MASTER THESIS

Thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Engineering at the University of Applied Sciences Technikum Wien - Degree Program Software Engineering

API Design in Distributed Systems: A Comparison between GraphQL and REST

By: Thomas Eizinger, BSc

Student Number: 1510299001

Supervisors: Mag. DI Bernhard Löwenstein
Dr. Lukasz Juszczyk

Vienna, May 4, 2017

FH University of Applied Sciences
TECHNIKUM WIEN

Declaration

“As author and creator of this work to hand, I confirm with my signature knowledge of the relevant copyright regulations governed by higher education acts (for example see §§21, 42f and 57 UrhG (Austrian copyright law) as amended as well as §14 of the Statute on Studies Act Provisions / Examination Regulations of the UAS Technikum Wien).

I hereby declare that I completed the present work independently and that any ideas, whether written by others or by myself, have been fully sourced and referenced. I am aware of any consequences I may face on the part of the degree program director if there should be evidence of missing autonomy and independence or evidence of any intent to fraudulently achieve a pass mark for this work (see §11 para. 1 Statute on Studies Act Provisions / Examination Regulations of the UAS Technikum Wien).

I further declare that up to this date I have not published the work to hand nor have I presented it to another examination board in the same or similar form. I affirm that the version submitted matches the version in the upload tool.“

Vienna, May 4, 2017 Signature

Kurzfassung

Viele Entwickler begrüßten die neue Art, Schnittstellen für Client-Anwendungen zu entwickeln, welche durch die Veröffentlichung der GraphQL-Spezifikation durch Facebook im Jahre 2015 ermöglicht wurde. GraphQL löst einige Probleme im Vergleich zu einem ressourcenorientierten Schnittstellen-Design, welches oft fälschlicherweise bereits als RESTful bezeichnet wird. Diese Arbeit identifiziert wichtige Aspekte im Hinblick auf das Schnittstellendesign von verteilten Systemen und unterscheidet hierbei klar zwischen rein ressourcen-basierten Ansätzen und tatsächlich REST-konformen Systemen. Durch den Vergleich von GraphQL und REST wird der Einfluss des jeweiligen Ansatzes auf diese Aspekte analysiert. Eine der wichtigsten Erkenntnisse diese Vergleichs ist die Verschiebung von Verantwortlichkeiten weg vom Server hin zum Client. Dies erhöht, je nach Anwendungsfall, die Komplexität des Clients deutlich.

Schlagworte: Software Architektur, Hypermedia, Auffindbarkeit, Entkoppelung, Web Services

Abstract

When Facebook open-sourced the GraphQL specification in 2015, many developers welcomed this new way of writing APIs for their clients. GraphQL solves a few problems that can occur when using a resource-based approach for the API design, an approach that is often falsely labeled as RESTful. This thesis makes an effort in clearly to distinguish between such resource-based approaches and actually RESTful systems. It identifies key aspects in the design of APIs and compares the impact of GraphQL and REST on these aspects. As a result of the comparison, several consequences are identified and presented. A consequence that stands out through its importance is the shift in responsibility between client and server which causes, depending on the use-case, highly increased complexity of the client.

Keywords: Software Architecture, Hypermedia, Discoverability, Decoupling, Web Services

Acknowledgements

Writing this thesis has been very interesting and inspiring, but also a hard time. Projects like these do not come without ups and downs but, retrospectively, even those were very insightful. Many people have contributed to this thesis through their ideas, experience, knowledge and general support. Referring to it as the work of a sole individual would be inappropriate.

I want to express my deepest gratitude to my supervisor Bernhard Löwenstein for his support and time. His feedback and experience helped me not only to limit the scope of the work but also allowed me to structure the gained knowledge in a reasonable and meaningful way. His insights and ideas were significant to the final result.

In addition, I also want to thank my second supervisor and work colleague Lukasz Juszczyk for taking the time to support me with his knowledge and experience despite his time consuming day-to-day job. At times where I was confronted with seemingly unsolvable problems, he helped me to focus on the central aspects by asking the right questions.

Worthwhile to be mentioned is also my room-mate and university colleague David Leitner. He assisted me in nailing down the point of my work by taking part in discussions and listing to countless elaborations on the topic.

Thanks also goes to Martin Eizinger and Lauren Kisly. They proofread the work and thereby contributed their excellent knowledge about English writing. Their support allowed me to present the achieved work in a way that is enjoyable[1] to read.

I also want to thank my parents for supporting me both financially and emotionally during my time at the university. Their support allowed me to fully focus on studying, freed me from money-related problems and thus helped me to achieve what I have today.

Credits also deserve to be given to the University of Applied Sciences Technikum Wien. It provided me with an environment in which I was able to learn a lot in a very short term.

Thank you all for taking part in the creation of this thesis. It was a pleasure to work with all of you.

[1] Hopefully also for you, future reader.

Contents

Introduction

Over the lifetime of a system, changes are often requested by stakeholders for various reasons, for example, in order to meet new market needs. Technological and conceptual decisions concerning the system that were made in the past often no longer fit the new requirements and should therefore be re-evaluated. Often, this re-evaluation is not done, which over time leads to a phenomenon developers refer to as legacy code or brownfield. Greenfield on the other hand refers to the situation in which a development team can start from scratch, without inconvenient constraints from earlier design decisions that no longer serve their original need thus hindering the team in their work. Developing greenfield is fun to engineers because they do not have to take into account old decisions that cannot be undone. It is the freedom of choice that makes developers enjoy greenfield development. However, the question arises, how to correctly get along in this forest of choices a development team is confronted with at the beginning of a project? More concretely, how can architectural decisions be made correctly and with confidence?

Fielding defined Representational State Transfer (REST) [1] as the architectural style around the thinking model used in the creation of the standards that describe the Web: HyperText Transfer Protocol (HTTP), Uniform Resource Identifier (URI) and HyperText Markup Language (HTML) [2]. Although HTTP is defined extensible in the way that it supports any media type, at the time of the definition of REST, it mostly dealt with use cases that involved media types intended to be directly rendered to the user, like HTML. The thinking model used in designing the key elements of the web, later defined as REST, allowed the Web to scale to the point of today. In addition to the massive growth in the number of websites, HTTP was increasingly used as the primary technology for building Application Programming Interfaces (APIs), interfaces intended to be consumed by machines instead of humans. REST and HTTP APIs superseded technologies like Simple Object Access Protocol (SOAP) in many areas mainly because of their simplicity. Richardson and Ruby argue in [3] that this simplicity is induced by the uniform interface of HTTP. It is the common understanding that for example invoking `GET` on a resource always retrieves a representation, independently of who created the resource, which technology the server is powered by or the actual physical storage location of the resource. However, just as with any other technology, there is no silver bullet. Perhaps it is the incredible success of the Web that led to the assumption that REST and HTTP are also a good choice for any API, but until 2015, building a good API almost always resulted in developers picking REST as their concept of choice, regardless of the actual use case.

In 2015, Facebook open-sourced the specification to a technology called GraphQL. It is an approach that features a declarative query language for fetching data from an API. It demonstrated that there are use cases in which something different than HTTP is necessary in order to meet the requirements of the system. Confronted with two approaches for creating APIs, the question arises, which one to choose for a given use case? Implementing systems is costly. It is therefore crucial to correctly decide upfront which approach to pick for a given use case.

The scope of this thesis is the comparison between the technology GraphQL against the principles and constraints defined in the REST style in order to evaluate their key differences. These differences are then discussed and should assist designers in making an informed decision when imposed with the task of choosing the best approach for a given use case. While Fielding compared several architectural styles for a prototypical hypermedia system, this thesis deals with systems known as APIs. This is different primarily in the way that using an API adds another layer of abstraction between what the user sees and acts upon and the actual communication between client and server. The following research questions are therefore defined for this thesis:

1. How can GraphQL be compared to the architectural style REST?
2. What are the differences between the two approaches?
3. How do both approaches tackle recurring problems in the context of API design?

Through the comparison of GraphQL and REST and by answering these research questions, this thesis makes the following contributions:

- Identification of key aspects for designing APIs
- Identification of the architectural principles behind GraphQL
- An effort in applying Fielding's model in a new context, that is, for comparing a concrete technology to an architectural style

1 Software Architecture

1.1 History

Many efforts have been made to define the term software architecture. This section aims to provide a brief overview over the history of definitions for this term.

Parnas is one of the pioneers in the research field of software and software architecture. His work on module design [4], decomposition of systems [5] and structures in software systems [6] [7] built the foundation for a lot of further research and definitions. Among those, Perry and Wolf [8] were the first to consider data and therefore a run time aspect in their definition of software architecture. As Fielding reasons in [1, Section 1.2.3], previous definitions fail to account for that. The model by Perry and Wolf defines software architecture as a set of architectural *elements* which have a particular *form* due to a specific *rationale*. They further categorize these elements into *processing*, *data*, and *connecting* elements. Fielding further extends this model by excluding rationale [1, Section 1.2]. He reasons that, although important for the evolution of a system and its architecture, once it is implemented, the way the system works is independent from its original rationale. The ANSI/IEEE 1471-2000 standard [9], later superseded by ISO/IEC/IEEE 42010:2011 [10] takes a similar approach by defining software architecture as "fundamental concepts or properties of a system in its environment embodied in its elements, relationships, and in the principles of its design and evolution" [10]. Interestingly, this definition mentions the design principles as a part of the architecture, although Fielding already reasonably argued that rationale cannot be a part of software architecture.

Inspired by the model of Perry and Wolf, Fielding defined software architecture as "an abstraction of the run time elements of a software system during some phase of its operation" [1]. This definition stresses two points:

1. The architecture is an abstraction of a system's run time elements.
2. It can vary with the phase of operation.

Fielding mentions the example of a configuration file [1, Section 1.1] that is considered a data element during the start-up phase but looses its attribute of an architectural element during normal processing.

Clements *et al.* [11, Prologue 4.1] and Bass *et al.* [12, Section 1.2] later build on this definition and explicitly define components as a run time entity, thereby further manifesting the run time aspect of software architecture in this research field. Bass *et al.* come to the conclusion that "the software architecture of a system is the set of structures needed to reason about the system, which comprise software elements, relations among them, and properties of both" [12, Section 1.1], a definition they refined over the period of 15 years [12]–[14]. A single structure in their definition is a set of elements that is held together by a relation. They reason that no single structure can completely describe the architecture of a system and therefore define it as a set of structures. These are classified into three different types:

- Module structures
- Component-and-connector structures
- Allocation structures

The first type deals with the partitioning of systems into implementation units, thereby dealing solely with static representations of the software. Contrary to these modules, components and connectors consider the run time aspect of the software. A single module, like the client in a Client/Server (CS)-system, could result in 10 components at run time. For this reason, it is important to recognize the possible relationships between modules and components. A single module can be used in several components at run time, whereas a component is usually composed of multiple modules. Fielding also identified the need for this distinction [1, Section 1.1]. The third type, allocation structures, deals with the mapping of a system's component to non-software structures like hardware, such as Central Processing Units (CPUs).

Through this definition, they extend the scope of software architecture beyond the run time aspect. Considerations like the decomposition of the source code into modules are also referred to as structures and therefore part of the software architecture. This is contrary to previous definitions by Fielding or Perry and Wolf which only considered the system at run time to define the software architecture. Interestingly, Bass *et al.* still reason that every system has an architecture, despite the lack of documentation and source code or people who know about it. They argue that every kind of diagram or visualization is only a representation [12, Section 1.1]. However, if a system's software architecture is present despite the lack of source code, how can module structures be considered architectural elements?

Another interesting, though less formal, view of software architecture is presented by Fowler. He identifies software architecture as "the things that are hard to change" [15]. Klusener *et al.* later propose a similar definition of software architecture: "those aspects that are the hardest to change" [16]. Fowler reasons that, in contrast to building architecture, software architecture is not limited by physics. This allows to design every element of a software system to be easily (ex-)changed – something that is not trivial to achieve in building architecture. However, as complexity increases for every changeable element, accounting for all possible changes is not

feasible [15]. The primary purpose of software architecture is to ensure that the system's quality and behavioral requirements are met in order to satisfy business goals [11, Prologue 1]. As a result, software architecture should encompass all necessary details but at the same time as little as possible. This corresponds with the definition by Bass *et al.* as they only include those structures that are needed in order to reason about a system. Still, their choice of structures for doing that is debatable, as argued above.

In summary, the term software architecture can be manifested into abstractions of run time aspects of a software system. Abstractions serve the need for hiding complexity and detail for the purpose of simplification. However, defining abstractions is always a trade-off between details that allow fine-grained control and generalizations that simplify things. The designer has to choose which details should be hidden and therefore are easier to change later on. However, changing the abstraction itself is hard and costly. This is the rationale behind the definitions of Fowler and Klusener *et al.* As architecture consists of abstractions, it allows certain parts (the details) to be easily changed. Nevertheless changing the abstractions themselves is hard because they are usually layered, building on top of each other.

1.2 Architecture and Design

Clements *et al.* define the difference between architecture and design as "architecture is design, but not all design is architectural" [11, Prologue 1.2]. The design of something is the result of designing it, a process that is usually driven by an intent for the result to meet a certain requirement. Software architecture can therefore also be seen as a result of a design process. The important difference though is that architecture only deals with those aspects that are crucial for the system to meet its requirements [11, Prologue 1]. For example, a requirement demanding data to be replicated will result in design decisions around data persistence to be of architectural significance. Contrary, if the requirements do not place any restrictions on the persistence of data but just want it to happen, the architecture is unconcerned by the actual implementation and thus, the design process and its decisions are not of architectural relevance. As a result, the level of detail is not an appropriate indicator for architectural design decisions. The example illustrates that it depends on the requirements, whether or not details are of architectural relevance. Still, it is often the case that non-architectural and detailed design overlap. As architecture is about abstractions, design decisions hidden by those abstractions are often non-architectural.

