Haaga-Helia
University of Applied Sciences

GraphQL: The API Design Revolution

Aleksi Ritsilä

Bachelor's Thesis
Degree Programme in Bit
2017

Haaga-Helia
University of Applied Sciences

Abstract

Date

Author(s)
Aleksi Ritsilä

Degree programme
Bit

Thesis title
GraphQL: The API Design Revolution

Number of pages and appendix pages
31 + 2

APIs allow software to speak to each other. Currently the dominant design model is REST. Though it is known to build long lasting APIs, it has issues that come with today's applications, especially with the rise of mobile app consumers. Queries done with REST require many HTTP endpoints that return fixed data structures which can easily result excess data. Issues come also with frequent updates to the applications since knowing what individual clients need is very difficult.

GraphQL was built by Facebook in 2012 for their mobile and web apps. It was designed precisely for the need of more flexibility and efficiency in client, server communication. But will it take over REST?

GraphQL has one "smart" end point that can take complex operations and return only the data the client needs. The shape of a responses matches the shape of the queries. Thus, GraphQL does minimize the amount of data transferred and enables easy evolution of applications.

GraphCool is an open-sourced framework for the development and deployment of serverless GraphQL backends. With it I created a database and accessed it for purposes of my example project. GraphCools documentation is well done and the environment is clear and easy to work with. It provides an excellent way and place for learning GraphQL.

REST is an architectural concept where as GraphQL is a query language. REST can utilize HTTP content-types, -methods and -caching. However, if a vast field of clients with different needs require limited data, with inexpensive queries, GraphQLs API could be what you are looking for. It´s not really a question of "can GraphQL do more than REST" but rather it seems many things are just a lot easier with GraphQL.

Keywords
GraphQL API REST

Table of contents

1 Introduction

Application programming interfaces (API´s) ease the work of the web developers when building applications. They allow software to speak to each other, a call is made from a client to the server and data is given back. The dialogue is done via requests.

Most likely all of the applications you see have an API, currently most of them Representational state transfer (REST) API´s. When done well REST produces long lasting API´s thus over the last decade it has become the standard of today. However, flexibility and performance optimizing aren´t really in focus. Queries done with REST require many HTTP endpoints that return fixed data structures which can easily result with excess data. Issues come also with frequent updates to the applications since knowing what individual clients need is very difficult. These are issues that come important with today's applications, especially with the rise of mobile app consumers.

GraphQL, developed by Facebook in 2012, and publicly released in 2015, was designed precisely for the need of more flexibility and efficiency in client, server communication. GraphQL uses types and fields to describe data and schemas. Queries to API result in precise, predictable results without anything excess. This is done by exposing only one HTTP endpoint because of a very flexible data structure.

It is easy to see why GraphQL is quickly making name for itself as a revolutionary API query language. Though it already has gotten a big following including Twitter, Pinterest and Shopify to name a few, the question is if GraphQL truly is the future of API designs. Is there perhaps a place for both REST and GraphQL, or could they even work together? What do the experts of the field have to say, and what do I find from personal hands on experience.

2 The Context

What is the context in which GraphQL works in? GraphQL is an API query language but what exactly is an API and what makes an API good? Data is passed between the client and the server. But what is the data exactly like?

2.1 API

API stands for Application programming interface. The “A” in API can be a piece of an application, the whole application or a whole server, or just about any piece of software. APIs provide an interface to stored data fitting the needs of an application allowing software to speak to each other. This can be a websites server requesting data from another server where the data is stored in a database. And many of the servers out there today have some kind of an API. (Gazarov 2016).

2.1.1 What does an API do?

To get a good idea how it works we can imagine a restaurant where a customer makes an order, a waiter delivers the order to the kitchen where the meal is made. When the meal is ready it is given from the kitchen to the waiter and passed to the customer. The waiter doesn´t need to understand what is going on exactly on either side of the order but it has to be able to pass it to the kitchen in a way it can be understood there, and deliver back the correct dish (Mulesoft 2015).

An API can be seen as the waiter in the previous metaphor. Client makes a request from the server much like the customer does from the kitchen. Once the data is received from the server it is given to the client by the API. This is all done with public methods and properties. API doesn´t need to know what happens on either side of the request nor do the server and client need to understand each other, they have their own private inner logics outside the scope of the API (Gazarov 2016).

What are the tasks of this “waiter” in computer programming? An API is in charge of the following: (Buna 2016)

- Controller between protected raw data services and software clients
- Parsing client’s requests
- Constructing join statements
- Structuring raw data by clients demands
- Responding with demanded formats

2.1.2 Connectivity between applications

Where do we see APIs in real life? A good example is a website where a person can look for flights. When a person is looking for flights to Barcelona in December the site sends requests to multiple airlines API´s asking for flight data for that particular destination and time. Now the site can display all the desired flights for the person to choose from. When the person is ready to book the flights, they can fill in the needed information right there on the website. The information is then passed back to the API of the airline to finish the order. APIs allow users to view information and complete actions located elsewhere without leaving a website or an app they are using (Gazarov 2016).

The same connectivity between different platforms, provided by APIs, can be seen everywhere. Instagram photos can be shared on Facebook pages. Data from Facebook pages can be displayed on other websites. Google maps with desired locations can be implemented to services and applications. App and game developers can let their user's login with Facebook or Google accounts for social media interactions (Uzayr 2016, 1).

From a web developers side, API´s allow the usage of already built features without attempting to recreate them. This increases productivity and reveals all new possibilities for innovation (Berlind 2015). Applications can tap into features of others creating connectivity between different platforms in a secure and controlled way. APIs open up all new business possibilities for commercializing services and features.

2.1.3 What is a good API?

API, like its name suggests, is an interface and should be designed for developers. Choosing a protocol for an API should be done with a best possible developer experience in mind. Any familiarities to protocols that the developers already have should be taken into consideration. (Doglio 2015, 2)

A good API needs good documentation. It lets the developer know how to exchange data between software. The documentation can consist of sample requests descriptions and snippets making the modern APIs standardised and developer friendly (Mulesoft 2015). This will shorten the developers learning curve for a system and provide help for error handling. The documentation should be always up to date with possible changes.

Handling all the traffic and performing well at the same time is key. This can be achieved by spending resources only when and where needed. (Doglio 2015, 2) A good API should also be easily updated and modified. How will a new version affect the existing work, queries and clients, and is it backwards-compatible.

Lastly the APIs should be designed to be secure. Who will be granted the access to the API (Authentication) and once logged in, what resources will be accessible inside the system (Authorization). Standardizing an API also improves security by enabling good monitoring and management. (Mulesoft 2015) HTTPS should be a must to encrypt the data between the client and server. (Doglio 2015, 2).

