Microservice architecture patterns with GraphQL

Ville Touronen

Helsinki March 24, 2019

UNIVERSITY OF HELSINKI
Department of Computer Science

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Koulutusohjelma — Studieprogram — Study Programme
Faculty of Science	Computer Science

Tekijä — Författare — Author
Ville Touronen

Työn nimi — Arbetets titel — Title

Microservice architecture patterns with GraphQL

Ohjaajat — Handledare — Supervisors
Jussi Kangasharju, Matti Luukkainen

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
M.Sc. Thesis	March 24, 2019	55 pages

Tiivistelmä — Referat — Abstract

In this thesis the GraphQL query language was studied in the context of the microservice architecture. The thesis was a combination of a literary study and expert interviews. Most prevalent microservice architecture patterns were gathered from peer reviewed studies and grey literature. Four expert interviews were held to detect which patterns were used in projects which used a microservice architecture and GraphQL APIs.

Based on the interviews, a typical GraphQL microservice architecture consists of up to 10 REST or GraphQL microservices exposed to public networks using a GraphQL API gateway. The studied microservice architectures typically had a flat structure where a GraphQL API gateway exposed the other microservices in a protected network, which used a persistence layer to store data. Most of the microservice architectures seemed to rely on synchronous communication patterns, using a platform provided service discovery mechanisms to implement load balancing. The most prominent authorization mechanism was to use web tokens via an authorization header.

An aggregating GraphQL service seemed to work well as an API gateway layer, which exposes the functionality of underlying microservices to clients. A pattern called schema stitching was succesfully used in some of the interviewed projects to automatically combine multiple microservice GraphQL schemas together. Overall the interviewed developers had a positive experience using GraphQL in their projects. The API exploration tools and built-in documentation enabled by the GraphQL schema introspection functionality made client application development easier.

ACM Computing Classification System (CCS):
Computer systems organization → Architectures → Distributed architectures
Information systems → World Wide Web → Web services

Avainsanat — Nyckelord — Keywords
GraphQL, microservice architecture, web services, REST

Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — övriga uppgifter — Additional information

Thesis for the Networking and Services subprogramme

Contents

1 Introduction

Microservices are a modern service-oriented architecture (SOA) related software development approach where application logic is divided and implemented as separate functional domains, instead of implementing all application logic in a single monolithic application. Compared to SOA services, microservices are typically considered to be more lightweight, less coupled and more fine-grained [Mir]. Microservices are self-contained and can be separately deployed and removed, whereas SOA architectures might be more tightly coupled and depend on a shared messaging layer such as the Enterprise Service Bus (ESB) or other infrastructure services, like service discovery. SOA based services are more coupled into the service's ecosystem compared to microservices, which typically can have their own deployment and testing pipelines from the rest of the microservices. The popularity of microservices have been increasing and it is used by many known companies such as Netflix and Soundcloud [AAE16].

GraphQL is a novel API query language created by Facebook and is an alternative to using typical JSON or XML API endpoints in a server application. The GraphQL request determines which data is returned by the GraphQL server, whereas a JSON endpoint would typically return a fixed predetermined selection of data. The query language's key selling points are being client-centric, developer friendly and reducing roundtrips to the server by making it possible to request all the needed data in one request [Sch15]. Also, only the requested fields are returned, which reduces unnecessary network traffic and server resource usage.

Architecture patterns are reusable solutions to specific architecture or design problems in software development [Ric18, p.20]. The patterns solve a problem by structuring the application in a specific way, or using a specific strategy to fulfill functional or quality requirements. Applying architecture patterns might cause new issues in the application that need to be solved by other architecture patterns. There typically isn't a superior set of patterns that should be always used, but rather the patterns are used to solve specific issues caused by the application's structure or requirements.

The purpose of this thesis is to study architecture patterns usable in the context of GraphQL and microservices. Typical microservice patterns are discussed and evaluated. Also GraphQL related patterns such as schema stitching are discussed and evaluated when applied in the context of the microservice architecture.

Chapter 2 presents the methodology and research questions of the thesis. In chapters 3 and 4 the microservice architecture and GraphQL specification are covered as the basis of the thesis. In chapter 5 typical microservice architecture patterns are gathered from scientific studies and grey literature, such as conference presentations and technical articles. In chapter 6 possible GraphQL related patterns are presented. The chapter 7 lays out the interview method used in this thesis. The results of the interviews are presented in chapter 8. The interview results are compared to research findings in chapter 9. Lastly, recommendations are suggested in chapter 10 based on the findings.

2 Methodology

In this thesis literature sources are researched for any common architecture patterns, which are used in the context of microservices and GraphQL. Also, semi-structured interviews are used to study the incidence of the presented architecture patterns in commercial products. In each interview a past or ongoing project of the interviewee is discussed. A qualitative analysis of the interviews is done to evaluate the incidence of covered patterns, and their shortfalls and benefits if those were mentioned in the interviews. Current best practices from literary sources and interview data are combined to make a recommendation on how to structure a microservice architecture with GraphQL interfaces.

The research questions of the thesis are the following

- **RQ1**: What is the typical architecture layout in a microservice architecture project using GraphQL?
- **RQ2**: What is the overall developer experience in real world microservice projects using GraphQL?
- **RQ3**: Does GraphQL impose any limitations in a microservice architecture compared to alternative API methods?

3 Microservice architecture

Service oriented software development is a style of software development which splits application's business logic into separate autonomous applications, called *services*

[Fow14]. *Services* implement a piece of the business domain, typically one business capability, and they can be used by other clients or services over the network [CDP17]. Microservices architecture is a fairly new trend in service oriented software development, which promotes high isolation in microservice's infrastructure and deployment pipeline. A microservice should work independently from other microservices, which is achieved with loose coupling by using synchronous remote procedure calls (RPC) or asynchronous messaging to handle inter-process communication (IPC). The microservice should own their data model, which makes it possible to have separate persistence layers of each microservice, deployed in separate databases.

3.1 Comparison to Monolithic architecture

Microservice architecture is a software architecture which is used to implement a web application using loosely coupled single-purpose services [BHJ+]. Microservice architecture might be selected over a *monolithic architecture*, which has all the business logic implemented in a single software package [Hun17, p.1]. The benefits of using a monolithic architecture is its simplicity. At smaller sizes the monolithic application is generally easier to develop and the deployment is easier because only one package has to be deployed. Also scaling is to some extent easier because only one software package has to be deployed multiple times and the whole application stays functionally the same. There's no inter-process communication that should be taken into account when scaling a monolith. Integration testing is simpler than in a distributed system, and also the simpler architecture makes drastic changes possible into the system [Ric18, p.4].

In a comparison of the two architectures, an application with microservices will have separation of concerns by nature [Hun17, p.1]. Also each service in an application can be developed with different set of tools, such as databases, programming languages, monitoring, testing and build pipelines. Other reasons to split a monolithic application into many microservices is to cut costs when using modern pay-by-usage cloud platform-as-a-service (PAAS) providers such as AWS or Azure. Splitting to microservices enables scaling only those parts of the application that need scaling instead of scaling the whole monolithic application [BHJ+]. Also different modules can have differing computational requirements – one module might have special memory requirements as another module might use CPU a lot. The breaking of modules into different microservices makes it possible to tailor the servers exactly

to their needs, which also cuts costs [Ric18, p.6]. Implementing applications using microservices also prevents vendor lock-in [BHJ+], because the service as a whole is less dependent on one service provider, programming language or framework. A monolithic application on the other hand will have the same language, conventions and tools everywhere which might make its functions easier to develop across software teams. Although adding new features might be easier across domain teams in a monolithic application, the deployment might be more difficult because the whole application has to be tested and working instead of just deploying the changed parts of the application.

DevOps related concepts like *continuous integration* (CI) suits microservices well. Because one microservice's scope is limited to a small set of functions, they are more easy to test in isolation. Continuous integration means that changes are applied to the main code branch in short development cycles while keeping the whole application functional. Continuous deployment is also easier with microservice architecture. Continuous deployment means that the application is deployed automatically when changes are added into the main code branch. DevOps and organizational patterns are covered in more detail in chapter 5.6.

Microservices will typically be split so that one service handles tasks related to one entity in the business domain [Hun17, p.4]. Microservices might be split even futher if a specific functionality is important enough. Because of the added complexity that intercommunication between microservices adds to the application, testing the whole system becomes more difficult because network issues such as latency and availability have to be taken into account [Ric18, p.17]. In a monolithic application communication works simply through function calls, which is simple and fast but it results in greater coupling across the application. It is recommended to use a monolithic architecture if the application domain is small and it won't need to be extended in the future. Monolithic architecture can also be used as a starting point for a microservice architecture. Fowler calls this approach the *Monolith-first* pattern [Fow]. Using the Monolith-first pattern allows the team to study the business domain before investing into a more complex microservice architecture.

There are drawbacks to using microservice architecture other than increased testing complexity. One is the difficulty of splitting the business domain in a sensible way. The domains have to be non-dependant from each other so the microservices can be deployed separately and so they are only loosely coupled. If the microservices are dependant from each other, the whole architecture has to be deployed as one and can

be thought as a *distributed monolithic application* [Ric18, p.17]. The third drawback is the difficulty of deploying new features which span across multiple microservices, because of the added complexity of microservice intercommunication and also the timing of deploying all the updated microservices.

3.2 Comparison to Service-oriented Architecture

Service-oriented architecture (SOA) is a broad term with no clear definition [CDP17]. Common characteristics of SOA and microservice architecture are building an application using separate composable services. Services are invoked using remote procedure calls. A SOA uses an enterprise service bus (ESB) and heavier protocols for communication with other services [Mir]. Also SOA services can share the persistence layer whereas microservices should have their own databases. Sharing a service bus for communication and databases makes SOA services more coupled. Cerny et al. describes SOA's and microservice architecture's main differences as where the application's business rules are implemented [CDT18]. In SOA the business rules are applied through the shared ESB and other platform services and the SOA services are kept simple. In microservice architecture the business rules are applied and enforced by the microservices themselves, and the communication layer is kept simple. Having the business rules applied through the communication layer requires heavier protocols [CDT18]. Microservices are expected to use light messaging protocols such as REST [DRS17]. Sources on the definition of SOA are scarse and vague at best [CDP17], whereas there seems to be more clear definitions available for the microservice architecture [Fow14, Ric18].

3.3 Inter-process communication

Because application's functionality and data is scattered across multiple services, inter-process communication has to be used to fulfill requests that need data from other services. Communication strategies between microservices can be divided into two categories: synchronous and asynchronous [Pac18]. The synchronous one-to-one communication is called request-response communication and can be implemented with protocols such as HTTP, gRPC and Apache Thrift. Synchronous communication means that the client that made the request will wait for the response before continuing code execution. Asynchronous communication methods don't stop code execution when the request is sent. The result of the request has to be polled from

the server afterwards separately. Typically asynchronous communication methods are used for triggering tasks on other microservices that don't need an immediate response or the task needs a lot of time to complete, for example generating a file. A message broker such as RabbitMQ or Apache Kafka is often used to relay asynchronous messages between microservices.

3.4 CAP theorem, microservice availability and consistency

CAP theorem is an idea presented by Eric Brewer which states that a distributed database can only have two qualities from the following three: data **consistency**, **availability** and tolerance to network **partitions** [Bre12]. Network partitioning means that some operations happen only locally because a database is unavailable, which means that the database operations are eventually applied to the database, when the database is available again. When a network partition happens either availability or consistency has to be sacrificed. If availability is prefered, it means that the data across the systems will become inconsistent until recovery from network partition has happened. If consistency is prefered the service has to become unavailable until the network partition has ended. Blocking all requests will keep the data consistent, but availability has been sacrificed. CAP therem can also be applied to microservice architecture because of the distributed nature of the application logic and data storage [Ric18, p.113-114]. For instance CAP theorem can be applied to IPC when choosing from synchronous or asynchronous communication protocols. Synchronous communication keeps data consistent because the request waits for the data to be returned. If cascading synchronous data fetches happen for the request, e.g. other data is requested from other services to fulfill the original request, the services become even more unavailable in favor of consistency. Richardson states that availability is usually favored in microservice architectures [Ric18, p.114]. To make services more available, asynchronous communication and *eventual consistency* should be used where possible [Ric18, p.103]. Eventual consistency means that if data writes span across microservices, the changes are applied over time to enable faster response times [Vog09]. Abadi describes an extension to the CAP theorem called PACELC, which adds latency as a variable in to the mix [Aba12]. Abadi argues that even without network partitions, a trade-off exists between consistency and latency in a distributed system. Availability and latency can also be linked together, where unavailable service essentially is a service with a very long latency. Abadi states that to achieve high availability and low latency,

data has to be replicated in a distributed system [Aba12]. Richardson also presents a microservice architecture specific solution where a local copy of the data is stored from other services [Ric18, p.105]. A microservice subscribes to events of other services to keep their local copies of external entities up to date. With replicating data there's a drawback of potentially duplicating large amounts of data across services.