1.3 Patterns and Styles

Both terms, patterns and styles describe software architecture on the meta-level: i.e. they do not represent software architectures themselves. The term architectural style has first been used by Perry and Wolf. They borrow it from the field of building architecture in order to name a set of constraints on design elements and the relations between those elements [8]. Before being explicitly named, Shaw already identified recurring abstractions and concepts in software [17]. Fielding's definition [1, Section 1.5] of an architectural style is heavily influenced by Perry and Wolf. Yet, he stresses that the set of constraints needs to be coordinated because the impact of applying a style is not only defined through its individual constraints but also by their combination [1, Section 2.2]. Influenced by the work of Gamma *et al.* [18] in the field of Object-Oriented Programming (OOP), Buschmann [19] defined patterns for software architecture.

Although the terms architectural styles and patterns are typically treated differently [1] [11] [20], some researchers consider them as synonyms [14]. Taylor *et al.* [20] define architectural patterns as more concrete than architectural styles as they are usually applied to a concrete problem than just in a specific context. In their examples, a problem would be the separation between business logic and data management, whereas a context would be "distributed systems". Clements *et al.* further clarify this distinction by defining architectural styles as "the term for a package of design decisions that explains a generic design approach for a software system" [11]. In contrast, an architecture pattern is the triple of a problem in a specific context with an associated approach as the solution. Similarly, Bass *et al.* later define architectural patterns as "packaged strategies for solving some of the problems facing a system" [12]. The term pattern is thereby further manifested as a solution for a concrete problem, whereas a style is more abstract.

1.3.1 Implications of applying a style to a system

In order to clearly differentiate between software architectures and their styles, the following section presents an example describing the relations between constraints, the induced properties and the rationale behind them.

Given the CS-style, we are faced with the constraint of having to separate our system into two components with different responsibilities. A single server usually communicates with several clients, whereas those only communicate with the server. One of the induced properties of this style is simplicity as the server is not concerned with the User Interface (UI) its clients present to the user. This allows the UI to be changed easily, without affecting the server. However, the server also acts as the central point of communication, hence representing a single point of failure. In addition, the server component is worthwhile to be compromised as it is concerned with storing the data of the system, a property that makes it interesting for attackers. Depending

on the application domain, these drawbacks may be unacceptable. For example, in the context of crypto currencies like bitcoin [21], decentralization is one of the main requirements. Having a central server that controls payments would defeat one of the initial ideas behind bitcoin as a payment system that cannot be comprised by gaining control over the central servers. Bitcoin follows the peer-to-peer style and uses a majority voting technique for determining “the truth”. For that reason, one would have to take over or supply more than half of all involved nodes of a bitcoin network.

The drawbacks and benefits illustrated in the example can be attributed back to the CS-style and are independent of the actual technology that is used to implement the system. Architectural styles allow reasoning about benefits and drawbacks of their application without having to actually implement them. Implementing the system would allow observation but is usually too costly.

The architecture of a software system on the other hand is the result of applying an architectural style. It is important to note that architectures do not necessarily have to conform to the initially applied style. They can degrade over time through changes in the software. Of course, the properties a style promises to induce in a system are only induced as long as the architecture conforms to the style.

In summary, architectural styles can be described as the meta-level of software architecture, allowing the discussion of architectures without actually implementing them.

1.4 Documentation

The need for a documentation of software architecture and its necessary diversity was already identified by Parnas [4]. Perry and Wolf later define “views” that allow to describe different perspectives on the software architecture of a system [8]. On top of that, Kruchten defines the “4+1” view of software architecture, a fundamental aspect of the Rational Unified Process [22]. Clements *et al.* elaborate in detail on those efforts and documentation of software architecture in general [11].

2 Material and Methods

The following chapter introduces terminology, concepts, and technologies that are used in the remaining thesis. It elaborates on the definition and scope of the term API, explains the basics of GraphQL and similar technologies and discusses differences between REST and HTTP.

2.1 APIs in Software Systems

The term API is used in different contexts in the field of software engineering. In general, it refers to the interface of a software element that can be called or executed. These elements appear in different levels of abstraction in a software system. Relevant in the context of this thesis are only those elements that appear on the highest level: components and connectors. In order to further narrow down the scope, we constrain the number of interesting systems to those that are only consumed by machines rather than by humans. An API in the context of this thesis is therefore a component that offers certain functionality and data to other components via a defined interface that allows them to perform various tasks but does not offer a UI that is directly presented to the user.

2.2 APIs and the Web

The main difference between an API and a regular website is the added layer of abstraction between the UI that is presented to the user and the communication with the server. This additional indirection introduces a few problems and discussion topics that are not present in regular Web development. Most importantly, we now need to differentiate between actions the user performs on the UI and the interactions between the client and server. For this discussion, we introduce the concept of *operations*. An operation is the smallest unit-of-work a client component can perform or request from a server component. Invoking an operation leads to an *interaction* between two components. However, several operations can be potentially invoked in a single interaction. The name operation is therefore not to be confused with a procedure call like in the Remote Procedure Call (RPC) style.

An operation in the context of a REST API for example can be the invocation of the `GET` method of a resource. However, for HTTP APIs in general, the granularity is not necessarily defined by the HTTP verb. Systems using HTTP solely for tunnelling purposes might define an interface that allows to call multiple operations in a single HTTP request, therefore supporting bulk operations. In summary, invoking an operation always results in an interaction between two components but several operations can be grouped into a single interaction. Interactions only describe the necessary communication in distributed systems between components if they want to collaborate in achieving a certain goal.

It is important to consider that the possibility of invoking multiple operations within a single interaction is completely dependent on the design of the system. For example, when working with HTTP, an application-level protocol, designers usually go for a one-to-one mapping between operations and interactions as HTTP is rich enough to describe the semantics of an operation on the protocol level. GraphQL on the other hand allows to execute multiple mutations in a single interaction.

2.3 GraphQL

GraphQL is a technology initially [23] created at Facebook in 2012. A specification draft [24] was open sourced in 2015. It is described as “a query language for your API” [25] and was built in response to a performance problem during Facebook’s shift to native mobile apps [23]. Basically, it is technology for building APIs and is intended as an alternative to REST. Despite its name, it does not only allow to query data from the server but also modify it. The following section explains how GraphQL works.

2.3.1 Schema

Central to GraphQL is a schema. It describes available types and their relations, as well as the entry points for the clients; queries and mutations. Code 2.1 shows an exemplary GraphQL schema. At its root, it defines the *schema* element that is further divided into *query* and *mutation*. A concrete type like *User* has a number of typed fields. An exclamation mark indicates non-nullability, i.e. the field cannot be null. Square brackets around the type definition indicate an array of the enclosed type. Parameters as in queries and mutations are named and can therefore be specified in any order.

Code 2.1: A GraphQL schema definition, illustrating the definition of types and their relations.

```
schema {
  query: Query
  mutation: Mutation
}

type User {
  id: ID!
  nickName: String
  posts: [Post]!
  followees: [User]!
  followers: [User]!
}

type Post {
  id: ID!
  author: User
  content: String
  # The ISO representation of the date when the post was created.
  createdAt: String
  replies: [Post]!
  replyTo: Post
}

type Query {
  # Retrieve the timeline of a user identified by their nickname.
  timeline(of: String): [Post]!
  user(nickName: String): User
  users: [User]!
}

type Mutation {
  writePost(authorNick: String, content: String, replyTo: String = null): Post
  newUser(nickName: String): User
  followUser(me: String, other: String): User
}
```

2.3.2 Queries

In order to retrieve data from a GraphQL service, a query is sent to it. Code 2.2 shows such a query, whereas Code 2.3 illustrates an exemplary result. It retrieves the content and the author's nickname of all posts on the timeline of the user *MaxMustermann*. Note how the query starts with one of the fields of the *Query* type: *timeline*. In the schema, the query *timeline* is defined to take a single parameter named *of* and return an array of *Post*. Code 2.2 also illustrates the use of fragments, reusing the field-set used for describing the desired fields of each timeline

post to also fetch those fields for all replies of each post. Applying a fragment is similar to the Spread-operator in ECMAScript 2015 [26]. When issuing a query, GraphQL requires all requested fields of the query to be primitive types in order to improve predictability [23].

Code 2.2: A GraphQL query against the schema defined in Code 2.1.

```
query {
  timeline(of: "MaxMustermann") {
    ...basicPostFields
    replies {
      ...basicPostFields
    }
  }
}

fragment basicPostFields on Post {
  content
  author {
    nickName
  }
}
```

Code 2.3: Exemplary result of the query illustrated in Code 2.2.

```
{
  "data": {
    "timeline": [
      {
        "content": "My first blogpost!",
        "author": {
          "nickName": "JohnDoe"
        },
        "replies": [
          {
            "content": "Super cool!",
            "author": {
              "nickName": "MaxMustermann"
            }
          }
        ]
      }
    ]
  }
}
```

2.3.3 Mutations

Mutations are similar to queries, except that they are allowed to cause side-effects, although this is not enforced by GraphQL [25] but is only a convention. Mutations are therefore used to change data on the server. Code 2.4 shows an exemplary mutation. Mutations return values, just as queries do. Hence, a client can query data based on the return value of a mutation.

Code 2.4: A GraphQL mutation against the schema defined in Code 2.1, adding a new blog post as the user *JohnDoe*.

```
mutation {
  writePost(authorNick: "JohnDoe", content: "Blogging all day long!") {
    author {
      posts {
        content
      }
    }
  }
}
```

Code 2.5: Result of the mutation illustrated in Code 2.4.

```
{
  "data": {
    "writePost": {
      "author": {
        "posts": [
          {
            "content": "My first blogpost!"
          },
          {
            "content": "Blogging all day long!"
          }
        ]
      }
    }
  }
}
```

2.3.4 Transport protocol

GraphQL itself is only a specification for a “server-side run time for executing queries” [25]. By that, it is unconcerned with the transport protocol that is used between client and server for transmitting these queries and mutations. As GraphQL encodes all necessary information in

the payload, it also works over transport protocols like Transmission Control Protocol (TCP) or User Datagram Protocol (UDP). When used in conjunction with HTTP, a single endpoint is typically provided that accepts `POST` requests by the client, transporting the payload in the HTTP body.

2.3.5 Similar technologies

GraphQL is not the first technology of its kind. The following section describes four technologies that make an effort in solving the same or similar problems.

Feed Item Query Language

Feed Item Query Language (FIQL) "is a simple but flexible, URI-friendly syntax for expressing filters across the entries in a syndicated feed" [27]. It is an effort by Nottingham in standardizing filtering and sorting requirements in the context of HTTP resources. However, it has no support for joining together several resources in a single request, like GraphQL does by traversing the relations of entities.

OData

OData [28] is a standard from the Organization for the Advancement of Structured Information Standards (OASIS). It features a very rich interface on top of URIs for performing tasks like filtering, searching, ordering, or expanding [28]. The *expand* feature is similar to what GraphQL allows the client to do by including related resources in the representation of the requested one. An OData request that retrieves the same data as the query in Code 2.2 is shown in Code 2.6. OData differs from GraphQL in the way that it describes the desired subset of the requested resource through special query parameters.

Code 2.6: A `GET` request, fetching the same data from an OData service as the GraphQL query in Code 2.2 by using a combination of *$expand* and *$select*. $select constrains the returned properties for an entity to the listed ones and $expand includes a related entity.

```
GET serviceRoot/Timeline('MaxMustermann')?$select=id,content&$expand=Author($select=
    nickName),Replies($select=id,content;$expand=Author($select=nickName))
```

Falcor

Falcor [29] is a JavaScript library developed by Netflix that promises efficient data fetching. In contrast to GraphQL it is not a specification but just a library. Falcor makes an effort to combine the small message sizes and low latency of RPC with the cache consistency and loose coupling of REST [30]. It does that by providing access to a model as a single resource on the server. A Falcor client can then access arbitrary data from that model as it would access a local JavaScript object. The client library of Falcor takes care of fetching the necessary sub-graphs of the model from the server if they are not already present on the client and stores them in a normalized way.

JPA Named Entity Graphs

The Java Specification Request (JSR) 338 defines a concept that is called Named Entity Graphs [31, Section 3.7] which achieves similar goals as GraphQL but operates on the database level. It is not intended to be used for communication between components but primarily acts as a performance optimization for components that use the Java Persistence API (JPA) for handling data persistence. Entity graphs allow to name a set of attributes which can later be used to fetch exactly this set of attributes from the database when retrieving a single entity or a list of entities, independently of the fetch strategy (*EAGER* or *LAZY*) that is specified for the individual attribute. Entity graphs allow JPA to build efficient database queries and avoid running into problems with lazy loading after the session has already been closed.

2.3.6 Operations in GraphQL

The concept of an operation maps to queries and mutations in GraphQL. The only differences between those two is the convention that queries should not cause side-effects [25]. GraphQL allows multiple queries or mutations to be executed in a single interactions, thus each individual query or mutation is a single operation.

2.4 REST

REST is defined as an architectural style [1, Chapter 5]. It is the formalized version of Fielding's model created by him in response to his work on the HTTP standard and several other ones [2]. However, HTTP does not implement all constraints defined by REST. One constraint the application programmer has to take care of is hypermedia. It is one of the four sub-constraints of the uniform interface constraint and is referred to as: "Hypermedia as the engine of application

state" [1, Section 5.1.5]. The uniform interface constraint in general and especially the hypermedia constraint discriminate REST from other styles [1] [32]. Understanding and implementing the hypermedia constraint is therefore a critical part in the process of building a REST API.

A common misconception is that simply the use of HTTP qualifies an API as RESTful. As a result, many services exist that label themselves "RESTful" although they do not account for all constraints [33]. Richardson introduced a model [34] to discriminate web services that make use of HTTP. It classifies a service into one of four levels according to its maturity in terms of the REST style. Level 0 indicates URI tunnelling, level 1 adds dedicated resources, level 2 uses HTTP verbs for describing actions and level 3 finally adds hypermedia [34].

