UNIVERSITY OF VAASA

SCHOOL OF TECHNOLOGY AND INNOVATION

AUTOMATION AND COMPUTER SCIENCE

Jeremias Snellman

IMPLEMENTATION AND EVALUATION OF A GRAPHQL-BASED WEB APPLICATION FOR PROJECT FOLLOW UP

Vaasa 27.08.2019

Supervisor Prof. Timo Mantere

Instructor Petri Välisuo & Tim Wallin, Gambit

CONTENTS

LIST OF ABBREVIATIONS 4

ABSTRACT 5

1 INTRODUCTION 6

1.1 Research question and purpose 6

1.2 Limitations 7

1.3 Structure 8

1.4 Gambit Labs Ab Oy 8

2 WEB API 10

2.1 Client-server architecture 10

2.2 Hypertext Transfer Protocol (HTTP) 10

2.2.1 The history of HTTP 11

2.2.2 Structure of messages 12

2.3 Data formats 14

2.3.1 JavaScript Object Notation (JSON) 14

2.3.2 Extensible Markup Language (XML) 15

3 GRAPHQL 16

3.1 The Type System 18

3.1.1 Schema 18

3.1.2 Types 19

3.2 Queries, Mutations and Subscriptions 21

3.3 GraphQL API Architecture and Design 24

3.3.1 Domain analysis 25

3.3.2 Architectural design 25

3.3.3 Prototyping 25

3.3.4 Implementing for production 26

3.3.5 Publishing 26

3.3.6 Maintenance 26

4 REPRESENTATIONAL STATE TRANSFER 27

4.1 The architectural style 27

4.2 RESTful API Design 28

4.2.1 Resources 29

4.2.2 Representation 30

4.2.3 Parameters 30

4.2.4 Methods 31

5 RELATED WORK 33

5.1 Web API Capacity Study 33

5.2 Performance analysis of Web Services 34

5.3 API Design in Distributed Systems 34

5.4 Semantics and Complexity of GraphQL 36

6 PLANNING AND IMPLEMENTATION 37

6.1 Toggl 37

6.1.1 Toggl API 38

6.1.2 Reports API 38

6.2 Requirements 40

6.3 Selection of technologies 42

6.3.1 Selection criteria 42

6.3.2 Server-side 42

6.3.3 Client-side 43

6.4 System design and architecture 44

6.4.1 Data model 45

6.4.2 Synchronization service 46

6.5 Creating the GraphQL API 48

6.6 Consuming the GraphQL API 49

7 EVALUATION AND RESULTS 52

8 CONCLUSIONS AND DISCUSSION 57

REFERENCES 59

APPENDIX 63

A. List of HTTP status codes and reason phrases 63

B. List of supported API requests for the Toggl API 64

LIST OF ABBREVIATIONS

AST	Abstract Syntax Tree
API	Application Programming Interface
CMS	Content Management System
HTML	Hypertext Markup Language
HTTP	Hypertext Transport Protocol
JSON	JavaScript Object Notation
REST	Representational State Transfer
SOAP	Simple Object Access Protocol
SPA	Single-Page-Application
TCP	Transmission Control Protocol
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
XML	Extensible Markup Language

UNIVERSITY OF VAASA
Faculty of technology

Author:	Jeremias Snellman
Topic of the Thesis:	Implementation and evaluation of a GraphQL-based Web application for project follow up
Supervisor:	Timo Mantere
Instructor:	Tim Wallin, Petri Välisuo
Degree:	Master of Science in Technology
Degree Programme:	Degree Programme in Energy- and Information Technology
Major:	Automation and Computer Science
Year of Entering the University:	2014
Year of Completing the Thesis:	2019 **Pages:** 66

ABSTRACT

This thesis is about APIs and web development. Technologies specifically presented include GraphQL and REST as a basis for the implementation of the web application. The purpose of the thesis was to create a web-based tool for project follow up. The main goal of the tool is to provide reporting functionalities and views for project follow up based on data from the public APIs provided by Toggl.

In the first part of the thesis, relevant theory about GraphQL, REST and APIs is provided. Web APIs are presented, along with common protocols, such as HTTP, as well as different data formats for the serialization of the data to be transmitted over the network. A literature review is also performed on the current state of the research on GraphQL-based APIs as well as on comparisons on GraphQL-, and RESTful-APIs.

The second part consists of the design and implementation of the application. Toggl, a time tracking service, is described with its two different APIs, the Toggl API, and the Report API. Further, the decision process of selecting the technologies for the developed tool is presented. One of the main parts of the tool consists of the synchronization mechanism for keeping the data in the database up to date with the data stored in Toggl. The other major part is about exposing the data via a GraphQL-API and consuming it in a single-page-application created in React.

The developed application is a minimum viable product, fulfilling its purpose of providing reporting functionalities for projects based on the data provided by Toggl. It was developed with usability, flexibility and testability in mind enabling further development in the future. GraphQL was the choice of API technology and was considered a suitable approach in this application.

KEYWORDS: GraphQL, REST, API, Web application, Toggl

1 INTRODUCTION

After the World Wide Web was invented and implemented, a need for standards for communicating between different computer systems rose in the 1990s. Different protocols were developed, such as Hypertext Transfer Protocol (HTTP) and Simple Object Access Protocol (SOAP). Later, a web architectural style called Representational State Transfer (REST) was developed by Roy Fielding. (2000). With the emerge of SOAP and REST, computer systems could communicate with each other via the Web using HTTP, utilizing Application Programming Interfaces (API).

Together with Web APIs, Asynchronous JavaScript and XML (Ajax) enabled the possibility to create dynamic sites which run in the browser. The term Single Page Application (SPA) was coined in the 2000s, which is used for a web application which runs completely in the browser. The needed resources, for example, JavaScript files or Hypertext Markup Language (HTML) documents are fetched from the server and form a standalone application in the browser without the need of loading a new page when the user clicks on a menu, for example. A large number of frameworks have been developed which builds on this principle. React, developed by Facebook, Angular, which Google is the inventor of, and Vue, founded by Evan You, to mention a few.

As the complexity of data has increased, the need for new ways for fetching data from the server has emerged. As a response to this, Facebook developed a query language and runtime called GraphQL, enabling a flexible and extensible way for the client to fetch data and communicate with the server.

1.1 Research question and purpose

This thesis is about evaluating GraphQL from a practical approach, in the implementation of a web application. Two different approaches for creating APIs, namely GraphQL and REST, will be presented and evaluated. A literature review will also be performed on the current state of research on GraphQL and REST.

A need for a tool was discovered in the company Gambit Labs Ab Oy (see chapter 1.4), to ease the follow-up of projects. A time tracking service called Toggl is in use by the company. Toggl offers integrations to different project management tools, along with the possibility to create custom integrations, using its public API. Toggl is used for tracking the time developers, designers and project manager spend on different clients and projects, thus making it easier to invoice the customers as the total time spent and other queries are easy to access from the Toggl reports.

The purpose of the thesis is to utilize Toggl's public API in creating a tool for fetching and presenting the data, thus creating views for the follow-up on estimates, time used, and invoices on specific tasks within a project. On the short term, this tool primarily serves in a retrospective way, whereas in the long term, it enables to serve as a foundation for the salesforce on giving more precise estimates and offers for new potential projects.

1.2 Limitations

The thesis is limited to presenting only two different API technologies, namely GraphQL and RESTful APIs, as these are very popular today. Terms such as Single Page Application (SPA) will be explained, together with an exploration of different frontend and backend frameworks. These will, however, not be evaluated nor compared to each other, rather they only serve as a theoretical foundation for the later chapters in which the development process is explained in more detail.

The tool created is not a complete project management tool, with all the different features for managing a project, but a rather focused tool and a minimum viable product (MVP), which aims to fulfill its purpose mentioned in the previous section. This will help to keep the complexity down but still, enable further development in the future. Further utilization of the Toggl data is effortlessly achievable, due the the flexibility and customizability of GraphQL, which is used, thus enabling rapid creation of new user interfaces.

1.3 Structure

After the introduction chapter, Web APIs in general are covered. HTTP, a commonly used protocol, is explained along with different data formats for sending data between the server and the client. In the following chapter GraphQL is explained in detail. Terms such as schema, types, queries and mutations are covered, along with different use cases for GraphQL and scenarios, where a different choice of technology is better suited.

In chapter four, alternatives to GraphQL, REST-based APIs are covered. The architectural style will be presented along with its constraints. Finally, the term RESTful web service, the implementation of REST, will be explained.

Chapter five provides a literature review of some of the research about GraphQL which at the time of writing have been done. The research papers presented in this thesis cover essentially comparisons between GraphQL and REST, as well as evaluations of the GraphQL language itself.

In the initial part of chapter six, Toggl, the time tracking service in use in Gambit, is explained. The usage as well as the public API with its different report types are covered. Furthermore, the requirements of the tool is presented along with a section covering the selection of technologies. Different UML diagrams are created in support of the tool, covering the functionality and the structure of the application. Among these are class and sequence diagrams.

Finally, in chapter seven, the results of the thesis are explained, covering the outcome of the implemented tool and an evaluation of the application and GraphQL, from a practical approach. In chapter eight the project is concluded and discussed.

1.4 Gambit Labs Ab Oy

Gambit Labs Ab Oy, hereafter called Gambit, is a company founded in 2011. It is one of the first companies in Finland with the intention of implementing smart mobile applications to customers. Gambit employs a total of 21 employees and is located in Runsor, in the Finnish city of Vaasa. (Gambit 2019.)

Gambit has gained a broad portfolio of diverse projects and customers. These vary from creating websites for small companies, to more complex solutions including mobile apps and web services customized for solving a specific problem. (Gambit 2019.)

2 WEB API

An API is an interface to a computer program, which exposes some data and functionality to allow different systems to communicate with each other, and thus exchange information. This can be done via a web service, which is a web server, whose purpose is to support an application's or a site's needs. A Web API is the utilization of the two, exposing a web server's functionality allowing client programs to interact with the server. (Masse 2011:5.)

2.1 Client-server architecture

Web APIs typically operate in a client-server architecture. This model can be described as two separate, autonomous processes, which operates in a request-response way. That is, the client requests a resource from the server, which then processes the request and sends the resource in a response back to the client. *(*Singh & Yadav 2009:1). This is illustrated in Figure 1.

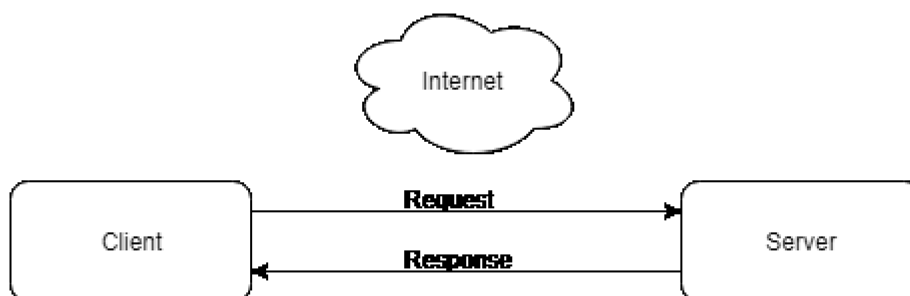

Figure 1. A general view of a client-server architecture.

