Experiences on Migrating RESTful Web Services to GraphQL

Maximilian Vogel[1,2(✉)], Sebastian Weber[2], and Christian Zirpins[1]

[1] Karlsruhe University of Applied Sciences, Moltkestr. 30, 76133 Karlsruhe, Germany
christian.zirpins@hs-karlsruhe.de
[2] diva-e Netpioneer GmbH, Ludwig-Erhard-Allee 20, 76131 Karlsruhe, Germany
{maximilian.vogel,sebastian.weber}@diva-e.com

Abstract. Web service APIs are central hubs of modern cloud-based application systems. Over recent years, REST has become a de facto standard for their architectural style. Yet in scenarios like mobile apps, flexible client-centric data fetching approaches have emerged as a promising alternative. This gives rise to the question whether RESTful systems can be migrated to a technique like GraphQL and benefit from the new approach. In this paper we report on our experiences during such migration of a real world smart home application. Our observations have underpinned some of the conceptual benefits but also identified challenging aspects where further research is required.

Keywords: Software service architecture · REST · GraphQL

1 Introduction

Many modern service-oriented systems – especially those based on microservice abstractions [14] – embrace the REST-style of resource-oriented distributed software architecture [8]. Beyond offering a simple and solid solution for developers to realize interprocess communication, the REST architectural style leads t some favorable characteristics for distributed systems. Among them are *loose coupling* and *composability* of services as well as *flexibility*, *robustness* and *scalability* of resulting systems.

REST promotes the decomposition of systems into sets of linked resources with a certain level of granularity. This leads to difficult trade-offs between reusability and performance that are well known in general software service architecture (e.g. [4,7]). We generally seek less granular and more cohesive (micro) services fostering loose coupling and high reusability. But this might require cumbersome client server conversations with multiple consecutive requests traversing the resource graph (aka "under-fetching"). The inverse approach of the *Coarse-Grained Remote Interfaces* pattern trades fewer requests with less networking overhead against lower cohesion and reusability [13]. Moreover, this often leads to the transfer of too much data for individual requests (aka "over-fetching").

L. Braubach et al. (Eds.): ICSOC Workshops 2017, LNCS 10797, pp. 283–295, 2018.
https://doi.org/10.1007/978-3-319-91764-1_23

Finally, service granularity might be optimized for specific use cases, but these are hard to reuse triggering an ongoing proliferation of variants.

Beyond architectural quality, another point to consider is the *ease of using service APIs* from a client perspective. As Byron puts it, app developers *"... don't think of data in terms of resource URLs ... (but) in terms of a graph of objects ..."* relevant to their apps [1]. This has led to a number of *declarative data fetching languages* like Facebooks' *GraphQL* [5]. GraphQL enables app developers to run queries against a server API that traverse a graph of application objects and fetch exactly the right amount and format of data in a single request. Such APIs omit over- as well as under-fetching without the need for specialized services, thus yielding performance benefits while being widely reusable.

The potentials of declarative data fetching have recently triggered discussions on whether a technology like GraphQL is an alternative (and thus in contradiction) or a possible extension (and therefore in conformance) to the REST architectural style and its implementing frameworks. This discussion is also relevant for the case of existing RESTful systems with respect to possibilities of their modification in order to benefit from declarative data fetching. To this end, questions arise on two different levels:

- Individual Software Services
  - How can existing RESTful services be transformed or extended to offer declarative data fetching capabilities?
  - What are the performance costs and technical challenges of query execution and are they always justified?
- Service-Oriented Systems and Architecture
  - Are services and systems with declarative data fetching capabilities (still) RESTful and exhibit the related architectural benefits?
  - Which architectural design patterns emerge for such systems?

In this paper we don't seek to finally answer the above questions. Instead, we offer an experience report along with a discussion on the migration of purely REST-based software services to GraphQL. We show an industrial case study that builds on a smart home platform and demonstrate the consequences of migrating parts of its RESTful API to GraphQL. Our results underpin the conceptual advantages of declarative data fetching in practice. Yet, we have identified challenges on different levels that should be considered during migration.

