argument and have value less than the limit
- **Weight** - When no fields or actions have a set weight works as a selection limit, you can however customize the weights from default

Setting the limits in settings.py:

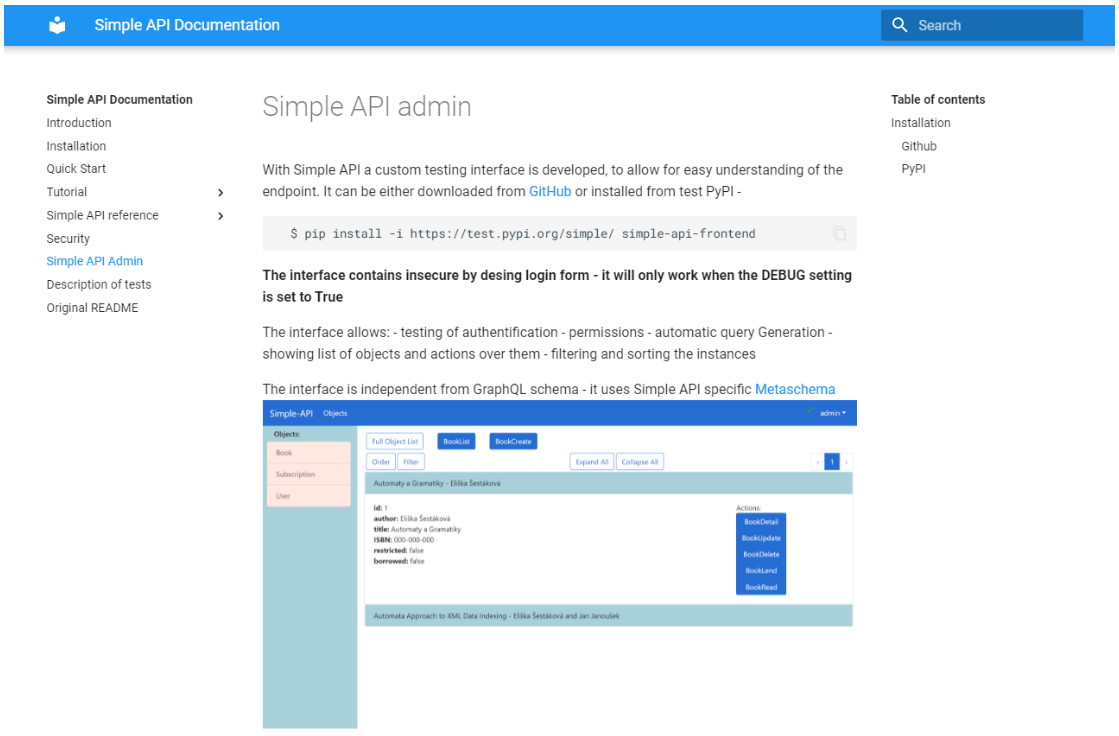
```python
SIMPLE_API = {
    "SECURITY": {
        "LIST_LIMIT": 100,
        "DEPTH_LIMIT": 20,
        "WEIGHT_LIMIT": 200000
    }
}
```

Setting the values for the weight limit:

For actions:

```python
actions = {
    "Heavy_Action": Action(return_value=StringType(), exec_fn=ping, action_weight=
    "Light_Action": Action(return_value=StringType(), exec_fn=ping, action_weight=
}
```

Figure D.3 Overview and instructions for usage of query security limits



Simple API admin

With Simple API a custom testing interface is developed, to allow for easy understanding of the endpoint. It can be either downloaded from GitHub or installed from test PyPI -

```
$ pip install -i https://test.pypi.org/simple/ simple-api-frontend
```

The interface contains insecure by desing login form - it will only work when the DEBUG setting is set to True

The interface allows: - testing of authentification - permissions - automatic query Generation - showing list of objects and actions over them - filtering and sorting the instances