A common example of a client-server model with network communication, is the browser acting as a client, which requests web sites or other resources from a web server. (Singh & Yadav 2009:157.)

2.2 Hypertext Transfer Protocol (HTTP)

The communication between a client and the server is often handled via HTTP. In general, a client requests a resource from the server, using a GET or POST method or another HTTP method, in which the server processes the request, and creates a response describing the performed actions, hence returning the required resource to the client. The resource can be of various formats, for example, images, audio and text, which could be formatted in JSON or XML (described in chapter 2.3).

HTTP is a protocol at the application-level for distributed, collaborative, hypertext information systems, stateless in its nature. It relies on reliable data-transmission protocols such as Transmission Control Protocol (TCP), which ensures that the data received will be the same as the data sent from the server. (Totty, Gourley, Sayer, Aggarwal & S. Reddy 2009). HTTP, built upon the previously mentioned client-server paradigm is often used as the underlying protocol when designing and implementing web APIs.

2.2.1 The history of HTTP

The first prototype version of HTTP, version 0.9, was proposed by Tim Berners-Lee in 1991. As a protocol, it was rather limited. The only supported request type was a GET request, which will be explained further later in this chapter. (Totty et al. 2009).

HTTP/0.9 was shortly replaced by version 1.0. This protocol was the first widely adopted version, with support of more request methods such as GET, POST and DELETE. In addition to the methods, HTTP headers and multimedia object handling were introduced, allowing for creating content with a graphically appealing user interface. (Totty et al. 2009).

With the increased popularity, many features were added to HTTP by commonly used web clients and servers. Even though they were unofficial, the features became the facto standards, resulting in HTTP/1.0+. Its descendant, version 1.1 is the standard version of HTTP. This version is since its publication used by most clients and is still very popular today. In this version significant performance optimizations were made along with corrections for many architectural flaws. (Totty et al. 2009.)

The latest version of HTTP is version 2.0. Up until 2.0, the data are transmitted in plain ASCII text. For increased performance, HTTP/2.0 transfers the data in binary form. HTTP/2.0 also includes several other improvements and optimizations. The usage of HTTP/2.0 is increasing, with a percentage usage of 33.3% of all websites, at the time of writing. (W3Techs 2019.)

2.2.2 Structure of messages

An HTTP message consists of three parts; the start line, the headers and the entity body. The start line and the headers, written in ASCII text, are separated by a two-character end-of-line sequence. The entity body is an optional block in the message. It may contain data in text or binary format, which is specified in the header block. (Totty et al. 2009). The message is either a request or a response, both having the same structure of the message.

The start line in a message defines the type of method, the Uniform Resource Identifier (URI) of the request and the HTTP version to use. The available methods, according to the current HTTP specification, version 1.1, are shown in table 1.

Table 1. Table representation of all methods specified in HTTP/1.1, as provided by Fielding and Reschke. (2014:22.)

Method	Description
GET	Transfer a current representation of the target resource.
HEAD	Same as GET, but only transfer the status line and header section.
POST	Perform resource-specific processing on the request payload.
PUT	Replace all current representations of the target resource with the request payload.
DELETE	Remove all current representations of the target resource.
CONNECT	Establish a tunnel to the server identified by the target resource.
OPTIONS	Describe the communication options for the target resource.
TRACE	Perform a message loop-back test along the path to the target resource.

In RFC 7231, in the Internet Engineering Task Force (IETF), Fielding and Reschke state that the only required methods, which general-purpose servers must support, are the GET and HEAD method, while the others are optional. (Fielding & Reschke 2014:22). As

HTTP was designed to be easily extensible, other methods may be implemented by servers. These methods are called “extension methods”. (Totty et al. 2009.)

The start line in the response differs from the request. In the response, the star line contains the HTTP version, a status-code and a reason-phrase. The status-code informs the client about the occurred reactions during processing the request. It consists of a three-digit sequence, with the first digit expressing the class of the response, for example “success”, “error” et cetera. (Totty et al. 2009f). The following list describes the various classes of status codes, and their description, as presented by Fielding and Reschke (2014:47):

- 1xx (Informational): The request was received, continuing process.
- 2xx (Successful): The request was successfully received, understood and accepted.
- 3xx (Redirection): Further action needs to be taken in order to complete the request.
- 4xx (Client Error): The request contains bad syntax or cannot be fulfilled.
- 5xx (Server Error): The server failed to fulfill an apparently valid request.

In appendix A, a list of common status codes and corresponding reason phrases is presented. The reason phrases are only for human interpretation of the response and can be changed without affecting the protocol.

The header section contains additional information regarding the request and response. HTTP headers can be classified in the following categories; General-, Request-, Response-, Entity-, and Extension-headers. The headers consist of a list of key-value pairs, that is, a name of the header and its corresponding value, separated by a colon. An example of a common header is “Content-length” which defines the length of the body in bytes.

Another example is “Content-type”, which specifies the type of the body. For example, “Content-type: image/jpeg” defines that the body is a jpeg image. (Totty et al. 2009.)

Finally, the entity body of the HTTP message is the payload. Entity bodies carry the actual data, which was requested, for example HTML documents, images, videos et cetera. This section is optional, thus providing it both in the request and the response is not needed. (Totty et al. 2009.)

2.3 Data formats

As different systems communicate with each other, in a web-context often via HTTP, the systems need to understand the transmitted data. A client application, for example a JavaScript application, must be able to understand the entity body of an HTTP request sent to a server application, for example written in Python. For this case, different data formats have been established. The purpose of the data formats is to retain common standards for serializing data structures or other data in an application.

2.3.1 JavaScript Object Notation (JSON)

JSON is a lightweight, text-based format which is designed to be human readable, for exchanging information typically in a client-server architecture. JSON originated from JavaScript and uses JavaScript syntax. However, JSON is not dependent on JavaScript or any other language. Many languages support the data format, among these are Java, PHP, C# and Python. (Sriparasa 2013.)

Essentially, JSON is a format for serializing data. The format supports four primitive types: numbers, booleans, strings and null. It is also able to represent two structured types: objects and arrays. Strings in JSON are defined to be any sequence of any length, including zero, of Unicode characters. Objects are a collection of key-value pairs, where the key is a string representing the name of the value, and the value itself may be any of the

primitive or structured types as defined in JSON. An array in JSON is an ordered list of values of any length. (Bray 2017.)

2.3.2 Extensible Markup Language (XML)

XML is a subset of the Standard Generalized Markup Language (SGML) and was formed in 1996 by a working group under the auspices of the World Wide Web Consortium (W3C). The design purpose for XML is, among others, that it should be straightforwardly usable over the Internet, it should be easy to create programs which handle and parse XML documents and that the XML documents should be clear and human-readable. (Bray, Paoli, Sperberg-McQueen, Maler & Yergeau. 2008.)

An XML document is built upon storage units, called *entities*. These entities contain data and are usually identified by an entity name. The entities contain parsed or unparsed data. Parsed data consists of characters which form data or markup. Character data contains the representation of the data while the markup encodes a description of the layout and logical structure of the document. (Bray et al. 2008.)

3 GRAPHQL

GraphQL is a query language and a runtime. A client and a server which both understand the query language, may request and respectively respond to the data requirements, which is declaratively communicated by the client. The runtime lays on top of the server application and is an execution layer, which ensures that the server is able to respond to the request, made in the GraphQL language. In his book "Learning GraphQL and Relay", Samer Buna describes the relationship between the GraphQL language and runtime as the following quotation:

> Anyone can invent a new language and start speaking it, but no one would understand them without learning the new language first or having someone translate it for them. That's why we need to implement a runtime for GraphQL on the backend servers. (Buna 2016:7.)

Furthermore, the runtime layer is a graph-based schema, which is independent of any language. The schema defines the capabilities of the corresponding data service. Based on the schema, a client can perform any query against the runtime, as long as it lies within the constraints of the schema. Using this approach, both the client and the server are decoupled from each other, enabling each to scale as well as evolve independently. (Buna 2016:6 – 7.)

When Facebook developed GraphQL, a set of design principles where followed: (Facebook 2018a.)

- Hierarchical: In order for clients to describe the data requirements, a query is hierarchically structured, in the same way as the data returned in the response. This achieves congruency in the creation and manipulation of view hierarchies.

- Product-centric: GraphQL has a client-side first approach. Instead of starting from the backend API designer's perspective, everything starts from the front-end engineer's way of thinking and requirements.

- Strong-typing: As every GraphQL server has its own type system, Query validation can be performed already at development time, using certain tools. With strong-typing, the structure of and the nature of a response can be guaranteed at a certain level before execution.

- Client-specified queries: Given the type system, the capabilities of a GraphQL server is published to the client. The responsibility then lays in the client to define what data should be returned on a detailed level. This is the opposite to most client-server applications not using GraphQL, as the data returned is specified in the server.

- Introspective: As defined in the specification, a GraphQL server's type system can be queried in itself. A client can query the server for which kind of queries can be performed, for example. This feature allows for the creation of common tools and client libraries, such as GraphiQL.

At its core, a GraphQL service is about defining types which contain fields, and then provide functions for the fields. In figure 2, an example is shown, made by GraphQL (2019a). In this example a service providing the current user in a system is declared. Figure 3 shows a corresponding implementation of the functions used for retrieving the data.

```
type Query {
  me: User
}

type User {
  id: ID
  name: String
}
```

Figure 2. Example of two GraphQL types defined. (GraphQL 2019a.)

```
function Query_me(request) {
  return request.auth.user;
}

function User_name(user) {
  return user.getName();
}
```

Figure 3. Example functions for processing a GraphQL query. (GraphQL 2019a.)

Furthermore, when proceeding with the example above, figure 4 shows how the query can be stated in this specific GraphQL server. The corresponding result of the query is shown in figure 5.

```
{
  me {
    name
  }
}
```

Figure 4. GraphQL query example. (GraphQL 2019a.)

```
{
  "me": {
    "name": "Luke Skywalker"
  }
}
```

Figure 5. Example response of a GraphQL query. (GraphQL 2019a.)

3.1 The Type System

The GraphQL Type system defines the capabilities of the server. It is used to determine whether a query is valid and whether the provided values in a query contains valid data. The standard language of describing the type system, is the GraphQL Schema Definition Language (SDL), a kind of Interface Description Language (IDL), which is included in the GraphQL specifications. (Facebook 2018b.)

3.1.1 Schema

In order to know what kind of operations are supported in a GraphQL server, the schema must contain information about the Root Operation Types, which can be a query, mutation or subscription. Among these, only queries are mandatory, with mutations and subscriptions being optional. (Facebook 2018b.)

