TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

Taavet Tamm 185007IADB

GraphQL API Implementation in a Distributed System

Bachelor's thesis

Supervisor: Aleksei Talisainen
MSc

Tallinn 2021

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Taavet Tamm 185007IADB

GraphQL API rakendamine hajussüsteemis

Bakalaureusetöö

Juhendaja: Aleksei Talisainen
MSc

Tallinn 2021

Author’s declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Taavet Tamm

17.05.2021

Abstract

The aim of thesis was to create an extendable and documented API layer for existing collection of microservices, to serve a new mobile application. The goal was to reduce response times for clients and avoid exposing every service to public web.

The analysis evaluates suitable technologies with future maintainability and performance in mind. The implementation focuses on tools and approaches that are usable within chosen technology: GraphQL.

The result was set of additional microservices and defined approaches for building future services. The client integration became easier to use and change.

This thesis is written in English and is 33 pages long, including 7 chapters, 17 figures and 1 table.

Annotatsioon

GraphQL API implementatsioon hajussüsteemis

Käesoleva lõputöö eesmärk oli luua laiendatav ja selgelt dokumenteeritud API kiht kasutades varem loodud mikroteenuseid. Töös käsitletud taustteenuse ehitamisega samaaegselt loodi uus mobiili rakendus. Lõputöös käsitletud projektile esitatavad nõuded keskenduvad eelkõige päringute kiirusele, turvalisuse ja halllatavuse aspektidele.

Lõputöö analüütiline osa hindab erinevate tehnoloogiate sobivust laiendatava API loomiseks. Praktiline osa keskendub valitud tehnoloogia (GraphQL) implementatsioonile mikroteenustest koosnevas süsteemis ja kirjeldades kasutatud lahendusi.

Töö tulemusena loodi mitmed uued mikroteenused ja defineeriti lähenemised, mida saab kasutada edaspidi GraphQL teenuseid luues. Mobiilirakenduse suhtlus taustteenusega kiirenes ning arenduskiirus kasvas.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 33 leheküljel, 7 peatükki, 17 joonist, 1 tabelit.

List of abbreviations and terms

ACID	An acronym to remember the key principles of a transactional database system, stands for Atomic, Consistent, Isolation, and Durability
API	Application Programming Interface
Golang	Statically typed, compiled programming language designed at Google
Horizontal scaling	Scaling by adding more machines to your pool of resources
JWT	An open standard (RFC 7519) that defines a compact and self-contained way for securely transmitting information between parties as a JSON object, stands for JSON Web Token
Middleware	Software that acts as a bridge, functions that execute during the lifecycle of a request
MongoDB	Source-available cross-platform document-oriented database program
Node.js	JavaScript runtime environment
npm	Node.js package manager
PostgreSQL	Free and open-source relational database management system emphasizing extensibility and SQL compliance
Projection	Taking a vertical subset from the columns of a single table that retains the unique rows
Push notification	Style of Internet-based communication where the request for a given transaction is initiated by the publisher or central server.
React Native	Open-source mobile application framework
REST	Representational State Transfer
RESTful	An API that conforms to REST architectural style.
ServiceArea	Polygon on map where customers can use a product.
SMS	Short messaging service
UI	User Interface
Yaml	Human-readable data-serialization language, commonly used for configuration files

Table of contents

List of figures

List of tables

1 Introduction

When applications grow so grows the complexity and need to experiment and change the features these provide to users. As team grows and the amount of code to maintain increases then having a good set of documented, reusable end optimised backend API (application programming interface) services, becomes increasingly important. The following thesis analyses the approaches of building backend services as a distributed system and looks at ways to provide consistent and extendable API layer to a mobile application.

As an example, the author covers implementation of food delivery application. In the scope of 6 months an existing application prototype was completely redesigned and rewritten from scratch as a React Native (open-source mobile application framework) application. As a part of this redesign effort existing API layer was to be redesigned to build an extendable and reliable platform. In scope of this thesis the author covers implementation and design of backend system, focusing on problems and solutions around creating a well-documented and extendable API.

At the start of the project, it was decided to approach the solutions from new designs and not consider the limitations of existing prototype implementation with existing data models and API layer. The project was framed into 6-month long window a launch date was set and feature list was composed based on the userbase of previous prototype application. The development was conducted in two distinct teams, where backend developers and application developers kept separate backlogs.

The author of this thesis participated in the project as a backend developer in a team of five people, with the focus on designing and implementing the API structure that application can use. Author of this thesis had previous experience with GraphQL API, so in this project his role was to implement schema sharing solutions and choose overall approaches for GraphQL based microservices microservices.