The rest of this paper is structured as follows: In Sect. 2 we first discuss related work. We then give a short introduction to GraphQL in Sect. 3. Based on this background, we present our case study on migrating REST-based systems to GraphQL in Sect. 4. Next, we discuss challenges of migration on the level of service-oriented systems and architecture (Sect. 5.1) as well as individual software services (Sect. 5.2). Finally in Sect. 6, we summarize our work and give an outlook.

2 Related Work

From an academic standpoint, little research has been published specifically about GraphQL yet. Currently, Facebook provides only an informal documentation of GraphQL [5]. Beyond that, Hartig and Pérez have studied the language from a theoretical perspective and provided a formal query semantics [10].

Falcor by Netflix and *OData (Open Data Protocol)* are flexible data fetching libraries and constitute GraphQL alternatives [11,17,18]. Helfer categorizes them as Web API technologies and brings them into a chronological sequence with REST and SOAP [11]. Cupek and Huczala compared OData with REST [3] but not with GraphQL. Falcor as well as GraphQL are JSON-oriented technologies that both emerged in 2015 and act as middleware to query backend data sources [15,16]. At the time of writing, no academic work exists on Falcor. However, several web-based resources provide comparisons of Falcor, OData and GraphQL [12,15,16]. As Meredith concludes, *"The Falcor data model is a graph, and the GraphQL data model is a tree."* [15]. In this paper, we solely focus on GraphQL.

Recently, some student research has emerged on the topic. The thesis by Cederlund is in the field of performance and also compares GraphQL with Falcor [2]. In our own work [21] we have studied how a REST-based backend can be migrated to a GraphQL server and examined the performance implications on different types of requests (i.e. consecutive REST calls vs. single GraphQL queries). In the following sections, we describe our experience with GraphQL in a real world project and provide a comparison of GraphQL with REST.

3 GraphQL

GraphQL is a strongly typed query language, which provides a flexible syntax for describing data requirements and interactions for building client applications [6]. It is not an implementation though, but represents a standard for developing GraphQL server solutions and use it on the client-side. A *GraphQL server* implements the language features and required characteristics defined in this specification [5]. The language itself provides a type system, which specifies the types and expressions that are supported. Each GraphQL server supports different types representing an application-specific type system. The possibilities of a GraphQL server are described by a *schema*, which specifies all supported types and operations. In the context of a given schema, a server can verify that a request is syntactically correct, unambiguous and mistake-free. As shown in Listing 1.1, the GraphQL schema always has one `schema` type that defines the entry point for all client operations.

Listing 1.1. Example of a GraphQL schema

```
schema {
   query : Query,
   mutation: Mutation,
   subscription: Subscription
}

type Query {
 maintenance(id: String): Maintenance
}

type Property{
   id : String!,
   customer: Customer
   ...
}

type Customer {
   id : String!,
   country: String!,
   city: String,
   ...
}

type Maintenance{
   id : String!,
   start: String!,
   property: Property
   ...
}
```

The following *operation types* can be defined in a schema:

Query. The *query type* is the entry point for each client-side request. Within this type, all possibilities for querying data are defined by the inclusion of further object types.

Mutation. If it is necessary to allow the clients to add or modify records, a *mutation type* can be defined as an entry point for respective requests.

Subscription. The *subscription type* offers an entry point for *real-time exchange* of data between clients and the server.

An *object type* represents a grouping of attributes that are referred to as *fields*. Each field contains a name, a type and an optional argument. It can also be specified, if attributes are non-nullable or arrays. The type is either a *scalar type* representing a single scalar value or an *object type*.

To issue a request, a *query document* must be created in a JSON-like syntax. A query document contains multiple definitions of *operations* and *fragments*. Each operation starts with an *operation type*. Depending on the schema, *query*, *mutation* and *subscription* might be available. In an operation (e.g. query), field arguments might be given to select objects and field names must be declared to be included in the results.

Fragments allow to reuse common repeated selections of fields. Fields can be either specified directly or the *spread operator* (...) can be used to include a fragment. All fields defined by a fragment will be added at the same level as the fragment invocation.