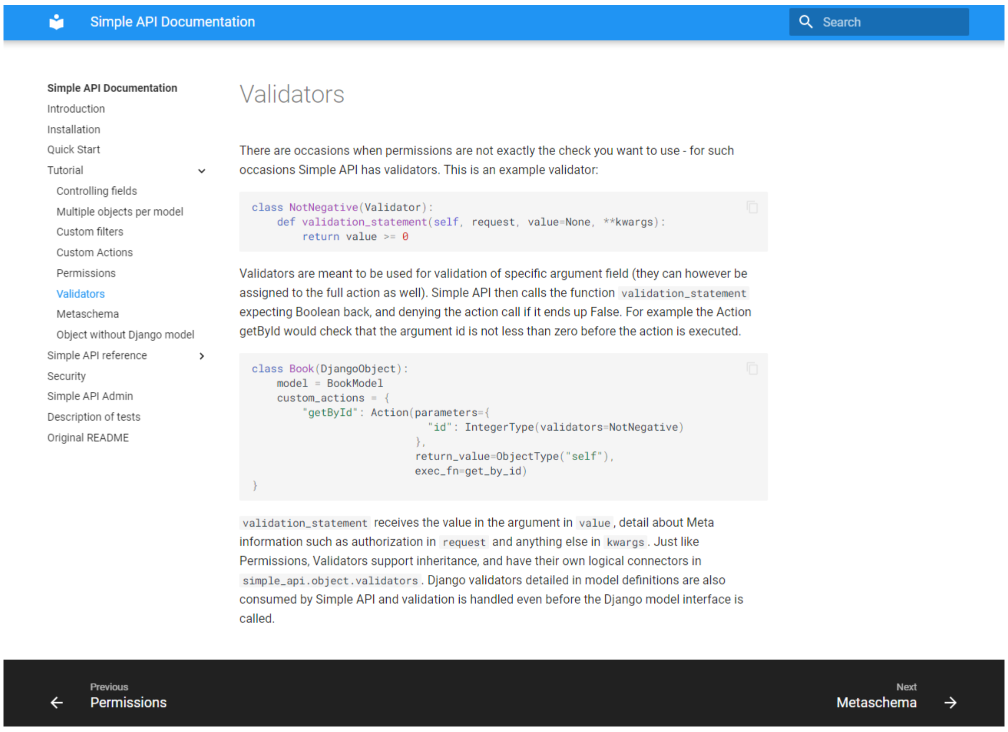
The interface is independent from GraphQL schema - it uses Simple API specific Metaschema

Automaty a Gramatiky - Eliška Šestáková

id: 1
author: Eliška Šestáková
title: Automaty a Gramatiky
ISBN: 000-000-000
restricted: false
borrowed: false

Automata Approach to XML Data Indexing - Eliška Šestáková and Jan Janoušek

Figure D.4 Instructions and installation information for the web interface



Validators

There are occasions when permissions are not exactly the check you want to use - for such occasions Simple API has validators. This is an example validator:

```
class NotNegative(Validator):
    def validation_statement(self, request, value=None, **kwargs):
        return value >= 0
```

Validators are meant to be used for validation of specific argument field (they can however be assigned to the full action as well). Simple API then calls the function `validation_statement` expecting Boolean back, and denying the action call if it ends up False. For example the Action getById would check that the argument id is not less than zero before the action is executed.

```
class Book(DjangoObject):
    model = BookModel
    custom_actions = {
        "getById": Action(parameters={
                        "id": IntegerType(validators=NotNegative)
                    },
                    return_value=ObjectType("self"),
                    exec_fn=get_by_id)
}
```

`validation_statement` receives the value in the argument in `value`, detail about Meta information such as authorization in `request` and anything else in `kwargs`. Just like Permissions, Validators support inheritance, and have their own logical connectors in `simple_api.object.validators`. Django validators detailed in model definitions are also consumed by Simple API and validation is handled even before the Django model interface is called.

Figure D.5 Documentation for usage of Validators

Appendix E

Additional code examples

```
from simple_api.adapters.graphql.graphql import GraphQLAdapter
from simple_api.adapters.utils import generate
from simple_api.django_object.django_object import DjangoObject

from simple_api.adapters.graphql.utils import build_patterns

from .models import CustomUser as CustomUserModel, Post as PostModel


class CustomUser(DjangoObject):
    model = CustomUserModel


class Post(DjangoObject):
    model = PostModel


schema = generate(GraphQLAdapter)
patterns = build_patterns("api/", schema)
```

Listing E.1 Full example of `objects.py` definition for Simple API

```
query Metaschema{
    __types{
        typename
        fields{
            name
            typename
        }
    }
    __objects{
        name
        pk_field
        actions{
            name
            parameters{
                name
                typename
                default
            }
            data {
                name
                typename
                default
            }
            mutation
            return_type
            permitted
            deny_reason
            retry_in
        }
    }
    __actions{
        name
        parameters{
            name
            typename
            default
        }
        data {
            name
            typename
            default
        }
        mutation
        return_type
        permitted
        deny_reason
        retry_in
    }
}
```