The engineering department accountable for this consumer facing application, and several other services, had settled on set of technologies. Node.js and Golang are used as programming languages, and shared set of internal libraries are maintained for these languages. MongoDB and PostgreSQL are used as database solutions with toolset around availability. It was agreed that when choosing technologies for project in question, the cost of maintaining is considered and extending the existing set needs to have clear and substantial benefits.

Existing prototype application had served the company well in gaining an initial userbase and validating product market fit. But as the count of customers increased, the application started to be perceived as slow and unreliable. It was identified that to continue growing an increasing number of features need to be implemented and experimented with. The main driver of project covered in this thesis is the decision to start using react-native as mobile application platform and rewrite the existing prototype, to increase speed of feature development. While implementing backend services the author had to keep in mind the available tools that react-native has for data transfer.

2 System requirements

The following chapter covers the relevant descriptions of domain models and non-functional requirements set for backend service. Only the requirements classified as essential for first launch of the application are covered here. The feature list is derived from existing prototype application and new designs approved by internal stakeholders.

2.1 Browsing list of venues

Each user of application should be able to view venues based on the geographical coordinates of the delivery destination they choose. Each venue is displayed with status based on the working hours. Venues include a list of items with stock levels. Each venue and item in venue include at least a name and an image but can include various other properties. Items can have several variations, for example pizza with extra cheese, and belong to categories. To list relevant venues, each one should be grouped by locality (ServiceArea) it exists in. Figure 1 illustrates the most important relationships in the data model.

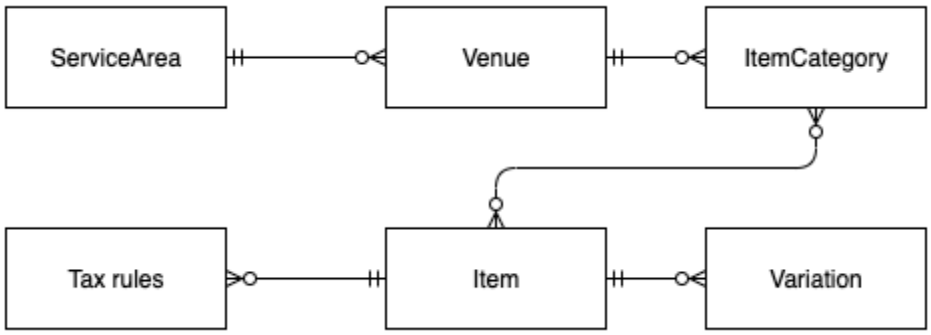

Figure 1. General overview of required data model.

2.2 Composing shopping cart

Each user of application should be able to add items into shopping cart, unfinished shopping cart contents are persisted in device local storage. Shopping cart includes item references and quantities. Shopping cart contents are validated each time an item is placed into it, taxes are recalculated, item stock level is rechecked.

2.3 Placing an order

Each user of application should be able to choose a payment method and place an order with shopping cart contents. The application should support several different payment methods. Before placing an order, a user must verify their phone number and provide an email address. Email address is used to forward final receipt to the user and phone number is used to notify the user about delivery.

2.4 Tracking progress of an order

Each user of application should be able to place one order at a time. When a user had an order that has not been completed yet, they should be able to see the location and status of the order. Notifications about order progress should be sent to the user in form of push notifications and SMS. When an order reaches delivery point the user should be able to pick up the order and mark it received on the application.

2.5 Non-functional requirements

Following chapter highlights relevant non-functional requirements that were defined for the mobile application backend service.

Firstly the users should be identified by verified phone number, a reinstall with the same phone number should be the same user. Each user can have multiple devices. Devices are identified by reference given to each application installation and should not reference physical devices. Users should not be prompted to login at the start of each session, but the app should keep a password in device secure storage and use authentication tokens to communicate with the API service.

The backend service should be deployed as a docker container and should be allow for horizontal scaling. The API definitions that backend service should be documented and provide a way to generate client definitions with react-native tools. Backend service should follow internal company observability guidelines. The mean response time of backend responses should be under one second with every type of query. The service should be written in either Golang or Node.js and use PostgreSQL or MongoDB as database.

3 Analysis of relevant technologies

The following chapters analyse the choices for technologies, considering functional and no-functional requirements.

3.1 API Schema definitions

The project requirements state that all the API definitions should be well documented and allow for generating client code in react-native. Two common ways to provide schema definitions are Openapi and GraphQL. Following chapter analyses the suitability of these technologies in the context of this project.

Openapi specification [1] defines language interfaces for RESTful APIs. The tools for editing schema and generating server and client code are well established. Swagger open-source tools make using Openapi schema definitions easier, providing human readable web interface automatically from definition and allowing for code generation in several languages. Swagger development started already in 2010 [2] and there are many examples and guides for implementation. Using this approach implies that the API follows REST conventions [3], changes should to versioned, endpoints should be defined based on data model and queries should use relevant http methods: GET, POST, PUT, DELETE.