In order to differentiate more clearly between the varying understandings of REST, subsection 2.4.1 describes the CRUD over HTTP pattern and subsection 2.4.2 explains the concept of hypermedia.

2.4.1 CRUD over HTTP

CRUD describe four primitive types of interaction with a data entity. The CRUD over HTTP pattern therefore refers to services that expose these operations on a set of data entities over HTTP. The mapping between these four operations and the corresponding HTTP verb is illustrated in Table 1. In addition, such services often directly map resources to data elements of the underlying storage mechanism.

Webber *et al.* [35] elaborate in detail on how to build such a service. They reason that while CRUD services are useful to extend the reach of a system that primarily manipulates records, more complex scenarios and workflows require richer interaction models than CRUD.

Table 1: Mapping between CRUD interactions and the HTTP verbs.

Interaction	**HTTP verb**
CREATE	POST
READ	GET
UPDATE	PUT
DELETE	DELETE

2.4.2 Hypermedia

The term hypermedia originates from the term hypertext which was originally defined by Nelson [36] and can be described as text that refers to other text. In REST, hy-

permedia is described as control data that is embedded in the representations of resources. The representation of a resource and – as a result – that of the hypermedia controls embedded in them, is determined by the media type. Examples for media types that define hypermedia elements are `application/html` [37], `application/atom+xml` [38], `application/vnd.uber+xml|json` [39], `application/vnd.siren+json` [40] or `application/hal+json` [41]. Media types like `application/json` [42] or `application/xml` [43] are only serialization formats and do not specify any hypermedia elements. However, they are often used as building blocks for media types with hypermedia elements by defining semantics of certain attributes. An example for that are the two attributes `_links` and `_embedded` of the `application/hal+json` media type [41] that is based on the JavaScript Object Notation (JSON) serialization format. Code 4.4 on Page 34 is an example for a representation that is heavily influenced by the `application/hal+json` media type by grouping links under the `_links` property and embedded resources under `_embedded`.

Hypermedia elements typically contain a relation attribute that specifies their semantics. For example, in `application/hal+json` these are the object-keys of the `_links` and `_embedded` properties. The concept of relations in general is defined as Web Linking in [44], which also defines a link relation registry that is currently managed by the Internet Assigned Numbers Authority (IANA) which hosts officially registered link relations and their documentation. If none of the official link relations are appropriate for a given use case, Extension Relation Types [44, Section 4.2] can be used. They appear in the form of URIs that may point to an actually accessible resource that further describes the semantics of the link relation, however this is not a requirement. Code 2.7 illustrates the use of such an Extension Relation Type.

Code 2.7: A resource-representation that contains an Extension Link Relation.

```
{
  "_links": {
    "http://api.example.org/rels/reply": {
      "href": "..."
    }
  }
}
```

The link relation attribute of a hypermedia element describes the semantics of the relation between the linked resources and thereby also defines the possible interactions with the linked resource. For example, the link relation `next` is defined as “refers to the next resource in a ordered series of resources” [44]. A paginated resource that displays a number of search results may, for example, use this relation to point to the next page of search results.

2.4.3 Operations in REST

The central concept concerning interface design in REST is a resource [1, Section 5.2.1.1]. It is an abstraction around a certain piece of information. To understand the concepts of operations in REST, it is important to differentiate between the concept of resources, their representations and actions.

A resource is a named concept that can have any number of representations for a particular point in time, which means, the representation of a resource can vary. This variation cannot only come in the form of a different encoding, like the use of Extensible Markup Language (XML) instead of JSON but also actually different information. What must not change is the semantic. For example, a resource that describes a person's best friends may refer to different persons depending on the point in time. The information encoded in the representation of the resource changed, yet it represents the list of the person's best friends, leaving the semantics of the resource untouched. In addition, the available actions can vary as well. An order that has already been placed may no longer support the addition of further items, thus denying a particular action, although it was allowed at a previous point in time [1].

An operation as defined in section 2.2 maps to a composition of three aspects in REST: the resource that is addressed, the action that should be performed on the resource and the metadata that is included in the request. Exactly this triple uniquely describes the desired operation. In HTTP[1] for example, the action could be one of `GET`, `POST`, `PUT`, or `DELETE`, the resource is addressed by an URI and the metadata comes in the form of HTTP headers.

2.5 Comparison Approaches

Comparing GraphQL and REST is not trivial. While the former is a specification for a technology, the latter is an architectural style and thus a lot less restrictive in terms of the implementation. A specification directly and ideally completely describes the desired behavior of a system. An architectural style on the other hand only defines constraints. The resulting architecture can be designed freely within these constraints. Thus we are faced with the comparison of two concepts that are defined on different levels of abstraction. Basically, there are three possibilities of how this comparison can be conducted: Specialization of the one, generalization of the other or a comparison of them as they are.

The following section presents each one of these approaches and concludes with reasoning on the eventually chosen approach.

[1] A comprehensive list of the available methods for HTTP is described in [45].

2.5.1 Approach 1: Specializing REST

The first approach deals with the specialization of REST in order to compare it to GraphQL. Specializing REST has two implications. First, concrete technologies have to be chosen that allow to build a RESTful system. HTTP is a good start here, as it already implements most constraints of the REST-style. What has to be taken care of in addition is the hypermedia-constraint. This is not as trivial because hypermedia is concerned with semantics of the domain and therefore operates on the application-level. Although there are domain-agnostic media types, a choice has to be made after all.

A comparison between two concrete technologies can be conducted by defining two use cases, a set of comparison criteria and implementing each use case for each technology, resulting in a total of four implementations. Therefore, and this is the second implication, for a meaningful comparison, the use cases should be chosen in a way that allows diversity between its implementations, e.g. it is pointless to compare the implementations of two use cases if they only differ in minor details. Appropriately defining these use cases is also not trivial.

Both challenges, the need for choosing a concrete technology for REST and defining meaningful use cases, are the major drawbacks of this approach. It is hard if not impossible to draw conclusions from a concrete technology to its abstract definition. One can never be sure if a concrete advantage, disadvantage or problem is caused by the technology or if it can be attributed back to the architectural style one tried to follow in the implementation of the system.

2.5.2 Approach 2: Generalize GraphQL as an Architectural Style

The second approach deals with the generalization of GraphQL as an architectural style, as an effort to close the gap between the different levels of abstraction in the definition of both concepts. The premise of approach 2 is therefore to find an existing, matching style that describes GraphQL. This in turn requires the identification of its architectural principles. Efforts to fulfill this premise yielded two style candidates: Remote Evaluation (REV) and Remote Data Access (RDA). Upon successful generalization, Fielding's model [1] for comparing architectural style could be applied to compare REST against the identified style.

Architectural principles

Three architectural principles for GraphQL have been identified.

Client-Server is the foundation for a lot of architectural styles dealing with network-based applications [1, Section 3.4.1], including REST, REV, and RDA.

Stateless communication is an important aspect of a communication protocol concerning scalability and recovery from partial failure. GraphQL does not have a concept of server-side sessions and thus features a stateless communication protocol.

Declarative languages have the pleasant property of describing only the desired result instead of the way on how to achieve it. Although GraphQL is basically a Network Communication Language (NCL), it is constrained to be solely declarative.

Remote Evaluation

REV was first defined by Fuggetta *et al.* in [46] as one of four different styles for code mobility. In the REV style, a client has *know-how* that is transferred to a *computational* component that also has access to the necessary *resources* in order to perform a computation [46]. Fielding further generalized these four styles under the Virtual Machine (VM) style because the code, independent of its initial location, has to be interpreted by another component [1]. Language-based communication in general was first defined by Falcone in [47]. He identified increased flexibility through the possibility of components that program each other.

One aspect that is neither considered by Fuggetta *et al.* nor by Fielding is the differentiation between declarative and imperative code. While the latter is hard to reason about by simply looking at it, the former by definition has to be interpreted that way. Thus, there is a difference in the visibility of a communication protocol that uses declarative and one that uses imperative code. An intermediary aware of the grammar should be able to reason about the communication between two components that is based on a declarative language.

While the REV style may seem to be a good fit at first sight, it does not take into account the declarative nature of GraphQL. Thus, conclusions drawn only from the constraints of the REV style are not valid in some cases for an architecture making use of GraphQL; for example, lacking visibility as identified by Fielding [1, Sectin 3.5.2].

Choosing REV as a representative for GraphQL in the comparison to REST does not account for all rationale behind its definition. Such a comparison would therefore be biased and is thus deemed not meaningful.

Remote Data Access

Another approach for classifying GraphQL as an architectural style is RDA. It was coined by Fielding as a deviation of the CS style with the added constraint of using a standardized language for querying data [1, Section 3.4.7]. Fielding mentions Structured Query Language

(SQL) as a concrete example and reasons that this style decreases scalability through server-side session state and simplicity because both components need to understand the same database manipulation language. Considering only the constraints (CS and declarative language), GraphQL seems to directly qualify for this style. However, Fielding's description of this style is directly coupled to databases: "A client sends a database query in a standard format [...]" [1].

Two problems arise when RDA is used to describe GraphQL. First, GraphQL is not tied to any database technology because it operates on the application layer and second, GraphQL uses a stateless communication protocol and therefore does not suffer from the problems of server-side session state. Both aspects basically eliminate the possibility to describe GraphQL as an instance of RDA, at least in the form described by Fielding. Interestingly, the RDA style in general could be generalized to be included in the family of VM styles because of its language nature.

Even if one would interpret the definition of this style more loosely, describing something by the use of a concept that was initially defined differently only leads to confusion. A more loose interpretation of the word database in the sense of "a component that provides access to data", would allow to classify GraphQL as an instance of RDA by simply adding a constraint that enforces a declarative language. However, redefining the term RDA would only lead to confusion and thus does not offer any benefit.

2.5.3 Approach 3: Compare Both Concepts Based on Their Architectural Principles

Approach 1 and 2 try to close the gap between the levels of abstractions concerning the definitions of GraphQL and REST. The third approach describes the effort of comparing those concepts as they are, without specializing the one or generalizing the other. To do this, the approach identifies desired properties of a system offering an API and investigates how REST and GraphQL influence these properties and deal with occurring problems. This model is influenced by Fielding's in that the impact of architectural decisions imposed by the constraints of a style on the resulting system are observed.

While this approach is very similar to Fielding's, its application differs slightly. Fielding compared architectural styles in his dissertation [1] in the context of a network-based hypermedia system; the World Wide Web. In contrast to the Web, APIs typically do not directly provide a UI that is exposed to end users but are rather intended to be consumed by machines.

2.5.4 Approach Evaluation

While approach 1 and 2 both have drawbacks or fail to even allow a meaningful comparison, approach 3 describes a way that allows to compare GraphQL and REST without the need for redefinitions, generalizations, or specializations. The following subsection shortly describes why approach 3 is chosen for the comparison.

Fowler describes architectural decisions as those that are “hard to change” [15]. Being able to reason about architectural styles without actually implementing a system that makes use of this style is therefore very cost-effective and thus desirable. This is the main motivation for comparing approaches on an abstract level, like by the means of properties that are induced by the constraints of a style in a system [1, Section 2.2].

Comparing GraphQL and REST according to approach 1 includes defining two use cases, implementing both of them and later comparing the implementations. However, as argued above, implementing systems is costly and an abstract comparison is desirable because it allows further comparisons to be easily conducted as well. In addition, it is of little value to actually implement the use cases because, as with any design process, the API design has to be done at least partly upfront. Hardly any conclusion that is applicable in a general sense can be drawn from the artifacts of a concrete implementation that could not also have been drawn from the artifacts of the design process.

Approach 2 and 3 describe a way for a comparison on an abstract level, however, identifying an architectural style that matches the architectural properties of GraphQL was not successful and thus, the premise for approach 2 is not fulfilled. As Fielding reasons in [1, Section 2.2], new architectural styles can be derived by adding constraints to existing styles. Therefore, a new architectural style could be formed by formalizing the identified architectural principles as constraints. However, defining a new style would exceed the scope of this thesis, which leaves approach 3 as the only viable way for comparing GraphQL and REST.

2.6 Related Work

Fielding conducts a survey on architectural styles in the context of a hypermedia-based network architecture and examines the properties induced by the constraints of these styles [1, Section 3.1]. He also introduced the architectural style REST, a formalization of the mental model he applied in the design of several standards concerning the Web, like HTTP or URI [1, Section 6.1] [2].

Stenlund and Gustavsson make an attempt to compare GraphQL and REST in [48]. However, they do not take the hypermedia constraint of REST into account in their comparison because according to them, it is not relevant as the majority of self-labeled RESTful web services do not

use it. Independent of that, any conclusions about the REST style are ruled out if it is not taken into account fully, because a style does not only define itself through the individual constraints it is composed of but also of their combination [1]. Considering the maturity model by Richardson [34] , their comparison only accounts for APIs reaching level 2 in the referenced model. They also conduct performance measurements in terms of CPU-load and memory-usage in order to compare their implemented prototypes. Both metrics are implementation dependent and therefore inappropriate for drawing conclusions concerned with the design of APIs.

Pautasso *et al.* [49] as well as Stenlund and Gustavsson [48] make use of a decision model for comparing REST against other approaches. While using such a model may provide a clear and reconstructible decision path at first sight, their practical usefulness is debatable. The problem with such decision models is that, like every model, they are an abstraction. Per definition, abstractions hide details. Choosing which details to hide can only be done while being aware of the actual use case the model will be used in. A general decision model for deciding on a particular technology or approach therefore has to very strictly define the use cases in which it should be applied. Without that, applying a decision model to a new use case always raises the question if the model accounted for all details important to this particular use case or if some of them were left out.