2.2 JSON

JavaScript Object Notation, JSON is a standardized textual representation of data. Data passed between a server and an application is often JSON. It has become the standard data transfer format over past years. It only includes directly related data to the queries and therefore is light weight. The format is also very simple, and easy to write and read improving the developer experience. It also allows the developer to use different data types providing additional information to the transfers. (Doglio 2015, 2)

A JSON consists of a key and a value creating a pair. The key is a string enclosed in quotation marks, and it names the JSON. A value can be a string, a number, a boolen, an object or an Array. An object is key/value pair of its own and an Array a list of values. (Smith 2015, 4)

- A key/value pair: “Key” : ”Value”
- An object: { “Key” : ”Value” }
- An array: [“value”, “value”]

3 REST

Representational state transfer or REST was first introduced to the world in 2000 by Roy Fielding in his doctoral dissertation (Doglio 2015, 1). Over the past years it has become the dominant Web service design model because of its simplicity over other designs available at the time. REST is an architectural concept aiming to help create and organize distributed systems, meaning it´s not a standard with rules written in stone, rather a style to show how a good system behaves (Doglio 2015, 1). The architecture aims to create well performing, transformable and long-lasting systems.

3.1 Resources, Representations and Requests

Like its name suggests REST transfers a representation of resources. The resources define the type of information transferred, and actions and services of a system. They can be anything conceptualized, a text or an image for instance. Client sends a specific header asking for a resource to the server. The API is responsible of finding the best representation of that resource on the server side. The resources can be represented in many forms. The data can be represented for example as a JSON or XML file, and one resource can have multiple representation. The client will read and parse the information and request the most suitable form of the resource. (Doglio 2015, 1)

A resource has an identifier in a form of an URL (Doglio 2015, 1). The URLs used should be logical and accessible by other parts of the system, as well as unique for one specific resource. Resources also have metadata to describe them, for instance a content type. The resources are queried with the identifier URL and a request. GET request is used for read-only purpose queries and for more complex queries that alter the stage of the data (create, update and delete) POST is often used (Uzayr 2016, 1).

- GET /api/resource .json
- GET /api/resource .xml

In addition to the previous verbs, PUT is used to create- and DELETE to remove a resource. Some actions such as searching and filtering resources aren´t done directly with any of the requests mentioned. These complex actions are done with a “?” sign which can be applied to all of the verbs. (Doglio 2015, 1)

- GET /api/resource ?q=[SEARCH-TERM]
- GET /api/resource ?filters=[COMMA SEPARATED LIST OF FILTERS]

3.2 Constraints

REST defines a system with constraints. These constraints are discussed in the next sub chapters. By adding the following constraints one by one, on top of the last, to all components, harmony will be reached in system interactions (Doglio 2015, 1).

3.2.1 Client-Server

This constraint describes the basic nature of APIs. Server is in charge of services, and the request regarding its services. Client makes the requests to the server regarding its services. This separates the frontend code, and the server side code storing and processing the data. Allowing independent development and evolution for both. (Doglio 2015, 1)

3.2.2 Stateless

The requests coming from the client must include all information that is needed for the server to fully understand that request. This will improve system monitoring and reliability. It also means data won´t be stored between request which results in faster freeing of resources and easier implementation. (Doglio 2015, 1)

The downside of the constraint is potential overhead of the requests. When all information is required for the requests from the server it is easy to get excess data and cause extra traffic.

3.2.3 Cacheable

Responses to requests should cached. All ready requested data can be fetched from cache layer living either on the client or the server. This results in faster performance. The downside, if the caching is designed poorly, is that the data returned can be no longer fresh. (Doglio 2015, 1)

3.2.4 Uniform Interface

Setting a uniform interface for the components of a system will simplify interactions between the client and the server. Unique identifiers for resources should be used, and CRUD (create, retrieve, update, delete) actions provided for the requests giving independent clients clear set of rules to follow when implementing (Doglio 2015, 1). Keeping in

mind that HTTP isn´t the only protocol for REST this can be done with unique HTTP paths for each resource (/api/resource), as well as using HTTP verbs (GET, PUT, POST, DELETE) for the interactions.

This constraint can have its downside to it. Uniform interface will standardize all communications even if a more suitable form would be available for a certain piece of software. Most often when using REST and moving away from web based systems, where REST is optimized, performance can he harmed. (Doglio 2015, 1)

3.2.5 Layered System

Separating the systems components into multiple layers can simplify the system, improve safety and level the traffic loads. It also allows new layers to be added, and existing ones deleted or modified without tempering with the whole system.

3.2.6 HATEOAS

HATEOAS, short for Hypermedia as the Engine of Application State is a constraint of REST, distinguee from other network architectures. Unlike in many other architectures with interactions guided only in the documentation or language, it allows the client to interact with an application entirely through hypermedia links without prior knowledge of how to interact with the application. HATEOAS aims to provide systems longevity and good possibilities for evolution.

HATEOAS resembles a familiar process of entering a web page. The client will have a fixed URL, a root endpoint, to enter the application from. There they are provided with set of links for possible first steps, a resource list. Each resource has metadata, in which there should be a set of hypermedia links. These links tell the client what to do with that resource. This means all the next steps regarding that resource are there for the client. This is how the client transitions through the application and makes possible manipulations to the representations of resources. The metadata also includes a possibility of specifying the content-type of a representation of a resource, if there are more than one. The client can then choose the best suiting one for the particular use case. (Doglio 2015, 1).

The root endpoint exposes links that are used throughout the client's transition in the system. Not all links are accessible through every endpoint. The API only returns the links

needed for that particular use case. The client will still have access to everything exposed in the root endpoint for displaying information to an end user. (Doglio 2015, 1)

HATEOAS improves the decoupling of the client and server significantly. It also allows changes to the resources and functionalities on the server side without affecting the client. Thus, the REST system clients can start interactions knowing only one endpoint – the root end point and keep interacting through the evolving of the server side without complications.

4 GraphQL

GraphQL is a data query language for APIs and runtime for fulfilling those queries with your existing data (GraphQL.org). It is a language that can be taught to a software client application. The application can then communicate with a backend service, also speaking GraphQL, to request data. The language is close to JSON and has operations for reading (queries) and writing (mutations) data. The operations are strings for which the GraphQL service can then respond in desired format, often JSON. (Buna 2016, 1)

It is a run time, a layer, written in any language for a server application to understand- and respond to GraphQL requests. The layer defines a graph-based schema of possible data services for the client. Backend servers speak their own languages and this layer can be implemented on top of existing server logic allowing it to pass the GraphQL request down the logic and get the requested data. This separates clients from servers and allows independent evolving for both sides. (Buna 2016, 1) GraphQL can be confused as being a database technology which it is not. It is a query language for APIs, not for databases (Howtographql.com, Introduction).