GraphQL is not as old and established as Openapi specification, but development is still active, and specification is defined. There are examples other companies using this technology and achieving reliable results, for example Netflix [4] and Airbnb [5]. GraphQL has been gaining popularity in recent years, and there are several frameworks to choose from. This specification does not imply that API is built following RESTful conventions, instead it defines a query language and requires that all communication uses GraphQL query syntax. GraphQL frameworks also provide web UI, which in use while estimating felt easier and more consistent to use for the author. All GraphQL queries use single endpoint and POST method, and all monitoring must rely on request body. GraphQL query language allows the client to fetch only the data it needs, changes to the schema are not versioned.

Table 1 compares most relevant features of two schema definition approaches mentioned above, considering the main differences in the context of example project.

Table 1. Comparision of GraphQL and Openapi technologies

	GraphQL	Openapi
Data consumption	Fetch what is needed for UI	Prone to under-fetching or over-fetching of data
Query count	Batch fetch multiple objects in single http request	Requires multiple roundtrips for multiple entities
Maturity	Not as well established	Relies on proven RESTful approaches

This project started because existing prototype application was slow to develop and did not provide smooth experience to users. It is expected that there is high value in having an API layer which allows to change the data queries while using exiting endpoints. Using GraphQL allows for less data usage between client and server, leading to more speed in some cases [6].

3.2 Data storage

The non-functional requirements of this project allow for using two data stores, MongoDB, and PostgreSQL, in this chapter suitability for different GraphQL queries is analysed in context of two different data stores.

Several data models already exist, and should be reused, as other services and applications rely on these objects. MongoDB is extensively used to keep configurations for venues and items, these are mostly models that change rarely and store loosely structured data. PostgreSQL is used for entities that are more transactional to store payment records and order receipts.

GraphQL is data store agnostic, but there are several ways of fetching data efficiently. In general query resolvers should rely on projections and [7] to fetch data from database and avoid creating n+1 queries for nested data structures.

3.4 Apollo platform

At the time of this project the most widely supported tools for creating and maintaining GraphQL service is Apollo Data Graph Platform [7]. Although some alternatives exist for example Nautilus library, Apollo tools are most relevant in this project as existing system is already composed of few separate services. Apollo tools also have comprehensive documentation and significant amount of starts in GitHub. Key feature that Apollo platform provides is schema federation, which allows the project to use several microservices but expose only a single gateway service to mobile application. Apollo platform also provides libraries for http server and middlewares which are important from observability standpoint. By using a consistent approach for serving API definitions in form of Apollo server library it is simpler to develop and share separate microservices as they are more similar. Apollo platform tools are well documented and standardized, federation is also supported by Golang tools for GraphQL.

4 Implementation of distributed GraphQL schema

The following chapters illustrates challenges that using GraphQL creates, and describes solutions used in the context of building food delivery application.

4.1 GraphQL overview

This chapter attempts to describe general concepts of GraphQL for reader who might not be familiar with the query language and aims to highlight relevant parts from query language specification [8].

GraphQL is a language used to make requests to application services that have capabilities defined in this specification. It provides a product centric view for building applications on top of data models, allowing to consider the data needed for a feature without the need to create new data sources. GraphQL defines an application specific type system and requires that all services that are exposed to end user facing application follow a strongly typed schema definition. GraphQL allows for nested data structures and requires that every client defines a query for every request, describing explicitly what the API service should respond with. Figure 2 and figure 3 illustrate the format of simple GraphQL request, where only the name attribute of user is fetched.

```
{
  user(id: 4) {
    name
  }
}
```

Figure 2 Example GraphQL query for name of a user with id 4

```
{
  "user": {
    "name": "Mark Zuckerberg"
  }
}
```

Figure 3 Example GraphQL query response (in JSON)

GraphQL supports two main operations *Query* and *Mutation.* Queries are intended to fetch data from server, every query is explicitly defined in the schema and accepts a set of inputs. Similarly, every query defines the type that client can expect in response. Schema can define required and optional fields. Schema fragment defined in code examples on figure 4 and figure 5 includes a query for fetching only available venues

Query for active venus returns an array of *Venue* objects where exclamation mark denotes a non-null value, this query returns empty list or a list of venues.

```
{
  type Query {
    activeVanues(serviceAreaId: ID!): [Venue!]!
  }
}
```

Figure 4 Schema fragment with query definition for available venues

```
{
  query {
    activeVenues("5990a67459d177ad96dbe5ee”) {
      _id
      title
      items {
        _id
        name
      }
    }
  }
}
```