Listing E.2 Metaschema query for Simple API in GraphQL

Bibliography

1. O’REILLY, Tim. *What is web 2.0: Design Patterns and Business Models for the Next Generation of Software* [online]. ” O’Reilly Media, Inc.”, 2005 [visited on 2021-04-15]. Available from: `https://www.oreilly.com/pub/a/web2/archive/what-is-web-20.html`.
2. COYLE, Frank P. *XML, Web Services, and the Data Revolution* [online]. Addison-Wesley Professional, 2002 [visited on 2021-03-15]. ISBN 9780596002244. Available from: `https://books.google.cz/books?id=B8tKBAIOu-UC`.
3. CERAMI, Ethan. *Web Services Essentials.* ”O’Reilly Media, Inc.”, 2002. ISBN 9780596002244. Available also from: `https://www.oreilly.com/library/view/web-services-essentials/0596002246/`.
4. GARRETT, Jesse James. *Ajax: A New Approach to Web Applications* [online]. 2005-02 [visited on 2021-04-20]. Available from: `https://www.scriptol.fr/ajax/ajax_adaptive_path.pdf`.
5. FIELDING, Roy Thomas. *Architectural Styles and the Design of Network-based Software Architectures* [online]. Irwine: University of California, 2000 [visited on 2021-04-04]. Available from: `https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm`.
6. *What is REST* [online]. 2020 [visited on 2021-04-24]. Available from: `https://restfulapi.net/`.
7. NOTTINGHAM, Mark. Web Linking - RFC 5988 [online]. 2010 [visited on 2021-05-01]. Available from DOI: `10.17487/RFC5988`.
8. NOTTINGHAM, Mark. Web Linking - RFC 8288 [online]. 2017 [visited on 2021-05-01]. Available from DOI: `10.17487/RFC8288`.
9. KELLY, Mike. *JSON Hypertext Application Lanuage* [online]. 2016-05. Version 08 [visited on 2021-05-01]. Available from: `https://tools.ietf.org/html/draft-kelly-json-hal-08`.
10. BYRON, Lee. *Keynote: A Brief History of GraphQL* [online]. 2019 [visited on 2021-04-12]. Available from: `https://www.youtube.com/watch?v=VjHWkBr3tjI`.
11. *Facebook Class A Common Stock initial public offering declaration* [online]. SEC [visited on 2021-04-05]. Available from: `https://www.sec.gov/Archives/edgar/data/1326801/000119312512240111/d287954d424b4.htm`.
12. RESELMAN, Bob. *What is GraphQL and How Did It Evolve From REST and Other API Technologies?* [Online]. 2019-07 [visited on 2021-04-04]. Available from: `https://www.programmableweb.com/news/what-graphql-and-how-did-it-evolve-rest-and-other-api-technologies/analysis/2019/07/31`.
13. *GraphQL - Specification* [online] [visited on 2021-04-04]. Available from: `https://spec.graphql.org/draft/`.

14. BYRON, Lee. *Exploring GraphQL* [online]. 2015 [visited on 2021-04-15]. Available from: `https://www.youtube.com/watch?v=WQLzZf34FJ8`.
15. *GraphQL - Frequently Asked Questions (FAQ)* [online] [visited on 2021-04-04]. Available from: `https://graphql.org/faq/`.
16. STURGEON, Phil. *GraphQL vs REST: Overview* [online]. 2017-01 [visited on 2021-04-04]. Available from: `https://phil.tech/2017/graphql-vs-rest-overview/`.
17. EIZINGER Thomas, BSc. *API Design in Distributed Systems: A Comparison between GraphQL and REST* [online]. University of Applied Sciences Wien, 2017 [visited on 2021-04-04]. Available from: `https://eizinger.io/assets/Master-Thesis.pdf` . Supervised by Mag. DI Bernhard LÖWENSTEIN.
18. OGGIER, Camille. *How fast GraphQL is compared to REST APIs* [online]. 2020 [visited on 2021-04-04]. Available from: `https://www.theseus.fi/bitstream/handle/10024/340318/Thesis_Camille_Oggier.pdf`.
19. FIELDING, Roy Thomas. *REST APIs must be hypertext-driven* [online]. 2008-10 [visited on 2021-04-04]. Available from: `https://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven`.
20. *Stackoverflow 2020 developer survey - most popular technologies* [online]. 2020 [visited on 2021-04-04]. Available from: `https://insights.stackoverflow.com/survey/2020#most-popular-technologies`.
21. *Django Documentation - FAQ* [online]. 2020 [visited on 2021-04-04]. Available from: `https://docs.djangoproject.com/en/3.1/faq/general/#why-does-this-project-exist`.
22. *Django-Tastypie on PyPi* [online]. 2020 [visited on 2021-04-06]. Available from: `https://pypi.org/project/django-tastypie/`.
23. *Django-Packages - API creation* [online]. 2020 [visited on 2021-04-06]. Available from: `https://djangopackages.org/grids/g/api/`.
24. *Django REST Framework - offical website* [online]. 2020 [visited on 2021-04-06]. Available from: `https://www.django-rest-framework.org/`.
25. *Django Packages : API Creation* [online] [visited on 2020-05-04]. Available from: `https://djangopackages.org/grids/g/api/`.
26. *Benchmark from Django Ninja Documentation* [online]. 2020 [visited on 2021-04-06]. Available from: `https://raw.githubusercontent.com/vitalik/django-ninja/master/docs/docs/img/benchmark.png`.
27. *Proposal for implementation of model based actions* [online]. 2020-08 [visited on 2021-04-06]. Available from: `https://django-ninja.rest-framework.com/proposals/models/`.
28. *Documentation for django-graphene to use django authorization* [online]. 2020 [visited on 2021-04-06]. Available from: `https://docs.graphene-python.org/projects/django/en/latest/authorization/`.
29. *GraphQL - OWASP Cheat Sheet Series* [online]. 2021 [visited on 2021-04-09]. Available from: `https://cheatsheetseries.owasp.org/cheatsheets/GraphQL_Cheat_Sheet.html`.
30. *Validators | Django documentation | Django* [online]. 2020 [visited on 2021-04-13]. Available from: `https://docs.djangoproject.com/en/3.2/ref/validators/`.
31. *Securing Your GraphQL API from Malicious Queries* [online]. 2018-02 [visited on 2021-04-29]. Available from: `https://www.apollographql.com/blog/securing-your-graphql-api-from-malicious-queries-16130a324a6b/`.
32. *devkral/graphene-protector | GitHub* [online]. 2020 [visited on 2021-04-14]. Available from: `https://github.com/devkral/graphene-protector`.