In this chapter, using http://graphql.org/learn/ as a guide, I will demonstrate what results different operations will get.

4.1 History of GraphQL

Designed and used by Facebook since 2012 (publicly released and open sourced in 2015) for their web- and mobile applications, GraphQL was created to solve the issues of the times APIs, especially with over fetching. This was caused by the basic structure of the queries, causing a series of requests with excess data. This is especially an issue with mobile apps that are used with devices and network conditions that can´t handle huge

amounts of data and payloads. Today it is maintained by a vast community across the globe consisting of both companies and individuals (Howtographql.com).

Facebook needed an API strong enough to describe all of Facebooks data-fetching and at the same time easy to use and learn for their product developers. Facebooks mobile apps were becoming more complex resulting in performance issues and crashing. The news feed of theirs had been delivered as HTML and an API data version of it was seen to be needed. Facebook tried RESTful server resources and FQL tables to solve their issues but were frustrated with the difference of data used in apps and in server queries. They wanted a graph of object with used models like JSON instead of resource URLs, secondary keys and join tables. There was also a lot of code for the server when preparing the data and for the client to parse. (Byron 2015)

So, what did Facebook come up with to solve the issues? GraphQL has one “smart” end point instead of multiple ones like REST has. This endpoint can take complex operations and return only the data the client needs. (Greif 2017) Facebook wasn´t the only one looking into making API interactions more efficient but once GraphQL was open-sourced many programs such as Coursera were cancelled as they hopped on to GraphQL. (Howtographql.com).

4.2 Precise queries and results

The communication is done with text documents in GraphQL query language. These documents contain one or multiple read or write operations. In GraphQL the read operations are called queries and write operations mutations. The queries ask for specific fields of objects. A field is like a function, it will return a response of a primitive value, an object or an array of objects. With these fields, the results of a query can be narrowed to only the data that is wanted. The server knows exactly what is asked, and the client will get back what it expects. The fields map the properties of objects. A query starts with a root query object, an entry point to the data. Queries are formed of selection sets that can be nested. Each selection set is represented with curly braces, and they tell the GraphQL server what properties to read from the field. (Buna 2016, 2)

Here a name field of a hero object is queried. The result of a query will have the same structure as the query itself, as seen on the table 1. We don’t want to receive everything about a hero object, only the name.

Query	Result

{
  hero {
  name
  }
}
	{
	  "data": {
	  "hero": {
	  "name": "R2-D2"
	  }
	  }
	}

Table 1 GraphQL Query

4.3 Many resources with one request

The queries have a possibility of sub-selections of fields. This allows the client to query related data in only one request instead of many roundtrips. It is also possible to add arguments to the fields. With these arguments data can be already transformed into desired form.

An id argument will specify a human object and a unit argument will transform the height field into a form wanted. The sub field of friends will result only returning the names of the friend object of the human object with id:1000.

Query	Result

{
  human(id: "1000") {
  name
  height(unit: METER)
  friends {
  name
  }
  }
}
	{
	  "data": {
	  "human": {
	  "name": "Luke Skywalker",
	  "height": 1.72,
	  "friends": [
	  {
	  "name": "Han Solo"
	  },
	  {
	  "name": "Leia Organa"
	  },
	  {
	  "name": "C-3PO"
	  },

	{
	"name": "R2-D2"
	}
	]
	}
	}

Table 2 GraphQL Query with arguments

4.3.1 Aliases

The objet fields of of the queries results match the names of the fields in the query itself. However for some purposes the the names the UI is using can be different from the ones of the server. In graphQL any fields name can be customized with aliases. These aliases should be named describtive to the particular query or by the usage purpose of the UI. The query will still ask for the same data but the return will use the specified names in the responses. Thus the client will not need to a work extra processing the data before using it. Aliases also allow the same field to be asked multiple times if needed. (Buna 2016, 2)

Query	Result

{
empireHero: hero(episode: EMPIRE) {
name
}
jediHero: hero(episode: JEDI) {
name
}
}
	{
	"data": {
	"empireHero": {
	"name": "Luke Skywalker"
	},
	"jediHero": {
	"name": "R2-D2"
	}
	}
	}

Table 3 GraphQL Query with aliases

4.3.2 Fragmentation

GraphQL also allows fragmentation in a query. Instead of repeating fields, for comparing two sets of data, for example, it is possible include reusable fragment units to queries where they are needed. Sub-components and multiple views can ask for isolated data without duplicating the query logic.

Fragments can´t be used on their own since they are just partial operations. They must be prefixed with a spread operator (three dots) to be used inside a full operation. When GraphQL server sees the three dots followed by a name inside a query, it will look for a fragment with the same name. The content of the fragment will be then placed in the spread operator. The fragment has to fit in the place where it is used and thus can only be used within the selection. (Buna 2016, 2)

Query	Result
{	{
leftComparison: hero(episode: EMPIRE) {	"data": {
...comparisonFields	"leftComparison": {
}	"name": "Luke Skywalker",
rightComparison: hero(episode: JEDI) {	"appearsIn": [
...comparisonFields	"NEWHOPE",
}	"EMPIRE",
}	"JEDI"
	]
fragment comparisonFields on Character {	},
name	"rightComparison": {
appearsIn	"name": "R2-D2",
}	"appearsIn": [
	"NEWHOPE",
	"EMPIRE",
	"JEDI"
	]
	}
	}
	}

Table 4 GraphQL Query with fragments

4.3.3 Variables

Using variables as an input will make the GraphQL queries dynamic and reusable. By replacing a static value in a query with $variableName followed by its type, declaring it as an accepted variable of the query and passing a value (variableName: value) in a separate dictionary for transport-specific variables (usually JSON). After the variable is defined at

the top of the query it can be used anywhere inside that query operation (Buna 2016, 2). Same variable names can be used in different operations but have to be unique inside each operation (Buna 2016, 2).

The variables can be required for a query by using a “!” sign after the type ($variable-Name: type!). Default values can be also assigned for the queries after the type declaration part. If a variable is passed for the query, it will override the default one.