Figure 5 Example query that a client can execute against query definition of venues

Similarly, GraphQL supports definition of mutation that is intended to execute change operation against API service. Mutation types are defined in GraphQL schema and client can define the structure of expected data. Code example in figure 6 illustrates schema fragment that defines mutation for creating a new order from shopping cart contents.

```
{
  type Mutation{
    createOrder(createOrderInput: CartData!): [Order!]!
  }
}
```

Figure 6 Example mutation definition for creating a new order

One of biggest benefits of GraphQL is a developer friendly web interface to interact with backend API. This web UI is useful in development mode, where developers can enjoy a query interface with autocompletes and try out data fetching approaches. Having strongly typed schema and consistent documentation makes it easy to start new applications in the future or share API definitions. Figure 7 includes a screenshot of GraphQL web UI to illustrate what the documentation looks like.

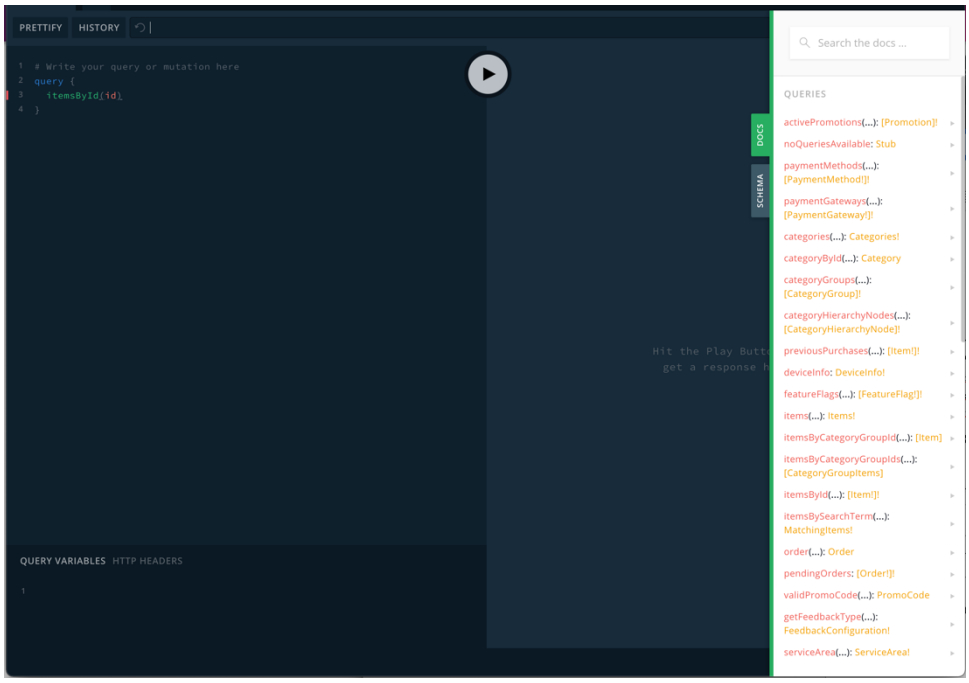

Figure 7 An example of GraphQL web UI

4.2 Implementation of GraphQL gateway

This chapter describes the concept of federation in GraphQL and covers implementation of API gateway using Apollo tools in Node.js.

Having different data models in different services creates a system where features that the application provides to end-users needs to fetch data across multiple different domains, databases, and services. To make application logic easy to change and facilitate future features a separate service that composes a general schema across all relevant services is used. This approach allows client to easily combine different data sources and change requested data based on screens that end-user sees. As an alternative it was considered to include different endpoints into application itself, but this approach would make splitting services or moving them harder. Figure 8 includes a diagram of simplified system architecture using a separate gateway service.

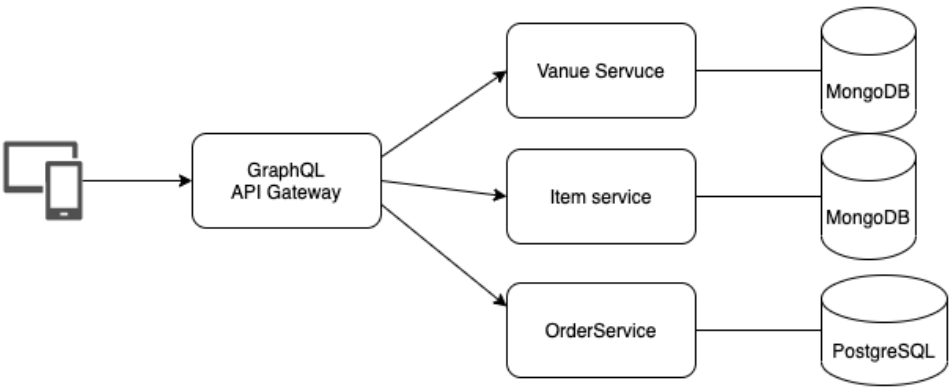

Figure 8 Simplified structure of distributed system

Apollo GraphQL Platform provides a concept of federation to solve this problem. An Apollo Federation architecture consists of two parts. First component is a collection of implementing services that each define a distinct GraphQL schema. Second part is a gateway that composes the distinct schemas into a single data graph and executes queries across the services in the graph. Apollo tooling includes a query planner which finds the most optimal path to calling other services.