3.5 Research

Two systematic mapping studies were found on microservice architectures [AAE16, TLP18]. Alshuqayran et al. studied the incidence of *challenge* topics which have to be addressed when using a microservice architecture [AAE16]. The most commonly covered topics were related to deployment and communication. Other challenges covered in the research material were related to service discovery, fault-tolerance, performance, security and logging. Case studies, solution proposals and validation of solution proposals were found for both deployment and communication. The rest of the topics had mostly just solution proposals with no validation studies. Alshuqayran et al. states that studies in microservice architecture is still in its infancy and important topics such as security are largely uncovered [AAE16]. A newer mapping study by Taibi et al. on microservice architecture patterns supports these claims as they found that the most studied pattern categories were related to communication and coordination, deployment, and patterns related to data storage [TLP18].

4 GraphQL

GraphQL is an API query language specification designed by Facebook. Facebook has also built a reference implementation in Javascript which is widely used as a basis for other GraphQL implementations [Facc]. This section will explain the essential properties of the GraphQL specification.

4.1 Description

GraphQL is an API query language [Fac15b] which can be used to describe an arbitrary selection of objects and their fields that a GraphQL server should return. [Fac18a] The query is parsed and resolved by a GraphQL server, and data is returned

to the client that made the query. GraphQL specification defines the query language, but it doesn't define how the query should be resolved. The query resolving can happen in any way and it differs between GraphQL implementations. A GraphQL implementation is typically a small layer in the server that only parses the query and calls data resolvers accordingly.

As pointed out by one interviewee, GraphQL language is usually associated with the HTTP transport protocol, but the language can be used with any transport protocol which can delivery the query to the GraphQL server. Examples of using GraphQL over WebSockets or other inter-process communications exist [Apo19c].

4.2 Specification

The first GraphQL specification was released in 2015 [Fac15b, Fac15a]. The GraphQL implementation used at Facebook predates the specification by three years and was in production use [Sch15]. The design principles for the GraphQL specification were to provide a client-first API query language, where the developer can freely define which data is fetched from the backend [Sch15]. The latest specification was published June 2018 [Fac18b, Fac18a] and it is the first stable release of the GraphQL specification. The specification defines the language and its grammar, type system, introspection system, and describes the execution and validation engines that can be used to resolve the queries.

4.3 Language

A request, referred as a *document* in the specification, can contain operations. By specification operations are called either *queries* or *mutations* [Fac18a], but other operations also exist in GraphQL implementations such as the *subscription*. Roughly, *queries* are operations that fetch data and *mutations* are operations that mutate the application's state or data. *Subscription* is an example of a library specific operation, which describe queries that push data to the client when changes happen. A typical way to implement *subscriptions* would be to use the WebSocket protocol, which supports pushing data to the client. The request document can also contain *fragments* which can be used in the aforementioned operations to define a set of fields that should be resolved by the server. Fragments are used to define reusable parts of requests such as interfaces which are implemented by GraphQL objects. Interfaces are explained in section 4.5.4.

The document consists of lexical tokens which can be *Punctuators* that describe the document's structure, *Names* which can be operation or field names, and *Values* of integer, float, and strings.

Listing 1: A simple GraphQL request document

```
query GetBook {
  book(isbn: "9780345391803") {
    title
    pageCount
    synopsis
    releaseDate
  }
}
```

Listing 2: A GraphQL JSON response

```
{
  {
  "data": {
    "book": {
      "title": "The Hitchhiker's Guide to the Galaxy"
      "pageCount": 224
      "synopsis": "One Thursday lunchtime Earth is unexpectedly der
      "releaseDate": "1979-1-1"
    }
  }
}
```

Listing 1 shows a simple example of a GraphQL document. The document has a named query "GetBook". The document describes a query with a *SelectionSet* of a book with an argument *isbn*. SelectionSets are denoted with curly brackets. The book also has a *SelectionSet* which selects *fields* from the book object. If a field is a complex object such as the book in the example, it has to include a *SelectionSet* to choose which values are returned by the server. Listing 2 shows a response to the listing 1

Listing 3 is an example of a more typical request document with a reusable *fragment* and *variables*. Variables are denoted with '$' characters and are followed by a type declaration. The character '!' means that the variable is required and cannot be a null value. Variables are used to parametrize operations so the operation can be reused on client-side without building a separate request document string for every request.

Listing 3: GraphQL document example

```
query GetBook($isbn: String!) {
  book(isbn: $isbn) {
    title
    pageCount
    synopsis
    releaseDate

    author {
      ...personFields
    }

    editor {
      ...personFields
    }
  }
}

fragment personFields on Person {
  name
  imageUrl
}
```

Behaviour of resolving field data can be altered with directives. Directives can be used to specify additional information about field's format, conditional field resolution, field skipping for instance. Listing 4 shows an example of using a directive specifically to include a field when a variable is set to be true.

Listing 4: Example of using directives

```
query GetBook($isbn: String!, $includeSynopsis: Boolean) {
  book(isbn: $isbn) {
    title
    pageCount
    synopsis @include(if: $includeSynopsis)
    releaseDate
  }
}
```

4.4 GraphQL Schema

GraphQL specification includes a type system language which describes GraphQL server's *schema* [Fac18a]. The GraphQL schema tells which queries, mutations,

types and directives exist on the server. The schema can include inline documentation by using multiline strings next to the documented types. The inline documentation is then used by the developers who will develop their application against the implemented GraphQL server. The schema can also be used for instance by client libraries for code-generation and by servers for generating stub functions for query resolvers. GraphQL schema can also describe extensions on another schema which might come from an another API. These schema extensions are one feature that can be used when combining microservices with *Schema Stitching*, and will be discussed later in this thesis.

4.5 Types

GraphQL type system consists of objects and scalar types which are defined in the application's GraphQL schema.

4.5.1 Scalars

GraphQL specification has primitive types called Scalars which can be integers, floating point numbers, booleans, and strings [Fac18a]. The schema can also define custom Scalars that resolves into a string, but represents for instance a date. The GraphQL response's leaves are always Scalars, so they always resolve into a basic primitive type. The specification describes strict rules on type coersions, e.g. how scalars can be typecasted when resolving a value. Primitive values can also be represented as Enums, if there's only a limited amount of values which the primitive can resolve into.

4.5.2 Objects

Objects are entities that hold other Objects, Scalars, or Enums. Objects form the hierarchical structure of the GraphQL schema [Fac18a]. Objects contain named fields that represent other Objects, Scalars, Enums or Unions. Fields can be named anything other than strings starting with double underscore '__', which are reserved for the schema introspection system in GraphQL.

Listing 5: GraphQL Object example

```
type Book {
  title: String!
```

```
  pageCount: Int
  synopsis: String
  releaseDate: Date
  author: Person
  editor: Person
  related: [Book]
}
```

Listing 5 shows an example object schema where 'title', 'pageCount', 'synopsis', 'releaseDate' are scalars, so they will be the GraphQL response's leaf nodes. Fields 'author' and 'editor' are other objects so they act as intermediate nodes in the response data. 'Related' field is a *List* of other Book types. GraphQL specification allows recursive references in the type definitions [Fac18a]. When objects are requested from the GraphQL server, the specification states that the fields should appear in the response in the same order as they were requested. Objects can also be used as variable input values or field arguments.

4.5.3 Nullable types

GraphQL field's value can be null by default, unless specifically denoted as a required field [Fac18a]. In Listing 5 field 'title' is marked as a required field. If the field is queried from any Book type, a non-null value is guaranteed to be returned from the server.

4.5.4 Interfaces

GraphQL specification supports declaring *Interfaces* which objects can implement [Fac18a]. Listing 6 shows an example of using interfaces to define shared fields between objects. The listing shows two interfaces which are both implemented on Book and Article types. The implementing types define their own fields on top of the interfaces. Listing 7 shows an example query for an 'entity' which returns EntityInterface typed object. To add more fields to the query which reference the implementing objects and other interfaces, the query has to use the dot notation shown in the listing.

Listing 6: Interface definition example

```
interface TitleInterface {
  title: String!
}
```

```
interface EntityInterface {
  id: ID!
}

type Book implements EntityInterface & TitleInterface {
  id: ID!
  title: String!
  isbn: String
}

type Article implements EntityInterface & TitleInterface {
  id: ID!
  title: String!
  doi: String
}
```

Listing 7: Interface query example

```
query GetEntity($id: ID!) {
  entity(id: $id) {
    id
    ... on TitleInterface {
      title
    }
    ... on Book {
      isbn
    }
    ... on Article {
      doi
    }
  }
}
```

4.6 Introspection

GraphQL specification also defines an introspection schema that can be used for querying the schema of a GraphQL server [Fac18a]. The introspection schema has a __Schema type which contains all the types and directives of the server. A __Type field is also available which can be used for querying information about a single field. The introspection functionality of GraphQL can be utilized for instance in developer tools that expose the API schema in an easily readable format. In figure 1 a screenshot of the GraphiQL tool is presented. On the left side a query

can be written which can be executed on the API. The result or errors are output on the middle column. In the example a film is queried with a hashed ID. The fields *id*, *title*, *director* and *publishDate* are used as the selection set for the film. A JSON object is returned by the API of the movie "New Hope" with the selected fields. On the right side a documentation column is present where the schema can be explored. For instance in this example the entity *Film* is open to see which fields the *Film* entity exposes.

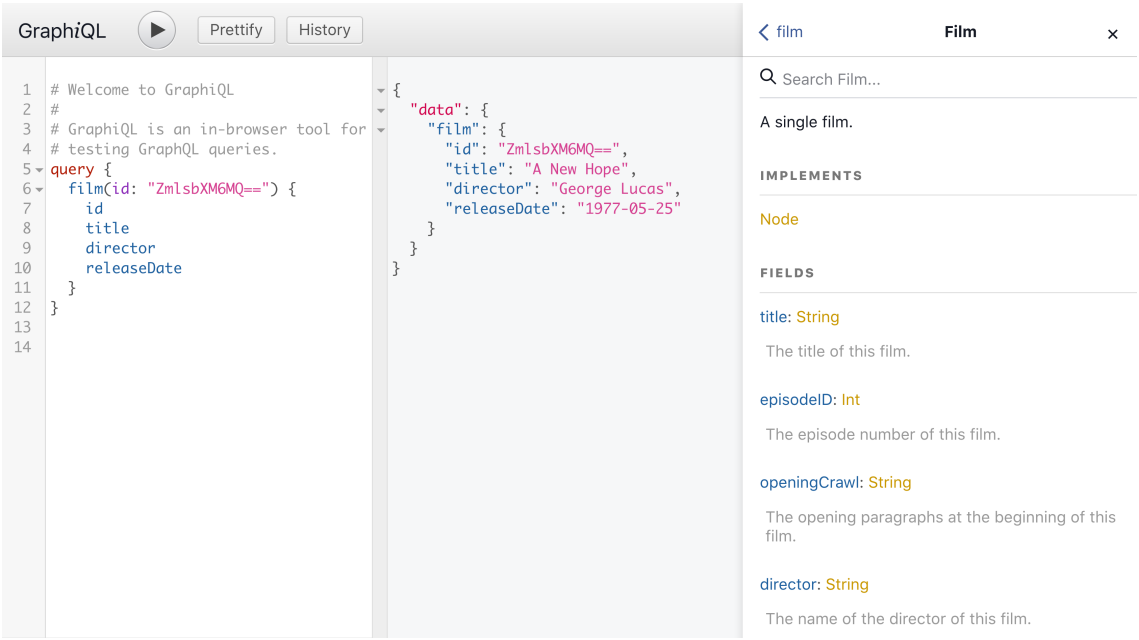


Figure 1: GraphiQL tool for inspecting GraphQL APIs

4.7 Query execution

For executing a request the GraphQL server needs a schema and a request document at minimum [Fac18a]. If variables are used in the request document, the request should also include all the non-optional variables required by the request document. When fulfilling the request, all validations against the server's GraphQL schema should pass. This means that all requested fields and their types are defined in the request document the same way as defined in the schema. If validations don't pass, the request should return a list of errors and completely fail the original request. Queries should select and return data from the server based on the fields defined in the request document. Fields at same level can be resolved in parallel, but the ordering in the result has to be the same as in the request document.