Mesbah and Deursen describe SPIAR, an architectural style for Asynchronous JavaScript and XML (AJAX) applications [50]. Their work is related because GraphQL is commonly used for Single Page Applications (SPAs), the successor of what Mesbah and Deursen describe as “AJAX applications”. They correctly identify missing considerations in the REST-style for building such applications, like delta-communication. However, instead of viewing such systems as compositions of styles, they create a new style that accounts for many aspects that are already described by previous styles. Systems making use of SPAs can be viewed as compositions, for example they may use the Model-View-Controller (MVC)-style for the UI and the REST-style for communication with the API. Just because they are distributed over HTTP and also use HTTP as their communication protocol for exchanging data does not make them different from desktop applications concerning the interaction patterns and therefore the API design.

In [51], Shaw and Clements present a classification of architectural styles based on their constituent parts; components and connectors, control issues, data issues, and the interaction of control and data issues. However, as Fielding reasons in [1, Section 3.8.1], their view of architectural styles does not assist a designer in the choice of an architectural style because their classification is only concerned with technical properties like the shape or topology of the resulting system and not connected to the actual needs of the application. For the same reason, their classification methodology is not applicable to the problem at hand because comparing the topology of GraphQL and REST does not allow to gain knowledge about when to chose which approach for building an API.

Kazman *et al.* introduce the Software Architecture Analysis Method (SAAM) in [52]. They re-

commend the use of scenarios for analyzing the software architecture of an existing system. Their method may have helped with the approach explained in subsection 2.5.1. However, they only focus on analyzing the modifiability and maintainability of a software system. Dobrica and Niemela further compare several architecture analysis methods in [53], including SAAM and three extensions of it.

3 Comparison Criteria

Fielding defined a number of architectural properties relevant to his survey on architectural styles [1, Section 2.3]. The criteria chosen for the comparison of GraphQL and REST consist of a subset of the properties defined by Fielding, extended by further aspects that are induced by the added layer of abstraction between the actual UI that is presented to the user and the API. The following section introduces the criteria used for the comparison.

3.1 Operation Reusability

In the CS-style, a single server provides functionality and data to several, potentially even different types of clients, like mobile apps and desktop applications. The clients invoke operations on the server in order to retrieve or change the stored data. A single operation may therefore be used by different types of clients, however this is only possible if the operations are reusable, i.e. an operation that was created earlier can be used by a new type of client. The reusability of an operation depends on a number of factors. To discuss these, we first need to consider the different types of operations. One way to categorize these is by looking at the side-effects they cause. Data retrieval operations typically do not cause any and are therefore considered safe, i.e. they can be invoked, possibly several times subsequently, without causing change. Unsafe operations on the other hand are allowed to cause side-effects, therefore all operations that modify or delete existing data or add new data to the system are considered unsafe.

Concerning the reusability of an operation, safe operations are reusable for a new type of client if they return just the data that is needed. Although too much data is not a deal-breaker, it is still considered suboptimal. Not enough data is obviously a problem because clients have to invoke additional operations in order to retrieve the data they need, thus lowering the reusability of this operation. Unsafe operations cause side-effects because they change some part of the data that is stored on the server. Such an operation is reusable if it only changes those parts the client intends to change. Contrary to safe operations, unsafe ones that do less than what is intended by the client is less of a problem than operations that change too much data. A client can always orchestrate over multiple, more fine-grained operations in order to achieve a certain goal. However, breaking apart an already existing operation that performs a set of changes is not possible. The reusability of an operation is therefore always related to the goal a specific client wants to achieve.

For a client, it would be ideal if it would not have to reuse an existing operation but rather invoke one that is specifically suited for the task at hand, thereby improving its simplicity [47]. Considering that even different types of clients in the same application domain want to achieve similar goals, providing specialized operations for every client type results in duplication on the server. Designing operations is therefore centered around a trade-off between generalization and specialization. A very general approach to operation design results in less duplication on the server but forces the clients to orchestrate over many individual operations to achieve their goal. Specialized operations on the other hand cause duplication on the server and make the overall interface more complex because the number of available operations increases.

For unsafe operations, generalization results in a high granularity, up to the point where each individual piece of data can be freely modified. This of course would not be feasible because it prohibits the server from performing integrity checks on the modifications requested by the clients. For safe operations, the highest level of generality is achieved by merging all retrieval operations into one, returning all the data that is stored in the system. Of course, this approach is also not applicable in practice for several reasons like performance or access-restrictions.

In a real-world system, all operations are abstractions over the data model used by the server. The aspect of operation reusability deals with the different approaches to the trade-off between generality and specialization in the design of operations and examines, how GraphQL and REST deal with this trade-off.

3.2 Discoverability

Discoverability refers to the degree by which something can be found and thereby discovered. Within the lines of this thesis, discoverability is not to be confused with service discovery concepts of Service-Oriented Architecture (SOA) such as Universal Description, Discovery and Integration (UDDI) [54].

3.2.1 Static and Dynamic Discoverability

An API features a high discoverability if it is self-descriptive and therefore can be discovered on its own, without the need for out-of-band information like external documentation. Type systems offer a kind of discoverability that can be described as static discoverability because it works context-less, i.e. there is no need for a running system in order for other tools or services to make use of it. In contrast to that, dynamic discoverability involves an actually running system and operates context-aware. For example, providing metadata at run time is a way to achieve dynamic discoverability. Static as well as dynamic discoverability mechanisms can be used at development time to support the documentation of a system. For example, static

discoverability mechanisms provide the foundation for auto-generated documentation, ahead-of-time compatibility checks and coding assistance. As dynamic discoverability mechanisms offer insight into running systems, developers can gain viable information from them through exploration.

3.2.2 Levels of Dynamic Discoverability

An important aspect concerning dynamic discoverability is, if and how dependencies between operations can be discovered. The concept of operation dependencies describes the preconditions that have to be met for a single operation to be successful. Some of them can be resolved by calling operations in a particular order. For example, in a system providing operations for placing orders, products may only be added to an order until the order is placed and thereby finalized. Therefore, the operation for adding further products can only be invoked until the operation for placing the order is invoked. After that the operation fails. There is therefore a dependency between those two operations. Apart from that, operations may also depend on other properties such as authorization information, restricting successful invocations to clients providing appropriate credentials. APIs can be grouped into three levels on how they expose the dependencies between operations:

Level zero

Level zero indicates no support for dynamic discoverability, i.e. the API does not provide any information concerning the dependencies between operations. Clients of such an API therefore have to replicate the server side control logic in order to successfully invoke operations in the correct order.

Level one

Dynamic discoverability of level one is achieved if an API provides information on whether a certain operation can be invoked or not. This information can for example be conveyed by a list of operation identifiers. Presence of an identifier in this list indicates that the operation can be invoked (and its absence the opposite). In order to successfully invoke an operation, clients of such an API need to have additional out-of-band information such as parameter names. As already mentioned in section 2.2, the term operation is not to be confused with a procedure as in RPC. Similarly, parameters in this context just refer to any information that is passed to the server along with the operation identifier and do not necessarily refer to procedure or method arguments. In HTTP for example, the body and query parameters can be used to parametrize an operation.

Level two

An API featuring dynamic discoverability level two not only describes if an operation can be invoked but also contains information about how the operation should be invoked. This refers to the parameter names that are to be passed to the operation, how these parameters should be encoded, which of them are optional and so forth. Basically, any kind of information that is needed from a technical point of view to successfully invoke the operation. Clients of a level two API therefore do not need any additional information except from the parameter values themselves. Yet, those are provided at run time by various sources like the application's users. Everything else is provided in-band, i.e. through the communication with the server.

3.3 Component Responsibility

The responsibilities of components can be described in terms of the allocation of functionality in a system. Functionality can be described solely from a technical point of view, such as the persistence of data or the user interface. The distribution of functionality induces responsibilities in the components in terms of the business rules and requirements that should be implemented in the system.

Through the added layer of abstraction between a client and a server component in an API scenario, the server is completely unconcerned with the UI that is presented to the user. In the context of a website, the server at least provides the declarative code that is used by the client to render the UI. Even in the context of SPAs, the server that provides the API does not necessarily need to serve the SPA's code. In summary, the aspect of component responsibility discusses the allocation of responsibilities that is induced by either using GraphQL or REST for the design of an API.

3.4 Simplicity

Fielding identifies the principle of separation of concerns as the primary means for inducing simplicity [1, Section 2.3.3]. He refers to the simplicity of the resulting architecture. In software development, simplicity is a very important concern because anything that is too complex to understand cannot be changed or used. From the perspective of the end-user of the system, improving simplicity is also known as improving the usability of a system. However, this does not only apply to the end-users of a system but also to its developers. For them, the variety and richness of their tools also impacts how well they can work with a certain technology. The aspect of simplicity could be rephrased as "usability for developers", however the term usability

is strongly related to end-users in software engineering. Therefore, the discussion focuses on the simplicity of designing and building an API using either GraphQL or REST.

3.5 Performance

User-perceived performance is the “impact on the user in front of an application” [1]. In the basic form of the Web, that is, without JavaScript, every user action leads to a network request being issued. In general, the user-perceived performance of such a system is primarily driven by the network performance, the level of efficiency concerning the usage of the network and the server’s processing time for each operation. In contrast to that, users interacting with a client of an API do not necessarily generate network requests through their actions; certain actions may as well be performed locally, without any interaction with the server. Although this has an influence on the user-perceived performance, the only concern the client can address, if it has to use the network, is efficiency. The network performance itself and the processing time for the operations are not within the client’s influence.

Hence, only network efficiency is discussed in the comparison between GraphQL and REST. Efficient use of the network can be achieved through a variety of ways. In general, it is about avoiding unnecessity, for example in terms of requests or payload content. There are several strategies that address these unnecessities like limitation of the payload size, caching of previous responses, or customization of responses. Mesbah and Deursen [50] identified the need for “delta-communication” between client and server in order to improve network efficiency.

Performance is a concern of every software system. What varies between systems is the level at which the performance is unacceptable and renders an application unusable. In the context of APIs, two often observed performance problems are discussed: Over-fetching and multiple round-trips. In addition, options for caching in each approach are included in the comparison.

3.5.1 Over-fetching

Over-fetching describes the situation in which a client receives more data than it needs, i.e. unnecessary payload content. The problem can originate from an overly general interface that tries to satisfy too many needs at once, thus returning too much data for a specific use case. Over-fetching is always context-related because it is something the client experiences. Consider an operation $\mathtt{O}$ that fetches information related to an entity $\mathtt{E}$ and a client $\mathtt{C}_1$ that uses this operation to fetch the attributes $\mathtt{A}_1$ and $\mathtt{A}_2$ of entity $\mathtt{E}$. Client $\mathtt{C}_2$ also needs to fetch attribute $\mathtt{A}_1$ but does not need $\mathtt{A}_2$. If the client $\mathtt{C}_2$ reuses operation $\mathtt{O}$ it experiences the problem of over-fetching because it is provided with the value of attribute $\mathtt{A}_2$ although it does not need it. In

general, over-fetching can only occur if the response of an operation is defined by the server and not by the client.

In addition to over-fetching of individual attributes, the problem can also occur when fetching collections of resources although a particular client is only interesting in one specific element.

3.5.2 Multiple Round-Trips

The need for multiple round-trips occurs if a client is not able to fetch the data it needs for a specific use case in one interaction, i.e. generating unnecessary requests. Multiple, sequentially executed interactions increase the user-perceived latency [50] of the system. The problem of multiple round-trips occurs due to the added layer of abstraction between the UI and the API. A single user action on the UI may result in multiple operations being invoked in the API. If the API does not support the composition of several operations in a single interaction, multiple round-trips are needed in order to invoke all operations.

The problem also occurs if the client needs to fetch data from the server it can not yet address because of missing identifiers (e.g. URIs). Browsing through the API until the desired identifier is found also leads to multiple rounds trips until the desired information can be retrieved.

3.5.3 Caching

The concept of caching is a double-edged sword. On the one hand, it can provide a big boost in improving the efficiency of an application if responses of previous requests can be used later on. On the other hand, these responses consume resources on the client in the form of storage capacity and they can be become stale, no longer representing the current version of the entity. In order to avoid stale data, caches need to be invalidated or refreshed. Unfortunately, appropriately invalidating caches is considered one of the hardest problems in computer science[1].

3.6 Interaction Visibility

Interactions between two components that feature a high visibility can be monitored or mediated by intermediaries [1]. In order for interactions to be visible, they need to be understood by those intermediaries. A stateless communication protocol greatly increases visibility as each interaction can be understood on its own and an intermediary does not need to keep track of previous interactions. Fielding identified several benefits that result from visibility: “improved

[1] An often cited quote that is attributed to Phil Karlton says: “There are only two hard things in Computer Science: cache invalidation and naming things.”

performance via shared caching of interactions, scalability through layered services, reliability through reflective monitoring, and security by allowing the interactions to be inspected by mediators (e.g., network firewalls)" [1].

3.7 Customizability

REST and partly GraphQL offer a way to temporarily specialize another component. This feature is known as customizability [46]. Temporary specialization means that an architectural element, for example a component or a connector, is extended at run time in a way that it can perform a service it was not able to do before.

4 Comparison

The following chapter contains the comparison between GraphQL and REST according to the criteria defined in the previous chapter.

4.1 Operation Reusability

Through its language-nature, GraphQL offers completely different ways for modeling operations compared to REST. The following section elaborates on these differences and their implications.

4.1.1 GraphQL

As already described in subsection 2.5.2, GraphQL is related to NCLs. Due to its language nature, it allows the client to be as specific as necessary about the data that should be retrieved. The server on the other hand only provides a very minimal and general interface that can be reused by many clients. Basically, GraphQL allows arbitrary traversal of the object graph that is returned from either a query or a mutation. Although this introduces great flexibility concerning the actually returned data, the client is limited to the entry points that are provided by the schema.