Listing 1.2 (left) presents an example of a request, where specific fields of the `Maintenance`-object with the `ID` of `19` are requested. As shown in Listing 1.2 (right), the server returns a JSON object with the requested data.

Listing 1.2. Example query (left) and result (right)

```
{
  maintenance(id: "19") {
    id
   ...
    Property {
      id
      ...
      Customer {
        id
        ...
}}}}
```

```
{
   "data": {
      "Maintenance": {
         "id": "19",
         "Property": {
            "id": "24",
            "Customer": {
               "id": "33"
}}}}}
```

The GraphQL specification offers more aspects of schema definition and the query language. For the sake of giving a first impression, we confine the discussion to the above selection of basic GraphQL features.

4 Migrating a Smart Home App from REST to GraphQL

This section presents the integration of GraphQL into the customer project *SmartHome* of diva-e Netpioneer GmbH. First we introduce the original architecture and present different approaches to integrate GraphQL. Then we show how a single service can be migrated to GraphQL and describe which adjustments had to be made. Subsequently, we present the results of an experimental evaluation that compares the performance before and after migration.

4.1 Original SmartHome System and Architecture

The SmartHome system provides a platform through which IoT devices can be managed. The backend architecture is organized in three layers for *web API*, *services* and *persistence*. Web API and services had been originally implemented as a Java application with Spring Boot[1]. The API layer provides various RESTful services to request or transmit data by a number of different clients.

Listing 1.3 shows parts of the existing REST schema used in the performance analysis (Sect. 4.4). Each of the available resources in the system can be requested using a URI. For example, the resource customer can be requested by the URI `../1.0/customer/{customerID}`. The data model of the REST API represents relationships between resources through primary keys (`id`).

[1] https://projects.spring.io/spring-boot/.

Listing 1.3. Example of the original REST data model

```
Maintenance {
   String id;
   Long start;
   Long end;
   String refProperty;
   ...
}
```

```
Property {
   String id;
   String street;
   String city;
   String comment;
   String refCustomer;
   ...
}
```

```
Customer {
   String id;
   String name;
   String street;
   String zip;
   String city;
   ...
}
```

In addition to an Angular single-page web app[2], various iOS and Android apps as well as IoT devices communicate with the server. In these applications, the views are often composed of several interdependent resources. For example, for a dashboard in the web app, the resources `Property`, `Maintenance`, and `Customer` shown in Listing 1.3 are required. Since the relationships require sequential resolution, several REST requests are necessary. Here, GraphQL is intended to improve the performance and flexibility of data exchange.

4.2 Architectural Considerations

In the following, we will discuss the integration of GraphQL. To use GraphQL, an interface is required, which is provided by a server. The server must be able to process GraphQL queries and return a data set specified by the client. As shown in Fig. 1, a possibility is to use GraphQL as an *API gateway* that can be used to encapsulate access to the external systems [9].

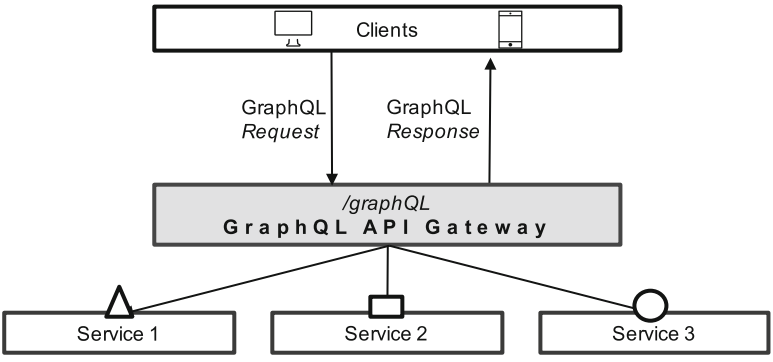

Fig. 1. GraphQL as an API gateway

This approach can be used in a microservice environment for encapsulating internal service interfaces and provide a unified web API. In a monolithic system, GraphQL is often directly integrated as a part of the API layer with direct access

[2] https://angular.io.

to intra-process services of the business layer. This approach has the advantage that no further network requests are necessary for a GraphQL query that could have negative effects on the performance.