Mutations typically perform side-effects in the server. After the side-effects are done, a selection set is returned from the mutation's return type. A good convention is to return the modified object nested in a special mutation result type, which can include other metadata fields later on if needed [Mer]. If a subscription is supplied in the request document, a connection is established between the client and server and new items in the event stream are sent to the subscriber.

4.7.1 Schema extension

The GraphQL specification defines extension types for Objects and Interfaces [Fac18a]. Schema extension can be used for extending local object types or interfaces, or types in a remote schema. An example usage of schema extension is *schema stitching* which combines multiple local or remote schemas together into one single schema. Schema stitching is covered in more detail in chapter 6.2.1. Another example would be to extend existing remote schema with new local functionality.

4.8 GraphQL server implementation

A GraphQL reference implementation is provided by Facebook, which is implemented in Javascript [Gra19]. The reference implementation accepts a type schema object which includes *resolvers* for all leaf nodes of defined GraphQLObjectsTypes. *Resolving* data is a typical term used in GraphQL implementations and specification, which means returning a value or object for a given field. Thus, functions that return the values or objects for field definitions are called *resolvers*. Listing 8 shows an minimal setup for a GraphQL server that returns book objects based on their id. In the example the root query schema is constructed with one GraphQLObjectType *bookType*. The imported *getBook* function is expected to match the schema definition of the bookType. Using a static type checker such as flow-typed or a javascript dialect like TypeScript can help with matching the type definitions of the schema and resolvers.

Listing 8: A minimal example of using the reference implementation in Javascript ES6 syntax [Gra19]

```
import {
  graphql,
  GraphQLSchema,
  GraphQLObjectType,
  GraphQLString
```

```
  GraphQLInt
} from 'graphql';

import {getBook} from './models/book'

const schema = new GraphQLSchema({
  query: new GraphQLObjectType({
    name: 'Query',
    fields: {
      book: {
        type: bookType,
        args: {
          id: {
            description: 'internal id of the book',
            type: GraphQLNonNull(GraphQLString),
          },
        },
        resolve: (root, { id }) => getBook(id),
      }
    }
  })
});

const bookType = new GraphQLObjectType({
  name: 'Book',
  description: 'A book with an ISBN code',
  fields: () => ({
    id: {
      type: GraphQLNonNull(GraphQLString),
      description: 'The internal identifier of the book',
    },
    title: {
      type: GraphQLString,
      description: 'Book title',
    },
    synopsis: {
      type: GraphQLString,
      description: 'Synopsis',
    },
    isbn: {
      type: GraphQLString,
      description: 'ISBN code',
    },
    pageCount: {
      type: GraphQLInt,
```

```
      description: 'Page count',
    },
  }),
});

export const BookSchema = new GraphQLSchema({
  query: schema,
  types: [bookType],
});
```

4.9 Alternatives

GraphQL is commonly compared to API implementations using the SOAP protocol or the REST architectural style.

4.9.1 REST, Representational State Transfer

GraphQL and REST APIs are considered as alternative solutions for each other. Representational State Transfer (REST) is an architectural style defined by Fielding in his doctoral dissertation [FT00]. REST architecture style maps domain entities as resources, identified by resource identifiers. REST architecture can be applied to any transport protocol, but REST is often associated with the HTTP protocol using URIs as resource identifiers. A common way to apply REST architecture on HTTP is to use HTTP Verbs `GET`, `POST`, `PUT` and `DELETE` to indicate which action should be done on the resource. `GET` is used for fetching resource's data, `POST` for creating a resource, `PUT` for updating a resource, and `DELETE` for removing the resource. Although the four HTTP verbs are commonly associated with corresponding CRUD operations, the `PUT` and `DELETE` verbs are less used in practice [Fow10]. Each operation returns the new state of the modified resource. HTTP status codes should be used to inform the client what happened during the operation. Additional constraints to the architecture model are that requests are stateless, e.g. don't depend on the server's state, and that the response's cacheability is explicitly defined. Uniform interface principle in the REST definition, also known as *Hypermedia as the engine of application state* (HATEOAS), states that the responses should include information about possible actions on a resource, which makes the API locations and actions discoverable from the clients.

REST principles are applied to HTTP endpoints in varying degrees. The Richardson

maturity model of REST APIs explain the different levels in which the architecture style can be applied on HTTP endpoints [Fow10]. In the lowest level the HTTP endpoints don't map entities as resources and custom remote procedure calls are used to modify the entities. In the highest level the domain entities are presented as resources, HTTP verbs are used for describing operation intent and links are provided in the responses for possible resource actions.

GraphQL endpoint's API can be explored using the schema introspection functionality of the specification. This is a common benefit listed in REST and GraphQL comparisons. REST APIs can also be made discoverable by using the HATEOAS principle or by creating API definitions using a specification such as the OpenAPI [Swa19]. The OpenAPI definitions can then be explored using tools such as Swagger UI. OpenAPI defines the schema of the data model, and also information about the service like the service's location. Another GraphQL Schema equivalent for JSON would be the JSON Schema, which describes the format of a JSON document. JSON schema can be used to describe JSON data models and to validate the produced data on the server, or the received data on the client. The GraphQL schema is similarily used to validate data on the server and also on the client [Fac18a]. Eventhough it's possible to implement a REST API that is explorable, has schema definitions and good documentation, they are rarely implemented with all those features [Fow10]. In GraphQL all the mentioned features are implemented by default.

Many entities can be fetched in a single GraphQL query, which commonly isn't possible in REST APIs. This can be circumvented by custom solutions such as Facebook Graph API's batched API calls [Fac19]. The batch endpoint processes multiple requests encoded as a JSON array of JSON request objects. The JSON request objects define the endpoints, parameters and HTTP verbs of the operations.

Nemek argues in his presentation and blog post that implementing a properly designed REST API and with the use of HTTP/2 protocol, data overfetching isn't an issue, because consecutive multiple requests are multiplexed with the newer transport protocol [Nem18]. By adding hyperlink controls (HATEOAS) and adding addresses for linked data the JSON objects can be explored and fetched from different endpoints. This means that REST endpoints can expose only minimal amount of fields and all the required data can be fetched from multiple endpoints without much overhead. Still, multiple rounds of multiplexed requests are required to fetch any additional data about linked objects, compared to making a single request with GraphQL which describes all the fields that should be resolved in the request.

4.9.2 SOAP

Simple Object Access Protocol (SOAP) is a lightweight communication protocol which uses XML as the data exchange format [GHM+07]. The protocol is considered light because it only defines the data structure and processing rules of the messages, but states no guarantees or requirements about how the messages should be delivered. XML data format on the other hand is a verbose data format which adds network traffic compared to a solution that uses JSON. The specification defines an XML structure that is used for communication between computer systems. A SOAP message consists of a message header and a message body wrapped in an SOAP envelope element. The SOAP header consists of parts that define how and by who the message header should be consumed or processed. The SOAP headers might be read by intermediary nodes, but the body is always meant for the end recipient node. The body consists of the operation that is invoked by the requester, or has the response of the requested operation. SOAP isn't a query language, and doesn't define support for GraphQL like field selection. Compared to a GraphQL API, SOAP works more like an RPC protocol where data is requested by calling methods by using the SOAP messages. The SOAP operations map to functions, instead of mapping to resources (REST) or entities and their fields (GraphQL). Similar to GraphQL, a SOAP service is called through a single endpoint which processes the XML SOAP requests.

XML Schema can be used as the GraphQL schema equivalent of SOAP endpoints. XML Schema is typically used within a WSDL contract to define inputs and outputs of an web service's interface [CMRW07]. WSDL contract is a document that is used to describe web service's endpoints and interfaces. Although WSDL can be used to describe other endpoints, it is typically used to describe SOAP web services.

4.9.3 Netflix's Falcor

A more direct competitor to GraphQL would be Netflix's Falcor library which uses a virtual JSON object model definitions instead of a schema [Net19]. Resolvers for the virtual JSON object's fields are defined in a similar fashion as in GraphQL server implementations. The Falcor client library is used to query fields from the model. Setting the virtual object's fields are also supported through the model. The Falcor virtual JSON model isn't typed whereas a GraphQL Schema is strongly typed. Falcor is a single library whereas GraphQL is a language specification with

many implementations.

5 Microservice architecture patterns

Taibi et al. gathered microservice architectural patterns in their systematic mapping study [TLP18]. The study analysed 42 peer reviewed papers and two grey literature articles and combined all the found patterns in a pattern catalogue. Three main categories of patterns seemed to emerge from the papers. Orchestration and Coordination-based patterns describe ways to handle communication between microservices. Second category was related to microservice deployment methods. The third category of patterns was related to data storage and communication. The first and third categories will be presented in detail in subchapters 5.2 and 5.3. There seems to be a concensus of using either containers or VMs for deployment of microservices [TLP18]. Another common pattern for designing the architecture of microservices is using Domain-driven design (DDD) [DRS17]. DDD itself doesn't describe functional details of the architecture such as the database setup or deployment options, but it concerns itself more with how to define separate subdomains in the business domain [Eva03]. The identified subdomains can be used to define how to split functionality between the microservices [DRS17]. Domain-driven design is thought to complement microservice architecture design well in many sources [Ric18, DRS17, Fow14].

5.1 Domain Driven Design

Domain-driven design (DDD) is a collection of design patterns which can be used to model the business domain of a software project, introduced by Eric Evans in his book 'Domain-driven design' [Eva03]. In DDD the software team and the domain experts form a common language, called *ubiquitous language*, which is used to describe the business domain's models and entities. The language has shared names for all the entities in a problem domain. The model is used and refined in discussions throughout the project, as it is the basis for all software implementations in the project. DDD is thought to fit microservice architecture well, because business capabilities are modeled as separate *bounded contexts* without trying to form unified shared entities between the contexts. A bounded context has a limited self-contained view on its entities, which prevents creating highly coupled *god classes*

in the project, which try to combine all the functionality of the entity across the bounded contexts [Ric18, p.54-55].

5.1.1 Domains, subdomains and bounded contexts

Using domain driven design can help with separating concerns in the whole application domain [Fow14]. In a large organization the whole business domain can be splitted into separated areas, called subdomains, which will have their own concepts and also partially shared concepts [DRS17]. The subdomain is the problem space of the particular business area [Eva03, p.57]. A subdomain consists of tightly coupled concepts of one business area. Inside a subdomain a smaller area can be drawn with a fewer concepts, which represents the solution space of the problem. The solution space is called bounded context [Ver13, p.57] and is used as the basis for the software implementation [Fow14]. Difference to naive business capability modeling is that bounded contexts don't share their entities between other bounded contexts, which suits microservice architecture design well. This means that although a common entity might exist all over the business domain, such as an entity *customer*, the bounded context describes only part of the business entity which is relevant to the problem space in question. Let's consider the two bounded contexts in figure 2. Two bounded contexts *Customer service* and *Billing* share three business entities *Customer*, *Product* and *Invoice*. Both bounded contexts will have a model for *Product*, but they will contain different fields. In *Customer service* the *Product* might contain fields such as *previousMaintenanceDate* or *warrantyInfo* and *Billing* context *Product* might contain *dateOfSale* and *quantity*.

5.1.2 Context map

Concepts that describe the same entity in the business domain are called shared concepts. Shared concepts connect bounded contexts together. Eventhough the shared concepts might be named the same, they might have slightly different meaning between the different contexts. *Customer* is a typical shared concept between bounded contexts and it will mean different things in it's separate contexts, as shown in figure 2. Strategies like adding ontologies to the bounded context models can reduce problems such as overlapping concepts with different naming and also differing attribute names within the concepts [DRS17]. A *context map* is the graphical or literal presentation of the connected bounded contexts. Maintaining a context map

will prevent incompababilities between implementations of bounded contexts and keep the overall model coherent [Eva03]. To make bounded contexts' shared concepts compatible, a translation service can be used to convert between the bounded contexts. Typically the service itself is responsible of using this translational service when it needs to use data from the other bounded contexts [Eva03].