To use gateway library from apollo platform a separate service was created and deployed by the author of this thesis. This server is implemented in Node.js to take advantage of npm libraries that apollo platform maintains. To create a webserver latest version of *apollo-server* library was used. For composing federated graphql *@apollo/gateway* was included. The code example on figure 9 illustrates the usage of these libraries, and creation of gateway consisting of three other services, finally setting up the server to listen for traffic with *listen* function.

```
import { ApolloServer } from 'apollo-server';
import { ApolloGateway } from '@apollo/gateway';
const gateway = new ApolloGateway({
  serviceList: [
    { name: 'venues', url: 'http://venue-service' },
    { name: 'venues', url: 'http://item-service' },
    { name: 'venues', url: 'http://order-service' },
  ],
});
const server = new ApolloServer({
  gateway,
  subscriptions: false,
});
server.listen();
```

Figure 9 Node.js Snippet to create the initial GraphQL gateway server

4.3 Implementation of microservice for item fetching

This chapter covers the implementation of one mongo backed service which is part of general federated schema. Implementation of other services is similar, here specific implementation of item service is considered as an example.

Item microservice must provide schema to search and fetch products for venue views. Schema parts can be split among several files inside item service, a tag *gql* is used to define GraphQL schema definitions. The following code example on figure 10 illustrates the definition of two queries *itemsByCategory* and *itemsBySearchTerm,* latter query requires slightly more complicated parameters therefore the input grouped into separate input type definition. GraphQL specification defines a common *PageInfo* type, which is intended for paginating results with a pointer, providing last pointer as *after* parameter along with *limit* to indicate how many additional items should be returned to the client.

```
import { gql } from 'apollo-server-core';
const typeDefs = gql`
    type Item {
        _id: ID!
        name: String
        status: ItemStatus
    }
    type Items {
        data: [Item!]!
        pageInfo: PageInfo!
    }
    input ItemSearchInput {
        serviceAssignmentId: ID!
        searchTerm: String!
        after: String
        limit: Int = 10
    }
    type MatchingItems {
        _id: Int!
        timestamp: Int!
        searchTerm: String!
        data: [Item!]!
        pageInfo: PageInfo!
        totalItems: Int!
    }
    extend type Query {
        itemsByCategory(categoryId: ID!, after: String,
limit: Int = 10): Items!
        itemsBySearchTerm(itemsBySearchTermInput:
ItemSearchInput!): MatchingItems!
    }
`;
```

Figure 10 Code example of item service schema definition

With GraphQL schema defined in code it is possible to define relevant query resolvers. The resolvers are kept as minimal as possible and considered as controllers, choosing which service layer element to reference. The following code example on figure 11 illustrates fetching of items from different service layer entities using *byCategory* and *byAssignment* functions of *itemService.*

```
const queryResolvers = {
  itemsByCategory: async (parent, { categoryId, after, limit },
context) => {
    const itemService = new ItemService();
    return paginatedResults(itemService.byCategory(context.span,
categoryId, after, limit));
  },
  itemsBySearchTerm: async (
    parent,
    { itemsBySearchTermInput: { serviceAssignmentId, searchTerm,
after, limit } },
    ctx
  ) => {
    const itemService = new ItemService();
    const result = await
paginatedResults(itemService.byAssignment(ctx.span,
serviceAssignmentId, searchTerm, after, limit));
    return {
      searchTerm,
      serviceAssignmentId,
      ...result,
    };
  },
};
```

Figure 11 Code example of item service resolvers

Both query resolvers rely on a helper function to provide pagination definitions along with requested data. Following code example on figure 12 illustrates this helper function, defining cursor values and *pageInfo* parameters based on GraphQL specification. The client application can then rely on these values to fetch more items, as necessary.

```
async function paginatedResults(dataQuery) {
  const { results, previous, hasPrevious, next, hasNext } =
      await dataQuery;
  return {
    data: results,
    pageInfo: {
      startCursor: previous || '',
      endCursor: next || '',
      hasNextPage: hasNext,
      hasPreviousPage: hasPrevious,
    },
  };
}
```