Query	Result

query HeroNameAndFriends($episode: Episode)
{
  hero(episode: $episode) {
  name
  }
}
	{
	  "data": {
	  "hero": {
	  "name": "R2-D2"
	  }
	  }
	}

Variables	{
	  "episode": "JEDI"
	}

Table 5 GraphQL Query with variables

4.3.4 Directives

The GraphQL server can customize the response of query based on a directive. The response can either include something more (@include(if: Boolean)) or skip something (@skip(if: Boolean)) based on if the variable is true or not. This allows for the UI to have a summarized and detailed view for example. (GrphQl.com)

Query	Result

query Hero($episode: Episode, $withFriends: Boolean!) {
  hero(episode: $episode) {
  name
  friends @include(if: $withFriends) {
  name
  }
  }
}
	{
	  "data": {
	  "hero": {
	  "name": "R2-D2",
	  "friends": [
	  {
	  "name": "Luke Skywalker"
	  },
	  {
	  "name": "Han Solo"
	  },
	  {
	  "name": "Leia Organa"
	  }
	  ]
	  }
	  }
	}

Variables	{
	  "episode": "JEDI",
	  "withFriends": true
	}

Table 6 GraphQL Query with directives

4.3.5 Mutations

Updating data can be done with mutations in GraphQL and fields as data inputs. The mutations fields will run one after another, in series, unlike with queries. This means many mutations can be done in a single query, and the first one is guaranteed to finish before the one after and so on.

4.4 The type system

Though the results of GraphQL queries result in predictable results, a schema should be defined. The schema explains the capabilities of the GraphQL server. It can be seen as a contract between the client and the server defining how the client will be able to access the data (HowtographQL.com). GraphQL has a type system to describe the data that is possible to query. Each service defines this set of types which the queries are then executed and validated on. (GraphQL.org, Schemas and types)

4.4.1 Object types

Object types are the core of the queries and represent what kind objects can be fetched and what fields they have. Each of the fields can have none or more arguments which can be required or not. If an argument is optional, it can have default value. All arguments are passed by a name and thus have to be always named. (GraphQL.org, Schemas and types)

4.4.2 Scalar types

In order for the fields to resolve actual data GraphQL has scalar types. Besides custom scalar types there are following default scalar types: (GraphQL.org, Schemas and types)

- Int: A signed 32-bit integer.
- Float: A signed double-precision floating-point value.
- String: A UTF-8 character sequence.
- Boolean: true or false.
- ID: a serialized unique identifier

Values can also be set to be from a certain set of values, this can be done with an enum type. Lists and non-null can be used outside objects, scalars, enums and declarations by adding additional type modifiers, ! for non-null and [ ] for lists.

4.4.3 Query- and Mutation types

In addition to object types every GraphQL service will have a query type and can have mutation types. These both serve as an entry point to the schema but are still just like object types and their fields work the same. (GraphQL.org, Schemas and types)

4.4.4 Subscription types

Some applications can require a real-time connection to the server to get immediate information about some events, this can be done with subscriptions in GraphQL. When an event is subscribed by a client a steady connection is opened to the server. Whenever that event happens data is pushed to the client from the server. This means that though written in the same syntax as queries and mutations, subscriptions represent a stream of data instead of a request-response-cycle. (howtographql.com, core-concepts)

4.4.5 Interface and Union types

Interfaces and unions are both abstract types used to group other types (Buna 2016, 3). If types have common fields, interface can be used to return that group of types as a whole. Interface will define fields that the implementation must contain and thus guarantees that those fields are always supported in that interface. Types used will have all fields of the interface but can also bring extra fields of their own. Inline fragments can be used to ask for these extra fields outside of the interface. (Buna 2016, 3).

Unions can be used when types have no common fields but are still wished to be grouped. The types have to be concrete, not other unions or fragments. Union will define a list of different implementations. Inline fragments are used to ask for the fields of the types represented by a union. (GraphQL.org, Schemas and types)

4.4.6 Input types

Input types look the same as object types in GraphQL schema, they just have keyword “input” instead of “type”. Inputs allow complex objects to be passed into fields. This is useful in mutations where new objects are created. The input object fields can`t have arguments and they can´t be mixed with output types. (GraphQL.org, Schemas and types)

5 Creating- and Connecting to a GraphQL API

Following a tutorial “React & Apollo Quickstart” from graph.cool I will create a database and access it for the purposes of my example project. The project will be a website for a farm. The farm produces wools of different sorts as well as wood. The website should have a page where all the different products are displayed and a detail page for each product.

5.1 GraphCool

GraphCool is an open-sourced framework for the development and deployment of server-less GraphQL backends. It assists the developers with its framework and uses the GraphQL SDL syntax for easy defying and evolving of the database schema. Data can be queried with GraphQL CRUD API safely by exposing what is wanted from the database schema to the frontend apps through the API Gateway. (graph.cool) Once signed into, GraphCool provides the user with the possibility to create projects in which one can create a schema add and modify data, test queries in the playground feature, as well as excellent documentation on all the features.

5.2 Setting up the API connection

First off, I will clone graphcools react application to get started with. In terminal: `git clone https://github.com/graphcool-examples/react-graphql.git` and move to the directory: `cd react-graphql/quickstart-with-apollo`. Next up there I will need to install the graphcool CLI: `npm install -g graphcool`. I will create a local file structure for the Graphcool service. This is done in the server directory: `graphcool init server`.

The data models and type definitions needed for the project will be written in types.graphql file, in the GraphQL Schema Definition Language. In addition to ID, created at and edited at fields, my example projects product type needs a name, a description, an image and an optional type fields:

```
type Product @model {
  id: ID! @isUnique
  createdAt: DateTime!
  updatedAt: DateTime!
```

```
  name: String!
  description: String!
  imageUrl: String!
  type: String!
}
```

Table 7 Product type

Since the Product type in the data model is now only local, next I should deploy the service. It is done in the server directory, running a command: `graphcool deploy`. Once the deployment is completed the service will be available from HTTP endpoints printed in the output of the deployment command or from a `graphcool info` command. The Product type is added to the data model, CRUD operations are generated and exposed by the GraphQL API.

To connect the react application to the GraphQL API from the Graphcool service, I need to insert the previously printed HTTP endpoint for a Simple API into "./src/index.js" file of our React app: "const httpLink = new HttpLink({ uri: '__SIMPLE_API_ENDPOINT__' })". Then I will launch the app and build the dependencies. In terminal: `yarn install` and `yarn start`.

5.3 Inserting data with GraphCool

In the GraphCool console for the project I can easily insert data for my application. Under DATA the required fields are filled and then the object can then be saved.