33. *manesioz/secure-graphene | GitHub* [online]. 2019 [visited on 2021-04-14]. Available from: `https://github.com/manesioz/secure-graphene`.

34. *Limiting introspection - Graphene GitHub* [online]. 2020 [visited on 2021-04-13]. Available from: `https://github.com/graphql-python/graphene/issues/524`.

35. *Python Enhancement proposal 227 - Statically Nested Scopes* [online]. 2000 [visited on 2021-04-14]. Available from: `https://www.python.org/dev/peps/pep-0227/`.

36. *Many-to-one relationships | Django Documentation | Django* [online]. 2000 [visited on 2021-04-14]. Available from: `https://docs.djangoproject.com/en/3.2/topics/db/examples/many_to_one/`.

37. *SQLite Foreign Key Support, SQLite documentation* [online]. 2021 [visited on 2021-04-17]. Available from: `https://www.sqlite.org/foreignkeys.html`.

38. *Configuring the cache - Client (React) - Apollo GraphQL Docs* [online]. 2021 [visited on 2021-04-14]. Available from: `https://www.apollographql.com/docs/react/caching/cache-configuration/`.

39. KINUTHIA, Paul. *Disabling Django Graphene Introspection Query* [online]. 2020-06 [visited on 2021-04-17]. Available from: `https://medium.com/@pkinuthia10/disabling-djanog-graphene-introspection-query-8042b341c675`.

40. *Protection against malicious queries | Graphene GitHub issues* [online]. 2020 [visited on 2021-04-17]. Available from: `https://github.com/graphql-python/graphene/issues/907`.

41. *The Django admin site | Django documentation | Django* [online]. 2020 [visited on 2021-04-17]. Available from: `https://docs.djangoproject.com/en/3.1/ref/contrib/admin/`.

42. *Create React App* [online] [visited on 2021-04-20]. Available from: `https://create-react-app.dev/`.

43. *Print exception traceback of GraphQLLocatedError | graphene-django |GitHub* [online]. 2021 [visited on 2021-04-18]. Available from: `https://github.com/graphql-python/graphene-django/issues/1121`.

44. *PEP 335 – Overloadable Boolean Operators* [online]. 2004 [visited on 2021-04-18]. Available from: `https://www.python.org/dev/peps/pep-0335/`.

45. *Resource limitations - GitHub Docs* [online]. 2021-03 [visited on 2021-04-25]. Available from: `https://docs.github.com/en/graphql/overview/resource-limitations`.

46. BEMMAN, Dennis. *Doesn't work with React 16 · pastaclub/react-bootstrap-tooltip-button* [online]. 2018-02 [visited on 2021-04-20]. Available from: `https://github.com/pastaclub/react-bootstrap-tooltip-button/issues/4`.

47. *Cross Site Request Forgery protection | Django Documentation | Django* [online]. 2021 [visited on 2021-04-18]. Available from: `https://docs.djangoproject.com/en/3.2/ref/csrf/`.

48. *Testing Tools | Django documentation | Django* [online]. 2020 [visited on 2021-04-26]. Available from: `https://docs.djangoproject.com/en/3.2/topics/testing/tools/`.

Contents of the included medium

```
readme.txt ........................ Short description of the contents
├── dist .......................... Folder with distributable modules
│   ├── simple_api
│   ├── simple_api_admin
│   └── documentation
├── src
│   ├── simple_api ................ Simple API source code with example project and tests
│   │   └── documentation ......... Sources for the documentation
│   ├── simple_api_admin .......... Simple API Admin source code and build tools
│   └── thesis .................... Sources for this thesis in LaTeX
└── thesis.pdf .................... This thesis in PDF format
```