Figure 12 Code example of pagination helper for GraphQL

Item objects can have many optional and nested attributes, for example modifiers, therefore MongoDB is used to store these entities. To access MongoDB item service uses

library called mongoose which makes it easier to define the schema for stored documents and query these. Following code example on figure 13 illustrates fetching of items based on categoryId. While using mongoose a separate library is included in every schema definition *mongo-cursor-pagination* this library allows using same style of cursors as GraphQL *pageInfo* and simplifies fetching of items.

```
class ItemService {
  async byCategory(span, categoryId, after, limit) {
    Item.model.paginate({
      query: {
        status: ItemStatus.ACTIVE,
        category: categoryId
      },
      next: after,
      paginatedField: 'name',
      limit: limit,
    })
  }
}
```

Figure 13 Code example of database query in item service

Item service serves the schema for products on endpoint */graphql,* this endpoint is in turn called by API gateway with introspection queries and gets included in API gateway to be served as part of schema available to mobile application. Item service itself is not exposed to public web traffic and is only available to other services. In development mode it is possible to execute item specific queries directly against item service without having an API gateway. In service implementation the usage of API gateway is not required – schema can be used by several different application or included into different gateways.

4.4 Implementation of order service

Some already existing services are implemented using Golang. This chapter covers implementation of one PostgreSQL backed service which is part of federated schema.

User of mobile application can compose a shopping cart and pass the contents along to order service. This service stores the records of user purchase including item price at the time and the quantity of each item. By creating an order, the user takes the responsibility of paying for the chosen goods. Due to nature of these entities, it was established that this

service should provide strong ACID guarantees. Order service was implemented using strongly typed programming language and uses a relational database to store the records.

To provide GraphQL definitions to mobile application order service uses a library gqlgen [10]. This library allows the service to keep GraphQL definitions in a separate file with *.graphqls* extension. Figure 14 illustrates the schema that order service defines. Here a separate *Stub* type is defined to have a single *@key* annotation, to ensure that generated schema includes type definitions that federation requires.

```
input CartItemInput {
    itemID: ID!
    qty: Int!
}
input CoordinatesInput {
  lat: Float
  lng: Float
}
input CreateOrderInput {
  venueID: ID!
  cartItems: [CartItemInput!]!
  pupCoordinates: CoordinatesInput!
}
type Order {
    _id: ID!
}
type Mutation {
  createOrder(createOrderInput: CreateOrderInput!): [Order!]!
}
type Stub @key(fields: "stub") {
  stub: String
}
```

Figure 14 Code example of order service schema definition

After defining order service schema gqlgen is used to generate type definitions in code based on the schema. Code generation commands need to be run every time schema changes are made. Example commands on figure 15 illustrate how to install the library and generate source code, after running these commands a folder called “graph” is created along with empty resolver functions. The library provides a configuration file in yaml format, which allows to configure the schema as *federated.*

```
> go get github.com/99designs/gqlgen
> go run github.com/99designs/gqlgen init
> gqlgen
```

Figure 15 Example commands for creating GraphQL schema in Golang

Resolver functions in order service are lightweight routes and reference domain repositories and internal code structures. Order service code logic is split into several folders including: *entity, service, repository* to organise and implement domain-based business logic. Order service is also in charge for communicating with message queue and changing the status of order, refunding orders, and removing out of stock items resulting in partially fulfilled orders. The service is available only for internal use and is not exposed to public web traffic. Order service publishes only the operations that mobile application should be able to perform on the order as GraphQL definitions and does not for example include refund queries which should be approved by customer support.

4.5 Extending federated graph types

In some instances, domain models are not shared as a whole, instead some fields are resolved by separate services. This chapter describes how to use Apollo federation tools to maintain a valid schema with shared fields between microservices.

As an example, recommendations service should define a relationship between the item and recommended items, without defining the fields already in item service schema. Apollo GraphQL gateway sets a limit to how schemas can be defined: each type should be defined once. Therefore, it is not possible to copy the same definition from one service to another. Federation defines several annotations that can be used to extend and reuse schema definitions. Figure 16 illustrates definition of item schema with federation *@key* directive, to tell other services what field is used as unique identifier in this case *_id.*

```
type Item @key(fields: "_id"){
    ...
}
```

Figure 16 Code example of extendable GraphQL schema type with @key annotation

Having federation directive in item service allows recommendation service to extend item type with an additional field. Figure 17 illustrates how recommendations service defines

schema to provide *recommendedItems* when mobile application requests this field. Here *RecommendedItems* are resolved based on item tags and name.

```
extend type Item @key(fields: "_id") {
  _id: ID! @external
  name: String! @external
  tags: [String!]! @external
  recommendedItems:
     [ServiceAssignmentRecommendation!]!
     @requires(fields: "name tags")
}
```

Figure 17 Code example extendable schema object with annotations

This approach allows recommendations service to still have a valid schema and accept queries directly against service endpoints, while not having complete schema definitions in the source code. Recommendation service relies on function *buildFederatedSchema* from Apollo federation library [10] to compose a fully qualified schema.