4.3 Integration of GraphQL

A requirement for migration was the coexistence of original architectural layers and REST interfaces together with an additional GraphQL interface. For this reason GraphQL was integrated as part of the existing server. As described in Fig. 2, only the API layer had to be expanded.

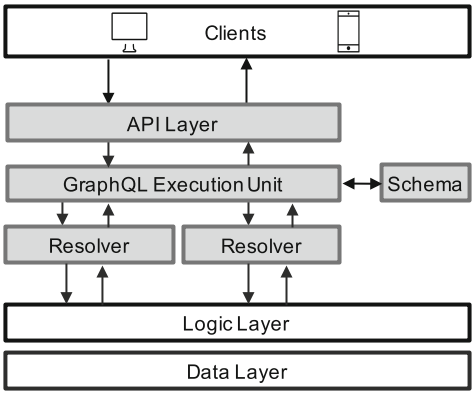

Fig. 2. GraphQL components in the migrated architecture

To this end, the *GraphQL endpoint* can be provided via Spring Boot and is (architecturally) located on the same level as existing REST interfaces. The GraphQL interface was created as a *POST interface*[3] via URI path `/graphQL`. The request payload contains a GraphQL query as specified by the client.

In order to implement the GraphQL specification, the open-source library graphql-java[4] was used. The library provides three components that must be added to the existing system: the GraphQL *execution unit*, a *schema* and *resolvers*. As explained in Sect. 3, the schema describes how types can be queried or modified via the GraphQL interface. In the specified endpoint, a GraphQL query document is passed to a GraphQL execution unit where it is validated against the schema. Each object type in this schema is represented by a resolver. Resolver methods are responsible for getting the required data fields of the query or mutation from the underlying logic layer.

To achieve a homogeneous API the GraphQL schema was derived from the existing REST resources. In the GraphQL schema references between REST

[3] See REST discussion in Sect. 5.1.
[4] https://github.com/graphql-java/graphql-java.

resources were mapped to references between GraphQL object types. Excerpts of the resulting GraphQL schema are shown in Listing 1.1 (corresponding to the REST schema in Listing 1.3).

In the end, integration of GraphQL doesn't affect existing REST interfaces. Parallel operation of REST and GraphQL is possible without restrictions.

4.4 Performance Analysis

In order to evaluate the envisioned performance gains of the GraphQL API compared to the original REST API, we have conducted performance tests. In the following, results of two experiments are presented, each of which consisting of 1000 requests. Results are shown as box plot diagrams (boxes represent 50% of results; upper/lower ends show max/min values, dotted lines represent medians).

In the first test scenario, an atomic resource `customer` was retrieved with three different approaches: 1. request against single REST API endpoint, 2. regular GraphQL query, 3. reduced GraphQL query. In the case of the reduced query, only a single field was queried instead of all fields to determine differences in query size. For all three retrieval approaches, one roundtrip was necessary to fetch the data record. Figure 3 shows the resulting box plot of experiment 1.