Mutations are allowed to cause side-effects and thus are GraphQL's primary mechanism to provide functionality that changes data. However, contrary to the freedom the client has concerning the returned data, the side-effects that are caused by a specific mutation are defined by the server and cannot be controlled by the client.

Concerning the identified trade-off between generalization and specialization, the language nature of GraphQL eliminates this at least for data retrieval use cases because query operations are dynamically composed by the client and interpreted by the server. What is limited in GraphQL are the entry points for the client in the form of the available queries and mutations. The flexibility in the definition of the exactly returned data allows clients to tailor it for their specific needs, thereby achieving highly reusable data retrieval operations.

4.1.2 REST

In REST, all interactions between client and server are modelled as messages. This is defined by the “manipulation of resources through representations” constraint that requires implementations of REST to send a representation along with an action to the server to indicate a desired state change [1, Section 5.1.5]. The shape and structure of the representation is furthermore defined by the media type that is used by the client to represent the information. The reusability of an operation therefore completely depends on the used media type.

This does not only apply to state changes, but also for data retrieval operations. In HTTP for example, the client can define a list of acceptable media types for a specific resource in the `Accept` header [45]. That means, in theory, a client could request a representation of the resource that perfectly fits its use case. However, for this to work the server also has to implement the semantics of the media type, otherwise it will not be able to produce the desired representation. For example, in a system with two types of clients, a mobile app and a desktop client, two representations of the same resource could be made available by the server: `application/vnd.timeline.mobile+json` and `application/vnd.timeline.desktop+json`. Each client requests a representation that is tailored for its use case. For example, the mobile variant could return less data in order to save network bandwidth. However, as an operation is a triple of resource, action, and metadata, we just created a new client-specific operation in order to meet the needs of each client. To achieve reusability, a media type must therefore support parametrization, for example, through query parameters.

Concerning the trade-off about reusability between generality and specialization, REST does not offer a solution per se. The actual representations of the resources and its possible parameterizations depend on the used media types and have to be defined ahead of time. Even if a media type supports parametrization, the scope and variety of these parameters also have to be defined upfront. A media type designed with generality in mind might return all sorts of information about a resource, unaware about the individual pieces of data the clients are actually looking for in their particular use case, leading to unnecessary data transfer. On the other hand, a media type offering representations specialized to the use cases of individual clients might satisfy this particular group of clients but may be unusable for others. This can lead to even more network requests, as required data has to be fetched from other resources as well. Supporting several specialized media types for different groups of clients increases the complexity of the server component as it has to implement support for each media type.

In summary, the reusability aspect of operations depends on the used media type and is consequently determined by their designer. In general, it is possible to achieve a certain degree of reusability through parametrization of individual operations. However, this increases complexity of the server.

4.2 Discoverability

GraphQL and REST differ in the way that GraphQL offers static and REST dynamic discoverability. The following sections describe advantages and disadvantages of both approaches.

4.2.1 GraphQL

One of GraphQL's key features is its static, reflective discoverability. It allows a client to reflectively inspect a running GraphQL server and gain information about the provided schema and the available queries and operations. This is achieved through the implemented type system that is exposed at a single entry point. A client able to understand GraphQL that is directed to such an entry point can discover further actions on its own without the need for out-of-band information. Although this introspection uses a running server, it is still considered static discoverability because it only uses the type system and the underlying schema.

Type systems allow the generation of documentation, a feature often welcomed by developers. GraphQL's reflective capability is, for example, used by GraphiQL [55] in order to provide features such as auto-completion. It is important to note that reflective features like that are always context-less, thus providing only information about what is generally available. For this reason, GraphQL's discoverability is referred to as static; it does not change unless the schema is changed.

As GraphQL's discoverability is static, it is considered a level zero approach concerning operation dependencies because it does not provide any information about them. Client developers therefore have to at least partially replicate the server side control logic in order to determine these dependencies.

An example for how this can be accomplished is illustrated in Code 4.1. The example embeds domain-specific metadata in the GraphQL schema. It defines a *state* field in the *Order* type that models the current state of the order. Clients can use this field to implement control logic that determines the availability of operations. For example, *placeOrder* can only be called on orders in the *OPEN* state. Although the server is now returning metadata concerning an order, clients still cannot discover if an operation is available or not because there is no direct coupling between the state of an order (e.g. *OPEN*) and the available operations (e.g. *placeOrder*).

Including a state field is only one of many possible solutions for adding enough information for the client to mimic the control flow. Independent of the chosen solution, GraphQL does not provide native support for representing dependencies between operations. This also causes an additional development effort if these dependencies are not static but rather dynamic, e.g. they change at some point. In that case, existing control logic that determines the dependencies has to be adapted.

Code 4.1: A GraphQL schema, modeling types and mutations for an exemplary ordering process.

```
schema {
  query: Query
  mutation: Mutation
}

type Order {
  id: ID!
  products: [Product]!
  state: OrderState!
}

enum OrderState {
  OPEN
  PLACED
  CANCELLED
}

type Product {
  name: String!
}

type Query {
  order(id: ID): Order
}

type Mutation {
  placeOrder(id: ID): Order
  cancelOrder(id: ID): Order
  addProductToOrder(orderId: ID, productName: String): Order
}
```

4.2.2 REST

REST explicitly requires data in responses to be annotated with metadata [1, Section 5.2.1.2]. This metadata comes in different forms where one is called control data. It describes actions the client should or has to follow, depending on the specification. In HTTP for example, caching is implemented through control data sent along with the actual representation of a resource, describing the representation's cache semantics. Code 4.2 illustrates a HTTP-response with caching control data: *max-age* [45].

Concerning discoverability, a more interesting type of control data are hypermedia controls. The "hypermedia as the engine of application state" constraint requires an application following the REST-style to only achieve state changes through the use of hypermedia [1, Section 5.2]. Hypermedia controls are a form of metadata and give REST its dynamic discoverability. As a

Code 4.2: An exemplary HTTP response, containing the “Cache-Control” HTTP header with a *max-age* directive, indicating a maximum age of 3600 seconds for the returned resource.

```
HTTP/1.1 200 OK
Date: Sun, 28 Mar 1993 13:37:00 GMT
Cache-Control: max-age=3600
Content-Length: 2
Content-Type: application/json

{}
```

result, actual data needs to be present in the service for this discoverability to provide value. In contrast, GraphQL's static discoverability works purely on the schema and is not coupled to the stored data.

The coupling between hypermedia and the data that is stored in the system is illustrated by the means of the already introduced ordering process and the identified operation dependencies. Adding more products to an already placed order does not make any sense. Similarly, invoking the operation for canceling an order is only meaningful for an already placed order. Code 4.3 illustrates the situation of an empty, not-yet placed order. The server indicates that further products can be added to the order by rendering the link with the *add-product* relation. As soon as a product is added to the order, a link for placing the order is included (Code 4.4). Code 4.5 illustrates the order after it was placed. In the new context, adding further products and placing the order is no longer meaningful. Instead, we are now offered with the possibility of *cancel*-ing the order. Also note how the link for removing a product from the order disappeared.

Code 4.3: An exemplary representation of an order resource representing an empty order along with a link for adding products to this order.

```
{
  "_embedded": {
    "products": [ ]
  },
  "_links": {
    "add-product": {
      "href": "..."
    }
  }
}
```

The referenced examples illustrate responses of an API with discoverability level one because they do not contain all the information that is necessary in order to successfully invoke the operations. They only contain the identifier (*add-product*) and information about the operation itself (the URI in the *href* attribute). As operations are defined by the triple of action, URI and

Code 4.4: An exemplary representation of an order resource containing a list of products and links for adding further products and placing the order.

```
{
  "_embedded": {
    "products": [
      {
        "productName": "Sunglasses",
        "quantity": 1,
        "_links": {
          "remove": {
            "href": "..."
          }
        }
      }
    ]
  },
  "_links": {
    "add-product": {
      "href": "..."
    },
    "place-order": {
      "href": "..."
    }
  }
}
```

Code 4.5: An exemplary representation of a resource showing an already placed order containing a list of the ordered products and a link for canceling the order.

```
{
  "_embedded": {
    "products": [
      {
        "productName": "Sunglasses",
        "quantity": 1,
        "_links": { }
      }
    ]
  },
  "_links": {
    "cancel": {
      "href": "..."
    }
  }
}
```

metadata, the missing parts have to be provided through out-of-band mechanisms to the client. Hypermedia controls are part of the representations, which implies that their semantics have to be defined by the used media type. The missing pieces of information therefore have to be contained in the documentation of the media type. Amundsen [56] elaborates on the design of media types and their documentation.

Depending on the media type, the information encoded in hypermedia controls can vary. For example, `application/hal+json` [41], which greatly inspired the representations of the examples above, only uses the link hypermedia control. Through its presence, the hypermedia control denotes that the operation can be invoked. In addition, the URI also provides information about the resource that is concerned with this operation. This information is enough to qualify an API as level one concerning discoverability. The `application/html` [37] media type is more sophisticated as it defines hypermedia controls like forms. A form additionally contains information concerning the HTTP method and further fields that can be sent along with the request. A system using the `application/html` media type hence qualifies as level two. All information that is needed for invoking the operation is present. This allows for completely generic clients to be written. Those can present a fully functional UI out of such a response as the clients do not need any out-of-band information for generating a request. For the `application/html` media type, these clients are known as web browsers.

In the context of APIs, where the response is not directly used to build up a UI, sophisticated hypermedia controls such as forms provide less value than in the scenario of the Web. Simpler ones such as links encode enough information for the client to control UI-logic like the current page in a workflow wizard, the state of buttons, or the visibility of entire UI-control groups. However, more sophisticated media types simplify clients as less information about the semantics of the elements has to be encoded out-of-band, e.g. through external documentation. When using a media type that supports forms, a client only has to have knowledge of the parameter names of a specific operation. The remaining information for building up a request can be figured out by the media type parser through the information provided in the resource representation. Forms are especially useful in scenarios that allow the server to embed them pre-populated with contextual information. As a result, a client only has to activate them and does not need to have any knowledge about the actual parameters. The media type `application/vnd.siren+json` [40] supports the concept of `actions` which is similar to `forms` in HTML. Code 4.6 illustrates how to pre-populate such an action with values, whereas Code 4.7 shows what the resulting HTTP request for this action could look like. The resource in Code 4.6 shows a single product (*productId* 3) in a possible list of products. The product embeds an action whose fields (*productId* and *quantity*) are already pre-populated with values. This allows a client to add this product to the current order by simply activating this action.

REST's dynamic discoverability through hypermedia can greatly simplify client application because less information has to be included out-of-band. It also supports documentation of these systems although dynamic discoverability implies that documentation either has to be gener-

ated using a running instance of the service or produced on-the-fly. Spring rest-docs [57], for example, makes use of a running instance of an HTTP service in order to generate documentation.

Code 4.6: A minimal representation of a product resource using the Siren-media type, illustrating the use of `actions`.

```
{
  "entities": [
    {
      "properties": {
        "productId": 3
      },
      "actions": [
        {
          "name": "add-to-current-order",
          "method": "POST",
          "href": "http://api.example.org/orders/1/products",
          "type": "application/x-www-form-urlencoded",
          "fields": [
            {
              "name": "productId",
              "value": "3"
            },
            {
              "name": "quantity",
              "value": "1"
            }
          ]
        }
      ]
    }
  ]
}
```

Code 4.7: An HTTP POST request as it could result from activating the action shown in Code 4.6.

```
POST /orders/1/products HTTP/1.1
Content-Type: application/x-www-form-urlencoded
Content-Length: 22

productId=3&quantity=1
```

4.3 Component Responsibility

As both approaches to API design implement the CS-style, the discussion about component responsibility is divided into client and server.

4.3.1 GraphQL

The central part of GraphQL is the schema that is designed by the developer. It serves as the contract for the communication between client and server and describes the types and their relations in the system.

Server

A GraphQL server hosts the schema and provides access to the data described by that schema through its interface. Its primary job is to validate and execute the queries and thereby serve the data that is requested by the clients or perform the necessary data manipulation if the request contains mutations.

Client

As already identified in section 2.5.2, a GraphQL client is in charge of the know-how by formulating queries and sending them to the server. Because the server only serves data, it is the client's responsibility to implement the control flow of the application.

4.3.2 REST

According to Fielding, "A REST API should spend almost all of its descriptive effort in defining the media type(s) used for representing resources and driving application state, or in defining extended relation names and/or hypertext-enabled mark-up for existing standard media types" [32]. The definition of REST and the cited elaboration of Fielding about REST API design leads to an interesting distribution of responsibility that may not be obvious at first. Central to the communication between the client and server component in a RESTful system are media types and link relations. The responsibilities are therefore divided into generating and parsing representations corresponding to these media types

Server

In a RESTful system, the server acts as the host of data and also implements the control flow through the application. It is responsible for generating and accepting representations of resources. This includes the responsibility to embed enough control data in the representations to allow the client to consume the API without out-of-band information.

Client

A client component of a REST API is in charge of parsing the representations received from the server and generating new ones out of the control data included in the representations. This forbids the client to generate requests “on-the-fly”, i.e. without having received appropriate control data that would suggest the creation of a request. The client is also in charge of parsing the metadata that is included in the response and act according to it. An example where the client acts based on instructions by the server is caching of responses. Zuzak *et al.* [58] identified the use of Finite State Machines (FSMs) for the implementation of RESTful clients and client frameworks.

4.4 Simplicity

The following section explores the simplicity of an architecture using either GraphQL or REST in general. In addition, the tool support is also evaluated.

4.4.1 GraphQL

GraphQL is a solution to a very specific problem which makes it simple and easy to understand. By nature, such solutions are very concrete and consequently allow for a variety of tools to be created that assist developers. For example, GraphQL’s introspection feature [24] drives GraphiQL [55], a generic in-browser Integrated Development Environment (IDE) for GraphQL servers. GraphiQL allows for exploring a running GraphQL server by sending queries and mutations. The introspection enables GraphiQL to provide features such as auto-completion, syntax highlighting, and error reporting. Especially auto-completion can greatly simplify the work of developers.

