Utveckling av en modern fullstack applikation med GraphQL

Marco Törnqvist

Examensarbete

Informationsteknik

2022

EXAMENSARBETE	
Arcada	
Utbildningsprogram:	Informationsteknik
Identifikationsnummer:	8401
Författare:	Marco Törnqvist
Arbetets namn:	Utveckling av en modern fullstack applikation med GraphQL
Handledare (Arcada):	Fredrik Welander
Uppdragsgivare:	-

Sammandrag:
Vid sidan av REST har GraphQL dykt upp och blivit en av de mest använda metoderna för att kommunicera data mellan en server och en webbapplikation eller mobilapplikation.

GraphQL löser ett flertal olika problem som uppstår med REST-API:er, där det främsta problemet som uppstår är over-fetching. Over-fetching, i all sin enkelhet, uppstår då en onödig mängd data skickas från en server till en klient. GraphQL löser problemet genom att genom att ge mer kontroll åt klienten över den data som hämtas från en server. GraphQL kräver mer kod på serversidan, men å andra sidan minskar det belastningen i samband med att data hämtas/skickas från servern.

Syftet med arbetet är att beskriva hur GraphQL och REST fungerar från en teoretisk synpunkt och hur de skiljer sig från varandra. Arbetet kommer även att visa hur man själv kan implementera GraphQL med Apollo och även hur man kan implementera GraphQL på klienten med hjälp av NextJS och Apollo client.

Arbetet kommer att visa och beskriva de olika operationer som kan göras inom GraphQL, som till exempel queries, mutations och subscriptions.

Nyckelord:	NextJS, TypeScript, GraphQL, Apollo, Prisma, PostgreSQL, REST, JEST, Resolver, Schema, Query, Mutation, Subscription.
Sidantal:	34
Språk:	
Datum för godkännande:	

DEGREE THESIS	
Arcadia	
Degree Programme:	Information analysis
Identification number:	8401
Author:	Marco Törnqvist
Title:	Development of a modern full-stack application with GraphQL
Supervisor (Arcada):	Fredrik Welander
Commissioned by:	-

Abstract:
Alongside REST, GraphQL has emerged and become one of the most widely used methods for communicating data between a server and a web application or mobile application.

The point of GraphQL is to solve several problems that arise with REST API, where one of the main ones is over-fetching. Over-fetching, in all its simplicity, occurs when an unnecessary amount of data is fetched from a server to a client. GraphQL solves the problem by giving more control to the client over the data that is fetched from a server. GraphQL requires more code on the server-side, but on the other hand, it reduces the load in relation to data being fetched/sent from the server.

The purpose of this work is to describe how GraphQL and REST work from a theoretical point of view and how they differ from each other. This work will also show how the reader can implement GraphQL with Apollo, and also how GraphQL can be implemented on the client with NextJS and Apollo client.

This work will also show and describe the various operations that can be done within GraphQL, such as queries, mutations and subscriptions.

Keywords:	NextJS, TypeScript, GraphQL, Apollo, Prisma, PostgreSQL, REST, JEST, Resolver, Schema, Query, Mutation, Subscription.
Number of pages:	34
Language:	Swedish
Datum för godkännande:	