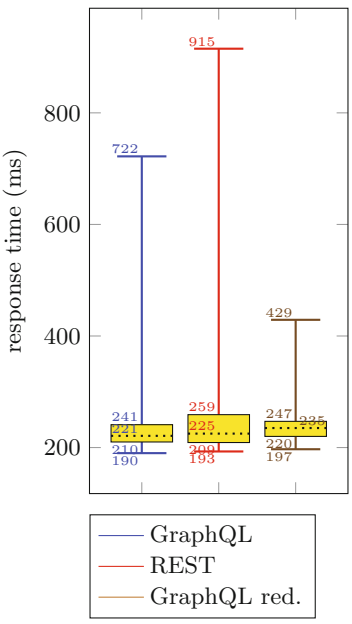

Fig. 3. Experiment 1: retrieving atomic resources

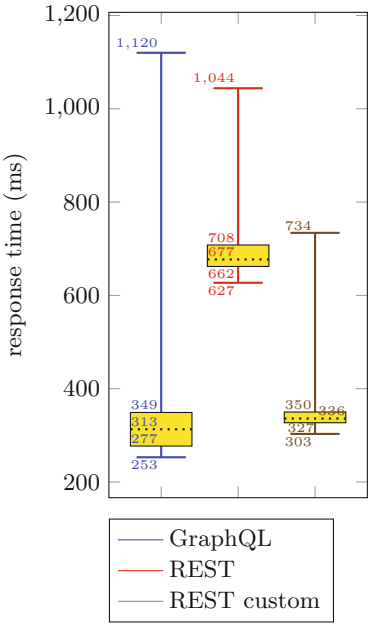

Fig. 4. Experiment 2: retrieving multiple linked resources

The median was 221 ms for performing the GraphQL query, and 225 ms for a request using the existing REST API. Thus, there is no significant difference

between the median of both retrieval approaches. The average query time of the reduced GraphQL query was 235 ms, so no significant difference could be found in the response times either. From this experiment we conclude that *GraphQL is not slower regarding query execution than REST for atomic resources.*

In experiment 2, the resources `Property`, `Maintenance`, and `Customer` depict in Listing 1.3 were retrieved requiring traversal of the original resource graph. The GraphQL approach uses a single query (shown in Listing 1.2) to request all objects from the server. The second variant utilizes the existing REST API to retrieve three resources consecutively one after the other. For the third variant, an optimized REST endpoint was created specifically for this application, which returns the resources in a single response. Figure 4 shows the resulting box plot of experiment 2.

The median was 313 ms for the GraphQL query and 677 ms for consecutive requests using the original REST API. For calling the specific endpoint, the median was 336 ms. Therefore, GraphQL required 46% of the time on average to retrieve all data from the server compared to the REST API. Response times of the specific endpoint were in a similar range than that of GraphQL.

5 Challenges of Migration

In the previous section, we have given a concrete example of migrating a static REST/HTTP API to GraphQL. It shows that the performance of client-server communication can be significantly improved. However, trade-offs had to be made for the migration and some issues have not been considered.

Challenges are caused by conceptual differences between REST and GraphQL on architectural level. These translate to concrete technical issues when creating individual services. Both angles are being discussed subsequently.

5.1 Challenges on Architectural Level

REST is an *architectural style* with general principles resulting in beneficial characteristics on system-level like flexibility, scalability and robustness. It is often interpreted as an implementation based on web standards (HTTP, URI, XML/JSON) leading to systems, which are simple, cheap and interoperable.

When migrating a REST/HTTP API to GraphQL/HTTP, a general challenge is not to break the design principles of the original architecture for the above reasons. Based on the four principles given by Pautasso et al. [19], we therefore analyze the compatibility of GraphQL with REST.

Resource Identification through URIs. REST encourages versatile resource-oriented architecture where self-contained cohesive resources are individually addressable by means of URIs. GraphQL on the other hand promotes a more data-centric model without architectural resources. A GraphQL service represents an *object graph* of data entities. These objects are collectively accessible through a single endpoint and URI.

Uniform Interface. REST APIs take full advantage of HTTP verbs and semantics (at least GET and POST should be considered for distinguishing idempotent operations). GraphQL introduces a high level query protocol on top of HTTP with individual operations (*query*, *mutation*, *subscription*). GraphQL requests might be alternatively mapped to GET or POST operations regardless of HTTP semantics.

Self-Descriptive Messages. REST promotes HTTP content negotiation, usually in conjunction with common formats like XML, JSON or HTML and HTTP metadata for resource representation. GraphQL prescribes a fixed query language for requests but is agnostic to the response (serialization) format (JSON is commonly used). Still the logical structure of a response is restricted by the specification and mirrors the intend of GraphQL as a data-fetching technology.

Stateful Interactions through Hyperlinks. Interactions with single REST resources are stateless and require self-contained messages. Conversation state is advanced by following hyperlinks that are transferred as part of resource representations in the former response. GraphQL interactions are stateless as well but are usually not meant as part of a stateful conversation. Traversal of the object graph happens during execution of a single query for the sake of collecting individual result sets.