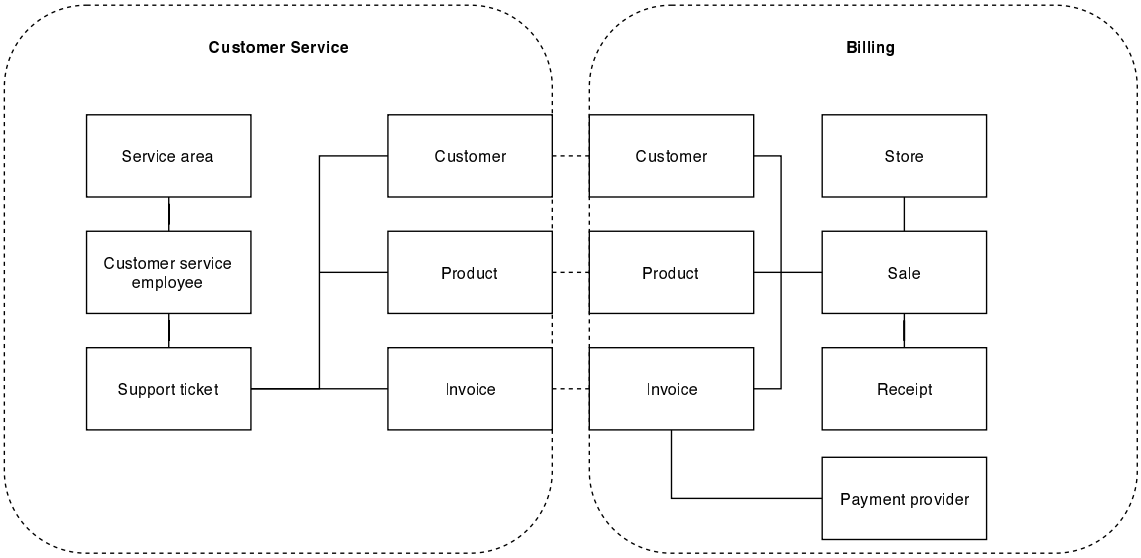


Figure 2: Two bounded contexts with shared concepts which connect the bounded contexts together.

5.1.3 Aggregates

Business entities can have complex associations to each other. In a system that models a book-related business domain, there could be a *book* microservice and a *retailer* microservice. A book microservice could track the list of retailers where the book is sold at. The retailer microservice might want to use the same book-retailer-relationship to show which books are in their inventory. In a monolithic application these relationships can be tracked in a single linking database table. On a distributed system such a linking table would tightly couple the microservices together and it isn't clear in which microservice the linking database table should be placed [Sar19]. In *aggregate modeling* the relationship is split in two parts: the retailer's inventory and the book's `soldAt` references. By splitting the relationship into two parts the data ownership becomes clear and the data updates can happen within a single microservice. Data outside the writing microservice should be eventually updated

to keep data consistent. Other entities in the retailer aggregate could be the address, its staff, and the retailers reviews.

Aggregates are a collection of connected domain entities which form a functional whole which can be updated and deleted together in a single transaction [Eva03]. Designing aggregates helps with figuring out the boundaries of data updates and removals. Transactional data updates should never span across aggregates, and anything else should be eventually updated if the aggregate update affects other aggregates in some way. For instance if the inventory of the retailer is updated, an asynchronous event could be used to update the corresponding book's `soldAt` references.

Aggregate modeling might also help with creating schema stitching compatible microservices. When creating a schema stitched GraphQL server, object references in one schema can be replaced with actual objects using *schema delegation*. The aggregate style relationship mapping works great both ways, because references on both sides can be resolved with direct mapping to another microservice's schema, instead of making a costly inverse relationship query from another microservice.

5.1.4 Continuous integration and shared kernels

Eventhough the goal of DDD is to manage a complex application domain through splitting it into smaller subdomains, sometimes sharing code can't be avoided. In the domain model this is shown as overlapping bounded contexts, and these areas are called *Shared kernels*. Shared kernel's appear commonly in the core domain of the application. In DDD continuous integration (CI) should be used to sync the implementations of the shared kernel between bounded contexts. A test suite will keep the shared code-base working between different contexts, but also it's a good indicator of how consistent the model is between software teams and implementations [Eva03].

5.2 API Gateway

API Gateway was identified by Taibi et al. as a popular *orchestration and coordination* based pattern in an application that uses microservices [TLP18]. API Gateway is a service on top of microservices which exposes and combines the functionality of microservices for client applications to use. API Gateway can provide customized endpoints for different client applications in the microservice ecosystem. The API

Gateway can also handle authorization and load balancing for the microservices. Benefits of using an API Gateway was reported to be extensibility and backwards compatability because of the additional level of freedom the customized API can provide. The reported downsides were scalability issues and increased complexity in the architecture. Because GraphQL enables clients to define their own customized queries, a single GraphQL endpoint can be used by multiple clients, thus GraphQL suits well to be used in the API Gateway pattern. Using GraphQL as the API Gateway layer will be discussed in section 6.2.

5.3 Data Storage Patterns

Taibi et al. identified three data storage patterns in their microservice architecture study [TLP18]. The most prominent pattern in the source case studies was to use separate databases for every microservice. Having separate databases has the benefit of making scaling and deployment easier, protecting data from getting accessed from other microservices. The second identified pattern was to use a database cluster for data storage where each microservice has only access to the database tables of their problem domain. Using a database cluster can be benefitial for easier scalability and when having high amounts of traffic, but the increased complexity and a shared point of failure is a risk when using a database cluster [TLP18]. Also a single shared database was used in a few case studies. A single database is typically present when migrating from a monolithic application. Using a database cluster or a single database can have the added benefit of increased data consistency through database rules, which can help when combining data in a schema stitched GraphQL API. Schema stiching will be discussed in chapter 6.

5.4 Command Query Responsibility Segregation

Command Query Responsibility Segregation (CQRS) is a software pattern, where reading and writing data is done by separate models [Fow11]. Richardson's definition of CQRS is to maintain a separate view of queryable data by using events [Ric18, p.228]. Typically data is written and read by a single model from the same data store, whereas in CQRS the written and read data can be saved into different data stores [Fow11]. CQRS can be a suitable architecture pattern for high throughput systems where writing or reading data has to be separately optimized. Fowler writes that CQRS is generally more suitable for complex domains and should not be used

unless necessary. Most of business cases fit the CRUD (create, read, update, delete) model which is a lot more easier for the developers to reason about, compared to CQRS [Fow11].

5.5 Authorization patterns

In a microservices context the authentication typically happens in a separate service outside the business logic layer. A Single Sign-on (SSO) service issues an authentication token for the user, which is saved locally in the client as a cookie or by other means [Ide15]. The token can be a value token, which presents claims about the user and provides a signature that can be verified with a public key by the application server. Typically the JWT standard is used for implementing value tokens [Rig19]. The token can also be a reference token, which contains an opaque session id which is translated into a value token in a microservice, or in a layer before the microservice as suggested by Ideskog [Ide15]. JWT value tokens work well to minimize identity related network traffic between the microservices and an identity service.

If a session is required by the application, a distributed session store should be used with the application's microservices [Ayo18]. An external database or key-value storage can work as the distributed sessions store. The only requirement is that the store is accessible from every microservice that needs access to the session data. This results in more network traffic, but it might be considered safer than relying completely on a value-based token, which can possibly be decrypted and the user's data extracted on the client side [Rig19].

5.6 Organizational patterns

DevOps is a modern set of practices where software testing, deployment and logging are highly automated with using code and tools such as a continuous integration (CI) pipeline [EGHS16]. The term DevOps comes from the combination of development and operations. Traditionally development and operations might have been responsibilities of different teams, whereas now a single team typically has both of the responsibilities. Cloud computing platforms such as the AWS or Azure are used for on-demand resources, where virtualization and containerization techniques are used for hosting the application. CI pipelines are used to automatically test and deploy new versions of the developed software. A CI pipeline pulls in changes from version control, tests, builds and deploys the code. The automatized deploy-

ment pipeline results in reproducible builds which in turn makes larger application scaling possible, because new servers can be launched with little effort if needed. Reproducible builds are important so server behaviour is predictable. Microservice architectures rely on DevOps workflows for deployment and management [Fow19c].

5.7 Testing microservices

Additional testing complexity is added by the distributed nature of the microservices architecture [Fow19d]. As with other architecture types, unit testing should be utilized to test business functionality. Unit testing can be split into two categories of *sociable unit testing*, where the module is tested through its interface and it's state is observed. The module uses other modules normally during the testing. *Solitary unit testing* is done by replacing other modules with mockups and observing the module's behaviour and interaction with the test doubles [Fow19d]. Fowler states that typically sociable unit testing is done on the business logic as the interactions are often complex and mocking the behaviour can be difficult. Solitary testing is used for external interfaces such as persistence layer code and REST endpoints.

Integration testing should be utilized to verify that the components inside a microservice function together correctly [Fow19d]. Integration testing can be done at any level of granularity. In microservices, integration testing is typically used for verifying integration code to other microservices in the ecosystem or with external services. The tests should focus on the tested microservice. Instead of writing comprehensive acceptance tests for other microservices, the testing should be limited to the tested microservice and not span to other microservices. Because microservices rely on IPC with external components, network issues such as timeouts and errors should be taken into account in the integration testing [Fow19d].

Component testing should be used for testing the microservice as a whole, where external services are replaced with test mockups. Databases can be replaced with a lightweight in-memory implementations [Fow19d].

Contract tests are run to verify that an outside microservice's API works as expected [Fow19d]. Contract tests should be run on the consuming microservice in the pipeline to detect faulty or regressed microservices. Consumer-driven contracts is a development style where contract tests are written for a new service to define which resources are expected from other services. The producing services are then developed accordingly. End-to-end tests are done to verify that the microservice

ecosystem works as a whole together. End-to-end testing can be used to detect network issues for instance between the services [Fow19d].

Using a microservice architecture can make testing easier, because of clear boundaries between components and the possibility of mocking other components when needed. Fowler recommends that microservice testing should concentrate on more fine-grained testing, having unit tests the most, and end-to-end tests the least [Fow19d].

6 Architecture Patterns with GraphQL

In this section architecture patterns that can be applied to GraphQL APIs are presented. Some patterns are applicable to a microservice architectures in general and some patterns are GraphQL specific, such as schema stitching (section 6.2.1). Because GraphQL is a fairly new specification, we rely mostly on grey literature, e.g. blog posts and websites, for information on combining microservice architecture patterns with GraphQL. The presence of these patterns are discussed with the interviewees to analyze the incidence of the patterns in real-world projects.

6.1 Validating GraphQL Schemas against DDD models

Many studies [DRS17, Fra17] explore ways of augmenting Domain Driven Design methods with domain-specific languages (DSL) such as OWL or graphical representations like UML. The GraphQL schema describes the type system of the GraphQL server, e.g. which entities and types can be requested from the server. If the microservice is implemented from the DSL model of the business domain, the resulting GraphQL schema's entities and their attributes should match the domain's model. To minimize inconsistencies between the original model and the microservice's GraphQL schema, we suggest either automated tools to validate the model against the schema, or using code-generation techniques to generate the schema from the business model's DSL. Schema validation can be problematic if the schema reflects the DDD model directly [Ver13, p.137-138]. This will create API instability because the GraphQL schema changes every time when the internal model gets updated. In the interviews, mirroring business models in the GraphQL schema was discussed.

6.2 Using GraphQL as an API Gateway layer

API Gateway is a popular microservice architecture pattern and was used in many case studies covered by Taibi et al. [TLP18]. A GraphQL service can be used as the API Gateway or as a complementary endpoint to other RESTful endpoints in an API Gateway [Gri18]. Figure 3 shows three suggestions how a GraphQL API Gateway can be used with microservices.