4.6 Observability

With distributed system consisting of several microservices it is important to follow uniform guidelines for observability. The following chapter illustrates main approaches used in GraphQL context to conform to established policies in the company.

With RESTful API servers request level middlewares are used to log query path, response time and http method. To have similar level of monitoring in GraphQL service the names of queries must be accounted for. Apollo server [8] provides similar features in form of plugins. Plugin allows services to get information from parsed query and use query name from request context. Logging full request body is avoided as some queries can include sensitive data in query variables. Several plugins were created to create uniform logs, gather business metrics based on GraphQL query names and to measure performance with existing toolsets used in the company.

The company is using distributed tracing [12] conventions to maintain overview of requests and services that are involved in producing responses to client requests. To maintain visibility into request lifecycle GraphQL gateway starts a new trace and defines an extended data source injecting a trace into http headers assigning consistent tags to every request. Every microservice including gateway itself has a middleware that can

continue trace from incoming request headers and add new steps to it. A consistent header naming is ensured by using internal libraries which serve as a wrapper functions for opensource tools.

4.7 Authentication

The requirements of application state that users should not sign up and should be able to use the application without an account. Certain operations still require customers to provide identifiable information and service security should still a non-functional requirement. Following chapter gives a broad overview of approaches used in securing mobile application in the context of example application.

Initial requirements state that customers should receive receipts to their email address and should be notified using SMS. In the context of this project email and phone number are personal information and are stored separately in auditable form with restricted access. This information is asked only when user places an order. Phone numbers are verified through third part verification library. Customer can choose to restore previous purchases based on the phone number and receive personalised recommendations.

Every time a customer installs the application a unique identifier is generated and stored in device secure storage. This identifier is used to register a device through a mutation in GraphQL schema. After registration process the application can request a JWT in boot up processes, this token is then validated every time the application client makes a request to the GraphQL gateway. Device context is propagated across services through HTTP headers, to provide consistent responses and serves as cache key where possible. Validating JWT in every request gives the system confidence that requests are indeed coming from the mobile application. Having device reference accommodates several features such as the consistent colour or sorting of design elements. Every customer can have several devices, but only a single device is connected to an order that the customer places.

5 Result

The project covered in this thesis was the first iteration of new mobile application release. The following chapters give an overview of performance and maintenance aspects after evaluating API usage in production environment.

5.1 Client GraphQL usage

After releasing the new mobile application to app stores a retrospective meeting was held to gather feedback and analyse the outcomes of the project. It was identified that mobile application development became much easier as developers can interact with GraphQL web UI to read API documentation and create test queries. Having a single gateway for requests also allowed mobile developers to use code generation and type definitions from GraphQL schema directly in the code base. Generating type definitions in prototype application was not possible as it communicated with several different services with inconsistent documentation and in some cases conflicting definitions. After two months it was noted that no new bug reports where the root cause is improper API query or a typo in field names have been created for new application.

It was possible to demonstrate that using GraphQL the mobile appication achieves an overall smaller mobile data usage for customer devices, adding additional fields to GraphQL schema does not impact previous query sizes. Having a consistency in microservices that serve mobile application enabled the analysis of overall application performance. New dashboards that visualize query level statistics were used to find specific queries that are slow and need optimization. From load balancer statistics it was possible to see that average response times reduced after the release of new mobile application.

Many individual microservices became accessible through internal network only. GraphQL gateway became a clear authentication layer between mobile application and backend services reducing the amount of known attack vectors that the system is vulnerable to. Having a single gateway simplifies throttling rules that load balancer can apply to API clients.

5.2 Downsides and future considerations

As every request in GraphQL uses the same http method and endpoint then tools used before to rate limit API requests from single client became harder to configure. Previously it was possible to maintain statistics based on RESTful API paths, but this can no longer be done as each query can include diverse set of fields, so the whole request body must be considered for every measurement. The tools for analysing POST body are not as well adopted and documented as endpoint-based alternatives.

Using Apollo Gateway as a central piece for external web traffic also made this into a single point of failure. A risk was identified that if API gateway would have any problems, then the mobile application would be completely unusable even if some set of other services remain functional. In the future this can be mitigated by using canary deployments and playing through failure scenarios in test environment and adding relevant alerts and recovery mechanisms.