4.4.2 REST

REST is defined as an architectural style and therefore naturally technology independent. The main focus and its primary discriminator to other styles are the constraints on the uniformity of the component interfaces and their interactions [1]. Richardson and Ruby [3] argue that it is the uniformity that makes RESTful services easy to understand. The constraints defined by REST allow for a lot of of standardization, for example resource identifiers (URI) and resource representation formats (media types). Standardizing these aspects in turn allows creating reusable software elements, like HTTP libraries, media type parsers and so forth. This further improves the simplicity of a RESTful system because developers can reuse many existing libraries and tools related to the Web.

4.5 Performance

The following section discusses the approaches GraphQL and REST take in improving the performance of a system.

4.5.1 GraphQL

One of GraphQL's primary design goals was a more efficient use of the network compared to a resource-based approach [23]. It uses a query language that allows the client to describe the data it needs. This moves the know-how to the client. The following sections discuss the impact of that on the problems of over-fetching and multiple round-trips.

Over-fetching

By moving the know-how about which data should be returned to the client, over-fetching is eliminated as the server only returns what was specified by the client. In GraphQL, the know-how is represented in the form of a query that is sent to the server. Another design concern of GraphQL is predictability [23]. To achieve that, a GraphQL client is forced to declare its desired fields only by the use of primitive types. It is therefore guaranteed that the server does not return more data than the client asks for, thus preventing over-fetching.

Multiple round-trips

GraphQL allows invoking multiple operations in one interaction. Code 4.8 illustrates how this can be done. Therefore, even if multiple operations are to be executed in response to a single

user action, GraphQL allows invoking them using a single interaction, thus preventing multiple round-trips. In addition, GraphQL also allows traversing the relations of types, e.g. to include the author of a post in the post itself. Code 2.2 on page 9 already illustrated that.

Code 4.8: A GraphQL query against the schema defined in Code 2.1, illustrating the use of multiple queries in a single interaction. The result of each query is assigned to a variable.

```
query {
  max: timeline(of: "MaxMustermann") {
    content
  }
  john: timeline(of: "JohnDoe") {
    content
  }
}
```

Code 4.9: An exemplary result for the query shown in Code 4.8. The variables defined in the query ("max" and "john") appear in the output, along with the result of the associated query.

```
{
  "data": {
    "max": [
      {
        "content": "My first blogpost!"
      }
    ],
    "john": []
  }
}
```

Caching

GraphQL itself does not define anything related to caching. Libraries like Relay [59] provide client-side support for caching of GraphQL responses. It uses a map-like structure to store parts of the returned graphs in a normalized way. Sole client-side caches always brings up problems concerning proper invalidation of caches. Relay partially addresses this through updates to this data structure with data returned from subsequent responses, thereby replacing potentially stale data.

4.5.2 REST

In order to improve network efficiency, REST defines a cache constraint which requires that "data within a response to a request be implicitly or explicitly labelled as cacheable or non-

cacheable" [1, Section 5.1.4]. In HTTP, this is achieved through the separation of each message into header and body, with the header containing metadata that describes the body. The data in the body is labeled as cacheable or noncacheable through certain header fields that are concerned with caching. Code 4.2 on Page 33 illustrates an HTTP-response containing an empty JSON-object that is labeled as cacheable for a duration of 3600 seconds. The following sections discuss the problems of over-fetching and multiple round-trips in the context of REST and explain problems concerned with caching.

Over-fetching

In HTTP, the representation that is returned for a resource is defined by the chosen media type. Content Negotiation [45, Section 12] can help in selecting an appropriate representation of a resource. Typically though, this mechanism is used to request a different representation of a resource that still encodes the same information, such as in a different language or encoding. To solve the problem of over-fetching, mechanisms for controlling the range of the returned data are needed. FIQL [27] is an effort by Nottingham in standardizing a syntax for expressing filters over a resource, thereby returning only those resources of a collection that are interesting for the client. OData, already described in section 2.3.5, also provides mechanisms for applying filters and selections to a resource.

In general, the problem of over-fetching occurs in HTTP because the representation is chosen by the server and can only be influenced by the client if the server provides explicit support for doing so.

Multiple round-trips

HTTP as a partial implementation of REST is an application-level protocol and hence intended to build a network-based API. This means that actions of the user should be modeled using the concepts of HTTP: resources in combination with appropriate actions. Operations are usually mapped in a one-to-one relation on interactions in HTTP. As a result, multiple operations in response to a single user action often lead to multiple interactions with the API.

Concerning the retrieval of data, that is, fetching representation of resources, all related resources could be embedded in one resource. The problem with this approach is that it destroys the consistency of the client's cache. REST defines caching on the resource level: i.e. the representation of a resource is the only level of granularity in the context of caching because the metadata only describes the complete response. The `multipart/*` media type [60, Section 5.1] allows multiple responses, each with its own header and body section, to be embedded into a single response. Yet, all header fields except those starting with `Content-` do not have any meaning in the body parts of a multipart response. As a result, all cache related headers

are ignored as well. URIs act as keys in the map-like storage structure of an HTTP cache. Hence, for caches to even properly handle cache control headers in those body parts, each body part would have to include a URI to designate its origin.

Figure 1 shows a sequence diagram that illustrates two sequential interactions of a client with a server. Figure 2 and Figure 3 illustrate the client's cache after each interaction. The representation of `R2` embeds `R1`. However, the cache can only store complete representations and is therefore unable to update the already existing representation of `R1`. A subsequent, direct request for a representation of `R1` may be answered by the cache, resulting in stale data although the UI may already display up-to-date data through the embedded representation of `R1` in `R2`. In order to preserve cache consistency, resources should either never be embedded or direct requests for representations of embedded resources should not be cached.

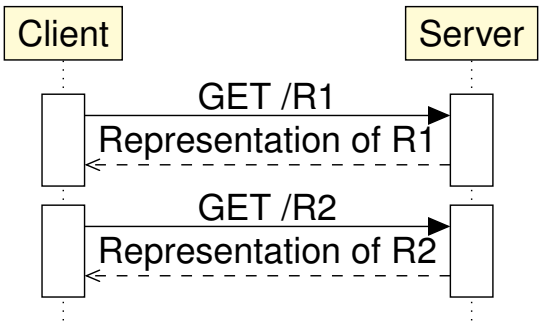

Figure 1: A sequence diagram showing two HTTP interactions of a client with a server.

In the context of operations that modify data, also known as unsafe operations, the multiple round-trip problem often occurs in CRUD over HTTP services, as they only expose primitive data manipulation operations on entities. As a result, the client has to orchestrate various data manipulation operations in order to achieve a certain goal. Yet, HTTP as an application-level protocol suggests to create a network-based API, thereby explicitly modeling the client's operations on the protocol level.

Caching

Caching as described in REST and consequently implemented in HTTP is conceptually different from other caching solutions. In REST, responses are annotated with metadata that describe the cache semantics of the contained data [1]. Thereby, if and how a response is cached is determined at run time. Other caching approaches typically feature static coupling between the invoked operation and the cache semantics. For example, the response of a certain operation is cached for 5 minutes, independent from the individual responses, whereas in REST, each response contains information about its cache semantics. Accordingly, subsequent invocations of the same operation could result in different cache behaviors if the server decides to change the cache semantics in-between.

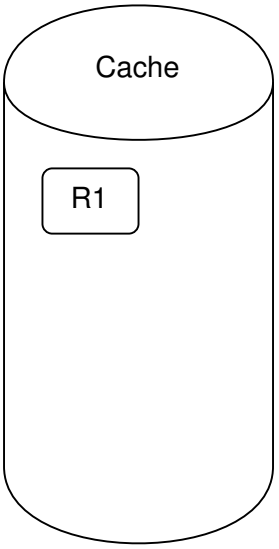

Figure 2: Illustration of the client's cache after the first request. `R1` denotes the retrieved representation.

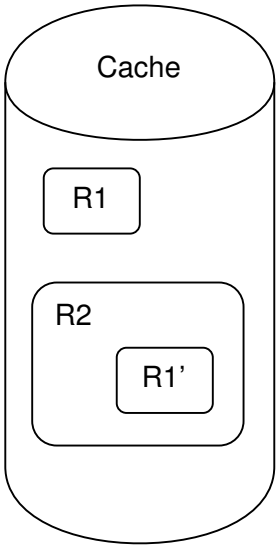

Figure 3: Illustration of the client's cache after the second request. The previously retrieved representation `R1` is still in the cache. In addition `R2` transparently embeds a newer version of it: `R1'`.

4.6 Interaction Visibility

The primary difference concerning visibility between GraphQL and REST is the use of a language vs. messages. The following sections elaborate on the influence of this difference on the visibility of their interaction style.

4.6.1 GraphQL

Through its dynamism, transfer of code is less visible than transfer of standardized messages. However, there is still a significant difference in the visibility of imperative and declarative code. The latter by definition has to be interpreted in order to be executed, which means an intermediary can also interpret the code and reason about it without actually executing any statements. PostScript [61] is an example for an imperative language used in the communication protocol between printers and computers. In order to reason about the result of such an interaction, an intermediary would have to execute the code as well and examine the result. With GraphQL featuring a declarative language, intermediaries capable of understanding the GraphQL grammar can at least partly reason about the communication between a client and a GraphQL server. For example, based on the convention that queries do not cause side-effects, an intermediary can reason that a query sent to a server did not change any data and may be subject to caching.

4.6.2 REST

REST was designed with a high visibility in mind. It features constraints that enforce standardized messages being exchanged over a uniform interface, using actions whose semantic meaning is well-defined. HTTP, for example, clearly defines that `GET` requests must not have any side-effects the user can be held responsible for [45]. They also have to be idempotent, building the foundation for shared, intermediary caches as they can infer from the message that another invocation would produce the same result. `PUT` and `DELETE` are defined with similar semantics, although they are only considered idempotent and not safe. `POST` requests though do not allow to infer anything. They do not have to be idempotent and can cause side-effects. That is the reason why many applications that use HTTP for tunneling purposes use the `POST` method; intermediaries will neither touch nor cache these requests as they cannot infer any semantics from them. Similar to the methods, identifiers (URIs) and metadata (HTTP headers) also have well defined semantics that can be interpreted by intermediaries.

4.7 Customizability

Both approaches support the concept of customizability through their use of mobile code that is either sent to the server or the client.

4.7.1 GraphQL

GraphQL is related to the REV-style as its primary means of communication is a declarative language. Hence, the client component specializes the server component to only return the data the client asks for. However, as GraphQL only features a declarative language, the capacity for specialization is limited in comparison to an imperative language.

4.7.2 REST

In REV, the client specializes the server. REST optionally includes the Code on Demand (COD)-style, which is inverse to the REV-style in the way that in COD, the server specializes the client. JavaScript is the Web's implementation of COD and can be executed by most[1] browsers.

[1] Lynx (http://lynx.browser.org) is a text-based web browser for shell environments and therefore does not support JavaScript.

As identified by Zuzak *et al.* [58], client libraries or client frameworks for RESTful APIs can also include a COD-engine for executing code that is sent by a server, similar to how web browsers execute retrieved JavaScript code. However, this does not have to be JavaScript but can be any language. The code that is retrieved from the server is just another resource and consequently has to be described by a media type. The media type in turn describes the semantics of the language. For example, the HTML element `script` has an attribute `src` that allows a Web page to link to an external JavaScript file. The HTML media type states that scripts referenced through this attribute have to be written in JavaScript, thus the browser can expect a response with the media type `application/javascript` when fetching the resource the URI in the `src` attribute refers to.

5 Discussion

5.1 Complexity

Some comparisons and discussions portray REST as a solution that is just fine for simple APIs, whereas one should consider moving to GraphQL as soon as things start to get complex. The problem with such conclusions is the missing definition of complexity. What is it that makes an API complex? Who has to deal with this complexity?

One approach to justify this conclusion would be that GraphQL makes fetching data in various ways really simple for the client. At the same time, a GraphQL server does not have to implement special support for individual client types. All clients can reuse the same API. In a resource-based approach, reusing operations is often not that trivial. The complexity of clients increases due to the need for orchestration over multiple operations. In fact, GraphQL was created as a response to problems that occurred by using a resource-based approach in some scenarios [23]. As a result, REST appears to be only suited for APIs that do not have to serve lots of data in different ways. However, I would consider this interpretation of complexity as quite limited. While the problem of efficiently fetching data is a serious one, operations which retrieve data have the pleasant attribute of being safe, not causing side-effects. Because of that, the only restrictions that are usually in place can broadly be categorized as access-restrictions, dealing with the problem of who is allowed to access which data. Restrictions in this context are to be understood as initiators of exceptional execution paths. They abort an operation before it can complete successfully in order to prohibit that the client receives data it should not receive. On the other hand, operations that do modify data additionally place constraints on the state they are about to modify. An example that has already been mentioned several times in this thesis is that of an ordering process. Adding more products to an order that has already been placed is likely to be prohibited by the system. Thus, the restriction that is in place here deals with the current state of the order and aborts the operation for adding another product in some cases. Reasoning about the state of a system introduces the dimension of time because state varies over time. Dealing with these restrictions and the resulting dependencies between operations also introduces complexity in the client. Hence, discussions about the complexity of an API should include problems like these.

So far, we have identified two sources of complexity introduced by an API in client applications: orchestration over multiple operations in order to fetch certain data and dependencies between

operations. The latter have to be handled by the client in order to allow the user to successfully use the application. Depending on the discoverability of the API, this can lead to replication of server-side control logic on the client. As the evaluation in subsection 4.2.2 showed, REST provides a way of expressing these dependencies via the use of hypermedia, or in this context, just hypertext, while GraphQL does not account for this problem.