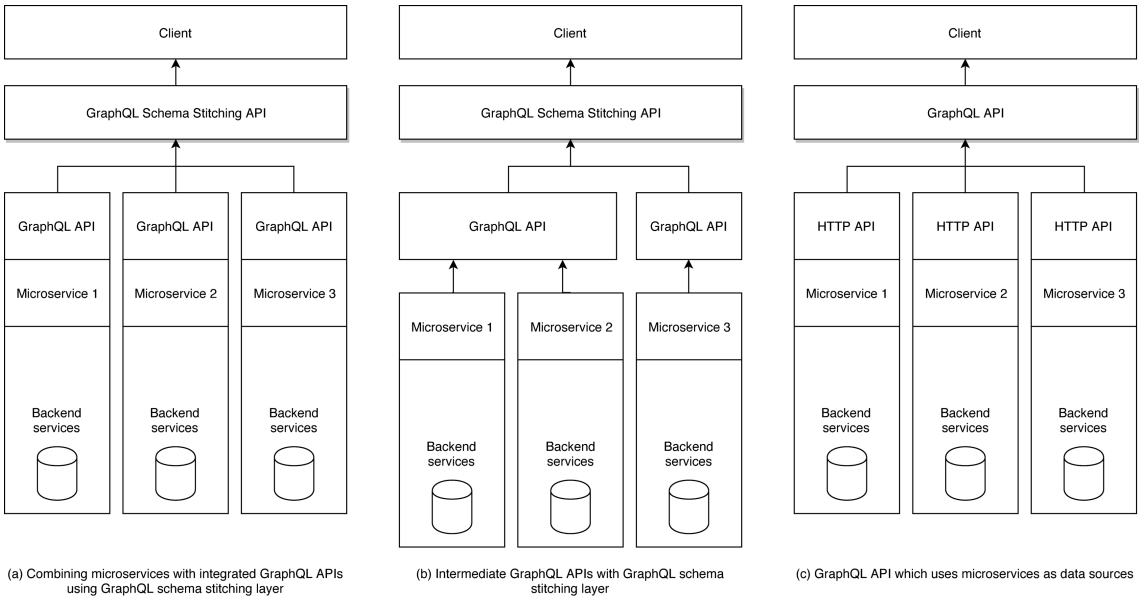


Figure 3: Examples of using GraphQL API to combine microservices

Vernon describes in his book 'Implementing Domain-driven Design' that using a RESTful interface for a DDD model promotes loose coupling. The same principles should also apply to an API Gateway. The decoupling of the HTTP interface from the core domain model helps with keeping compatability with clients when implemented correctly. Vernon says that directly exposing the core domain model should be avoided because it might cause client breakage when the model is changed. Implementing an API that translates to the core domain model for clients is considered a better practice. In the book the recommendation is to define a format specification which the RESTful HTTP interface will fulfill, so the clients can rely on the format eventhough the core domain might change [Ver13, p.137-138]. It can be drawn from these best practices that also a GraphQL API shouldn't mirror the core domain directly to prevent API breakage issues with clients. A fixed format specification can help with keeping the GraphQL entities from changing too much eventhough

the core domain changes. Schema changes due to changes in the domain model was discussed in the interviews.

Griffiths suggests that for better client compatability a RESTful HTTP interface and also an GraphQL API can be developed side-by-side [Gri18]. The suggested method is to develop primarily the GraphQL API and expose the API also through REST endpoints. The REST endpoints will call the GraphQL API with predetermined queries to reduce code duplication. The reported downsides would be missing the REST endpoint specific authorization, throttling, cache and load balancing functionalities that an API Gateways typically can provide [Ama19]. For a GraphQL API these API Gateway functionalities have to be typically implemented per service in the GraphQL resolvers or the business logic layer.

6.2.1 Schema stitching microservices into one API gateway

In figures 3b and 3c architecture models are presented where an intermediate GraphQL API connects and combines microservices in a layer below. The GraphQL API layer fetches resources from the application's microservices and sends them forward to the requesting client. Otemuyiwa explains connecting a GraphQL API on existing REST endpoints using the Apollo GraphQL library [Ote18]. An intermediate API layer can be used for migrating and connecting old microservices together into a single GraphQL API, without rewriting the old services. The GraphQL API layer fetches data from the underlying microservices using HTTP. The clients will then make calls to the GraphQL API instead of connecting to a RESTful API Gateway or directly connecting to the microservices.

Apollo GraphQL library also provides other tools for combining other microservices. If the microservices expose their functionality through a GraphQL API, a functionality called *Schema Stitching* can be used to combine the different GraphQL APIs together into a single API. This scenario is represented in figures 3a and 3b. Schema stitching's suggested use case is to make large codebases more manageable by separating an otherwise large API into separate smaller GraphQL services [Apo19b]. Schema stitching can be used to combine local GraphQL schemas and remote schemas together and also extending the types of the combined schemas with new fields, or adding fields or entities from other schemas. Extending the combined schema works with same principles as explained in chapter 4.7.1. Schema stitching is implemented in the Apollo GraphQL library. Apollo library provides a delegateToSchema function which can be used to add an entity from another schema into

another extended schema's entity. This can be useful for instance for populating entity ID fields with the actual entities from other schemas. One example would be to reference a user object with an ID in a separate schema, and populating the actual user data into the entity from an identity API.

An example of schema stiching can be seen in listings 9 and 10. In listing 9 two remote schemas and an schema extension is presented. The first two entities are defined in separate remote APIs. The third schema extension type defines an additional author field in to the Book's entity. Listing 10 shows an example how schema delegation can be used in the stitching API's resolver to fetch the author field from the User schema's API.

Listing 9: Example of an extended schema from another API

```
# Defined in GraphQL API 1
# A remote Book schema
type Book {
  id: ID!
  text: String
  authorId: ID!
}

# Defined in GraphQL API 2
# A remote User schema
type User {
  id: ID!
  name: String
  authorId: ID!
}

# Defined in the Stitching API
extend type Book {
  author: User
}
```

Listing 10: Javascript example of a resolver with schema delegation

```
Book: {
  author: {
    fragment: `... on Book { authorId }`,
    resolve(book, args, context, info) {
      return info.mergeInfo.delegateToSchema({
        schema: ApiTwoSchema,
        operation: 'query',
        fieldName: 'userById',
```

```
            args: {
              id: book.authorId,
            },
            context,
            info,
          });
      },
    },
  }
```

6.3 Communication strategies in a GraphQL API

Connections from a client to a GraphQL server are typically synchronous HTTP requests where the request contains the GraphQL request document and a corresponding JSON response is returned by the GraphQL server. GraphQL specification also describes the subscription operation, which subscribes the client to receive events from the server. Subscriptions require a sustained connection protocol between the GraphQL server and client such as Websockets to work.

As described by Pancheco, compared to monolithic architectures where communication happens through function calls, in microservice architecture the latency and processing time of requests become meaningful and have to be taken into account [Pac18]. Selecting a light weight data format instead of using JSON can reduce time which is used for parsing microservice requests.

Requests can be either synchronous or asynchronous between the GraphQL server and microservices. In the interviews, the case project's communication methods between the GraphQL server and microservices were discussed.

6.3.1 Inter-process communication using GraphQL

GraphQL can also be leveraged as an inter-process communication protocol [Abh19]. A solution presented by Aiyer uses a Javascript library *graphql-bindings*, which generates a Javascript API, the *bindings*, from a GraphQL API. The bindings can be created for a remote or local GraphQL API. The resulting API can be embedded easily into GraphQL resolvers for instance, to fetch data or to run mutations from other services, without having to define verbose request documents for each request. Using a GraphQL API for inter-process communication will add overhead in parsing the request documents and also in document size, compared to using simpler pro-

tocols such as REST or Protocol Buffers [Goo19]. Also there's an added overhead of implementing a GraphQL API for a microservice. The added overhead can be justified if the microservice's GraphQL APIs are schema stitched in an API Gateway layer, which makes it possible to define the schemas only once per microservice. Vernon states [Ver13, p.137-138] that the core domain model shouldn't be exposed directly by the API. Implementing a GraphQL API for a microservice can support this separation of API and the core model, much like implementing a REST API that doesn't mirror changes in the core domain directly.

6.3.2 Event subscriptions via GraphQL subscriptions

If the GraphQL library supports it, a *subscription* request document can be sent from the client to subscribe for push events for defined subscription fields. We suggest that GraphQL subscriptions can be used to subscribe to domain events in the microservice ecosystem. The pattern could be used for implementing a notification system for instance. As with recommendations not to mirror the core domain model in the API signature [Ver13, p.137-138], the same principles should most likely be applied for subscriptions.

6.4 Authorization in the GraphQL API microservice stack

The official GraphQL documentation recommends placing authorization logic in the business logic layer, which keeps all the authorization logic out of the API layer [Facb]. This also makes it easier to develop multiple APIs alongside each other, for example having a RESTful and GraphQL side-by-side as suggested by Griffiths [Gri18] because the authorization logic doesn't have to be duplicated in many places. Placing the authorization logic in the business logic layer instead of the API layer prevents differences in data access rights between GraphQL resolvers that use the same business logic layer functions. The GraphQL documentation also states that the business logic layer should receive the user object instead of an opaque user id or token, which keeps all authentication logic out of the business logic layer.

Apollo blog's stance on authorization is in line with GraphQL's official documentation. In complex server application the authorization logic should be placed in the business logic layer [Daw18]. Implementing authorization logic in the business layer is trivial in a monolithic application with a use of a security module in the application framework. A distributed authorization method presented in Chapter

5.5 is required for microservices to prevent duplicating authorization logic in the microservices [Ayo18]. Distributed session storage is also needed if calls to a particular microservice are load balanced from the GraphQL API.

According to guidelines in the GraphQL documentation and Apollo Blog the authorization logic should be placed below the API layer and populating the user context should happen before resolving data in the business logic layer [Facb, Daw18]. Based on these guidelines, populating the user context could happen inside a microservice through a middleware module which accesses a distributed session storage and does the user data fetching before resolving data in the business logic layer. This is a simpler option, but it might result in code duplication because the middleware library has to be recreated for every language used in the microservice ecosystem. This also increases network traffic and reduces availability in microservices, because a synchronous request has to be made to the identity server to fetch user data. Another option would be to have an additional API layer which fetches the user data before passing the request to the microservices, which is presented in figure 4. The user authentication layer can also be part of the API gateway as presented by Richarson [Ric18, p.354].

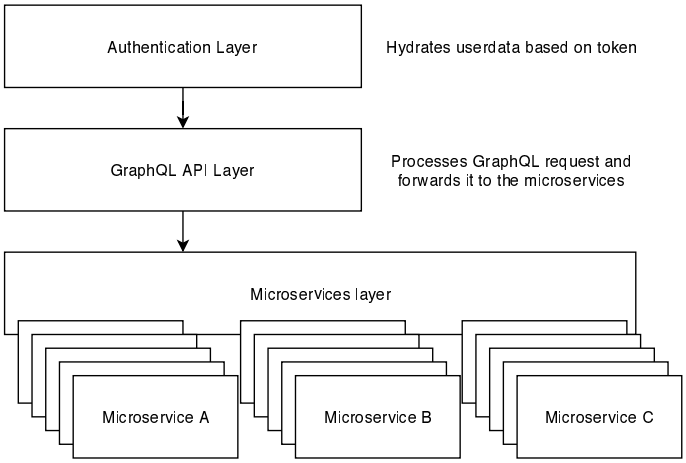


Figure 4: Suggested authentication layer in a request pipeline, which fetches the user data for the request.

6.4.1 Authentication and session mechanisms

Sending a JSON Web Token (JWTs) through an authorization HTTP header is the recommended first solution as the authentication mechanism according to the

GraphQL documentation [Faca]. Using JWTs doesn't have the same cross origin issues as session cookies when the GraphQL API is consumed from another domain. JWTs through authorization headers are more easy to use with Single Page Apps (SPAs) and mobile clients which access a cross-domain API. Open Web Application Security Project (OWASP) recommends to use sessionStorage over localStorage when saving the web token in a browser, to prevent access to the token after the website is closed [Rig19]. For added security against token sidejacking, a random fingerprint hash can be stored in the web token. The same unhashed fingerprint is sent to the client via hardened cookie, which isn't readable from Javascript. The token is accepted if the token fingerprint matches the cookie's fingerprint [Rig19].

If a cookie is used for storing the token, it is recommended to use hardened cookies with `HttpOnly` flag, which prevents accessing the cookie from Javascript. This protects the token from XSS attacks, but because cookies are sent automatically with every HTTP request, the web application has to be protected against cross-site request forgery (CSRF) [Rig19]. If cookies are used for token storage, the GraphQL API gateway has to either implement sticky sessions, which means that sessions are saved locally in the API gateway instance and the client is always served from the same API gateway instance [Ayo18]. Another way is to store session data in a disributed session storage that can be accessed from all API Gateway instances. The incidence of JWTs or sessions were discussed in the interviews. Also the token storage method was discussed.