As all the schema is served through API gateway then it must be considered in every schema change in downstream services that provide fragments in this schema. With performance in mind the gateway is configured in a way that schema is loaded only in boot time. Every time a downstream application release includes schema change it must trigger reloading of schema which makes release process longer and slower. To simplify this process Apollo platform provides registry component, which remains to be evaluated for this system.

6 Conclusion

The aim of this thesis was to create a maintainable and more secure API service for mobile application. Existing application and services were built with prototyping in mind and were intended to be an experiment instead of a scalable and reliable product that can compete on the market.

This thesis illustrated approaches that can be reused in context of building reliable services with emerging technology GraphQL. This technology allowed the existing service architecture to become consistently documented and reusable in different UI components without performance impact. Using GraphQL allowed mobile application to reduce data usage by fetching only specific fields that are needed for a UI element.

A new mobile application that was written as a client of GraphQL gateway is available on Apple and Google app stores and is gaining popularity. Additional features have been added and demonstrated that reuse is possible. The backend system implemented in this thesis has remained responsive and simplified scaling and maintenance. Additional microservices have been written using same libraries that were integrated as a part of this thesis.

7 References

[1] The Linux Foundation, "OpenAPI Specification Version 3.1.0," 15 February 2021. [Online]. Available: https://spec.openapis.org/oas/v3.1.0.

[2] IBM, “IBM Cloud Enterprise Sandbox,” IBM, May 2020. [Online]. Available: https://ibm-gsi-ecosystem.github.io/ibm-gsi-cloudnative-journey/developer-advanced-1/swagger-and-openapi/. [Accessed 3 May 2021].

[3] S. R. Leonard Richardson, RESTful Web Services, United States of America: O'reilly media, 2007.

[4] Netflix Technology, “Netflix Tech Blog,” Netflix Technology Blog, 11 December 2020. [Online]. Available: https://netflixtechblog.com/how-netflix-scales-its-api-with-graphql-federation-part-2-bbe71aaec44a. [Accessed 03 May 2021].

[5] A. Neary, “Airbnb Engineering,” medium, 5 December 2018. [Online]. Available: https://medium.com/airbnb-engineering/how-airbnb-is-moving-10x-faster-at-scale-with-graphql-and-apollo-aa4ec92d69e2. [Accessed 03 May 2021].

[6] T. M. M. T. V. Gleison Brito, “Migrating to GraphQL: A Practical Assessment,” in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, Hangzhou, China, 2019.

[7] R. Fadhil, “How to Use GraphQL DataLoader,” 29 05 2020. [Online]. Available: https://rahmanfadhil.com/graphql-dataloader/. [Accessed 11 05 2021].

[8] Apollo Graph Inc., “Apollo docs,” Apollo Graph Inc., [Online]. Available: https://www.apollographql.com/docs/. [Accessed 3 May 2021].

[9] Facebook Inc., “Graphql specification,” Graphql foundation, June 2018. [Online]. Available: https://spec.graphql.org/June2018/. [Accessed 3 May 2021].

[10] 99designg, “gqlgen documentation,” 21 09 2020. [Online]. Available: https://pkg.go.dev/github.com/99designs/gqlgen. [Accessed 11 05 2021].

[11] Apollo Federation, “Apollo Docs,” 09 10 2020. [Online]. Available: https://www.apollographql.com/docs/federation/api/apollo-federation/. [Accessed 11 05 2020].

[12] M. Subramanian, Jaeger - Distributed Tracing for Cloud Native Applications, Packt Publishing, 2020.

Appendix 1 – Non-exclusive licence for reproduction and publication of a graduation thesis[1]

I Taavet Tamm

1. Grant Tallinn University of Technology free licence (non-exclusive licence) for my thesis “GraphQL API implementation in a Distributed System,” supervised by Aleksei Talisainen
   1.1. to be reproduced for the purposes of preservation and electronic publication of the graduation thesis, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright;
   1.2. to be published via the web of Tallinn University of Technology, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright.
2. I am aware that the author also retains the rights specified in clause 1 of the non-exclusive licence.
3. I confirm that granting the non-exclusive licence does not infringe other persons' intellectual property rights, the rights arising from the Personal Data Protection Act or rights arising from other legislation.

15.05.2021

1 The non-exclusive licence is not valid during the validity of access restriction indicated in the student's application for restriction on access to the graduation thesis that has been signed by the school's dean, except in case of the university's right to reproduce the thesis for preservation purposes only. If a graduation thesis is based on the joint creative activity of two or more persons and the co-author(s) has/have not granted, by the set deadline, the student defending his/her graduation thesis consent to reproduce and publish the graduation thesis in compliance with clauses 1.1 and 1.2 of the non-exclusive licence, the non-exclusive license shall not be valid for the period.