In contrast to unsafe operations, safe operations can almost always be composed arbitrarily. For this reason, GraphQL can expose a generic, VM-like module that interprets declarative code in order to compose the data graph the client asks for. If fetching any of the individual fields in a GraphQL query would cause side-effects, this general approach would not be feasible. Through the same rationale, it is comprehensible that for mutations (operations with side-effects) GraphQL only provides an RPC-like approach that allows for state changes on a defined abstraction level, i.e. the fields of objects cannot be modified arbitrarily in GraphQL. Instead, the client has to invoke mutations which define modifications of state on a fixed level of abstraction. Contrary, the level of detail in queries is chosen arbitrarily by the client at run time. In REST, the server defines the abstraction level for both, safe and unsafe operations.

The discussion about the source and definition of complexity in an API is at least two-fold. For one, efficiently fetching data is certainly a problem that has to be addressed. On the other hand, dealing with the dependencies between provided operations is a topic that should not be underestimated. One of the intents of the CS-style is to simplify the overall system by separating concerns like UI and business-logic. If the client has to replicate the control logic because it cannot be guided by the server, those concerns are no longer strictly separated.

It appears that GraphQL and REST address different types of complexity. While GraphQL solves performance problems and simplifies fetching data in various ways, REST provides an excellent way for expressing dependencies between operations. This allows the server to guide its clients.

5.2 Responsibility Distribution

GraphQL makes fetching data really simple because once the server is up and running, the client can query any data within the schema. From a responsibility point of view, the server turns into a component that only serves data. This, in turn, means that clients of this server have to implement the required control flow that should be present in the system.

A problem that can arise once the amount of business logic increases, is the increased potential for violating the Tell, Don't Ask (TDA)-principle [62]. TDA is usually discussed in the context of OOP, however it can also be applied to distributed systems by viewing every system as an object that provides data and services. In short, the principle states that commands should

be sent to objects, telling them what to do (with the data they encapsulate) instead of querying (asking) objects for their data and performing checks and operations on that data elsewhere. Violating TDA leads to a chatty communication between objects, code duplication and an overall brittle system. This, in turn, causes those objects to be hard to change as they provide little abstraction and just act as data structures. The situation gets even worse in the context of entire systems. Moving functionality between systems is harder and more cost-intensive than between modules inside a single system. Violations of this principle often appear in CRUD over HTTP systems. Those often directly map the HTTP actions to the data manipulation actions and resources to data elements of the underlying storage medium, e.g. a relational database. The lack of abstractions leads to interaction patterns that first retrieve a number of resources and later change the state of some other resource solely based on the previously retrieved state. Not only does this affect performance due to unnecessary network calls but also fails to provide useful abstractions which would simplify the client components.

GraphQL is a generic data access layer and hence can encourage the violation of this principle. Through its dynamic interface, it allows retrieving just any data that is available in the schema. As soon as clients act upon the retrieved data solely for the purpose of changing some other state in the system, they very likely violate TDA. Yet, this is not always the case. Convenient access to data in a variety of ways can be useful without violating TDA. If all of the retrieved data is shown to the user and has to be retrieved anyway, it is preferable to have an efficient solution. In the end, TDA just discourages querying data from services or objects solely for the purpose of decision making. In such cases, it is easier to move the functionality to the data.

REST provides a uniform way to do that by treating information as a first-class citizen in the form of resources. The limited and primitive set of available methods forces designers to explicitly model functionality as commands to data-encapsulating entities instead of procedures that are remotely invoked using a set of parameters. Through hypermedia, those resources can describe relations among each other, leading to a technology independent, network-based API that encodes valuable information about domain concepts and possible interaction paths.

5.3 Standardization

REST was created as an architectural style in order to formally describe the thinking model on how the Web should work [1, Section 6.1]. Through the requirement for anarchical scalability across multiple trust boundaries, lots of effort has gone into designing it in a way that allows for a lot of standardization. Examples for that are HTTP as an application-level protocol that uses URIs to identify resources and media types that define their representations. The extensive use of metadata allows HTTP to be self-descriptive and extensible in a variety of ways. The media type concept enables a plugin-based architecture in which components can be dynamically

extended with new functionality, independently of other components. This thinking model allows the Web to continuously evolve.

Such a mentality was necessary for the Web because once a certain functionality was deployed, it was out there – almost impossible to ever be removed again. A good example for that was the initial misdesign of using HOST-relative paths in HTTP [1, Section 6.3.2.1]. Releasing a new version of the Web that fixes problems like these is simply impossible due to the number of independent parties across the world that already use it.

GraphQL is designed with a different vision in mind. As a concrete technology, it addresses performance problems like over-fetching and multiple round-trips that can occur in applications that need to fetch lots of data in various different ways. Using GraphQL, client developers can change the way they fetch data independently from the server. In a RESTful design, the representations of resources are defined by the server and hence cannot be altered by the client. Similar to REST, GraphQL is also formally defined by the means of a specification. This frees developers from coupling their applications to specific versions of certain libraries, a problem that also occurred during the creation of the Web with `libwww` [1, Section 5.1.4].

The shift to native mobile apps was the initial motivation for developers at Facebook to create GraphQL [23]. Mobile apps, similar to browsers in the context of the Web, are clients that are out of the control of their creators once they are deployed (i.e. installed by the end user). It is impossible to simply "redeploy" all of them because of a breaking change in the communication protocol. Therefore, the protocol has to be designed in a way that is extensible without breaking existing clients. GraphQL is Facebook's effort in achieving that for their use cases. In general, the way GraphQL is designed and defined would not allow it to scale to the size of the Web like REST did. It is simple not as extensible as REST. Yet, this has never been the intent [63].

In the context of APIs, the RESTful design model can be used to encode gained knowledge about a certain domain in a technology independent way. A designers primary tools for this job are links relations and media types. By following a "media type first" mentality, resources, their relations and state transitions can be defined abstractly, without thinking about concrete technologies like programming languages of clients or servers. This, in turn, allows developers to create reusable, domain-aware software modules like media type parsers for certain languages, greatly simplifying the work of developers. Similarly GraphQL permits defining a domain model in terms of the schema that is hosted on the GraphQL server. However, GraphQL lacks support for dynamic discoverability, hence the availability of state transitions cannot be expressed.

5.4 Performance

Improving the performance of a system solely through changes to the software always results in increasing the efficiency by which the hardware-provided resources are used. One

of GraphQL's primary design goals is to improve efficiency compared to a resource-based approach [23]. REST also accounts for potential performance problems by including the cache constraint: data that is included in a response has to be explicitly denoted as cacheable or not cacheable [1]. Although both approaches include a strategy for improving network efficiency, the solutions vary.

GraphQL's use of a declarative language eliminates over-fetching and avoids the multiple round-trip problem, thereby increasing network efficiency, because only the data that is actually requested by the client is transferred. Over-fetching and the problem of multiple round-trips often occur in the context of a resource-based approach such as REST because the server defines the resource layout. Contrary to REST, GraphQL does not account for caching in the specification, which has to be handled solely on the client.

The primary difference between both approaches is the way they handle caching and cache invalidation. In order for caching to be beneficial for a system, the application domain must allow the reuse of previous responses for subsequent requests. Data in a real-time system, for example, changes with a high frequency. In addition, such systems usually have to fulfill a requirement that demands the presented data to be almost always up to date. As caching can cause data to be stale, applying it in such a scenario is not useful.

The following section discusses the differences in handling caching and cache invalidation in detail.

5.4.1 Cache Invalidation

Data is considered stale if it no longer represents the actual state of an entity. Caching is one cause of stale data, as responses of previous requests are reused for subsequent ones. Discarding a cache entry is called invalidation and can happen through a number of different strategies. In REST, responses can contain metadata that describes the cache semantics of the data present in the message. A RESTful client can interpret this metadata and is allowed to cache the message according to it. This approach has two interesting implications. First, the server component is in charge of annotating the generated response with appropriate metadata that describes its cache semantics. Second, the client does not imply anything about the cacheability of the response but rather adheres to this metadata and caches responses according to it. This allows the server to instruct the client on a per-response basis to cache a certain representation. It is desirable to introduce this kind of coupling between the component that generates the response and the decision process about the cache semantics. The former knows best if the data in the response is static or computed on-the-fly. Such a dynamically generated response should probably not be cached at all because of its temporal nature. Thus, the server can attach appropriate metadata that tells the clients not to cache this response.

The cache that is included in the Relay-library [59] is never invalidated on its own. Instead it is just overwritten with new data from subsequent responses. In order to get rid of stale data, queries can be force-fetched, which means they bypass the cache and are directly sent to the server. In turn, the response overwrites the cache with new, up-to-date data.

If caches should be invalidated without metadata that is provided by the server, the client has to include invalidation rules on its own. These are typically coupled to individual operations. For example, data that is returned by an operation is cached for 15 minutes before being invalidated. The coupling between the cache configuration and the invoked operation is thus static and requires at least reconfiguration or, in the worst case, redeployment of the client in order to be changed. In REST and consequently HTTP, the client is just able to interpret the metadata that can be attached to each individual response. Changing the cache semantics of a resource involves zero changes to the client code. An approach like that was necessary for the Web due to its requirement for anarchic scaling [1].

5.4.2 Impact on the Server

Besides the performance gain for the application itself, caching also has a non-negligible impact on the server. If a client is able to reuse responses from previous requests, the server's load is reduced, improving scalability of the service. Naturally, the server cannot control the behavior of its clients. Despite that, providing them with information about the lifetime of a specific piece of data allows clients to behave "well" by adhering to the instructions of the server. Without metadata, clients do not even have the chance to do so but rather have to proactively request that data and compare it for changes locally.

6 Conclusion

The biggest implication of using GraphQL for the design of an API is the shift in a number of responsibilities from the server to the client. First, the client is in charge of implementing the workflow that should be represented in the application. Implementing the workflow in this context does not only mean in terms of the UI. This is the responsibility of the client after all. More importantly the client is also in charge of implementing the actual rules that determine which of the available paths are valid. Mostly through the lack of metadata in the responses from the server, the client has to figure out on its own, which steps or sequences of operations are valid. This requires the client to implement control logic in order for the user to successfully navigate through the application. Second, if the client uses a library that adds a cache, it also has to manage its invalidation, a task that should not be underestimated once the application grows in size.

Depending on the problem domain and the application's requirements, the impact of these consequences varies. Sometimes, caches are not needed or their use is even discouraged due to real-time requirements of the data. Some domains do not impose any or only few constraints on operations that cause dependencies between them, thus lowering the impact of lacking dynamic discoverability that would allow to infer such dependencies. For example, an API for an application that primarily provides multiple views onto the same data set will likely contain many safe operations for retrieving this data in various ways and fewer unsafe ones.

Technology decision making is a complex process that can hardly be automated due to the diversity of requirements. It is also notoriously hard to create a useful model that formally describes such a decision process. The problem with such models is that they are always an abstraction and abstractions hide details. It cannot be guaranteed that the creator of the model accounted for all details that are relevant in a specific context. In the end, it is the job of software engineers and architects to identify those details and decide based on the gained knowledge, which technology, concept, or approach brings the most value. Therefore, instead of concluding with a decision model on when to choose which approach, this thesis aims to give an overview over many relevant aspects in the context of API design. Whether or not a given approach is suitable in a concrete example always depends on the actual requirements which makes it impossible to state accurate recommendations.

7 Future work

Several areas for future work have been identified in this thesis. They are outlined in this final chapter.

Architectural style for GraphQL

The premise for one of the possible approaches to comparing GraphQL and REST was the classification of GraphQL as an architectural style. Although research identified two candidates, none of them described GraphQL well enough to serve as a meaningful abstraction. Fielding identified that architectural styles can be described as a deviation tree of constraints. Consequently, new architectural styles can be created by adding constraints to an existing style. Several problems have been described when GraphQL is classified as RDA or REV. Future work in this area could identify the missing constraints in these styles in order to mitigate those problems.

Hybrid solution

This thesis conducted a comparison between GraphQL and REST in order to identify their difference in the context of API design. Another topic for future research would include the analysis of hybrid solutions between both approaches. The REST architectural style has proven its usefulness in the design of the Web and allowed it to scale to today's size. However, as identified in this thesis, new challenges are introduced when REST is applied to the design of APIs, like the trade-off between specialization and generalization concerning reusability of operations. GraphQL can be considered a solution for problems like over-fetching and multiple round-trips that can arise from this trade-off. Future research could therefore investigate if and how both approaches can be combined in order to take advantage of the benefits of both.

GraphQL Meta-Graph

One of the identified problems of GraphQL is the lack of metadata for describing the returned data. In HTTP, each response is divided into a header and a body section with the header

containing metadata that describes the body. In comparison, GraphQL returns several individual entities in one response. Each instance of a type in a GraphQL-response can thus be considered equivalent to a complete response in REST. As a result, metadata would have to be included per-instance of a type in GraphQL. Future work in this area could identify the possibilities for adding metadata on this level. One approach that could be worth investigating is an additionally returned graph with the same structure that only contains metadata for each field, forming a “Meta-Graph”.

Bibliography

[1] R. T. Fielding, “Architectural Styles and the Design of Network-based Software Architectures”, University of California, Irvine, 2000, ISBN: 0-599-87118-0.

[2] C. Severance, “Understanding the REST style”, *Computer*, vol. 48, no. 6, pp. 7–9, 2015, ISSN: 0018-9162. DOI: 10.1109/MC.2015.170.

[3] L. Richardson and S. Ruby, *Restful Web Services*. O’Reilly, 2007, ISBN: 978-0-596-52926-0.

[4] D. L. Parnas, “On the Criteria to Be Used in Decomposing Systems into Modules”, *Commun. ACM*, vol. 15, no. 12, pp. 1053–1058, Dec. 1972, ISSN: 0001-0782. DOI: 10.1145/361598.361623.