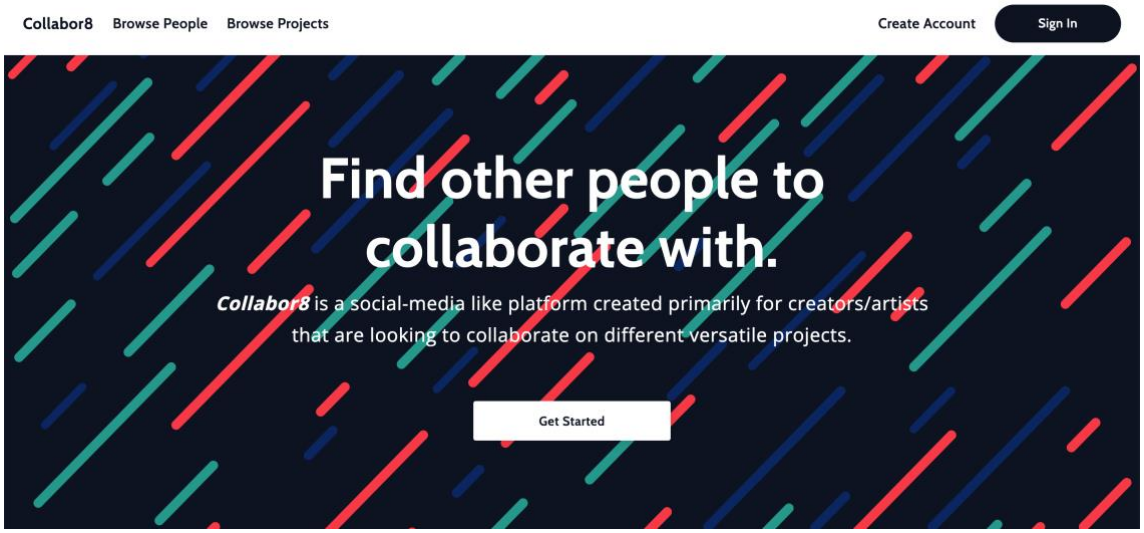
INNEHÅLL

FIGURER

TABELLER

1 INLEDNING

Collabor8 är en plattform för sociala medier riktat till alla sorters kreativa personer. Syftet är att hjälpa dessa kreativa personer hitta varandra: oavsett om man är en regissör som söker efter en skådespelare till sin nästa film, eller om man är en sångerska som söker sitt nästa band, så hittas dessa på hemsidan. Inom webbapplikationen kan man skapa olika *projekt* var man kan bjuda in individer och diskutera hur man kommer förverkliga *projektet*.



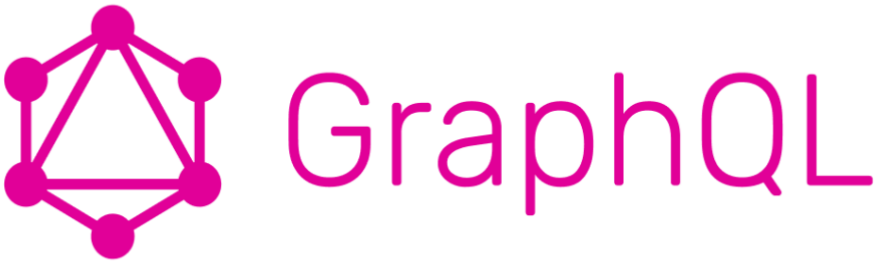
Figur 1. Collabor8 hemsidans inledning. (Collabor8. 2022)

Den praktiska delen av examensarbetet kommer långt basera sig på webbapplikationen *Collabor8*: en applikation jag själv utvecklat mellan sommaren 2021 och våren 2022 som ett portfolio-projekt. Applikationen omfattar funktionalitet som autentisering, skapandet av användare och profil, skapandet av projekt, chatt-system med mera. Chatt-systemet använder sig av olika GraphQL -koncept som *queries*, *mutations* och *subscriptions*. Själva källkoden webbapplikationen är offentlig och hittas under URL-adressen: https://github.com/marcotornqvist/collabor8. En live demoversion hittas under URL-adressen: https://collabor8-frontend.vercel.app/.

1.1 Bakgrund

Under de senaste åren har GraphQL blivit en av de vanligaste metoderna för en server att skicka data mellan en webbapplikation eller mobilapplikation, vid sidan om REST API:s. GraphQL skapades år 2012 av Facebook och har varit *open source* sedan år 2015.

Före GraphQL hade Facebook stora prestationsproblem och deras applikationer låg nere på en daglig basis, som ett resultat av att deras applikationer blev mer komplexa och belastningen på servrarna ökade. Facebook märkte att de existerande lösningarna inte var tillräckliga, eftersom man behövde göra så många olika förfrågningar till så många olika resurser för att i slutändan kunna komma åt en viss sorts data. Facebook behövde ett nytt alternativ för att kunna skicka data mellan servern och applikationen på ett utvecklarvänligt sätt, men samtidigt möta höga krav på prestanda. Målet med GraphQL är att kunna komma åt alla data via en endaste resurs, genom att kunna specificera de exakta data man vill att skall bli skickad från GraphQL API:er till klienten (Facebook Engineering, 2015). Vanligtvis kommer man åt denna resurs via en URL/graphql. Företag av olika storlekar, som blandat Starbucks, Github, Pinterest, och Shopify, har implementerat GraphQL inom sina applikationer (Graphql, 2022).



Figur 2. GraphQL logo. (GraphQL. 2022)

1.2 Syfte & Mål

Syftet med arbetet är att beskriva skillnaden mellan GraphQL och REST-gränssnitt, samt kartlägga för- och nackdelarna för både GraphQL och REST-gränssnitt. Målet med arbetet är visa skillnaderna på ett tydligt och intuitivt sätt, och är riktat till utvecklare med tidigare kunskap inom REST-gränssnitt. Arbetet kommer att täcka både teoretiska aspekter av skillnaderna, men även praktisk demonstrera hur GraphQL kan implementeras både på server- och klientsidan.