6.5 Service discovery

Service discovery is a pattern where the client application or server acquires the location of a microservice instance before requesting a resource. In client-side service discovery the client connects directly to the acquired microservice location. In server-side discovery the server routes the request to an available microservice instance. In both patterns a *service registry* is used to keep track of microservice instances. When a microservice is started, it registers itself to the service registry. The instance is removed from the registry when it is shut down.

6.5.1 Accessing microservices in client

Client-side discovery can be used in a situation where the client connects directly to multiple APIs. If client-side discovery is used, the microservice instance locations

can be determined at runtime from a service registry. Client-side service discovery is much less used compared to server-side or platform discovery [TLP18]. An example where client-side discovery might be useful is a multiplayer game where the client connects directly to a game server after querying the IP from a matchmaking service.

6.5.2 Server-side service discovery

Richardson presents an example of server-side discovery, where microservices do a self-registration on a service registry [Ric18, p.80-85]. The service registry is queried by the GraphQL server or API gateway, and the request is sent to the returned instance location. The solution has benefits that it is fairly simple to implement, and there's little infrastructure configuration. Also there are no DNS caching timeout issues where a stale server is still registered as a cached DNS entry. Service registry can instantly remove stale servers. The drawbacks are that the client and microservices get coupled with the service registry and service registry client libraries have to be written for all clients and services.

Taibi et al. found that most of the service discovery related solutions presented in the found articles used server-side service discovery [TLP18]. Server-side service discovery adds a load balancing node between the client and services which increases complexity. Instead of clients querying the service registry directly, the load balancer queries the address of an service instance from the service registry and forwards the original request to the service.

A platform provided service discovery is also typically used [Ric18, p.84-85]. A solution like the internal DNS system in Kubernetes can be used to map instances behind virtual hostnames. Server instances are launched within the platform, so the services don't have to implement self-registration. This makes the architecture simpler than compared to server-side service discovery. A typical drawback of using platform specific service discovery is that only services inside the platform can be automatically registered in the service registry. Outside services have more manual configuration if they are added to the load balancer.

6.5.3 Replacing service discovery with asynchronous messaging

Service discovery is associated with synchronous communication patterns, as it can be used to raise the availability of a service with load balancing between multiple service instances. Using a message broker effectively removes the need for service

discovery, because clients and services connect indirectly through a message broker to publish and consume messages. The services consume messages from the message broker as they become available.

6.6 Integration testing GraphQL

HTTP endpoints can be integration tested using strategies such as the *consumer-driven contracts* [Ric18, p.322-323]. Consumer-driven contracts are expected outputs defined in the consuming clients and services [Fow19b] which the providing APIs should produce. For a typical HTTP endpoints the schema could be a JSON schema or an XML schema which the provider is expected to fulfill. Extending the providing service happens when the consumers, the clients that make the requests, define new fields in the schema that is expected from the service.

We suggest that to use consumer-driven contracts in a GraphQL setting, the providing services should read the queries defined in the GraphQL clients and evolve their schemas accordingly. Deprecating unused fields can be achieved by logging GraphQL queries and keeping track which fields are used and which are not. Adding non-existing object fields in a request document results in validation errors, so adding new field proposals into the queries doesn't work. Unless a workaround is used for adding field proposals into the GraphQL queries, for instance using comments to add field proposals, the proposals could done between the developers by regular communication routes such as email.

Integration tests could be added for the GraphQL clients or servers. In clients the integration tests would warn if the expected GraphQL API has changed. In servers the integration tests could be used to keep the schema changes low to minimize client breakage.

6.7 Patterns for protecting a public API

A public API is vulnerable to outside attacks like denial-of-service (DoS) attacks and unintended API usage. Different strategies for protecting public GraphQL APIs were identified from grey literature and interviews [Sto19, Apo19a]. The Apollo GraphQL library supports a feature called *Persisted Queries*, which makes it possible to predefine allowed queries in the server [Apo19a]. The persisted queries support variables, so the query can still be parametrized. The queries are called using an

identifier, instead of sending the whole query document to the server. This reduces the size of the requests and also prevents unwanted API usage by limiting possible usable queries to ones that are used by known clients. Using persisted queries have the side-effect of adding coupling between the server and clients, because the queries have to be defined at build-time. Persisted queries are also available in other GraphQL server implementations, but it's not part of the official GraphQL specification.

Stoiber presents methods of analysing GraphQL queries for detecting malicous expensive queries aimed to overload the GraphQL server [Sto19]. Stoiber discusses various methods of reducing query complexity. The simplest methods are to limit the overall character length of the query and also limiting the length of requested lists types. Stoiber also presents the importance of depth analysis of the GraphQL query to prevent nested queries which get exponentially more complex when the depth increases. The most advanced way of detecting expensive queries is to do cost analysis on the queries. Libraries for implementing cost analysis is available and typically make use of GraphQL directives to decorate the types in the GraphQL schema with cost data [Sch19].

Circuit breaker is pattern which prevents a service from exhausting with open non-resolving synchronous requests [Fow19a]. Synchronous requests typically create a thread for each connection. The thread will wait until the response is returned and after that the thread will exit. If a service becomes unresponsive and requests to that service timeout, there is a possiblity that the calling service might exhaust with open connections. A function which makes a synchronous call to a remote service can be protected with a circuit breaker wrapper, which keeps track of timed out requests. If a predefined threshold of timeouts are reached within a certain time frame, the circuit breaker "trips" and fails all following requests instantly. The ciruit breaker can be manually reset, or there can be an automatic detection of correct behaviour in the remote service. Circuit breaker pattern could be applied to GraphQL resolvers, which would prevent exhausting the GraphQL API with open connections.

6.8 Command Query Responsibility Segregation with GraphQL

A few grey literature examples of using Command Query Responsibility Segragation (CQRS) can be found [And19]. Anderson states that using CQRS can be benefitial even for simple business cases and would make development easier, which contradicts

the article written by Fowler [And19, Fow11]. Anderson gives a simple Java example of a GraphQL server where User entities and Question entities can be viewed and updated. Although they are modeled and updated as separate entities, they can be returned as a single entity using GraphQL. It could be argued that Anderson is not using CQRS in the example, but using data aggregation via an API Gateway instead, because the User and Question models are still being updated and read from the same database and single models. To fit the Richardson's definition, the aggregated User and Question data should be stored and kept up-to-date in a separate store [Ric18, p.228].

In our interviews the usage of CQRS pattern was studied. To fulfill the CQRS pattern the studied system needed to store a read-only copy of information in a separate store from the database that is used for data updates. In one interview the use of CQRS pattern was identified. The CQRS pattern was used for an entity which was editable in an external service. Instead of aggregating the data directly from an external service, a local copy was kept up-to-date and served to increase throughput. Other local data could be aggregated to the entity using GraphQL.

7 Interview

We prepared a semi-structured interview to evaluate the incidence of proposed architecture patterns and also to validate the proposed architecture patterns directly by interviewees themselves. The interview process was conducted as a variation of *focused interview* presented by Hirsjärvi and Hurme [HH09], called the *theme interview*. The so called *theme interview* method is a semi-structured interview method where the interview is driven by open ended questions, which revolve around the pre-defined themes. The order and wording of the open ended questions are not strictly defined. The interview was split into two parts. In the first part the themes discussed are practical architectural solutions in a GraphQL microservice application. Qualitative analysis is done for the answers to find incidence of the proposed architectural patterns in chapter 6.

- General idea of the project
- General architecture of the project (how many microservices, was authentication used?)
- Why GraphQL was chosen instead of typical HTTP or RESTful endpoints

- Is GraphQL used alongside HTTP or RESTful endpoints
- What has been the client developer experience with GraphQL API?
- How was the business logic split into microservices?
- Similarities in business models and GraphQL Schema
- API changes, breakage and feature deprecation
- Placement of GraphQL APIs in the microservice stack
- Messaging solutions between microservices and between the GraphQL API
- Testing in the GraphQL layers
- Authentication and authorization solutions
- How did GraphQL API find the services?
- Organizational patterns (workflows, deployment pipeline, developer interaction between client and backend developers)

In the second part of the interview the following themes are discussed to validate and identify the presented architecture patterns directly. The architecture patterns are presented and explained briefly to the interviewee and then the patterns are discussed if they are present in any known projects to the interviewee. A general feasibility of the pattern is discussed and any variations of the pattern should be uncovered.

- Validating GraphQL schemas against domain models
- Using GraphQL as an API gateway layer
- Schema stitching microservices into one API gateway
- Communication strategies in a GraphQL API
- Inter-process communication using GraphQL
- Event subscriptions via GraphQL subscriptions
- Authorization in the GraphQL API microservice stack

- Authentication and session mechanisms
- Service discovery
- Integration testing GraphQL
- Circuit breaker pattern for queries
- CQRS pattern
- Attack prevention methods in public APIs
- Other solutions that weren't mentioned here

Additional analysis of the interviews are done to find more insights into discussed patterns and also uncovered patterns related to GraphQL.

8 Results

In total four interviews were conducted with experts who had real world experience of using GraphQL and microservices together. Case A was an application for handling customer related data, Case B was a content delivery platform, Case C was a high throughput system implemented using event sourcing and Case D was a system for handling technical devices, all using GraphQL in some layer of the microservice ecosystem. Table 1 displays all the detected patterns used in the projects, which were identified in the interviews.

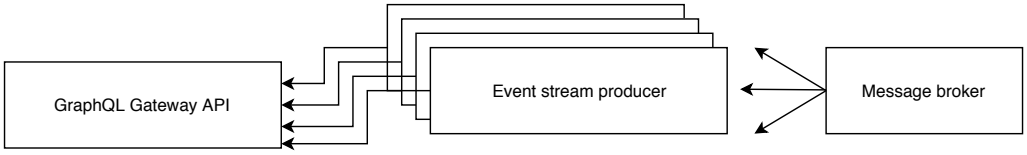
The reasons to choose GraphQL in the projects were the following.

- Positive experiences with the API exploration tools which improves the client development experience
- Possibility to develop the API with partial knowledge of the use cases and then filling out missing pieces as more parts of the domain is modeled
- Developing a unified API with vastly varying consumer needs
- To test the new API query language instead of implementing REST endpoints

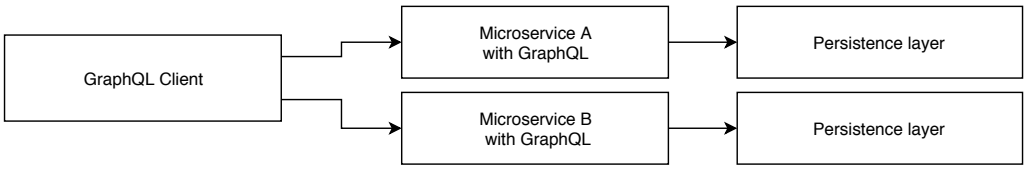
Nemek writes that developing a good REST API needs more careful domain modeling and understanding use cases in advance, whereas using GraphQL reduces the need for heavy API planning and mapping out API use cases. This supports the notion of developing the API with only partial knowledge of the use cases [Nem18].

8.1 Architectures

Cases B and D resembled the architecture layout shown in figure 3a the most. Case A and C differed from the presented architecture models. In case A the client connected directly to the microservices and consumed multiple GraphQL APIs (figure 5b). Case C didn't follow the presented architectures because of the event-based architecture that was being used. The event-based architecture is presented in figure 5a. The GraphQL API layer consumed processed event streams from microservices and passed them forward to clients.



a. Event-based microservice architecture with GraphQL API Gateway (Case C)



b. Connecting to multiple GraphQL microservices from client (Case A)

Figure 5: Additional architecture layouts in the case projects

8.2 Domain model

As Powell-Morse describes, business domains are typically more complex and convoluted compared to a technical project with clear technical specifications [PM19]. This seemed to be in line with our findings in the interviews, where a project with a more technical problem domain used a more straight-forward approach to modeling the domain. In these projects also the GraphQL schema seemed to follow the domain model closely. In the more business oriented cases a practice closer to DDD was used, although none of the business domain cases were complex enough for use of bounded contexts with context maps. The business entities were modeled as singular entities in a single domain, and not spread out across services. Thus, straight-forward business capability modeling was the prevalent method of modeling the business domains.