[5] ——, “Designing Software for Ease of Extension and Contraction”, *IEEE Transactions on Software Engineering*, vol. 5, no. 2, pp. 128–138, Mar. 1979, ISSN: 0098-5589. DOI: 10.1109/TSE.1979.234169.

[6] ——, “On a buzzword: Hierarchical structure”, 1974.

[7] D. L. Parnas, “On the Design and Development of Program Families”, *IEEE Transactions on Software Engineering*, vol. 2, no. 1, pp. 1–9, Jan. 1976, ISSN: 0098-5589. DOI: 10.1109/TSE.1976.233797.

[8] D. E. Perry and A. L. Wolf, “Foundations for the Study of Software Architecture”, *ACM SIGSOFT Software Engineering Notes*, vol. 17, no. 4, pp. 40–52, Oct. 1992, ISSN: 0163-5948. DOI: 10.1145/141874.141884.

[9] Institute of Electrical and Electronics Engineers, IEEE Computer Society, Software Engineering Standards Subcommittee, IEEE Standards Association, and IEEE Standards Board, *ANSI/IEEE 1471-2000, Recommended Practice for Architecture Description of Software-Intensive Systems*. New York: Institute of Electrical and Electronics Engineers, 2000, ISBN: 978-0-7381-2518-3.

[10] International Organization for Standardization, International Electrotechnical Commission, Institute of Electrical and Electronics Engineers, and IEEE-SA Standards Board, *ISO/IEC/IEEE 42010:2011: Systems and software engineering - architecture description*. Geneva; New York: ISO : IEC ; Institute of Electrical and Electronics Engineers, 2011, ISBN: 978-0-7381-7142-5.

[11] P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, P. Merson, R. Nord, and J. Stafford, Eds., *Documenting software architectures: Views and beyond*, 2nd ed., ser. SEI series in software engineering, Upper Saddle River, NJ: Addison-Wesley, 2011, 537 pp., ISBN: 978-0-321-55268-6.

[12] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, 3rd ed. Addison-Wesley Professional, 2012, ISBN: 978-0-321-81573-6.

[13] ——, *Software Architecture in Practice*, 1st ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1998, ISBN: 0-201-19930-0.

[14] ——, *Software Architecture in Practice*, 2nd ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003, ISBN: 0-321-15495-9.

[15] M. Fowler, "Design - Who needs an architect?", *IEEE Software*, vol. 20, no. 5, pp. 11–13, Sep. 2003, ISSN: 0740-7459. DOI: 10.1109/MS.2003.1231144.

[16] A. Klusener, R. Lämmel, and C. Verhoef, "Architectural modifications to deployed software", *Science of Computer Programming*, vol. 54, pp. 143–211, 2-3 Feb. 2005, ISSN: 01676423. DOI: 10.1016/j.scico.2004.03.012.

[17] M. Shaw, "Toward higher-level abstractions for software systems", *Data & Knowledge Engineering*, vol. 5, no. 2, pp. 119–128, Jul. 1990, ISSN: 0169023X. DOI: 10.1016/0169-023X(90)90008-2.

[18] E. Gamma, R. Helm, J. Vlissides, and R. Johnson, Eds., *Design patterns: Elements of reusable object-oriented software*, Addison-Wesley, 1995, 395 pp., ISBN: 978-0-201-63361-0.

[19] F. Buschmann, Ed., *Pattern-oriented software architecture: A system of patterns*, Chichester ; New York: Wiley, 1996, 457 pp., ISBN: 978-0-471-95869-7.

[20] R. N. Taylor, N. Medvidović, and E. M. Dashofy, *Software architecture: Foundations, theory, and practice*. Hoboken, NJ: John Wiley, 2010, 712 pp., OCLC: ocn231670965, ISBN: 978-0-470-16774-8.

[21] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system", 2008. [Online]. Available: https://bitcoin.org/bitcoin.pdf (visited on 2017-04-04).

[22] P. Kruchten, "The 4+1 View Model of Architecture", *IEEE Softw.*, vol. 12, no. 6, pp. 42–50, Nov. 1995, ISSN: 0740-7459. DOI: 10.1109/52.469759.

[23] L. Byron, "Designing a Data Language", presented at the Strange Loop 2016, St. Louis, Sep. 2016. [Online]. Available: https://www.youtube.com/watch?v=Oh5oC98ztvI (visited on 2017-04-04).

[24] (Jun. 2015). GraphQL is a query language and execution engine tied to any backend service., [Online]. Available: https://github.com/facebook/graphql (visited on 2017-04-08).

[25] (Sep. 2015). GraphQL-Documentation, [Online]. Available: https://github.com/graphql/graphql.github.io (visited on 2017-04-08).

[26] ECMA International, *ECMAScript 2015 Language Specification*, 5.1. Jun. 2015.

[27] M. Nottingham, "The Feed Item Query Language", IETF Secretariat, Internet-Draft, Dec. 2007. [Online]. Available: https://tools.ietf.org/id/draft-nottingham-atompub-fiql-00.txt.

[28] M. Pizzo, R. Handl, and M. Zurmuehl, Eds., *OData Version 4.0. Part 1: Protocol Plus Errata 03*, Feb. 6, 2016.

[29] (Feb. 2015). A JavaScript library for efficient data fetching, [Online]. Available: https://github.com/Netflix/falcor (visited on 2017-04-08).

[30] J. Husain, “JSON Graph: Reactive REST at Netflix”, ser. Applicative 2015, New York, NY, USA: ACM, 2015, ISBN: 978-1-4503-3527-0. DOI: 10.1145/2742580.2742640.

[31] L. DeMichiel, M. Bouschen, N. Seyvet, K. Sutter, P. Poddar, F. Benoit, G. Yorke, M. Keith, D. Anupalli, E. Bernard, S. Ebersole, S. Marlow, R. Schweigkoffer, E. Ireland, M. Byon, O. Gierke, M. Adams, A. Bien, B. Mueller, W. Keil, and C. von Kutzleben, “Java Persistence API Specification”, JSR 338, Apr. 2, 2013.

[32] R. T. Fielding. (Oct. 20, 2008). REST APIs must be hypertext-driven, [Online]. Available: http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven (visited on 2015-04-08).

[33] D. Renzel, P. Schlebusch, and R. Klamma, “Today’s Top “RESTful” Services and Why They Are Not RESTful”, in, Nov. 28, 2012, pp. 354–367, ISBN: 978-3-642-35063-4.

[34] L. Richardson. (2008). The Maturity Heuristic, [Online]. Available: https://www.crummy.com/writing/speaking/2008-QCon/act3.html (visited on 2017-03-19).

[35] J. Webber, S. Parastatidis, and I. Robinson, *REST in Practice: Hypermedia and Systems Architecture*, 1st ed. Farnham; Sebastopol, Calif.: O’Reilly Media, Sep. 27, 2010, 448 pp., ISBN: 978-0-596-80582-1.

[36] T. H. Nelson, “Complex Information Processing: A File Structure for the Complex, the Changing and the Indeterminate”, in *Proceedings of the 1965 20th National Conference*, ser. ACM ’65, New York, NY, USA: ACM, 1965, pp. 84–100. DOI: 10.1145/800197.806036.

[37] D. Connolly and L. Masinter, “The ’text/html’ Media Type”, RFC Editor, RFC 2854, Jun. 2000.

[38] M. Nottingham and R. Sayre, “The Atom Syndication Format”, RFC Editor, RFC 4287, Dec. 2005.

[39] M. Amundsen and I. Nadareishvili, “Uniform Basis for Exchanging Representations (UBER)”, May 15, 2015. [Online]. Available: http://rawgit.com/uber-hypermedia/specification/master/uber-hypermedia.html (visited on 2015-05-28).

[40] K. Swiber, “Siren v0.6.2”, Apr. 2017. DOI: https://doi.org/10.5281/zenodo.556783.

[41] M. Kelly, “JSON Hypertext Application Language”, IETF Secretariat, Internet-Draft, May 2016. [Online]. Available: https://tools.ietf.org/id/draft-kelly-json-hal-08.txt.

[42] T. Bray, “The JavaScript Object Notation (JSON) Data Interchange Format”, RFC Editor, RFC 7159, Mar. 2014.

[43] H. Thompson and C. Lilley, “XML Media Types”, RFC Editor, RFC 7303, Jul. 2014.

[44] M. Nottingham, “Web Linking”, RFC Editor, RFC 5988, Oct. 2010.

[45] R. T. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, “Hypertext Transfer Protocol – HTTP/1.1”, RFC Editor, 2616, Jun. 1999.

[46] A. Fuggetta, G. P. Picco, and G. Vigna, “Understanding code mobility”, *IEEE Transactions on Software Engineering*, vol. 24, no. 5, pp. 342–361, May 1998, ISSN: 0098-5589. DOI: 10.1109/32.685258.

[47] J. R. Falcone, “A Programmable Interface Language for Heterogeneous Distributed Systems”, *ACM Transactions on Computer Systems (TOCS)*, vol. 5, no. 4, pp. 330–351, Oct. 1987, ISSN: 0734-2071. DOI: 10.1145/29868.29870.

[48] E. Stenlund and K. Gustavsson, “Efficent data communication between a webclient and a cloud environment”, Jun. 23, 2016.

[49] C. Pautasso, O. Zimmermann, and F. Leymann, “Restful Web Services vs. "Big"’ Web Services: Making the Right Architectural Decision”, in *Proceedings of the 17th International Conference on World Wide Web*, ser. WWW ’08, Beijing, China: ACM, 2008, pp. 805–814, ISBN: 978-1-60558-085-2. DOI: 10.1145/1367497.1367606.

[50] A. Mesbah and A. V. Deursen, “An Architectural Style for Ajax”, in *2007 Working IEEE/IFIP Conference on Software Architecture (WICSA’07)*, Jan. 2007, pp. 9–9. DOI: 10.1109/WICSA.2007.7.

[51] M. Shaw and P. Clements, “A field guide to boxology: Preliminary classification of architectural styles for software systems”, in *Computer Software and Applications Conference, 1997. COMPSAC ’97. Proceedings., The Twenty-First Annual International*, Aug. 1997, pp. 6–13. DOI: 10.1109/CMPSAC.1997.624691.

[52] R. Kazman, G. Abowd, L. Bass, and P. Clements, “Scenario-based analysis of software architecture”, *IEEE Software*, vol. 13, no. 6, pp. 47–55, Nov. 1996, ISSN: 0740-7459. DOI: 10.1109/52.542294.

[53] L. Dobrica and E. Niemela, “A survey on software architecture analysis methods”, *IEEE Transactions on Software Engineering*, vol. 28, no. 7, pp. 638–653, Jul. 2002, ISSN: 0098-5589. DOI: 10.1109/TSE.2002.1019479.

[54] T. Bellwood, S. Capell, L. Clement, J. Colgrave, M. J. Dovey, D. Feygin, A. Hately, R. Kochman, P. Macias, M. Novotny, M. Paolucci, C. von Riegen, T. Rogers, K. Sycara, P. Wenzel, and Z. Wu, *UDDI Version 3.0*. Oct. 19, 2004.

[55] An in-browser ide for exploring graphql., [Online]. Available: https://github.com/graphql/graphiql (visited on 2017-05-02).

[56] M. Amundsen, *Building Hypermedia APIs with HTML5 and Node*, 1st ed. O’Reilly & Associates, Nov. 24, 2011, 240 pp., ISBN: 978-1-4493-0657-1.

[57] Test-driven documentation for restful services., [Online]. Available: https://projects.spring.io/spring-restdocs (visited on 2017-05-02).

[58] I. Zuzak, I. Budiselic, and G. Delac, “A Finite-state Machine Approach for Modeling and Analyzing Restful Systems”, *Journal of Web Engineering*, vol. 10, no. 4, pp. 353–390, Dec. 2011, ISSN: 1540-9589.

[59] Relay is a javascript framework for building data-driven react applications., [Online]. Available: https://github.com/facebook/relay (visited on 2017-05-02).

[60] N. Freed and N. Borenstein, “Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types”, RFC Editor, RFC 2046, Nov. 1996. DOI: 10.17487/rfc2046.

[61] Adobe Systems, Ed., *PostScript language reference*, 3rd ed., Reading, Mass: Addison-Wesley, 1999, 897 pp., ISBN: 978-0-201-37922-8.

[62] A. Hunt and D. Thomas, “The Art of Enbugging”, *IEEE Software*, vol. 20, no. 1, pp. 10–11, Jan. 2003, ISSN: 0740-7459. DOI: 10.1109/MS.2003.1159022.

[63] L. Byron. (Sep. 15, 2015). GraphQL: A data query language, [Online]. Available: https://code.facebook.com/posts/1691455094417024/graphql-a-data-query-language/.

List of Figures

List of Tables

List of Code

List of Acronyms

AJAX	Asynchronous JavaScript and XML
API	Application Programming Interface
COD	Code on Demand
CPU	Central Processing Unit
CRUD	CREATE, READ, UPDATE, DELETE
CS	Client/Server
FIQL	Feed Item Query Language
FSM	Finite State Machine
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
IANA	Internet Assigned Numbers Authority
IDE	Integrated Development Environment
JPA	Java Persistence API
JSON	JavaScript Object Notation
JSR	Java Specification Request
MVC	Model-View-Controller
NCL	Network Communication Language
OASIS	Organization for the Advancement of Structured Information Standards
OOP	Object-Oriented Programming
RDA	Remote Data Access
REST	Representational State Transfer
REV	Remote Evaluation

RPC Remote Procedure Call

SAAM Software Architecture Analysis Method

SOA Service-Oriented Architecture

SOAP Simple Object Access Protocol

SPA Single Page Application

SQL Structured Query Language

TCP Transmission Control Protocol

TDA Tell, Don't Ask

UDDI Universal Description, Discovery and Integration

UDP User Datagram Protocol

UI User Interface

URI Uniform Resource Identifier

VM Virtual Machine

XML Extensible Markup Language