1.3 Metoder

Utvecklingsprocessen av webbapplikationen Collabor8 kommer att demonstreras både genom visuella och teoretiska exempel. Examensarbetet kommer kortfattat gå igenom de viktigaste programmeringsspråken, samt biblioteken som använts för att bygga applikationen. Detta för att ge en helhetsbild över hur de olika teknikerna och GraphQL hänger ihop. Examensarbetet kommer även gå djupare in på hur diverse funktionalitet inom webbapplikation Collabor8 fungerar och implementerats. Funktionaliteten omfattar bland annat autentiseringen av användare, skapandet av projekt och chatt-funktionaliteten.

1.4 Avgränsningar

Arbetet kommer inte att behandla hur man konfigurerar en GraphQL & Apollo server. Arbetet kommer inte heller behandla konfigureringen av GraphQL & Apollo Client på klient-sidan. Examensarbetet kommer inte att fokusera på funktionaliteten av SASS eller HTML.

2 GRAPHQL VS REST

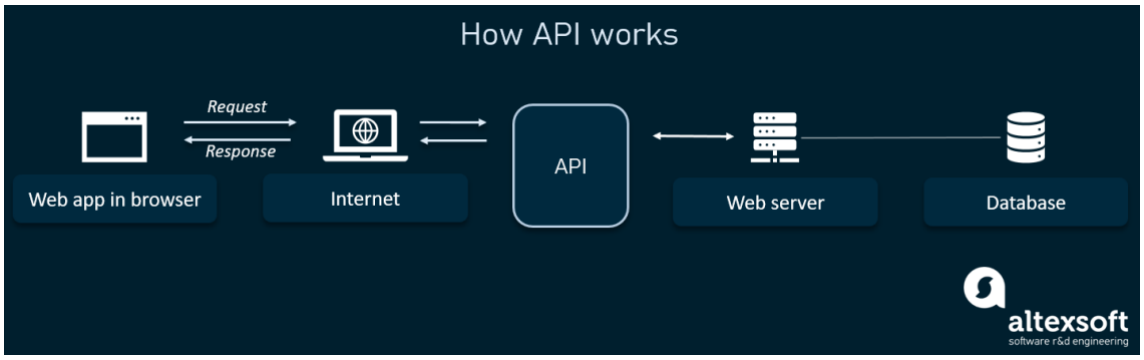
Detta kapitel kommer omfatta API:er, för- och nackdelar med GraphQL och REST-API:er, och gå igenom hur de skiljer sig ifrån varandra.

2.1 Vad är ett API?

API som står för *application programming interface,* eller programmeringsgränssnitt på svenska, är ett grundläggande ämne för att förstå sig på GraphQL och REST-API:er. API:er gör det möjligt för applikationer, servrar och enheter att skicka data mellan varandra genom olika sorters instruktioner. Det finns flera olika typer av API:er, som till exempel Soap, WebSocket, REST och GraphQL (postman.com, 2022).

Ett exempel av användandet av ett API är skapandet av ett blogginlägg på en hemsida. Man börjar med att fylla i ett formulär med textinnehåll och trycker ”skapa inlägg”, detta skapar en API begäran som skickar textinnehållet från klienten till en webbserver. Ifall

webbservern accepterat innehållet som API har specificerat, skapas ett inlägg med en http statuskod av 201. Detta tyder på att skapandet av ett nytt inlägg lyckades.



Figur 3. Vad är en API. (altexsoft, 2022)

WebSocket API:er möjliggör ett konstant dataflöde mellan en server och en applikation, utan att applikationen själv behöver göra en ny begäran av data och tvärtom (mozilla.org, 2022). En REST API däremot kräver att en applikation anropar servern varje gång man behöver komma åt ny data. Ett praktiskt exempel på var WebSocket API:s används är till exempel i chatt-system, var nya meddelande hämtas varje gång ett nytt meddelande skapas, utan att applikationen skilt behöver anropa en server varje gång. GraphQL använder sig av WebSockets för att kunna hämta och skicka data i realtid mellan klienten och servern (apollographql.com, 2022).