From the above discussion it becomes obvious that there are severe conceptual differences between the general models and practical realizations of REST and GraphQL. Consequently, a *substitution* of former REST resources by means of GraphQL services is likely to break the design principles of the original architecture. In particular, substituting the entire resource graph by a monolithic GraphQL service would be clearly invasive.

A more promising migration approach is to complement the original resource graph by additional endpoints representing subsets of linked resources. Still such complementary GraphQL services should be carefully designed, i.e. with respect to the selection of appropriate resources that lend themselves to data fetching as well as their granularity.

5.2 Challenges on Service-Level

Conceptual differences of GraphQL compared to the original REST architecture shows in a number of issues with respect to the implementation of GraphQL service endpoints. Degeneration of the HTTP-level as regards addressing of individual resources or semantics of operations leads to typical problems of utilizing common techniques for caching, fault tolerance or authorization. Some additional issues emerge on the level of the GraphQL protocol like new security threads. In the following, we will focus on caching and DoS threads.

Caching. GraphQL objects are not addressable via URIs. Thus, caching behavior of GraphQL queries cannot be specified by HTTP headers and generic HTTP

caching can't be applied. Instead, we need specific GraphQL caching that utilizes *object IDs* within queries. It is important to note that *globally unique IDs* are required for this purpose [6].

We might implement dynamic content caching by means of various approaches, depending on query and update behavior [20]. Full or partial *replication of data* improves situations with complex queries and infrequent updates. *Content-blind caching* hashes normalized queries as result keys. It works well for repeated queries but might pollute the cache otherwise. This approach works on HTTP-level, if GraphQL requests use HTTP GET with queries encoded as URL query parameters. *Content-aware caching* evaluates queries and locally stores resulting objects for a limited set of query templates. It is a trade-off between cache size and range of supported queries but puts more load on the cache for query evaluation and requires more involved consistency mechanisms.

As regards placement of cache content, it might be put on the GraphQL server (to avoid accessing remote data stores), dedicated proxies or the client-side. For the case of web apps, GraphQL clients such as Apollo[5] or Relay[6] already offer browser-located caching out of the box.

Denial of Service Attacks. GraphQL enables clients to submit individual queries to the server for execution. This feature might be exploited by perpetrators generating excessive load through overly complex (single) queries possibly leading to service failure. Therefore, we need to introduce countermeasures securing GraphQL APIs against malicious queries.

A naive approach against excessive requests is to check the *size of the query representation* before execution and restrict it to a certain limit. The GraphQL feature of *persistent queries* offers another simple way of protection. Here, all permitted query templates are stored on the server. Clients might only execute pre-defined queries, which can certainly been seen as benefit and drawback at the same time.

More sophisticated approaches introduce cost metrics quantifying the effort of query execution. *Resolver restriction* builds on the weighted number of resolver calls for a query or, alternatively, the duration of resolver execution. If a request exceeds some given threshold, the resolver interrupts it. Obviously, harmless queries might be killed and malicious ones still run for some time.

A better way would be to statically analyze queries prior to execution and disregard them in case of excessive complexity. Hartig and Pérez have shown that queries might lead to resulting object graphs of exponential size [10]. Luckily they also suggest that the size of a result object can be estimated with realistically low complexity. This would provide the theoretic foundation for effective DoS prevention to be build into GraphQL server frameworks.

[5] https://github.com/apollographql/apollo-client.
[6] https://facebook.github.io/relay/.

6 Conclusion and Outlook

In this paper we have studied the migration of distributed (e.g. web-based) apps from RESTful client-server interactions to GraphQL. We have compared the performance of complementary web APIs in the context of an industrial case study and demonstrated the benefits of the data fetching approach with GraphQL. While we restrict the study to a minimal part of the API, we consider the behavior of the real world backend (business and persistence layers) as representative for the performance measures.

Integrating complex semi-compliant technologies in the context of distributed systems naturally introduces challenges. Even the individual technologies themselves usually leave room for interpretation beyond their given rules or specification. At this point, best practices need to fill these gaps.