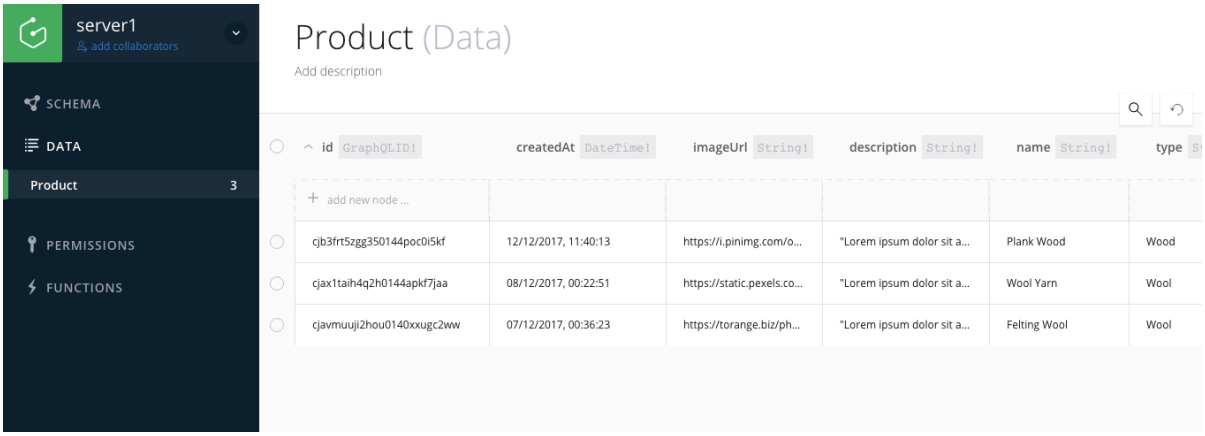


Figure 1 GraphCool console view

5.4 Querying data from the GraphQL API

In the playground feature provided by GraphCool I can test the queries needed for the project. The playground can be opened with a “graphcool playground” command.

Once the desired queries are formed they are added to the React application. On the files where data is queried I will “import gql from 'graphql-tag'” and add the queries to be used.

The application has a page where all of the products (image and name) of the farm are displayed:

```
const ALL_PRODUCTS_QUERY = gql`
    query AllProductsQuery {
      allProducts(orderBy: createdAt_DESC) {
        id
        name
        imageUrl
      }
    }
```

Table 8 All products query

And then return out the data received:

```
  componentWillReceiveProps(nextProps) {
    if (this.props.location.key !== nextProps.location.key) {
      this.props.allProductsQuery.refetch()
    }
  }

  render() {
     return (
      <div >
        <div className='list-container'>

          {this.props.allProductsQuery.allProducts && this.props.allProductsQuery.allProd-
ucts.map(product => (
            <Product
```

```
            key={product.id}
            product={product}
            refresh={() => this.props.allProductsQuery.refetch()}
          />
        ))}
      </div>
      {this.props.children}
    </div>
  )
 }
}
```

Table 9 Displaying all products

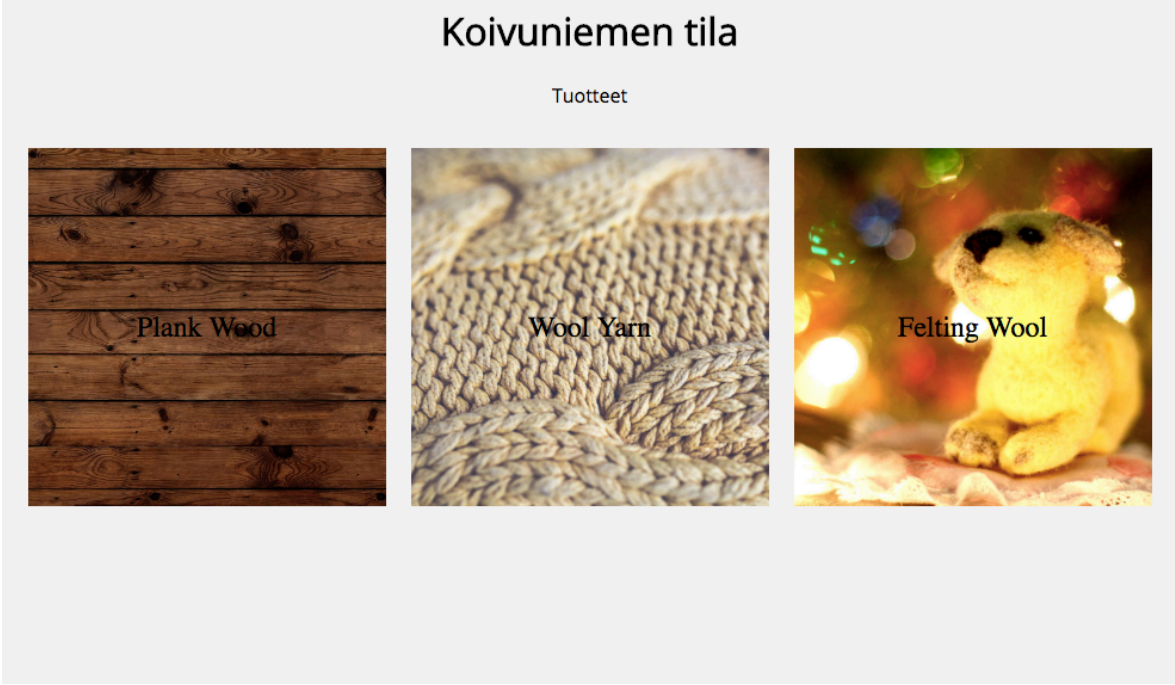


Figure 2 All products browser view

The detail page for each product should receive the name, image and description of that particular product:

```
const PRODUCT_QUERY = gql`
  query ProductQuery($id: ID!) {
    Product(id: $id) {
      id
      name
      imageUrl
```

```
    description
  }
}
```
Table 10 Detail page product query