3.1.2 Types

A fundamental unit in the schema is the type. There is a total of six different types: object-, scalar-, interface-, union-, enum- and input-object-types. Besides these types, there exist wrapping types, which can be either a List or a Non-nullable type and the root operation types, described in the previous section.

The object type is the most basic component of the schema. This represents the structure of the data of a certain object. An object type contains fields which can be of any of the other types. An example of an object type is as follows:

```
type Car {
  model: String!
  year: Int
  color: String
}
```

In this example, the object is a *Car*, which holds three fields, model, year and color. The fields are all scalar types, which will be explained later. According to the GraphQL SDL the exclamation mark (!) indicates that the model field is a Non-Null type. When a field which is marked as Non-Null is queried, the result can not contain a null value. (Facebook 2018c.)

The scalar types are a defined set of raw data values. The default types available in the specification are (GraphQL 2019b):

- 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: The ID scalar type represents a unique identifier, often used to re-fetch an object or as the key for a cache. The ID type is serialized in the same way as a String; however, defining it as an ID signifies that it is not intended to be human-readable.

Interfaces are similar to interfaces in object-oriented programming in the way that they define a set of fields, which an object type must include in order to implement the interface. Extending the *Car* example above we could have an interface named *Vehicle*, which the *Car* object implements:

```
interface Vehicle {
  id: ID!
  model: String!
  year: Int
  color: String
}

type Car implements Vehicle {
  id: ID!
  model: String!
  year: Int
  color: String
}

type Truck implements Vehicle {
  id: ID!
  model: String!
  year: Int
  color: String
}
```

In this example, there are two different kinds of objects, *Car* and *Truck*, both implementing the *Vehicle* interface. In addition to this, each object types may add object specific fields which do not belong to the interface.

Another type in the specification is union types. These specify that a type may be either one of the provided object types. In GraphQL's web site an example is provided defined as follows (GraphQL 2019b):

```
union SearchResult = Human | Droid | Starship
```

This example implies that when a query has a return type of *SearchResult*, the returned object might be either an object of type *Human*, *Droid*, or *Starship*.

The enum type defines a set of predefined possible values. The values in the enum type is a scalar value. This allows for the system to know, that the input or output of a query is either one of the values defined in the enum. (GraphQL 2019b). For example:

```
enum Color {
  RED
  GREEN
  BLUE
  BLACK
}
```

The last type of the named types is the input object type. These are a special type which is only used as a type when being passed as a query argument, but more commonly in mutations (discussed in section 3.2). The input type is similar to a regular object type with the difference in the keyword being used in the schema, *input*, instead of *type*. (GraphQL 2019b.)

3.2 Queries, Mutations and Subscriptions

Queries and mutations in GraphQL are means of interacting with the data. A query is a read operation, meaning that a client performs a query, which is sent to the GraphQL server. The server, in turn, processes the request and returns the requested data. This is being done without performing any so-called side-effects on the data, for instance updating, or deleting any data.

When the user wants to perform changes to data, for example create a new object, update an instance or delete some data, a mutation operation is performed.

Queries and mutations in GraphQL belong to the root operation types along with subscriptions. Similarly, as with the other types, queries and mutations are to be defined in the schema.

In order to define a query in the schema, usually a type named *Query* is defined, with each field representing a specific query, which can be performed. The name of the type is not required to be *Query*, but according to the specifications, the schema definition is not needed. Instead it can be omitted when the root types *query, mutation* and *subscription* are named *Query*, *Mutation* and *Subscription* respectively. An example of this is:

```
type Query {
  readCars: [Cars]!
}
```

This above example specifies that there must exist a query called *readCars,* which must return a list of car objects. The exclamation mark outside the brackets, allows the car objects themselves, but not the list, to be null. Similarly, to define a mutation, a type *Mutation* is added in the schema:

```
type Mutation {
  setCarColor(id: ID!, color: String!): Car
}
```

In this example, a mutation *setCarColor* is defined. This mutation takes two mandatory arguments, *id* of type *ID* and *color* of type *String*. The response of this mutation is an object of type *Car*.

When the queries and mutations are defined in the server schema similarly to the examples above, the client can utilize them by performing a request to the GraphQL server. If a client wants to get data about all cars stored in a database, it can use the *readCars* query in the following way:

```
query {
  readCars {
    id
    model
    year
  }
}
```

Updating the color of a car is performed by using the example mutation like this:

```
mutation {
  setCarColor(id = 1234, color = "green") {
    id
    color
    year
    model
  }
}
```

Subscriptions are a means of accessing near real-time updates to the client, when data or another event occurs in the server. Typically, this has been done using polling or web sockets. GraphQL subscriptions are defined in the same way as queries and mutations, for example:

```
type Subscription {
  carAdded: Car
}
```

This subscription definition implies that a client can subscribe to an event called *carAdded,* which will inform the client whenever a new car is saved. The events or notifications are often triggered when a change has occurred to the graph, via a mutation or another type of change, for instance, when new measurements from a sensor are received. (Biehl 2018:36). When the client wants to use the above subscription, a request to the GraphQL server is sent with the following payload:

```
subscription {
  carAdded {
    id
    model
    year
```

```
    }
}
```

After the subscription request is sent to the server, the client will receive a notification with the new car data every time a new car is added.

3.3 GraphQL API Architecture and Design

The architecture of a GraphQL server is often designed as a 3-tier architecture. These are the database layer, the business logic layer and the front facing layer. The GraphQL API lies in the front facing layer, along with possible other APIs, such as a RESTful API. This allows for the different APIs to reuse most of the application, only implementing API specific details in this layer. (Biehl 2018:43.)

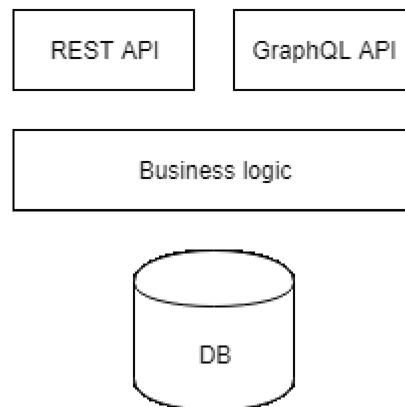

Figure 6. Simplified 3-tier architecture, as shown by Biehl. (2018:43).

When designing an API, Biehl states that the fundamental idea is to treat the API as a separate product, following that the API needs to be consumer-oriented (2018:45). Consumers of an API are all the developers, who build different clients utilizing the API. In its essence, being consumer-oriented means to prioritize the needs and desires of the consumer first when designing the API. Practically this means that it should be as approachable, simple, clear and clean from the consumer perspective, as possible. This implies that the designers need to know the type of solutions which are built using the API.

Reusability is an aspect to maintain when designing a consumer-oriented API. One of the traps to fall into when using a consumer-oriented design, is to design it too application specific, meaning that it is designed with a specific use case or solution in mind. The API needs to compromise between being too generic and narrow, and to maximize the reusability and the number of potential users, though being consumer-oriented at the same time. (Biehl 2018:45.)

The design phases presented by Biehl are following an agile approach, consisting of an iterative process with small incremental steps. These are the domain analysis, the architectural design, prototyping, implementing for production, publishing and maintenance. Each phase consists of a creative and a verification part. In the creative part, an artifact is created. The verification part provides early feedback in each phase, enabling relatively simple modification at low risk and cost. As the method is agile, not all requirements and information need to be available in the beginning, rather it is evolving as the project continues. (Biehl 2018:46.)

3.3.1 Domain analysis

In the domain analysis phase, the structure of the data is gathered and planned, to enable thinking from a consumer perspective. Questions to be asked are, who are the consumers of the API? What kind of solutions will be built with this API? How does the consumer want to interact with the data from the API? (Biehl 2018:47.)

3.3.2 Architectural design

The server architecture and an API philosophy are chosen in the architectural design phase. This phase involves evaluating different alternatives and choosing the best suitable architecture for the application to be implemented. As previously mentioned, the 3-tier architecture is commonly chosen as the architectural design in server environments. (Biehl 2018:47.)

3.3.3 Prototyping

Prototyping in the means of designing a GraphQL API, involves creating the actual schema and setting up mock data, to be used by the GraphQL server. This enables the consumers of the API to test and give feedback on the domain of the API. The schema and the domain can be updated after an iteration, often by extending the current schema to keep backward-compatibility. (Biehl 2018:47.)

3.3.4 Implementing for production

When implementing for production use, the schema is complete. This step involves proceeding towards using real data against the real application instead of using mock data. Other factors which come into focus in this phase are non-functional properties, such as performance, security and stability. (Biehl 2018:47.)

3.3.5 Publishing

After publishing the GraphQL API, further changes and improvements of the API need to be backward compatible with the original API. Hence, it is important that the API is tested properly and that enough feedback has been received from the consumers. (Biehl 2018:48.)

3.3.6 Maintenance

In the final phase, the API is maintained. Bug fixes included, but new features may be added as well. Backward compatibility is important and kept by permitting only additions of fields, queries or mutations, while obstructing the deleting of any functionality or field. In this phase, analytics may be added to gain insight into how and why consumers use the API, which assumes a direct communication with a consumer and an active community. (Biehl 2018:48.)

4 REPRESENTATIONAL STATE TRANSFER

In the previous chapter, GraphQL, the language and runtime were explained. Concepts such as the types system, queries and mutations were mentioned. Later API design with GraphQL was covered. This chapter provides information about a common architectural style for designing APIs in today's world of web development. REST and REST-based APIs, often called RESTful APIs, are terms which will be explained.

4.1 The architectural style

In the year of 2000, the term Representational State Transfer (REST) was composed by Roy Fielding in his doctoral dissertation. REST is an architectural style for distributed hypermedia systems and consists of a set of constraints which, as Fielding states: "emphasizes scalability of component interactions, generality of interfaces, independent deployment of components, and intermediary components to reduce interaction latency, enforce security, and encapsulate legacy systems" (Fielding 2000). The REST architectural style consists of six constraints defined as follows:

- Client-server: This constraint addresses the separation of concerns, separating the user interface from the data storage. This enables that both the client and the server may evolve independently and improves scalability, as the simplicity of the server may be kept down. (Masse 2016:3.)

- Uniform interface: The uniform interface constraint keeps the overall architecture simple and improves the visibility of interactions between the components. Another benefit is that it decouples the implementation of the service. A drawback is that decreased efficiency often follows, since the information is standardized in the transportation between components, rather than transferring information in an application specific format. (Fielding 2000.)

- Layered system: The layered system constraint ensures that each component can only see the closest layer for which the interaction happens. This enables intermediate network-based components, such as proxies or gateways to be deployed between a client and a server. Benefits from this constraint are that layers can be used to encapsulate legacy systems and improve scalability, by enabling load balancers, among others. (Fielding 2000.)