If generic solutions are found, *patterns* can be deduced. Some patterns have already emerged for GraphQL like *Connections* for pagination or *Globally Unique Object IDs* for caching. We expect more patterns to emerge in the architectural space and related to the integration of GraphQL with REST/HTTP. The ultimate goal would be to provide a systematically structured set of patterns as part of a more general pattern language for web APIs.

References

1. Byron, L.: GraphQL: A data query language. https://code.facebook.com/posts/1691455094417024/graphql-a-data-query-language/. Accessed 09 June 2017
2. Cederlund, M.: Performance of frameworks for declarative data fetching: an evaluation of Falcor and Relay+GraphQL. Master's thesis, KTH, School of Information and Communication Technology (ICT), Stockholm, Sweden (2016)
3. Cupek, R., Huczala, L.: OData for service-oriented business applications: Comparative analysis of communication technologies for flexible Service-Oriented IT architectures. In: 2015 IEEE International Conference on Industrial Technology (ICIT), Seville, pp. 1538–1543. IEEE (2015)
4. Erl, T.: Service-Oriented Architecture: Concepts, Technology, and Design. Pearson Education, Upper Saddle River (2005)
5. Facebook Inc.: GraphQL - Working Draft. http://facebook.github.io/graphql/. Accessed 11 Apr 2017
6. Facebook Inc.: Introduction to GraphQL. http://graphql.org/learn/. Accessed 19 Apr 2017
7. Feuerlicht, G., Lozina, J.: Understanding service reusability. In: 15th International Conference Systems Integration, Department of Information Technologies and Czech Society for Systems Integration, Prague, Czech Republic, pp. 144–150 (2007)
8. Fielding, R.T.: Architectural Styles and the Design of Network-based Software Architectures. Ph.D. thesis, University of California, Irvine (2000)
9. Fowler, M.: Patterns of Enterprise Application Architecture. Addison-Wesley Longman Publishing Co., Inc., Boston (2002)
10. Hartig, O., Perez, J.: An initial analysis of Facebook's GraphQL language. In: Proceedings of the 11th Alberto Mendelzon International Workshop on Foundations of Data Management (AMW), Montevideo, Uruguay, 5–9 June (2017)

11. Helfer, J.: GraphQL - Evolution or Revolution? https://speakerdeck.com/helfer/graphql-evolution-or-revolution. Accessed 27 June 2017
12. Helfer, J.: GraphQL vs. Falcor. https://dev-blog.apollodata.com/graphql-vs-falcor-4f1e9cbf7504. Accessed 17 July 2017
13. Larman, C.: Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development. Prentice Hall PTR (2004)
14. Lewis, J., Fowler, M.: Microservices. https://martinfowler.com/articles/microservices.html. Accessed 17 July 2017
15. Meredith, C.: The Falcor data model is a graph, and the GraphQL data model is a tree. https://edgecoders.com/the-falcor-data-model-is-a-graph-and-the-graphql-data-model-is-a-tree-6748ba53bb96. Accessed 03 Aug 2017
16. Miller, D.: graphql (facebook), falcor (netflix) and odata and ... http://appddeevvmeanderings.blogspot.com/2016/02/graphql-facebook-falcor-netflix-and.html. Accessed 17 July 2017
17. Netflix Inc.: Falcor: One Model Everywhere. https://netflix.github.io/falcor/. Accessed 17 July 2017
18. OData: OData - the Best Way to REST. http://www.odata.org/. Accessed 17 July 2017
19. Pautasso, C., Zimmermann, O., Leymann, F.: Restful web services vs. "big" web services: making the right architectural decision. In: Proceedings of the 17th international conference on World Wide Web, pp. 805–814. ACM (2008)
20. Tanenbaum, A., Van Steen, M.: Distributed Systems: Pearson New International Edition: Principles and Paradigms. Pearson Education Limited, Upper Saddle River (2013)
21. Vogel, M.: Potential von GraphQL in dynamischen Webanwendungen. Bachelor's Thesis, Karlsruhe University of Applied Sciences, Karlsruhe, Germany (2017)