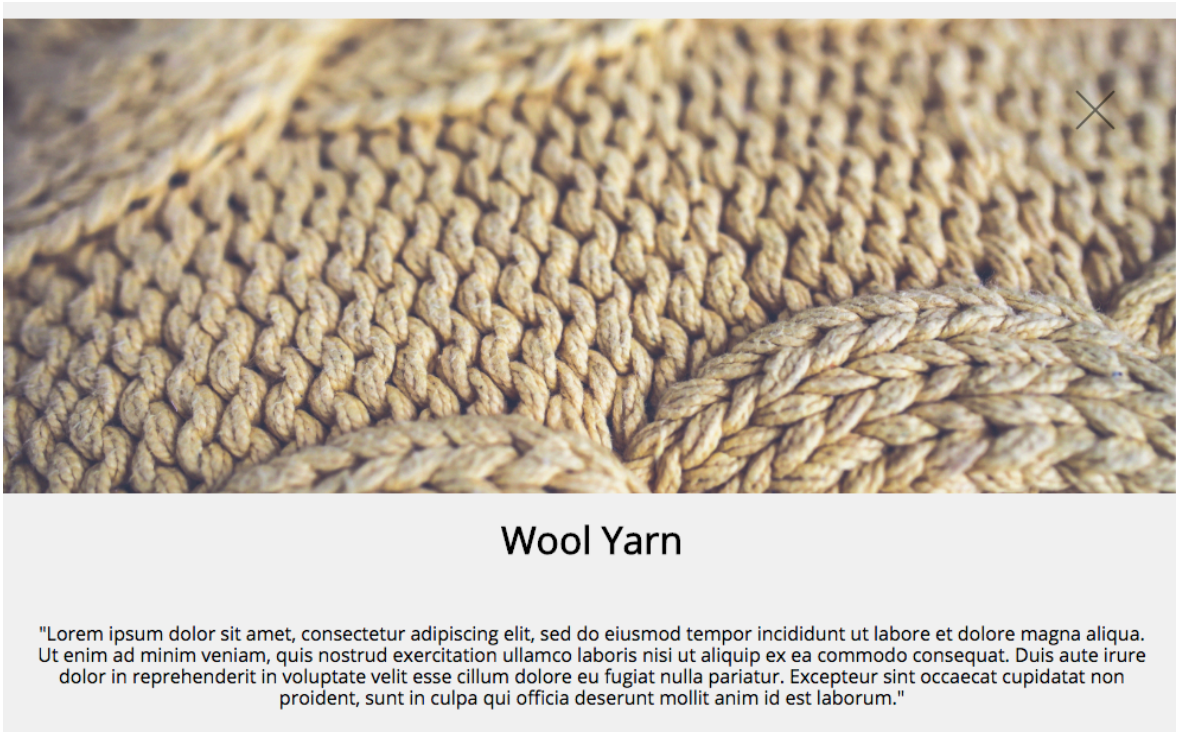
And again, we return the data received:

```
render() {


  const {Product} = this.props.productQuery

  return (
     <div className='main'>
     <div className='image'
        style={{
        backgroundImage: `url(${Product.imageUrl})`,
        backgroundSize: 'cover',
        backgroundPosition: 'center',

       }}>
     <div
      className='close  pointer'
      onClick={this.props.history.goBack}
     >
      <img src={require('../assets/close.svg')} alt=''/>
     </div>
     </div>
     <div>
      <div>
       <h1>{Product.name}</h1>
       <p className='text'>{Product.description}</p>
      </div>
     </div>
     </div>
  )
 }
}
```
Table 11 Displaying detailed product info



Wool Yarn

"Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum."

Figure 3 Product detail browser view

6 REST vs. GraphQL

Though GraphQL is often seen as a replacement for REST that isn´t necessarily the truth, they could be even used side by side. REST is an architectural concept where as GraphQL is a query language. REST focuses on creating long lasting APIs while GraphQLs strong points are in performance and flexibility. REST can utilize HTTP content-types, caching and so on, GraphQL has its own practices. REST has HATEOS which isn´t necessarily always used anyways. (Sturgeon 2017)

6.1 Requests

Both GraphQL and REST work around the idea of a resources specified by IDs. In REST, the identity of an object as well as the shape and size are called in the endpoint of the request while in GraphQL the identity is separate from the fetch and the format will be determined by the client. Both APIs will have a list of possible actions with separation of read and write operations. REST has list of endpoints beginning with GET or POST whereas GraphQL has a schema of queries and mutations. (Stubailo 2017)

HTTP verbs used by REST can be seen easier to understand for humans. GET will retrieve data, PATCH updates and DELETE request will delete. The verbs are quite obvious for the consumer. One knows what they are indented for, what will happen and that they are safe to use without even reading the documentation. GraphQL might not be as clear. The only certain way to know what a mutation for deleting an item is called is to look it from the documentation. API providers will have their own ways of naming operations and keeping everything consistent even within one API isn´t guaranteed. HTTP protocol can thus provide more consistency and predictability. The same goes for HTTP status codes, they are both machine and human readable. Using REST, we are able to know what happens just from the HTTP status. GraphQL queries will be “OK” even if the query wouldn´t have actually been successful. (Lauret 2017)

Using HTTP methods and responses bring limitations. REST accesses multiple endpoints to gather data. For example, first one endpoint to fetch a user, another to get all posts of that user and one for getting a list of followers. Often the developers are forced to heavy customization and interpreting to keep the payloads to a minimum. The clients however won´t have any control over the customization, unless they are given the rights to do so. Otherwise it’s all up to the developer to provide queries and responses needed. Still the two can be seen as completely different languages all together. Almost like asking something in English and getting an answer in Japanese. (Buna 2016)

GraphQLs way is the smallest possible requests, where REST will have the fullest (Sturgeon 2017). GraphQL queries can call multiple resources, have arguments in any fields and fetch related data according to relations creating a nested response. The shape of a responses in GraphQL are built by the execution library to match the shape of the queries. (Stubailo 2017)

6.2 Usage of the API

GraphQL was created out of the needs of today, especially the needs of mobile clients. Low-powered devices transfer data with not ideal networks. The mobile clients need power and ability to choose what data to consume. The mobile applications also evolve rapidly and their versions can´t be controlled well. Thus, GraphQL does minimize the amount of data transferred and enables new features to be added without removing the old ones. (Buna 2016)

Though both REST and GraphQL are most often used for sending data back and forth with REST it is possible to do more. Small files such as images can be sent with URL based uploads. This however won´t be useful anything bigger such as videos, large images etc. For those use purposes dedicated services should be used which is the GraphQL approach for all uploads. (Sturgeon 2017)

REST has a unique feature of HATEOS, allowing the users travel and explore the API through links. GraphQL requires more documentation to be clear for the user. (Morgan 2017)

6.3 Developer experience

The developer experience is driving the user experience and thus can be seen even more important. The language given to work with, should let the developers express the data requirements close to the way the data is used in the actual application. This how GraphQL was built and it´s strong type system also helps to avoid and report misuses. (Buna 2016)

Multiple different frontend frameworks and platforms ran by client applications cause problems when building and maintaining a single API. It is difficult to meet the requirements of them all. GraphQL can ease the process by giving each client access to just the date they need. (howtographql.com)