- Cache: The caching constraint is important in web architecture. This requires the server to declare whether a response may be cached or not, by the client. Caching content reduces latency as the payload over the network may be significantly reduced. Caching may decrease the reliability of data, in the case the data has been declared cacheable by the server and the data is updated on the server, while the client reuses the old data. (Fielding 2000.)

- Stateless: Adhering to the stateless constraint, the server does not keep track of application state, resulting in improved scalability, reliability and visibility. The stateless constraint indicates that each request must contain enough information for the server to understand and to be processed as a whole, without relying on any data left in a shared context on the server side. One of the drawbacks is increased network load, since the same data might be sent repetitively between the client and the server. (Fielding 2000.)

- Code-on-demand: The code-on-demand constraint is the only optional constraint in the REST architectural style. It allows for functionality on the client side to be downloaded and executed, in the form of scripts or applets. Benefits of the code-on-demand constraint are extensibility among others, by allowing functionality to be downloaded after deployment. According to Fielding, the reason for being an optional constraint is that it reduces visibility and tends to establish technology coupling. (Fielding 2000.)

4.2 RESTful API Design

APIs which fulfill the architectural constraints in REST are called RESTful APIs. This section describes some of the aspects regarding designing an API using the REST constraints.

4.2.1 Resources

A resource is a structure uniquely identifiable by a URI, which contains raw data of the business object. It is a central part of the API, thus making it difficult to change after the API has been published. Hence, it is important to elaborate on the correct model of the resources from the beginning. The design process of the resource model is similar to the designing of classes in object-oriented programming. However, a difference between these two is that the resource model in REST APIs is constrained by the HTTP methods, whereas in object-oriented programming, the methods may be application specific. Some aspects to consider in designing the resource model, is the scope of the resource, the attributes, relations to other resources et cetera. (Biehl 2016:91.)

Biehl mentions three different categories of resources; instance resources, collection resources and controller resources. (2016:103). The instance resource represents a single instance of a business object. For example, the resource at the URL *https://my-cars.com/cars/123* represents a resource of type *car,* which returns a JSON response containing the following data:

```
{
  “id”: 123,
  “model”: “audi”,
  “year”: “2019”,
  “color”: “red”
}
```

Collection resources represent a collection or a list of instance resources. Typically, the consumer of the API is interested not only in a single instance, but rather all, or a set of the existing instances of a resource. In the previous example, a *car* was fetched. To fetch all cars, the following URL could be used, *https://my-cars.com/cars*. This request would return all the available cars stored on the server side.

The last category of resources is the controller resource. Sometimes, in the case of a long-running process for example, the desired functionality cannot be described by a standard resource and an HTTP method. A controller resource can then be used, by sending a POST request to the URL, which triggers the process. The server then returns a response with the status 202 *Accepted*, which indicates that the process is submitted. The response might also include a *Location* header which provides a URI to the process. By performing a GET request to the URI, the current status of the process is returned to the client. (Biehl 2016:106.)

4.2.2 Representation

Representation of a resource is the point of serializing a resource in a transferrable way. As the resource in itself is merely raw data, the data needs to be serialized before transferring it between a server and a client. Furthermore, the client deserializes the representation back into a resource for additional processing.

There are various types of serialization rules. For example, in HTTP, the MIME-types can be used as serialization rules. JSON and XML are typically used in the context of web APIs. The representation of the resource can be found in the body of the request and response. (Biehl 2016:121.)

4.2.3 Parameters

An API is typically dynamic, meaning that the response is dependent on the input from the client, hence returning a specific response with regards to the input. In a RESTful API the input parameters are sent as HTTP parameters or in the HTTP body. The parameters can, depending on the use cases, be categorized in four different types. These are creation and update, filtering and sorting, locators and the final type, projections. For creating a new resource, the initial values of the resource need to be sent to the server. These are often sent as a POST request with the input sent as form parameters. (Biehl 2016:133.)

When the consumer fetches a list of resources, a collection resource, typically only a subset is relevant for the consumer. Transferring the complete set of resources from the server to the client for further processing, such as filtering and sorting is not efficient in terms of network bandwidth usage. Hence, the API needs to provide this functionality to the consumers. Filter and sorting criteria are often specified as query parameters, for example *https://my-cars.com/cars?model=audi&sort=year* would return a collection of Audis which is sorted by the year. (Biehl 2016:134.)

Projection is a means of providing a reduced set of fields or properties of a resource to the consumer. A consumer might be interested only in a small set of properties; therefore, the API should provide the functionality for reduced processing power and bandwidth. Similar to the filter and sorting criteria, the projection, or the selection of fields, can be provided as query parameters. Using the same example with cars, retrieving only the model property of the collection of cars is done with the following GET request: *https:/my-cars.com/cars?fields=model*. (Biehl 2016:136.)

4.2.4 Methods

Using the REST architectural style correctly, means that the HTTP protocol needs to be used correctly as well. Using the principle of URIs, a basic set of operations is defined, which can be performed on a resource. Encountering a new resource, the consumer will be familiar with the manners of interacting with it. This does not imply that all operations must be supported for each resource. The API provider determines which of the HTTP methods will be supported for each resource, often limited to a subset of all the HTTP methods. (Biehl 2016:142.)

The HTTP GET method is used for the retrieval of a resource. To fetch a list of resources, a GET request is performed to the URL of the corresponding collection resource. In the same way, retrieving an instance resource, a GET request is sent to the URI of the resource. When the resource exists in the server and the request is successful, a response with the status code *200 Ok* is sent back as a response, containing the resource in the

body. If the resource does not exist, a response with the status code *404 Not Found* is sent back. (Biehl 2016.143.)

Creating a new resource is often done by sending a POST request. The request is sent to the URL of a collection resource, of which the new resource is to be created. The initial values are sent in the body of the request as form parameters. A successful request is responded by a *201 Created* response, and a *Location* header, which contains a URI to the newly created resource. Performing a POST request to a URI of an instance resource will return a response with a status code *405 Method Not Allowed*. (Biehl 2016:144.)

To update a resource the PUT or PATCH method is used. The PUT method requires the complete representation of the resource and will update all fields of the resource. The representation is sent in the body of the request and the request itself is sent to the URI of the instance resource. A successful update of the resource results in a *200 Ok* response. Partially updating a resource is done using the PATCH method. It is sent to the URI of the instance resource, only containing the properties to be updated in the request body. (Biehl 2016:145.)

When a consumer needs to delete a resource, a DELETE request is sent to the server to the URI of the instance resource. A response status code *200 Ok* is returned if the resource exists or has ever existed. The consecutive responses fulfill the property of idempotent methods, meaning that the method can be repeated without altering the end result. Only in the case that the resource has never existed, the status code *404 Not Found* is returned. (Biehl 2016:146 – 147.)

The methods mentioned above are the most common HTTP methods, used for performing CRUD (Create, Read, Update, Delete) operations on a resource. Several other HTTP methods exist, for example the HEAD method, which is used to check the existence of a resource. This method returns only a status code determining if the resource exists, without sending the actual representation in the body. Another method which is used is the OPTIONS method. This returns a header called *Allow* with a set of allowed HTTP methods, which can be performed on the resource. (Biehl 2016:147.)

5 RELATED WORK

Both GraphQL and REST have been explained in the previous chapters. This chapter will provide a literature review of some of the existing research papers regarding evaluating GraphQL in itself, as well as comparing GraphQL and REST as a basis for designing APIs. Since GraphQL is a query language and REST is a set of constraints for an architectural style, they are not trivially comparable.

Because GraphQL was initially released to the public in 2015, the amount of research regarding GraphQL-based APIs, is still quite low. However, the research is increasing and several papers on both comparisons and evaluations of GraphQL have been made.

5.1 Web API Capacity Study

The thesis "Web API Capacity Study – A Comparison of REST and GraphQL" was made in 2018 by Tjip Pasma from the Aarhus University. A comparison between REST and GraphQL was done, in the context of managing a large amount of time series data, which is to be served to clients in real time. The data came from a wind power farm, which can contain hundreds of turbines in one farm, each of the turbine producing thousands of entries.

Three different APIs were implemented and evaluated; a RESTful API, a GraphQL RPC API and a GraphQL API using subscriptions. Three hypotheses were made which focused on comparing the capacity of different GraphQL implementations against a RESTful API. One hypothesis focused on the usability of GraphQL, and was formulated as "Will a GraphQL API have same or better usability metrics as REST based API?" (Pasma 2018:8 – 9.)

The results showed that with both GraphQL APIs, the capacity was increased with more than 100% compared to the REST-based API. The usability test also revealed that the

usability, based on a set of defined criteria, was improved by using the GraphQL based APIs. (Pasma 2018:52).

5.2 Performance analysis of Web Services

Another study which compares RESTful and GraphQL web services, is done by Arnar Freyr Helgason, from the University of Skövde, in 2017. This thesis investigates whether GraphQL can increase the performance of the dataflow, as well as whether network performance will increase by using GraphQL, with its single endpoint, compared to multiple endpoints for RESTful APIs.

The test environment used in the thesis, consisted of a Node.js server environment with a MySQL server, which stored the data to be fetched. In the client-side, the library jQuery was used for performing the requests. For measuring the latency, the total time from that a request was sent, until a response was received and parsed, was used. The measured time includes multiple steps, such as data transmission and the server processing the request. A packet sniffer was used for measuring the size of packets and the latency as well. (Helgason 2017:20.)

The results showed that for a simple structure of the data, the REST API outperformed GraphQL in terms of response time, whereas in a more realistic scenario, with a more complex table structure, using GraphQL improved the response time with approximately 25%. (Helgason 2017:37.)

5.3 API Design in Distributed Systems

The study by Thomas Eizinger, API Design in Distributed Systems: A Comparison between GraphQL and REST, (2017), from the University of Applied Sciences Technikum Wien, has a more theoretical approach in comparing GraphQL and REST. The purpose is to compare GraphQL against the architectural style REST, to find the key differences.

This should assist in making an educated decision of choosing the best approach for a specific task. The main questions which the thesis tries to answer are, how they can be compared, since GraphQL is a specification and REST is an architectural style, and what the main differences are, among others. (Eizinger 2017.)

The author starts by investigating different comparison approaches. The task of comparing the two is not trivial, as GraphQL and REST operate on a different level of abstraction. Therefore, three different comparison strategies are presented. The first strategy is to specialize REST, which means to choose concrete technologies for building a RESTful system, for example HTTP. This comparison can be done by defining a set of criteria, two use cases, and using both technologies in implementing the use cases. Another approach for comparing GraphQL and REST, is to generalize GraphQL as an architectural style. This means, identifying the principles in an architectural context behind GraphQL. Three principles were identified, client-server, stateless, and declarative languages. The last approach is to identify the desired properties of a system offering an API, and then comparing how GraphQL and REST influence the properties and manages occurring problems. This approach compares GraphQL and REST as is, without the need for specialization or generalization. Evaluation of the three strategies showed that the third alternative was the best for the scope of the thesis. (Eizinger 2017:15 – 18.)