Table 1: Interview data compiled into a pattern-case matrix

Pattern	**Case A**	**Case B**	**Case C**	**Case D**
Schema reflects modeled domain	Yes	Yes	No	Yes
Schema validation	No	Yes	No	No
API Gateway	No	Yes	Yes	Yes
Schema Stitching	No	Yes	No	Yes
GQL–microservice communication (Sync/Async/Both)	Sync	Both	Both	Sync
Inter-process communication using GraphQL	No	Yes	No	Yes
Event subscription	No	No	Yes	No
Authentication method	JWT	JWT	JWT	Session
Authorization delivery method (Header/Cookie/Other)	Header	Header	Header	Cookie
Authorization happens in	Models	Models	GQL	Database
Session	Stateless	Stateless	Stateless	Session
Service discovery	Server	Server	Platform	Platform
Integration testing GQL endpoints	No	Yes	Yes	No
Circuit breaker	No	No	No	No
CQRS	No	Yes	Yes	No
Attack prevention methods and feature limiting (Persisted Queries, Query Cost Analysis)	-	-	PQ	-
Other solutions	-	-	Code Generation	-

8.3 API Gateway and schema stitching

As displayed in the case data matrix 1, most of the projects used GraphQL as an API gateway layer. Two projects of the three API gateway cases used schema stitching to implement the API gateway layer. In the two other projects that didn't use schema stitching, some problems with the most popular schema stitching implementation was found which prevented the use of the pattern. For instance error propagation through the stitched API didn't work as expected in one of the projects.

8.4 Communication patterns

All projects used synchronous communication in some way, and most of the projects used it prominently (A,B,D). Only the event-based system (C) used mostly asynchronous communication methods using a message broker between microservices, and in the GraphQL layer. Case B also used asynchronous communication between microservices for coordination purposes, such as informing other services about updated data.

Two of the projects used GraphQL as a communication method between microservices (B,C). One project had GraphQL interfaces in the microservices, which used the same APIs for direct communication between the microservices. The other case microservices used the GraphQL API Gateway layer for fetching required data indirectly from other microservices.

None of the projects used the circuit breaker pattern in the GraphQL API, which prevents exhausting an API Gateway with open synchronous requests that would timeout with high probability.

8.5 GraphQL subscriptions

Only the event-sourced case system C used GraphQL subscriptions for pushing the event data for the application clients. Other cases used the standard `query` and `mutation` root object types for data fetching and side-effects.

8.6 Authentication and authorization

Only one of the case projects used session cookies (D) and the others relied on JWTs sent through the authorization header. The session cookie based authentication was

handled in the API Gateway layer where the user context was saved into request headers for the microservices to consume. The microservices live in a private network that can only be accessed through the GraphQL API from outside. Because of the simple user context, the user context data could be added directly into the HTTP request headers. Security and authorization was implemented in the persistence layer using database policies and rules. Two of the cases (A,B) used model-based authorization and the event-based system (C) implemented a resolver-based authorization. In resolver-based authorization the authorization logic is placed in the GraphQL server, which dictates if the user can query or mutate a field. Only one of the cases (B) used a distributed session store, eventhough the case's authentication mechanism was JWT.

8.7 Service discovery

All of the systems relied on platform provided service discovery, where the platform provider automatically keeps track of launched microservice nodes and the microservices are accessed either by a public URL or with private network hostname. Two of the systems (A,B) used load balancers in front of the microservices with GraphQL APIs to do load balancing and also rolling updates. In a rolling update the nodes are deployed one-by-one and the load balancer switches traffic gracefully to the new updated nodes without downtime.

8.8 Testing

In two of the systems some form of integration testing was done on the GraphQL API (B,C). A query or mutation was run against the GraphQL API to test correct responses. None of the projects did comprehensive integration testing of the APIs. The testing in general seemed to follow a pattern of testing the business model thoroughly with unit tests and testing critical parts of the API with integration or end-to-end tests. One of the problems identified with integration testing is that a big part of the system has to be replaced with mockups which makes the integration tests not useful and hard to maintain. One reason to do less testing of the GraphQL APIs was the trust in the strongly-typed schema validations that the GraphQL libraries implement. The strong typing and validation rules omits the need for certain types of faulty input value testing.

8.9 CQRS

Two of the projects (B,C) used CQRS because of different reasons. The content platform case (B) used an external content management system, from where a read-only copy was propagated to a local persistence layer for better availability and delivery. The event sourced case system (C) built views on the event stream data which reflected the current state of the system. As Fowler writes, CQRS suits well with event-based systems [Fow11]. Each microservice builds their own interpretation of the event data which is passed forward to consumers.

8.10 API protection mechanisms

None of the cases applied attack prevention mechanisms on the GraphQL APIs, although some methods were used for other purposes. Persisted queries were applied in case C to limit public API functionality for users. No query cost analysis was done on the requests on any of the cases. APIs were generally protected using authentication methods for the API clients.

8.11 Organizational patterns

The case projects covered in the interviews followed the usual DevOps organizational patterns. Variations of Scrum or Kanban agile methods were used to develop the case projects. The deployment of the microservices and GraphQL APIs were handled by the development teams. Continuous integration was implemented using automated testing. If tests passed on a pull request, the projects deployed automatically a staging version when the pull request was merged to the code main branch. Code quality was maintained using pull requests and code review workflows using the version control system. The GraphQL schema was used as a communication method for changes in the business logic layer. The GraphQL schema definitions could be kept in version control so code review would have to address also the changes in the schema and possible API breakage could be detected.

Mixed methods of API evolution were detected from the projects. In one project the schema changed during development a lot because of evolving requirements on the domain model. The API breakage was acceptable, because the client application was developed alongside the API. Many interviews presented the idea of handling GraphQL API breakage using a rolling introduction of updated fields or objects.

Updated features are brought alongside the deprecated fields or objects. After that the schema change was applied on client side. When the clients have moved to using the updated GraphQL schema, the deprecated features would be removed from the GraphQL API. Only one case (C) claimed no API breakage by only introducing new fields and objects to the schema. Feature deprecation wasn't needed on the project at the point of the interview.

8.12 Other patterns

One of the case projects made use of automatic schema generation from the database structure. The tooling generated a usable GraphQL server from the structure of the database using the PostGraphile library [Gil19]. Automatic schema generation was considered a really helpful tool in the API development, making the development faster and easier because of not having to write own resolvers for the data fetching, but there were some downsides also. The field and object names in the generated schema weren't client-developer friendly, which could be solved with schema transformations provided by the Apollo GraphQL library. Schema transformations is a feature where a GraphQL schema's object and field naming can be altered while keeping the old schema's functionality the same. The case project hadn't yet adopted the schema transformations because of added development costs of maintaining the schema transforms. Otherwise all the case projects used only well-known software patterns to implement their microservice ecosystem with GraphQL. GraphQL fit well in the role of an API Gateway for most projects.

9 Discussion

By studying scientific research and grey literature, and comparing the findings to the data extracted from interviews some common patterns can be formed for a microservice architecture which uses GraphQL. Eventhough the case projects used a microservice architecture and GraphQL, the problem domains weren't too complex and didn't use DDD practices to model the business domain. A conclusion might be drawn that most of the projects will have a simple problem domain and can get away with using a simpler approach to modeling the business domain. The suggested link between DDD aggregate modeling and GraphQL schema stitching could be studied futher in a follow-up study.

Based on the data, a typical architecture consists of 2-10 REST or GraphQL microservices exposed to public networks using a GraphQL API Gateway. The microservice ecosystem was typically flat, with no layered microservices. In the studied case projects, synchronous request-response communication pattern was most prominently used in the GraphQL APIs, with the exception of the event-based architecture which relied on asynchronous communication to push event streams to clients. Only the event-based architecture used GraphQL subscriptions for push based communication for clients. Asynchronous communication patterns were used in two other projects to make data structures across microservices eventually consistent. Some events were directly triggered from the GraphQL layer, but most of the asynchronous events were published from the business models. The synchronous communication points weren't protected with methods such as the circuit breaker pattern, which cuts connections to non-responsive microservices to prevent connection exhaustion. It could be that most of the projects are small enough to not ever reach connection exhaustion and in a simple microservice architecture cascading failures happen rarely. These advanced communication patterns are only needed in huge ecosystems [Chr19].

Most of the projects used web tokens as the authentication method. This was the recommended authentication method suggested in the GraphQL documentation [Facb]. All projects except the event-based system placed authorization logic in the business models. The event-based system placed authorization logic into the GraphQL resolvers. Placing the authorization logic in the models was recommended in the GraphQL documentation. It can be concluded that web tokens were succesfully used in practice with GraphQL and authorization was done typically in the business logic. Cookies and sessions were used only in one project. Using sessions will limit the API accessibility for other than web clients and for cross-domain access.

Service discovery methods were typically provided by the application platform. There was no clear suggestions for service discovery in the literature, but it was clear from the interviews that a platform-based approach was most prominently used. All the applications had the microservices behind a platform provided load balancer. The microservices were more or less manually configured – no totally automatic scaling was done in none of the projects. It might be that the scale of the projects were not big enough to justify automatic scaling.

Little contract, integration and end-to-end tests were done against the GraphQL API. Some critical queries were tested in two cases (B,C). Most of the testing efforts

were concentrated on unit testing in all the cases. This conforms to the *test pyramid* principle of writing unit tests the most and writing end-to-end test the least [Voc19].

CQRS was used in two cases (B,C). CQRS pattern seemed to work well with GraphQL for the projects. The CQRS pattern was suprisingly used in two of the four projects, eventhough the pattern is supposed to be only for advanced use-cases. The reasons to choose CQRS was because of using a 3rd party CMS as a service, and in the other case used it complementary to the event streams. Ilyenko writes that GraphQL query-mutation-subscription semantics fit well with the CQRS pattern [Ily19]. The mutations are used to trigger commands, which push events to an event store. A persistence layer reads events to store a queryable state, and GraphQL subscriptions can be used to subscribe on the event streams on client-side. GraphQL doesn't seem to prevent the usage of the CQRS pattern as proven by the case projects.

No GraphQL specific API protection mechanisms were used in the case projects, like query cost analysis. Projects mostly relied on authorization as the way to protect API functionality. API protection methods probably become more relevant at the middle of the project, and when API attacks are considered a great risk for application availability. Although the APIs were publicly accessible, none of the projects advertised their API for public use, which makes exploitation less likely.

All the case project organizations used similar DevOps practices. Deploying code automatically from main branch after completing automated tests succesfully was present in all cases. Pull requests and code reviews were used to keep code clean. Based on the findings we suggest that the similar setup can be expected in most of existing projects and should be used on any new microservice and GraphQL projects. Only effect of GraphQL in the organization seemed to be the added communication on API changes through pull requests.

Using code generation was shown in one project (D) where a GraphQL API was generated from a database schema automatically. Code generation wasn't part of the literary research part of this thesis, and would be an interesting subject for additional research.

Criticism for the used interview process might be that the amount of interviews did not meet the suggested minimum of 15 interviews, presented by Hirsjärvi and Hurme [HH09]. The presented results should be taken as preliminary results, and the subject should be more expansively mapped in a follow-up architecture study.

10 Conclusion

In this thesis the GraphQL, a query language for APIs, was studied in the context of the microservice architecture. The thesis was a combination of a literary study and expert interviews. Most prevalent microservice architecture patterns were gathered from peer reviewed studies and grey literature. Also patterns used with the GraphQL API query language was searched from mostly grey literary sources. Four expert interviews were held to detect which patterns were used in practice in projects which had a microservice architecture and implemented GraphQL APIs.

Based on the data, a typical GraphQL microservice architecture consists of up to 10 REST or GraphQL microservices exposed to public networks via a GraphQL API Gateway. The microservice ecosystem was typically flat 3-tier layout with API gateway in front, microservices in the middle, and persistence layer in the end (**RQ1**).

According to grey literature and the expert interviews, GraphQL language ecosystem seems to bring excellent developer experience for microservice architectures. A small learning and development investment has to be done on the part of back-end developers, which can be justified with more relaxed API design requirements upfront (**RQ2**).

Based on the expert interviews GraphQL was a viable alternative to using REST APIs in a microservice ecosystem. There was no perceived limitations compared to other API technologies (**RQ3**). GraphQL seemed to fit typical microservice patterns well. GraphQL was most likely used in the role of an API Gateway in the project. The GraphQL query-mutation-subscribe semantics seemed to fit well with an event-sourced system using CQRS pattern. The GraphQL API authentication patterns didn't seem to differ from REST API authentication – same HTTP transport level limitations were present in both architectures. To use cookie-based sessions with direct API calls, the API has to exist on the same domain as the calling website. Using web tokens via authorization header is more compatible with all kinds of clients and cross-domain requests which suits the GraphQL philosophy of figuring out the exact use-cases later on in the project.