All in all, GraphQL forces efficient queries and removes many design questions. With GraphQL clients can write their own queries and use the language of their choice instead of forcing the developers make custom endpoints, -representation or –APIs to solve issues with different clients and their needs. (Sturgeon 2017) The declarative language for data requirements in GraphQL is close to the way of thinking data requirement in English easing the thought work that goes into it. (Buna 2017)

6.4 Caching

REST can easily rely on HTTP caching and avoid re-fetching the same data because it is endpoint-based. With GraphQL the client has to handle caching. A client library such as Relay can provide a cache system. These systems are complex and since they might not provide information of the how long the data is valid, refreshing the cache is up to the consumer. (Lauret 2017)

Though GraphQL doesn´t have globally unique identifiers such as URLs for each request, similar identifiers can be exposed for the clients to use for caching purposes. Also as the requests come more customized the less cache hits are likely to happen meaning REST would be forced to go the same route of handling caching as GraphQL. (Sturgeon 2017)

6.5 Evolution of the API

Both REST and GraphQL can version and evolve but deprecations are GraphQLs strong point. Since monitoring field usage is made easy in GraphQL, developers can track and reach out to clients to whom new versioning's will affect. Unlike in GraphQL where clients are forced to specify what fields they wish to be returned, in REST knowing this is very difficult. This will result in entire new versions and the old one can´t be dropped until all of the clients have made the switch, even if the changes only affect the usage of some of them. (Sturgeon 2017)

Applications tend to have frequent updates with continuous deployment nowadays. REST APIs expose the data in a manner where modifications are most likely needed whenever design changes are done on the UI. The data needs change and more or less data might be needed to request caused by each change done in the frontend. With GraphQL, clients themselves specify the data requirements and backend adjustments won´t be necessary

when the design or data needs of the frontend change. Development and product iterations can thus be extremely fast. (howtographql.com)

REST APIs tend to be created simple at first and later have query language like features. GraphQL on the other hand will give a query language syntax and software development kits. Same results can be achieved with REST but it is still a concept with no written rules. (Sturgeon 2017) “With GraphQL, clients and servers are independent and they can be changed without affecting each other” (Buna 2016).

With web applications, it is easier to control API versions by simply pushing new code. Mobile applications are another story since versioning can´t be controlled easily at all. A user might download an app and keep using the version installed without updating it. GraphQL allows APIs to grow without versioning. New fields can be added without removing old ones as the graph can flexibly grow leaving paths for the for the old APIs. (Buna 2017)

6.6 Which one to choose

Which one to choose GraphQL or REST? You should consider the following questions: (Sturgeon 2017)

- What kind of clients do you have and are they different from each other? If the application will have a web and mobile versions, or even if it just a mobile app, the strengths of GraphQL could be where you base your decision.
- Are your clients handling caching? And even if they are, caching will be easier with REST.
- Do you want to give power to the client or keep them “dumb”? With GraphQL the clients will be in charge of a lot otherwise done by the developer.
- How much you value HTTP debugging- and cache proxies? If you have already a lot of HTTP knowledge you will benefit of it using REST.

If a REST API is following good practices the benefits of GraphQL might fall short. However, if a vast field of clients with different needs require limited data, with inexpensive queries, GraphQLs API could be what you are looking for. That being said, in real life of service-oriented-architectures, multiple different services with multiple APIs are often used. Some might be REST, some GraphQL. (Sturgeon 2017)

7 Discussion

As stated in the beginning of this thesis the question was if GraphQL truly is the future of API designs. After studying articles and books on the matter it is safe to say there definitely is demand for a “revolution” in the API world. GraphQL really successes to tackle the issues of REST and meets the demands APIs have today. Still we have to remember that it is a very new query language. This both means that it has the attraction of something new and exciting, but also that it will evolve and possibly make up for the flaws it has at the moment. As the study bases mostly in articles, I have tried to keep the previous points in mind staying trust worthy and objective.

7.1 Starting point

I´m not personally one to get excited straight away when something new comes out. I don´t think that new is always better, rather the opposite. I often feel many unnecessary changes are being made for the sake of staying on the “cutting edge” of technology. This can cause time and money being spent on something only making things more difficult. That being said everything evolves and that’s a good thing. It also means we have to sometimes move away from our trusted old habits to keep things moving in the right direction.

The rise in the mobile consumer base, complex nature and constant evolution of apps in general has created many issues and room for improvement for the APIs and the current standard, REST. The world has changed a lot since its launch and though REST has stand the test of time and evolved to meet many needs, now might be the time for a fresh start. RESTs biggest problems seem to be over fetching of data slowing down apps and the differences of many frontend applications that need work with one same API. This makes it very obvious for a company such as Facebook to have stepped in to change things.

7.2 Findings

After studying articles and books I feel I got a good view on the general opinions of some experts of the field. I do consider books more trustworthy than articles but unfortunately as the topic is still fairly fresh not many books were found on GraphQL. Luckily, I was able to find articles written objective and questionable manner giving me trust to the points made. There weren´t much controversy between the articles either. None of the articles claim

GraphQL to be magically solve all of the issues of APIs but they all are very excited about it, listing both weaknesses and advantages that come with it. The authors were aware of the fact that GraphQL does has a lot of “fuss” going around it and clearly made efforts trying to find its flaws.

It´s not really a question of “can GraphQL do more than REST” but rather it seems many things are just a lot easier with GraphQL. It is just build for the demand of today whereas REST was for the past. GraphQL will evolve during the times to come but even now it doesn´t seem to have any major short comings. The biggest issues when compared to REST pointed out by many authors were the lack HTTP verbs, status codes etc. and caching. But even when discussing about the previous the authors seem optimistic and don´t see them as a major problem.

For someone, such as myself, learning API languages today GraphQL might be the way to go. I don´t feel learning what an API is would have been any easier with REST rather the opposite. GraphQLs language and logic were easy to understand. Like the authors of the articles examined, I don´t think GraphQL is just a phase and will provide developers with many interesting and helping features in the times to come.

Both query languages have back end development frameworks such as graph cool. Which one prefers will mostly be based on what is wanted to be achieved with a framework. All of them have their own strong points. After trying out graph cool, all I can say is that their documentation is well done and the environment is clear and easy to work with. It provides an excellent way and place for learning GraphQL.

It would appear to be that if one is a good REST API developer they most likely will be good with GraphQL as well. I personally feel designing and developing an API is more about the logic than the tools. If GraphQL brings new challenges or things to wonder about they are mostly in the end something that a should be mastered which ever route a developer chooses to take anyway. Knowing both will always be the best. Even a freelance developer who could ideally choose which tools, languages etc. to work with, might end up in a situation where a framework works only, or just better, with “the other API”.