A set of comparison criteria were defined. Among these were operation reusability, discoverability, component reusability, simplicity, performance and over-fetching. These criteria were evaluated for both GraphQL and REST. In the conclusions, the author states that some of the biggest implications for using GraphQL are that a number of responsibilities shift from the server to the client. For example, the lack of metadata in the responses, implies that the client needs to know on its own, which steps of operations are valid. Another example is the handling of cache invalidation, since it is a convoluted task as the size and the complexity of the application grows. Finally, the author concluded that a decision model for choosing the right approach is not provided, since the right approach always is dependent on the requirements of the system. Instead, an overview of many relevant aspects within API design is given. (Eizinger 2017:52.)

5.4 Semantics and Complexity of GraphQL

The final research of this literature review is conducted by Olaf Hartig and Jorge Pérez in 2018. It is a technical evaluation of the semantics and the complexity of GraphQL. According to Hartig and Pérez, there is a need for a more fundamental understanding of the properties of the language, since several implementations already exist for the language, whereas a proper understanding of the properties does not exist at the time of writing. By investigating the properties, the authors want to clarify the limitations of the language and identify optimization possibilities of existing and potential new implementations of the specifications. (Hartig & Pérez 2018:1155.)

In the conclusions, the Hartig and Pérez state that the popularity of GraphQL as an alternative to traditional REST APIs has increased. They also confirm that GraphQL, similarly to other query languages, utilizes both a schema to describe the structure of the data, and a declarative language for clients to access the data. Finally, they provide a full formalization of the semantics of the language, showing that three different complexity related problems, efficiently can be solved. Thus they help developers in implementing more robust GraphQL interfaces. (Hartig & Pèrez 2018:1163.)

6 PLANNING AND IMPLEMENTATION

In the previous chapters, web APIs have been presented in general, as well as a detailed description of GraphQL-, and REST-based APIs. Previous studies of the two API technologies were also presented. The following sections cover the planning and the implementation of the web application conducted for Gambit, as well as a presentation of Toggl and the public APIs provided.

6.1 Toggl

Keeping track of time is an ever-important task, in businesses today. Especially in the field of software engineering, operating as IT consultants, time tracking is crucial. Projects are typically sold on a per hour basis, even though projects are occasionally offered for a fixed price, depending on the type of project and the complexity of it.

Toggl is a web-based tool for tracking time. It features team management and project time management, enabling the users to have several clients, projects, tasks within a project et cetera. Toggl also provides a public API for fetching data, and for managing time entries, which involves creating, updating, deleting and reading entries. The API has different types of endpoints, from fetching raw data, to creating more complex reports. Toggl, in combination with agile management, eases the time tracking and project management for different sizes of software development companies.

Toggl API is divided into two APIs called Toggl API and Reports API. The Toggl API is mainly for changing data, including creating new time entries. For read-only data, the Reports API is to be used. This API gives the user access to time entries within a workspace as well as aggregated data for reporting. Both APIs use the JSON data format for requests and responses. For authentication, basic access authentication is used. The user passes a header called *Authorization* with the value *Basic <credentiatials>,* where the credentials are the username and password joined by a single colon, encoded using base64 encoding. (Toggl 2017.)

6.1.1 Toggl API

The Toggl API provides CRUD operations on all the different entities. These are among other Clients, Projects, Tasks, Time entries, Users and Workspaces. The API is based on the REST architectural style. Hence, the most common HTTP methods are used for reading or updating data. Using Projects as an example, reading the details of a project can be done by performing a GET request to the URI *https://www.toggl.com/api/v8/projects/{project_id}*. Upon a successful response, a payload formatted in the JSON format is returned as shown in figure 7.


```
{
        "data": {
                "id":193838628,
                "wid":777,
                "cid":123398,
                "name":"Changed the name",
                "billable":false,
                "active":true,
                "at":"2013-03-06T12:15:37+00:00",
                "template":true,
                "color":"6"
        }
}
```


Figure 7. Example of a successful response from the Toggl API. (Toggl 2017b.)

Similarly, to update or to delete a project, a PUT request with the changed fields respectively a DELETE request is done to the same URI. A complete list of the available request within the Toggl API can be found in Appendix A.

6.1.2 Reports API

The Reports API consists of four different types of reports. These are the weekly report, detailed report, summary report and project dashboard. The reports share the same base URL of *https://toggl.com/reports/api/v2/{report_type}*. A request to any of the report

types consists of a set of standard request parameters, followed by the specific report type parameters. The list of standard request parameters is presented in table 2.

Table 2. List of the Reports API standard request parameters. (Toggl 2017c.)

Parameter	Description
user_agent	**Required**. The name of your application or your email address so we can get in touch in case you're doing something wrong.
workspace_id	**Required**. The workspace whose data you want to access.
since	ISO 8601 date (YYYY-MM-DD) format. Defaults to today - 6 days.
until	ISO 8601 date (YYYY-MM-DD) format. Note: Maximum date span (until - since) is one year. Defaults to today, unless since is in the future or more than a year ago, in this case until is since + 6 days.
billable	"yes", "no", or "both". Defaults to "both".
client_ids	A list of client IDs separated by a comma. Use "0" if you want to filter out time entries without a client.
project_ids	A list of project IDs separated by a comma. Use "0" if you want to filter out time entries without a project.
user_ids	A list of user IDs separated by a comma.
members_of_group_ids	A list of group IDs separated by a comma. These limits provided user_ids to the members of the given groups.
or_members_of_group_ids	A list of group IDs separated by a comma. This extends provided user_ids with the members of the given groups.
tag_ids	A list of tag IDs separated by a comma. Use "0" if you want to filter out time entries without a tag.
task_ids	A list of task IDs separated by a comma. Use "0" if you want to filter out time entries without a task.
time_entry_ids	A list of time entry IDs separated by a comma.
description	Matches against time entry descriptions.
without_description	"true" or "false". Filters out the time entries which do not have a description (literally "(no description)").
order_field	-For detailed reports: "date", "description", "duration", or "user" - For summary reports: "title", "duration", of "amount" - For weekly reports: "title", "day1", "day2", "day3", "day4", "day5", "day6", "day7", or "week_total"
order_desc	"on" for descending, or "off" for ascending order.
distinct_rates	"on" or "off". Defaults to "off".
rounding	"on" or "off". Defaults to "off". Rounds time according to workspace settings.

dis-play_hours	"decimal" or "minutes". Defaults to "minutes". Determines whether to display hours as a decimal number or with minutes.

A successful response consists of a set of standard fields along with report specific data as shown in figure 8.

```
{
  "total_grand":null,
  "total_billable":null,
  "total_currencies":[{"currency":null,"amount":null}],
  "data":[]
}
```

Figure 8. Example response of the common fields for each report type. (Toggl 2017c.)

The different report types enable the user to easily obtain an overview of the data in Toggl and provide aggregated data for different purposes, as well. The specific function of the different report types is presented in the following list.

- *Weekly report* – This report gives aggregated data over seven-day durations or earnings grouped by projects and/or users.

- *Detailed report* – The detailed report is used for fetching time entries based on some given request parameters/filters. Since the amount of data might be very high, this report returns paginated data.

- *Summary report* – Similarly to the detailed report, the summary report returns data regarding the time entries. However, the data is grouped on two different levels, according to the user's request parameters. The data can be grouped by projects, clients or users and sub-grouped by time entries, tasks and users, among others.

6.2 Requirements

Previously, Microsoft's Excel was used in Gambit for the purpose of following up on projects. Data was exported from Toggl manually to Excel, where, together with other financial data, a report of the current status of a project was created. The data included in the Excel template was the hours offered for a certain task, and the number of hours spent on each task. The invoiced hours were manually inserted from the accounting software. The hourly rate used for a customer was inserted, and based on the invoiced hours, a net hourly rate value could be calculated.

The goal of the developed tool is to replace this spreadsheet usage and ease the follow up of invoice hours for a specific task within a project. Another goal is to get an overview of the net hourly rate of projects. A requirement of the system is flexibility and extensibility, since the usage may vary, and the tool will be further developed later, as the requirements change. The development of the tool follows an agile process, meaning that the requirements must not be completely known in the planning phase, rather they evolve during the development as the tool is tested and evaluated. An initial general list of requirements is shown in table 3.

Table 3. The initial list of requirements.

Section	**Requirement**
Data	Fetch data from Toggl: customers, projects, tasks, time-entries et cetera.
	Synchronize the data every day, with the possibility to synchronize manually
	Extend the data, to include invoices, add the possibility to select project manager, notes about project invoicing, billing notes et cetera.
Security	The application must be secured with login.
Project-list	List all projects
	Filter on a given time interval
	Search for a project
Project details	List tasks within a project, with used and offered hours
	Add invoices for a task

6.3 Selection of technologies

Based on the general list of requirements above, the selection of technologies can be made. Choosing the right technology for an application is a complex matter, especially when the application is a commercial product, used by a lot of customers. However, in this case, the application is limited to internal use in a small company, allowing for a more flexible and unconstrained selection process.

6.3.1 Selection criteria

The following factors have been considered when deciding applicable technologies to use.

Security – The developed application handles sensitive information. Therefore, it is of utter importance that login is required for any access to the application.

Scalability & Performance – Since the application is merely for internal use, scalability is not a strict requirement. The application should be able to handle small loads, but no heavy peaks are expected. Likewise, the performance of the application does not need to be extremely optimized, as long as the user experience is considered somewhat good.

Development cost – The application will not be monetized, rather it will only decrease the amount of manual work of the project managers. Hence, the development time, thus the cost, will be kept low, utilizing ready-made technologies and components when possible.

Eco-system – The chosen technologies, should preferably hold rich eco-systems and an active development for further maintenance and development of the application.

6.3.2 Server-side

For the server-side a PHP-based framework called SilverStripe is used. SilverStripe is an open-source Content Management System (CMS), which allows creating a site fast with

small efforts, while also offering creation of custom functionality. The CMS includes user management, authentication and an administrator interface for configuring the application. Additionally, the framework possesses an Object-Relational Model (ORM) built-in, which handles all the underlying communication with the database, in this case, a MySQL database. Using the ORM in SilverStripe, the class *DataObject* is to be extended by a class representing a database table. When developing the application, the database schema is automatically generated based on the existing *DataObjects*. (SilverStripe 2019.)

The benefits of using SilverStripe in the server-side are due to the many built-in functionalities in the framework mentioned above. Due to these features, the focus can be held on developing the main functionalities of the application. One could argue that a dynamically typed language is not suitable for this kind of application. However, utilizing a well-structured framework, together with intelligent code completion, bring many of the benefits of a statically typed language.