One GraphQL specific pattern was uncovered in the case interviews, which was the use of schema stitching in conjunction with the API Gateway pattern. Although the feature is backed by the GraphQL specification in the schema extension sections, the implementations were considered somewhat experimental. Still, schema stitching

was succesfully used in production setting in two of the four case projects. In two other projects the technology was not seen reliable enough or lacking to be used in the cases.

The following ideas seemed less investigated in research and grey literature and show potential for follow-up studies.

- Measuring the API performance of the proposed combination of REST API and HTTP/2 multiplexing with cascading object fetches, compared to using GraphQL queries. The level of nesting in the GraphQL queries could be measured and compared to equivalent cascading REST fetch.
- Comparing and gathering authentication and authorization implementations used with GraphQL.
- Automatically generated GraphQL schemas for accessing database.
- Comparing caching mechanisms for GraphQL and REST endpoints.

References

AAE16 Alshuqayran, N., Ali, N. and Evans, R., A systematic mapping study in microservice architecture. *2016 IEEE 9th International Conference on Service-Oriented Computing and Applications (SOCA)*, Nov 2016, pages 44–51.

Aba12 Abadi, D., Consistency tradeoffs in modern distributed database system design: Cap is only part of the story. *Computer*, 45,2(2012), pages 37–42.

Abh19 Abhi, A., GraphQL Bindings for Service to Service Communication, 2019. `https://medium.com/open-graphql/graphql-bindings-for-service-to-service-communication-d1e89df66ecd`. [11.2.2019]

Ama19 Amazon, Amazon API Gateway Features, 2019. `https://aws.amazon.com/api-gateway/features/`. [9.1.2019]

And19 Anderson, K., Investigating the CQRS pattern using GraphQL and SPQR, 2019. `https://medium.com/@kenanderson_38113/the-cqrs-pattern-using-graphql-and-spqr-e69c57939582`. [18.2.2019]

Apo19a Apollo, Persisted GraphQL Queries with Apollo Client, 2019. https://blog.apollographql.com/persisted-graphql-queries-with-apollo-client-119fd7e6bba5. [20.2.2019]

Apo19b Apollo, Schema stitching - Combining multiple GraphQL APIs into one, 2019. https://www.apollographql.com/docs/graphql-tools/schema-stitching.html. [17.1.2019]

Apo19c Apollo, subscriptions-transport-ws - A GraphQL WebSocket server and client, 2019. https://github.com/apollographql/subscriptions-transport-ws. [20.2.2019]

Ayo18 Ayoub, M., Microservices Authentication and Authorization Solutions, 2018. https://medium.com/tech-tajawal/microservice-authentication-and-authorization-solutions-e0e5e74b248a. [24.4.2018]

BHJ+ Balalaie, A., Heydarnoori, A., Jamshidi, P., Tamburri, D. A. and Lynn, T., Microservices migration patterns. *Software: Practice and Experience*, 48,11, pages 2019–2042.

Bre12 Brewer, E., Cap twelve years later: How the" rules" have changed. *Computer*, 45,2(2012), pages 23–29.

CDP17 Cerny, T., Donahoo, M. J. and Pechanec, J., Disambiguation and comparison of soa, microservices and self-contained systems. *Proceedings of the International Conference on Research in Adaptive and Convergent Systems*, RACS '17, New York, NY, USA, 2017, ACM, pages 228–235.

CDT18 Cerny, T., Donahoo, M. J. and Trnka, M., Contextual understanding of microservice architecture: Current and future directions. *SIGAPP Appl. Comput. Rev.*, 17,4(2018), pages 29–45.

Chr19 Christensen, B., Introducing Hystrix for Resilience Engineering, 2019. https://medium.com/netflix-techblog/introducing-hystrix-for-resilience-engineering-13531c1ab362. [17.3.2019]

CMRW07 Chinnici, R., Moreau, J.-J., Ryman, A. and Weerawarana, S., Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language, 2007. https://www.w3.org/TR/wsdl20/. [25.02.2019]

Daw18 Dawkins, J., Authorization in GraphQL, 2018. https://blog.apollographql.com/authorization-in-graphql-452b1c402a9. [15.5.2018]

DRS17 Diepenbrock, A., Rademacher, F. and Sachweh, S., An ontology-based approach for domain-driven design of microservice architectures. *INFORMATIK 2017*, Eibl, M. and Gaedke, M., editors. Gesellschaft für Informatik, Bonn, 2017, pages 1777–1791.

EGHS16 Ebert, C., Gallardo, G., Hernantes, J. and Serrano, N., Devops. *IEEE Software*, 33,3(2016), pages 94–100.

Eva03 Evans, E., *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison Wesley, 2003.

Faca Facebook, GraphQL Learning Documentation: Authentication and Express Middleware. https://graphql.org/graphql-js/authentication-and-express-middleware/. [29.1.2019]

Facb Facebook, GraphQL Learning Documentation: Authorization. https://graphql.org/learn/authorization/. [8.1.2019]

Facc Facebook, GraphQL Website. https://graphql.org/. [16.11.2018]

Fac15a Facebook, GraphQL - Working Draft - July 2015, 2015. https://github.com/facebook/graphql/releases/tag/July2015. [16.11.2018]

Fac15b Facebook, GraphQL Release Notes July 2015, 2015. https://github.com/facebook/graphql/releases/tag/July2015. [16.11.2018]

Fac18a Facebook, GraphQL - June 2018 Edition, 2018. https://github.com/facebook/graphql/releases/tag/June2018. [16.11.2018]

Fac18b Facebook, GraphQL Release Notes June 2018, 2018. https://github.com/facebook/graphql/releases/tag/June2018. [16.11.2018]

Fac19 Facebook, Graph API, Making Batch Requests, 2019. https://developers.facebook.com/docs/graph-api/making-multiple-requests/. [21.2.2019]

Fow Fowler, M., MonolithFirst. https://martinfowler.com/bliki/MonolithFirst.html. [31.1.2019]

Fow10 Fowler, M., The Richardson Maturity Model, 2010. https://martinfowler.com/articles/richardsonMaturityModel.html. [21.2.2019]

Fow11 Fowler, M., CQRS - Command Query Responsibility Segregation, 2011. https://martinfowler.com/bliki/CQRS.html. [18.2.2019]

Fow14 Fowler, M., Microservices - a definition of this new architectural term, 2014. https://martinfowler.com/articles/microservices.html. [18.3.2019]

Fow19a Fowler, M., CircuitBreaker, 2019. https://martinfowler.com/bliki/CircuitBreaker.html. [27.2.2019]

Fow19b Fowler, M., Consumer-Driven Contracts: A Service Evolution Pattern, 2019. https://martinfowler.com/articles/consumerDrivenContracts.html. [14.2.2019]

Fow19c Fowler, M., MicroservicePrerequisites, 2019. https://martinfowler.com/bliki/MicroservicePrerequisites.html. [28.1.2019]

Fow19d Fowler, M., Testing Strategies in a Microservice Architecture, 2019. https://martinfowler.com/articles/microservice-testing/. [15.2.2019]

Fra17 Francesco, P. D., Architecting microservices. *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, April 2017, pages 224–229.

FT00 Fielding, R. T. and Taylor, R. N., *Architectural styles and the design of network-based software architectures*, volume 7. University of California, Irvine Irvine, USA, 2000.

GHM+07 Gudgin, M., Hadley, M., Mendelsohn, N., Moreau, J.-J., Nielsen, H. F., Karmarkar, A. and Lafon, Y., SOAP Version 1.2 Part 1: Messaging Framework (Second Edition), 2007. https://www.w3.org/TR/soap12-part1/. [22.02.2019]

Gil19 Gillam, B., PostGraphile Instant GraphQL API, 2019. https://www.graphile.org/postgraphile/. [17.3.2019]

Goo19 Google, Protocol Buffers, 2019. https://developers.google.com/protocol-buffers/. [17.3.2019]

Gra19 Graphql, graphql-js - A reference implementation of GraphQL for JavaScript , 2019. https://github.com/graphql/graphql-js. [5.2.2019]

Gri18 Griffiths, E., GraphQL as an API Gateway to Microservices, 2018. https://blog.codeship.com/graphql-as-an-api-gateway-to-micro-services/. [18.3.2019]

HH09 Hirsjärvi, S. and Hurme, H., *Tutkimushaastattelu*. Gaudeamus, 2009.

HP17 Hartig, O. and Pérez, J., An initial analysis of facebook's graphql language. *Proceedings of the 11th Alberto Mendelzon International Workshop on Foundations of Data Management and the Web. :*, volume 1912 of *CEUR Workshop Proceedings*. Juan Reutter, Divesh Srivastava, 2017.

Hun17 Hunter, Thomas, I., *Advanced microservices : a hands-on approach to microservice infrastructure and tooling*. Apress, San Francisco, California, 2017.

Ide15 Ideskog, J., How To Control User Identity Within Microservices, 2015. https://nordicapis.com/how-to-control-user-identity-within-microservices/. [14.5.2015]

Ily19 Ilyenko, O., Event-stream based GraphQL subscriptions, 2019. https://gist.github.com/OlegIlyenko/a5a9ab1b000ba0b5b1ad. [1.3.2019]

Mer Meredith, C., Designing GraphQL Mutations. https://blog.apollographql.com/designing-graphql-mutations-e09de826ed97. [28.03.2017]

Mir Miri, I., Microservices vs. SOA. https://dzone.com/articles/microservices-vs-soa-2. [4.1.2017]

Nem18 Nemek, Z., REST vs. GraphQL: A Critical Review, 2018. https://blog.goodapi.co/rest-vs-graphql-a-critical-review-5f77392658e7. [21.2.2019]

Net19 Netflix, Netflix Falcor, 2019. https://netflix.github.io/falcor/. [21.2.2019]

Ote18 Otemuyiwa, P., Layering GraphQL on top of REST - How to use Apollo with an existing PHP API, 2018. https://blog.apollographql.com/layering-graphql-on-top-of-rest-569c915083ad. [18.3.2019]

Pac18 Pacheco, V. F., *Microservice Patterns and Best Practices*. Packt, 2018.

PM19 Powell-Morse, A., Domain-Driven Design – What is it and how do you use it?, 2019. `https://airbrake.io/blog/software-design/domain-driven-design`. [21.2.2019]

Ric18 Richardson, C., *Microservice Patterns*. Manning Publications Company, 2018.

Rig19 Righetto, D., OWASP JSON Web Token Cheat Sheet for Java, 2019. `https://github.com/OWASP/CheatSheetSeries/blob/master/cheatsheets/JSON_Web_Token_Cheat_Sheet_for_Java.md`. [17.3.2019]

Sar19 Sarin, P., What's the point of the Aggregate pattern?, 2019. `https://medium.com/@philsarin/whats-the-point-of-the-aggregate-pattern-741a3132da5c`. [28.2.2019]

Sch15 Schrock, N., GraphQL Introduction, 2015. `https://reactjs.org/blog/2015/05/01/graphql-introduction.html`. [16.11.2018]

Sch19 Schrax, graphql-cost-analysis - GraphQL Query Cost Analysis for graphql-js, 2019. `https://github.com/pa-bru/graphql-cost-analysis`. [20.2.2019]

Sto19 Stoiber, M., Securing Your GraphQL API from Malicious Queries, 2019. `https://blog.apollographql.com/securing-your-graphql-api-from-malicious-queries-16130a324a6b`. [20.2.2019]

Swa19 Swagger, OpenAPI Specification, 2019. `https://swagger.io/specification/`. [21.2.2019]

TLP18 Taibi, D., Lenarduzzi, V. and Pahl, C., Architectural patterns for microservices: A systematic mapping study. 03 2018.

Ver13 Vernon, V., *Implementing Domain-Driven Design*. Pearson Education, 2013.

Voc19 Vocke, H., The Practical Test Pyramid, 2019. `https://martinfowler.com/articles/practical-test-pyramid.html`. [1.3.2019]

Vog09 Vogels, W., Eventually consistent. *Commun. ACM*, 52,1(2009), pages 40–44.