7.3 The thesis process and own learning

Looking back to the starting point I´m happy with the results of the thesis. I personally did not have any prior experience with APIs and therefore learned a lot in the process of this thesis. Tackling on such a big topic with a vast field of new terminology, principles and

logics wasn´t the easiest, especially at first. I managed to find good articles directed to novices such as myself easing the learning process. That is something I hope I have succeeded to do as well. I hope my thesis will be readable for an It-student wanting to understand APIs and GraphQL without prior knowledge.

As stated before I didn´t find many books on GraphQL as they yet to exist still. REST and APIs in general had more available. Luckily, with all the excitement around it, a lot of articles had been already written about GraphQL, many comparing it with REST. These articles provided knowledge and perspective not letting me feel it was only my opinions I had to base my study on. I gathered points that were the most mentioned throughout the study, as well as personally interesting or important factors.

I can´t say that I have become a master of GraphQL just yet. I however got a good understanding of APIs and the differences of REST and GraphQL. I also feel I now know what is expected from a good API today, what the factors are that bring challenges and how they can be overcome. It has to be pointed that for myself to form a strong opinion about the subject, I would need more hands-on experience with APIs. I am however very excited to put everything learned into practice continuing to get better.

Though my thesis won´t necessarily be the most exciting to a API guru, I am confident it will help new developers to get the hang of things API and GraphQL. As my initial question for the thesis was: “is GraphQL truly the future of API designs”, I have still slightly torn feelings if I did succeed or not answering the question. As things often are not black and white, the same applies here, there is no absolute answer. It is not a case where no one had tried to revolutionize the API designs since REST become the standard. REST just has been more or less without equal match, until GraphQL. It says a lot that many other companies looking into making API interactions more efficient cancelled their efforts and hopped on to GraphQL once it was made public. Nothing changes over night and REST will still be used for a long time, that I´m sure of. That being said there is no questioning that GraphQL is revolutionary and with someone such as Facebook behind it will be interesting to see where all things API are heading. If you ask me, where ever that might be GraphQL is currently leading the way.

References

Berlind, D. 2015. What Are APIs and How Do They Work. URL: https://www.programmableweb.com/api-university/what-are-apis-and-how-do-they-work Accessed 2 September 2017.

Buna, S. 2016. Learning GraphQL and Relay. Birmingham, UK.

Buna, S. 2017. REST APIs are REST-in-Peace APIs. Long Live GraphQL. URL: https://medium.freecodecamp.org/rest-apis-are-rest-in-peace-apis-long-live-graphql-d412e559d8e4 Accessed 27 September 2017

Byron, L. 2015. GraphQL: A data query language. URL: https://code.facebook.com/posts/1691455094417024/graphql-a-data-query-language/ Accessed 18 November 2017.

Doglio, F. 2015. Pro REST API Development with Node.js. La Paz, Canelones, Uruguay.

Gazarov, P. 2016. What is an API? In English, please. URL: https://medium.freecodecamp.org/what-is-an-api-in-english-please-b880a3214a82 Accessed 2 September 2017.

Greif, S. 2017. So what's this GraphQL thing I keep hearing about? URL: https://medium.freecodecamp.org/so-whats-this-graphql-thing-i-keep-hearing-about-baf4d36c20cf. Accessed 30 September 2017.

Howtographql.com. Basics Tutorial – Introduction. URL: https://www.howtographql.com/basics/0-introduction/ Accessed: 31 January 2018

Lauret, A. 2017. ...And GraphQL for all? A few things to think about before blindly dumping REST for GraphQL. URL: https://apihandyman.io/and-graphql-for-all-a-few-things-to-think-about-before-blindly-dumping-rest-for-graphql/ Accessed 4 January 2018

Morgan, A. 2017. GraphQL vs REST: Things to Consider. URL: https://www.infoq.com/news/2017/07/graphql-vs-rest Accessed 26 November 2017

Mulesoft. 2015. What is an API. URL: https://www.mulesoft.com/resources/api/what-is-an-api Accessed 7 September 2017.

Smith, B. 2015. Beginning JSON.

Stubailo, S. 2017. GraphQL vs. REST. URL: https://dev-blog.apollodata.com/graphql-vs-rest-5d425123e34b Accessed 22 November 2017

Sturgeon, P. 2017. GraphQL vs REST: Overview. URL: https://philsturgeon.uk/api/2017/01/24/graphql-vs-rest-overview/ Accessed 20 November 2017

Uzayr, S. 2016. Learning WordPress REST API.

WebConcepts 2014. REST API concepts and examples URL: https://www.youtube.com/watch?v=7YcW25PHnAA. Accessed 26 August 2017.

Wikipedia. 2017. Application programming interface. URL: https://en.wikipedia.org/wiki/Application_programming_interface. Accessed 25 August 2017.

Wikipedia. 2017. Representational state transfer. URL: https://en.wikipedia.org/wiki/Representational_state_transfer. Accessed 8 September 2017.

Appendices

Appendix 1. Traditional report structure

Cover page, abstract, table of contents
Introduction – general introduction – objectives, (research) problem setting, delimitation – concepts.
Theoretical part – theoretical and previous practical and experiential information – establishing a research space among earlier studies, theories and models, with reference to professional literature and other sources.
Empirical part – target of research – objective, problems, development task – methodological choices[1] or project plan[2] with justification – description of implementation or working methods – data and types of analysis used[1] – results[1] or product[2] – summary.
Discussion – consideration of results – trustworthiness of the research[1] – ethical viewpoints – conclusions and suggestions for development or further work – an evaluation of the thesis process and one's own learning.
References
Appendices – questionnaire/interview forms and analysis results[1] – the product (if possible to include in the report)[2]

[1]A research oriented thesis, including quantitative or qualitative research.
[2] A product-oriented, practice-based thesis, involving a product development or planning task, event, publication, multimedia product or the like.

Appendix 2. Zipper thesis structure

Cover page, abstract, table of contents
Introduction – objectives – delimitation – presentation of commissioning company – process description.
Topic A to be studied and developed – previous research or experiential information (theoretical part) – a description of the phenomenon as part of the target studied – results/product and suggestions for development.
Topic B to be studied and developed – previous research or experiential information (theoretical part) – a description of the phenomenon as part of the target studied – results/product and suggestions for development. [Followed by C, D… if needed.]
Discussion – trustworthiness/usability – summary and conclusions – an evaluation of one's own learning.
References
Appendices