6.3.3 Client-side

For the client side, the JavaScript library React is used. React is a declarative, efficient and flexible library for building user interfaces. A user interface built in React consists of small pieces, called *components,* which combined form the whole page. These components are isolated, enabling the creating of a modular and flexible user interface. The typical way of using React is creating a Single-Page-Application (SPA). In a SPA, the application resources are usually fetched upon visiting the site. The application then runs completely in the client-side as a JavaScript application, which composes the HTML instead of fetching the ready-made HTML resources from the server. When interaction with the server is needed, the application utilizes a web API.

Today, some of the most popular JavaScript frameworks/libraries are React, Angular and Vue. They all have their benefits and drawbacks. For the application in this thesis, the reasonable choice is to use one of these, and React was chosen. For the GraphQL client, Apollo client is used. Apollo is an open source implementation of GraphQL for both

server and client. The Apollo client includes integrations towards the view layers mentioned above. (Apollo 2019a.) For the design, a simplistic design is used, using a ready-made React library which is built on the Material Design specifications, made by Google.

6.4 System design and architecture

The first step in the application flow consists of consuming a RESTful API provided by Toggl. The data must be synchronized and kept up to date with Toggl, to avoid inconsistencies between the data in Toggl, and the data in the application database. For this, a synchronization service is created with a single purpose of keeping the application database in synchronization with Toggl, which in turn acts as the single source of truth. The rest of the server-side application is to provide necessary features, such as user management, and to secure the data for unauthorized users. Finally, a GraphQL API is provided for the SPA.

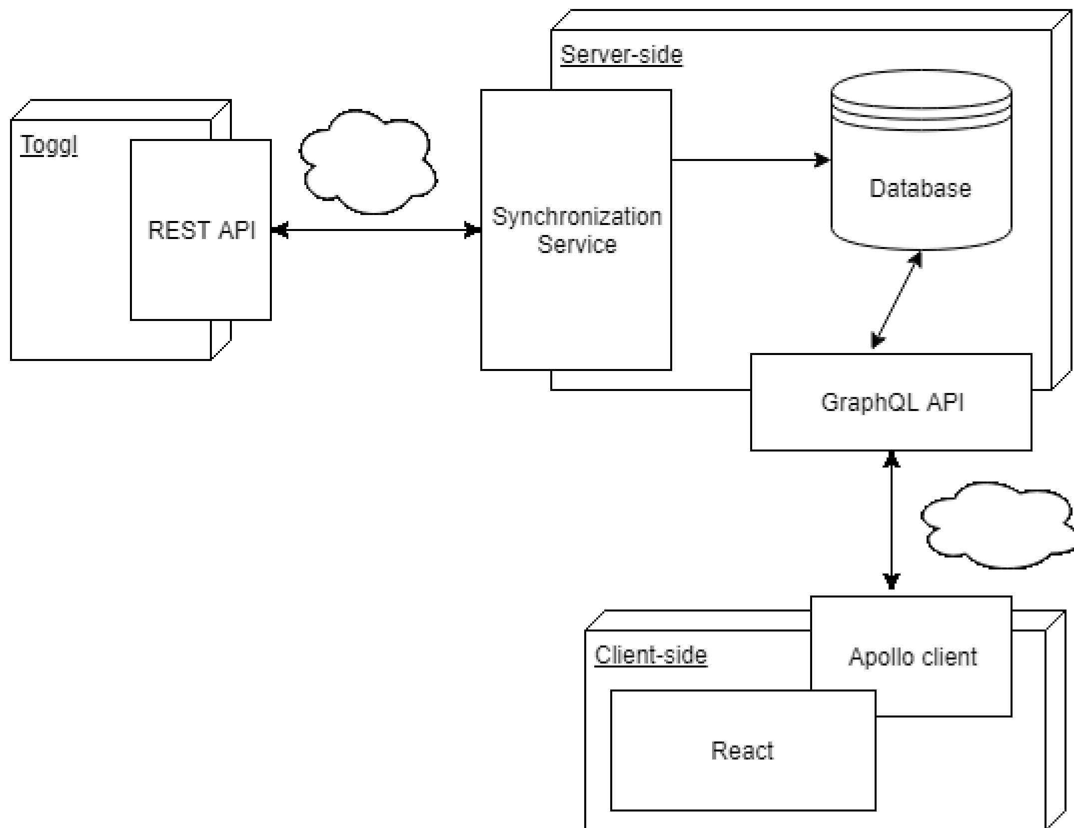

Figure 9. The architecture of the application.

6.4.1 Data model

One of the reasons for developing the application, was to gain extended functionality and data compared to the data and model provided by Toggl. The data model from Toggl is explorable by using the APIs. A derived simplified data model from Toggl is shown in figure 10.

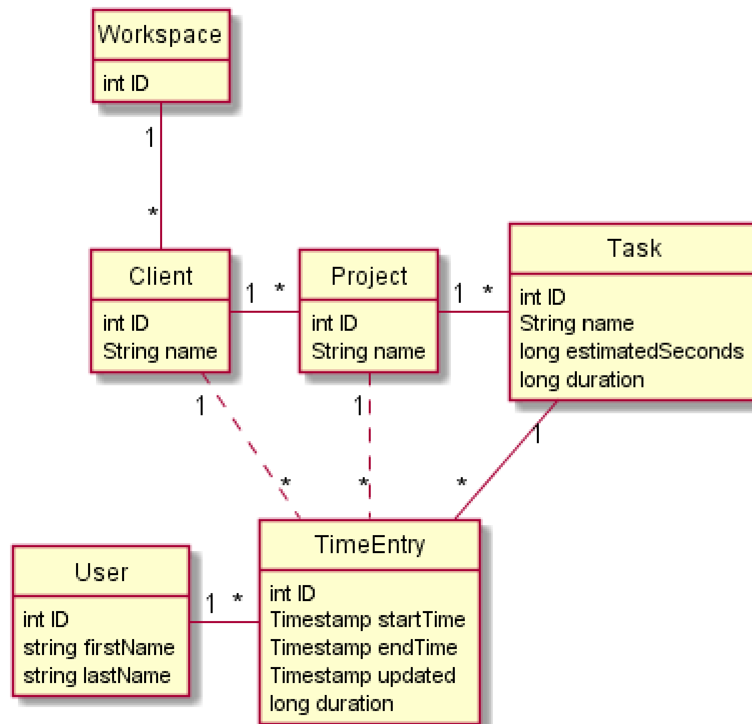

Figure 10. A simplified derived model of Toggl's data model.

6.4.2 Synchronization service

The synchronization service's task is to fetch all the data from Toggl for Gambit's workspace and store it in a database. This is not completely trivial since the data in Toggl may change after it has been fetched and stored in the database. This means that Toggl acts as a single source of truth, and the application needs to adapt the data accordingly. Keeping track of when an object has been updated in Toggl is easy, as Toggl in most cases provides a field called *LastUpdated*, which is a timestamp holding the value of the time when the object was last updated. By using this field, the data can be fetched from Toggl and the timestamp can be compared to the one saved in the database and updated when required.

For the case of the application developed in this thesis, both APIs from Toggl are utilized. The Toggl API is used for fetching all the data regarding the workspace, clients, projects, tasks and users. The process of fetching data is described in the following sequence diagram.

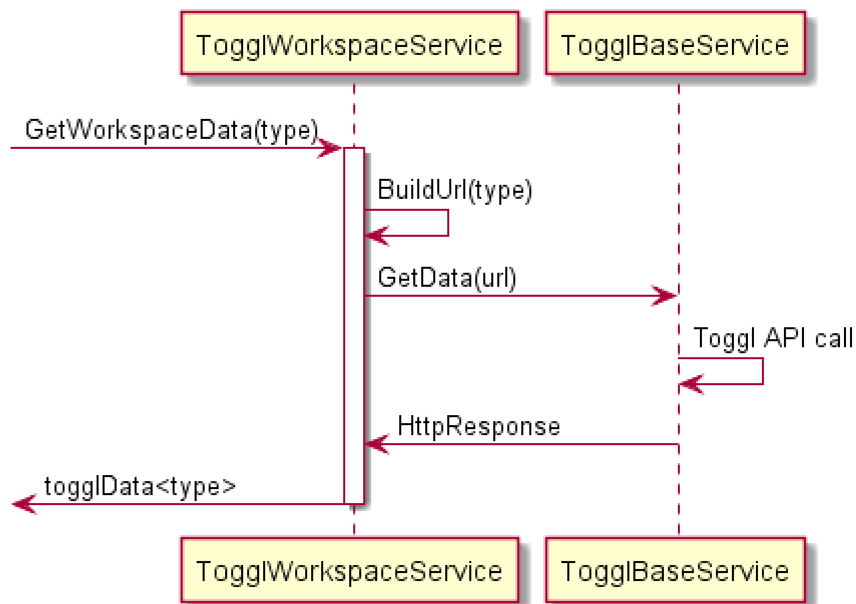

Figure 11. Diagram of fetching workspace specific data using the Toggl API.

The *TogglWorkspaceService* has a function called *GetWorkspaceData(type),* which has a parameter specifying the type of resource (projects, users, tasks and so on) to be fetched. Based on this, the complete URI is built and sent to the *TogglBaseService,* which performs the actual HTTP request to the Toggl API. The response is returned and parsed for further usage.

For the actual time entry data, the Report API is used. The process of fetching time entries involves pagination and is performed according to the sequence diagram below.

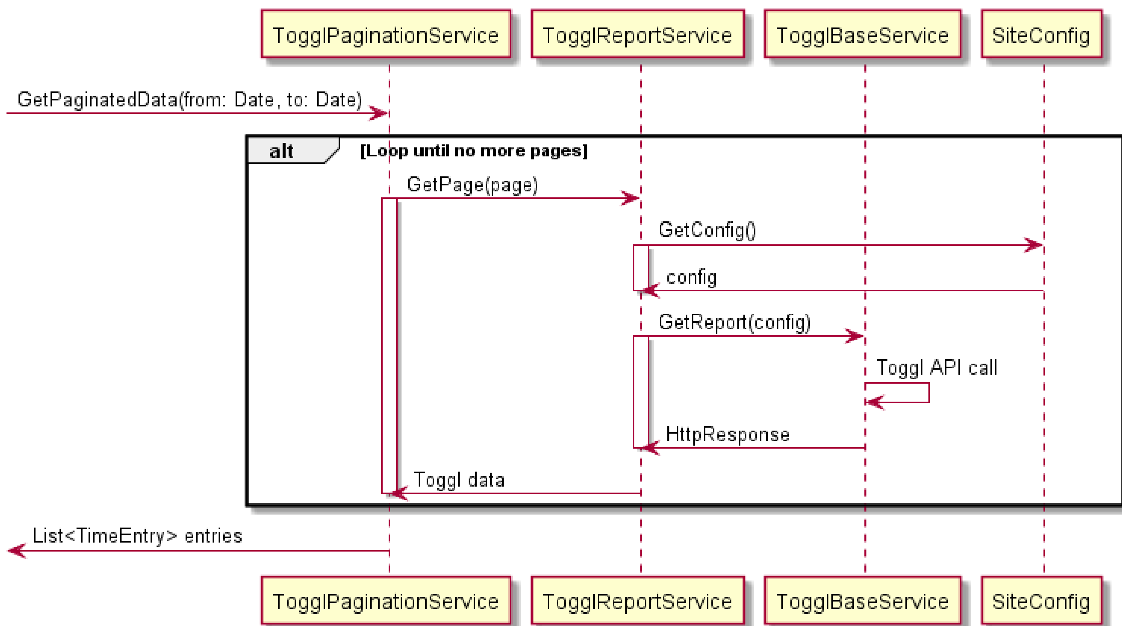

Figure 12. Sequence diagram of performing paginated requests for time entries using the Report API.

6.5 Creating the GraphQL API

Using SilverStripe and GraphQL is simple, since a GraphQL module is included in SilverStripe. The module uses scaffolding for exposing the *DataObjects*, thus enabling fast and simple creation of the GraphQL API. The scaffolding generates basic CRUD operations and provides pagination for exposing larger sets of data as well. To expose a *DataObject* the name of the class needs to be defined in a configuration file. Then, together with scaffolding, the final GraphQL schema can be generated.

```yaml
schema:
  scaffolding:
    types:
      Client:
        fields: [ID, Name]
        operations:
          read:
            paginate: false
        nestedQueries:
          Projects: true
```

Figure 13. A snippet of the configuration file for the generation of the GraphQL schema.

The example above is part of the configuration of the scaffolding, which enables the creation of the GraphQL schema. Here, the *DataObject* of type *Client* is provided, defining that the fields *ID* and *Name* should be exposed. Furthermore, the available operations are read-only, with the configuration of excluding pagination, hence all *Clients* can be fetched in a single query. The *nestedQueries* option specifies that the *Projects* for a certain *Client* can be fetched in a single query.

6.6 Consuming the GraphQL API

As mentioned previously, consuming the GraphQL API is done by using React and Apollo client. Apollo client provides components for queries and mutations in the same declarative way as other React components. The components handle all the networking and provide loading and error states. After the data has been successfully loaded it is injected into the render function of the component, thus allowing further processing and rendering of the data.

The query itself is written, using utilities provided by Apollo client for parsing GraphQL queries. These functions parse a GraphQL query from a string to a standard GraphQL Abstract Syntax Tree (AST). The GraphQL AST in turn enables readability and easy

modification of the query. The Query component has a *prop*, which in React is an arbitrary input for a component, called *query* which is the query parsed as a GraphQL AST. In the snippet shown below, a query called *readProjects* is defined, which has two arguments. These are two dates specifying the time interval from where the projects should be fetched. Basic information about the projects is fetched, that is, the *ID, Name, LastUpdated* timestamp, along with the assigned project manager for the project and the client to which the project belongs.

```
const query = gql`
  query readProjects($from: String, $to: String) {
    readProjects(From: $from, To: $to) {
      ID
      Name
      LastUpdated
      Assignee {
        ID
        FirstName
        Surname
      }
      Client {
        Name
      }
    }
  }
`;
```

Figure 14. GraphQL query for fetching all projects within a given time interval.

```
<Query
  query={query}
  variables={{ from: this.state.from, to: this.state.to }}
>
  {({ loading, error, data }) => {
    if (loading) return <Loading />;
    if (error) return "error";

    const rows = this.getRows(data);
    return (
      <MUIDataTable
        className={"projects"}
        columns={this.columns}
        data={rows}
        options={this.options}
      />
    );
  }}
</Query>
```

Figure 15. Render function of the Projects component.

The example in figures 14 and 15, shows how the *Query* component is used. The query parsed as a GraphQL AST is passed as a *prop* along with the variables for the time interval. When the *Query* component mounts, an observable is created for the query. The component then subscribes to the result. First, the result is searched for in the Apollo cache. If it is not found, it will be queried from the server. While fetching the result, the component is rendering the loading state. When a response is received or an error occurs, the error text is shown, or the data is received for further processing. In case of valid data, the projects are parsed from the response data and passed along with options and other data to the *MUIDataTable* component, which handles the rendering of the projects in a list. (Apollo 2019b.)

7 EVALUATION AND RESULTS

In this chapter, the results of the thesis and the developed application are presented. GraphQL is evaluated along with an evaluation of the application based on a set of criteria. The evaluation of GraphQL is done according to different aspects of using GraphQL in the development of the application. These aspects are subjectively evaluated together with a hypothetical corresponding implementation using a RESTful API. The relevant aspects in the evaluation are the implementation and consumption of the API, the complexity of data fetching as well as client-side caching of the content.

The results of the research papers presented in the literature review revealed that in some specific cases, as the complexity of the data increases, GraphQL might a better alternative for implementing the API compared to the traditional RESTful APIs. The studies also showed that GraphQL in many cases increases performance, as the number of requests to the server can be substantially decreased. This evaluation provides a more higher-level evaluation from the developer's perspective.

The implementation of the GraphQL API is dependent on the technologies used. In its essence, the schema needs to be generated, either manually or as in this case, generated by code and scaffolding techniques. Producing the GraphQL API is therefore easily done, after the data model is designed and implemented. Similarly, a RESTful API implementation often relies on higher-level libraries and technologies. In light of the technologies used in the development of the application, using GraphQL for exposing the data, is a fast and reliable way.

Consuming the API is likewise to producing it, often dependant on high-level libraries which abstracts much of the work needed to be done. For GraphQL, as Apollo Client is used in this application, it enables the developer to write queries immediately without regards on any lower-level details. Along with the introspective tool GraphiQL, the capabilities of the server can easily be inspected. Altogether, consuming the GraphQL API, is about writing a component directly with the only need of declaratively stating which data from the server to fetch, when needed. In a RESTful implementation, many good

libraries and principles exist, which give similar introspective capabilities as GraphQL. Also, consuming a RESTful version in modern JavaScript, could be easily done using a lightweight or no library at all. The benefits of using one over the other are therefore dependent on the language, frameworks and the libraries in use.

In the application, different levels of complex queries needed, for presenting the data. The GraphQL API allows for all the needed data to be fetched in a single request, minimizing bandwidth usage and latency. This is useful in some of the created components, as multiple resources often are needed for producing a view. However, even though the data can be specified in the exact structure wanted, in reality some complex queries take a rather long time to execute, leading to bad user experience. In these cases, the solution has often been to restructure the query. Comparing the complexity to a RESTful API, the benefits of GraphQL lay in the removal of over-fetching of the data. A RESTful implementation would in some cases need to perform up to 5 different requests for the rendering a single view, which in the GraphQL implementation can be done in one query.

Caching is a complex matter with no simple general solutions. In the case of this application caching is handled by the GraphQL client. The data is often changing, forcing the cache to update regularly. The Apollo Client caches all queries and their results. In case a mutation has been performed, which creates stale data in the cache, the cache must be updated. In simple cases, the cache is automatically updated, whereas in more complex scenarios, the cache needs to be manually updated. As caching is a constraint in the REST architectural style, caching is easier to manage, by allowing the browser to manage the cache. In that aspect, a RESTful implementation could be a better alternative. However, in the developed application, caching is not a strict requirement, and is managed well by the default cache configuration in Apollo Client.

In table 4, the previous aspects have been graded in a 1 – 5 scale according to the suitability of the technologies in the different areas. The table shows that in the developed application, GraphQL is the more suitable alternative for producing the API. Some aspects are dependent on third party libraries, with little to no differences in using either

API technology, whereas in terms of managing the complexity of the data, GraphQL is clearly the better alternative.

Table 4. Evaluation of a GraphQL implementation and a corresponding hypothetical RESTful implementation of the developed application's API.

Area	GraphQL	REST	Note
Producing the API	3	3	Dependent on the framework and language used.
Consuming the API	3	3	Dependent on the framework and language used.
Complexity	5	1	Complex queries are needed for different views. With GraphQL a single request can be made, fetching the exact data needed.
Caching	2	4	Out of the box, caching is more complex in GraphQL, since HTTP caching can not be used.

The developed application is a MVP, with the intention of being further developed in the future, which is outside the scope of this thesis. The evaluation of the application is two-folded, on one side, there is the user evaluation of the application. Things considered here are, usability and determining if the application fulfills the requirements. On the other hand, the application is evaluated from a developer's perspective. Factors to be considered here are, testability, flexibility and learnability.

When it comes to usability, the most important aspect is that the application is well designed in terms of user experience, providing an intuitive way for working and using the application on a daily basis. The users, in this case project managers, should in case needed, be provided with documentation and/or schooling, to get started with using the application. For achieving this, relevant schooling has been provided and for good user experience, material design was utilized. Material design is a design language consisting of guidelines, best practices, tools and components provided by Google and many open-source projects, for user interface design.

Since this is an MVP, flexibility is crucial for further development and addressing of issues. In the future, development of the application might include integrations to other software, for example accounting software. Other features could be more advanced analysis of the data, to find patterns of successful projects and estimates. For this, the foundation needs to be strong, with a good and thought-through data model. Flexibility in this case means that exposing new functionality should be simple.

Flexibility has been achieved by first and foremost investigating in a good data model. For this the Toggl API has been explored, to get an understanding of how Toggl has implemented their data model. Furthermore, using GraphQL enables the client side to specify the exact data needed. As the requirements changes and new functionalities are added, the client is able to directly specify and use the field. However, in case the data model is updated in a way that an existing field is removed or modified, this might lead to complications as the client side has to be updated accordingly in case it's dependant on the field.

Testability provides ways of ensuring that an application works in the intended way according to the requirements and the specification. Testing an application can be done in multiple steps, from the early design phase to the later steps of implementing and maintaining the developed application. In the design phase, the application is designed by composing small units, which if ensured performing as intended, the overall system is functioning as intended. When implementing the units, each unit is tested individually, which ensures that they function correctly. In the later steps when the units are composed together, forming the complete application, integration tests are performed. These ensure that the complete system fulfills its specification. (Mili & Tchier 2015:24.)

In the case of the developed application, there are some critical units which are testable. These are among others, the synchronization of the data. It is crucial that the data in the application is up to date with the data in Toggl. Other testable units are the CRUD operations on the data model. In the scope of this thesis, unit testing or integration testing is not performed but should be considered in the future development of the application.

The evaluation of the application is presented in table 5, where each different aspect has been subjectively graded on a scale of 1 – 5 of how well the application fulfills the aspect.

Table 5. Summary of the evaluation of the application.

Aspect	Grade	Note
Usability	4	Using material design creates a minimalistic and easily manageable user experience
Testability	2	The application is testable, but tests are not implemented in the scope of the thesis
Flexibility	5	A good data model along with GraphQL provides a good flexible foundation which can be further developed
Learnability	3	The user interface is logically structured and easily explorable

8 CONCLUSIONS AND DISCUSSION

The developed web application is a Single-Page-Application built with the popular JavaScript library React. For the user interface, Material Design was used as the design principles, by utilizing ready-made libraries developed in React. SilverStripe was selected as the server-side framework. Benefits of using the PHP-based framework lay in fast and easy setup of user management and previous experience of using the framework. When considering SilverStripe and GraphQL, the setup of the GraphQL runtime is done by utilizing the SilverStripe GraphQL module. This module provides helpful utilities such as the GraphiQL web-based tool for schema inspection. Scaffolding is used in the module, which allows the developers to declare the schema rapidly. In an environment with strict performance requirements, other technologies, based on a compiled language such as C# and .NET Core could be a better option. However, due to the loose requirements of performance, SilverStripe was chosen as the framework.

The application fulfills the initial requirements as an MVP and has been developed to be extensible and flexible. The structure of the application is testable; however, tests have not been written in the scope of this thesis. In the future for further development, and if the application is utilized in a more central role, it is crucial that at least the core logic of the application is tested with unit and integration tests.

GraphQL was the technology chosen in the implementation of the application. How would a corresponding RESTful API implementation look like? What would be the benefits and drawbacks of it? Taking it further, one could ask, is the server layer between the Toggl API and the client needed at all, or could the same application be developed, calling the Toggl REST API directly from the client?

A RESTful API implementation would certainly have some benefits over the chosen GraphQL implementation. The development time would be similar to the developing the GraphQL API, by using ready made modules and libraries for SilverStripe and React. One of the greatest benefit would be caching, as shown in chapter 7, since it would be handled by the browser, thus removing the need of a complex caching library. This is,

however, not a problem with the GraphQL implementation, since Apollo client manages caching well. A drawback of using a RESTful API would be in the complexity of fetching the data, which in some cases, would require the client to perform five different requests for rendering a view in the application. In the current implementation, GraphQL manages this in a single query.

Using the Toggl API directly, removing the middle layer is also considered. This would have reduced the development time greatly, since a considerable amount of time was spent on the synchronization process between Toggl and the server. However, the need of storing the data in a middle layer is inevitable, since the application is dependant on an extended data model, which is not provided by Toggl. Therefore, the purpose of the application would be eliminated without the middle layer.

Further research ideas derived from the thesis, would be to implement the same API using REST, and measure the performance, as well as, evaluate the process of implementation of the API from the developer's perspective. In that case, a better judgement could be done on the suitability of GraphQL, as a basis for the creating the API.

The process of developing the application has been exciting and fun. Developing an application from the beginning brings many aspects which need to carefully be designed and planned, in contrast to developing an existing application containing legacy code. I have learned a lot about API technologies, specifically about RESTful-, and GraphQL-APIs, along with the modern JavaScript library React. This kind of application is a good case for learning and utilizing new technologies.

In the field of web development, as the complexity of the applications developed will increase, we will probably see an increase in the usage of GraphQL as a means of creating APIs. We already see giants such as Facebook and Netflix, developing their own protocols for fetching data, as the standard approaches of using RESTful or other API technologies are not sufficient for the increased complexity in web applications.

REFERENCES

Apollo. (2019a). The Apollo GraphQL platform [online]. [17.04.2019]. Available at: https://www.apollographql.com/docs/intro/platform

Apollo. (2019b). Queries [online]. [08.05.2019]. Available at https://www.apollographql.com/docs/react/essentials/queries.

Biehl, M. (2016). *RESTful API Design. APIs your consumers will love.* API-University Press. ISBN 978-15147351169.

Biehl, M. (2018). *GraphQL API Design.* API-University Press. ISBN 978-1979717526.

Box D., D. Ehnebuske, G. Kakivaya, A. Layman, N, Mendelsohn, H. F. Nielsen, S. Thatte & D. Winer. (2000). Simple Object Access Protocol (SOAP) 1.1 [online]. [05.03.2019]. W3C Note 08 May 2000. Available at: https://www.w3.org/TR/2000/NOTE-SOAP-20000508/

Bray, T. (2017). The JavaScript Object Notation (JSON) Data Interchange Format. Internet Engineering Task Force (IETF) Request for Comments 8259 [online]. [05.03.2019]. 2-16. Available at: https://www.rfc-editor.org/rfc/pdfrfc/rfc8259.txt.pdf. ISSN 2070-1721.

Bray, T., J. Paoli, C. M. Sperberg-McQueen, M. Maler, F. Yergeau. (2008). Extensible Markup Language (XML) 1.0 (Fifth edition) [online]. [05.03.2019]. W3C Recommendation 26 November 2008. Available at: https://www.w3.org/TR/xml/

Buna S. (2016). *Learning GraphQL and Relay. Build data-driven React applications with eases using GraphQL and Relay.* Livery Place: Packt Publishing Ltd. ISBN 978-1-78646-575-7.

Eizinger T. (2017). *API Design in Distributed Systems. A Comparison between GraphQL and REST.* University of Applied Sciences Technikum Wien.

Facebook. (2018a). Overview [online]. [28.02.2019]. Available at: https://facebook.github.io/graphql/June2018/#sec-Overview

Facebook. (2018b). Schema [online]. [28.02.2019]. Available at: https://facebook.github.io/graphql/June2018/#sec-Schema

Facebook. (2018c). Non-Null [online]. [28.02.2019]. Available at: https://facebook.github.io/graphql/June2018/#sec-Type-System.Non-Null

Fielding R. T. (2000). *Architectural Styles and the Design of Network-based Software Architectures* [online]. Doctoral dissertation, University of California. Available at: https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm

Fielding R., Reschke J. (2014). *Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content* [online]. RFC 7231. Internet Engineering Task Force (IETF). Available at: https://tools.ietf.org/html/rfc7231

Gambit. (2019). Home page [online]. [28.02.2019]. Available at: https://www.gambit-group.fi

GraphQL. (2019a). Introduction to GraphQL [online]. [28.02.2019]. Available at: https://graphql.org/learn/

GraphQL. (2019b). Schemas and Types [online]. [28.02.2019]. Available at: https://graphql.github.io/learn/schema/

Hartig O. & Pérez J. (2018). *Semantics and Complexity of GraphQL.* Lyon, France. Proceedings of the 2018 World Wide Web. Pages 1155 – 1164. ISBN 978-1-4503-5639-8.

Helgason A. F. (2017). *Performance analysis of Web Services. Comparison between RESTful & GraphQL web services.* University of Skövde.

Masse M. (2016). *REST API Design Rulebook: Designing Consistent RESTful Web Service Interfaces*. California: O'Reilly Media Inc. ISBN 978-1-449-31050-9.

Mili A. & Tchier F. (2015). Software Testing: Concepts and Operations, John Wiley & Sons, Incorporated. ProQuest Ebook Central, https://ebookcentral-proquest-com.proxy.uwasa.fi/lib/tritonia-ebooks/detail.action?docID=4040909.

Pasma T. (2018). *Web API Capacity Study. A comparison of REST and GraphQL*. Aarhus University.

SilverStripe. (2019). Introduction to the Data Model and ORM [online]. [17.04.2019]. Available at: https://docs.silverstripe.org/en/4/developer_guides/model/data_model_and_orm/

Sriparasa S. (2013). JavaScript and JSON Essentials [online]. Packt Publishing Ltd. Available at: https://ebookcentral-proquest-com.proxy.uwasa.fi/lib/tritonia-ebooks/reader.action?docID=1481127.

Toggl. (2017a). Toggl API Documentation [online]. [07.05.2019]. Available at https://github.com/toggl/toggl_api_docs.

Toggl. (2017b). Projects [online]. [07.05.2019]. Available at https://github.com/toggl/toggl_api_docs/blob/master/chapters/projects.md.

Toggl. (2017c). Toggl Reports API v2 [online]. [07.05.2019]. Available at https://github.com/toggl/toggl_api_docs/blob/master/reports.md.

Toggl. (2017d). Toggl API v8 [online]. [24.05.2019]. Available at https://github.com/toggl/toggl_api_docs/blob/master/toggl_api.md.

Totty B., D. Gourley, M. Sayer, A. Aggarwal & S. Reddy. (2009). *HTTP: The Definitive Guide* [online]. California: O'Reilly Media Inc. ISBN 978-156-592509-0.

APPENDIX

A. List of HTTP status codes and reason phrases

Table 6. A non-exhaustive list of status codes and reason phrases as provided by Fielding & Reschke. (2014:49.)

Code	**Reason-Phrase**
100	Continue
101	Switching Protocols
200	OK
201	Created
202	Accepted
203	Non-authoritative Information
204	No Content
205	Reset Content
206	Partial Content
300	Multiple Choices
301	Moved Permanently
302	Found
303	See Other
304	Not Modified
305	Use Proxy
307	Temporary Redirect
400	Bad Request
401	Unauthorized
402	Payment Required
403	Forbidden
404	Not Found
405	Method Not Allowed
406	Not Acceptable
407	Proxy Authentication Required
408	Request Timeout
409	Conflict
410	Gone
411	Length Required
412	Precondition Failed
413	Payload Too Large

414	URI Too Long
415	Unsupported Media Type
416	Range Not Satisfiable
417	Expectation Failed
426	Upgrade Required
500	Internal Server Error
501	Not Implemented
502	Bad Gateway
503	Service Unavailable
504	Gateway Timeout
505	HTTP Version Not Supported

B. List of supported API requests for the Toggl API

Table 7. Complete list of possible API requests from the Toggl API. (Toggl 2017d.)

Type	**Available requests**
Authenticate and get user data	HTTP Basic Auth with e-mail and password
	HTTP Basic Auth with API token
	Authentication with a session cookie
	Destroy the session
Clients	Create a client
	Get client details
	Update a client
	Delete a client
	Get clients visible to the user
	Get client projects
Groups	Create a group
	Update a group
	Delete a group
Projects	Create a project
	Get project data
	Update project data
	Delete a project
	Get project users
	Get project tasks
	Delete multiple projects
Project users	Create a project user
	Update a project user

	Delete a project user
	Add multiple users to a project
	Update multiple project users
	Delete multiple project users
Tags	Create a tag
	Update a tag
	Delete a tag
Tasks	Create a task
	Get task details
	Update a task
	Delete a task
	Update multiple tasks
	Delete multiple tasks
Time entries	Create a time entry
	Start a time entry
	Stop a time entry
	Get time entry details
	Update time entry
	Delete time entry
	Get time entries started in a specific time range
	Bulk update time entries tags
Users	Get current user data and time entries
	Update current user data
	Reset API token
	Sign up a new user
Workspaces	Get user workspaces
	Get workspace users
	Get workspace clients
	Get workspace groups
	Get workspace projects
	Get workspace tasks
	Get workspace tags
Workspace users	Invite users to the workspace
	Update workspace user
	Delete workspace user
	Get workspace users for a workspace
Dashboard	Get a general overview of your team