2.2 REST API

REpresentational State Transfer eller REST är en arkitektonisk stil för skapandet av applikationsprogrammeringsgränssnitt. REST-API:er har fungerat som den populäraste stilen för att skapa API:er på webben sedan år 2005. När REST API utvecklades av Roy Fielding år 2000, var tankesättet att använda sig av så många tidigare funktioner inom HTTP som möjligt: ”Det finns många smarta funktioner i HTTP som gör att webben fungerar så bra som den gör. Tanken med REST är att utnyttja så mycket av detta som möjligt i stället för att hitta på nya sätt att hantera överföringen på” (newseed.se, 2022).

2.2.1 Metoder

REST använder sig av många olika HTTP-metoder för att olika tjänster skall kunna kommunicera med varandra. De vanligaste HTTP-metoderna är GET, POST, PUT och

DELETE. Med dessa metoder så kan man skapa CRUD-applikationer (Create, Read, Update och Delete).

Låt oss ta Collabor8 webbapplikationen som ett exempel: man kan skapa ett projekt med en POST begäran, uppdatera projektet med en PUT begäran, olika användare kan hämta projekt med en GET begäran och själva projektet kan bli raderat med en DELETE begäran. För att till exempel kunna radera ett inlägg, så behöver användaren vara inloggad och autentiserad genom att API godkänner användarens autentiseringstoken som skickas i samma DELETE-begäran (Budibase, 2021).

Method	Description
GET	Retrieve information about the REST API resource
POST	Create a REST API resource
PUT	Update a REST API resource
DELETE	Delete a REST API resource or related component

Figur 4. REST API Methods. (oracle docs, 2022)

2.2.2 Resurser

REST-API:er består oftast av många olika resurser som klienten kan skicka eller begära data ifrån. En resurs kan till exempel hämta data ifrån en databas med en SQL-query, som sedan i sin tur skickar den hämtade datan till klienten. Ett praktiskt exempel är ett formulär på klienten som endast tar en input av meddelanden. När användaren tryckt på skicka, så skickas detta meddelande via en POST begäran till den specificerade resursen där relevant logik specificerats av en utvecklare. Logiken kan till exempel validera att meddelandet innehåller tillräckligt många tecken. Ifall meddelandet uppfyller kraven, sparas meddelandet i en databas, annars returnerar resursen ett felmeddelande. Meddelandet kan sedan hämtas av den mottagande klienten med hjälp av en GET-begäran.

Figur 5. Data Fetching with REST vs GraphQL. (howtographql.com. 2022)

2.2.3 Fördelar

REST API:s är relativt simpla. Det krävs inte mycket kod för att komma i gång med ett REST API, till exempel med ett ramverk som *express.js*. Inlärningskurvan är också mycket simplare än de flesta andra alternativen p.g.a. det ingår betydligt mindre konfiguration.

REST API:s är *stateless*. Detta gör modifieringen av en REST API -resurs mindre komplicerat, eftersom man inte behöver oroa sig över att en resurs skulle gå sönder på grund av en tidigare version. Varje begäran är isolerad och påverkas inte av en tidigare begäran.

REST-API:er stöder cachelagring utan extra konfigurering på klienten, vilket resulterar i snabbare hastigheter hos klienten. Cachelagring möjliggör en tillfällig lagringsplats för data och filer på en dators RAM. Med hjälp av lagringen kan en klient återanvända sig av

tidigare hämtad data och filer, utan att behöva begära efter samma data samt filer från en server på nytt (Ashita Gopalakrishnan, 2021).

2.2.4 Nackdelar

Ett av de vanligaste problemen med REST-API:er är over- och under-fetching. Ofta när man begär data från en resurs med en REST API, så har man inte kontroll över vilka fält som returneras inom ett objekt hos klienten. Detta leder till att onödiga mängder av data skickas till klienten, vilket i sin tur resulterar i försämrad prestanda. Motsatsen till over-fetching är under-fetching. Under-fetching uppstår då klienten är tvungen att begära data från flera olika resurser för att komma åt all nödvändig data för dess funktion: ”over-fetching is fetching too much data, meaning there is data in the response you don't use. Under-fetching is not having enough data with a call to an endpoint, forcing you to call a second endpoint” (Yachaka. 2017).

REST-API:er erbjuder inte mycket flexibilitet. Oftast när en klient får mer funktionalitet, blir mer komplex eller modifieras, så uppstår det nya krav på data. Som ett resultat behöver oftast REST API:s även modifieras för att kunna möta kraven från klienten. Detta ökar beroende-förhållanden och tar mer tid, alltså försämrar produktiviteten eftersom att REST API:s måste modifieras och klienten måste vänta på modifieringarna för att kunna påbörja sin implementering (Ashita Gopalakrishnan, 2021).

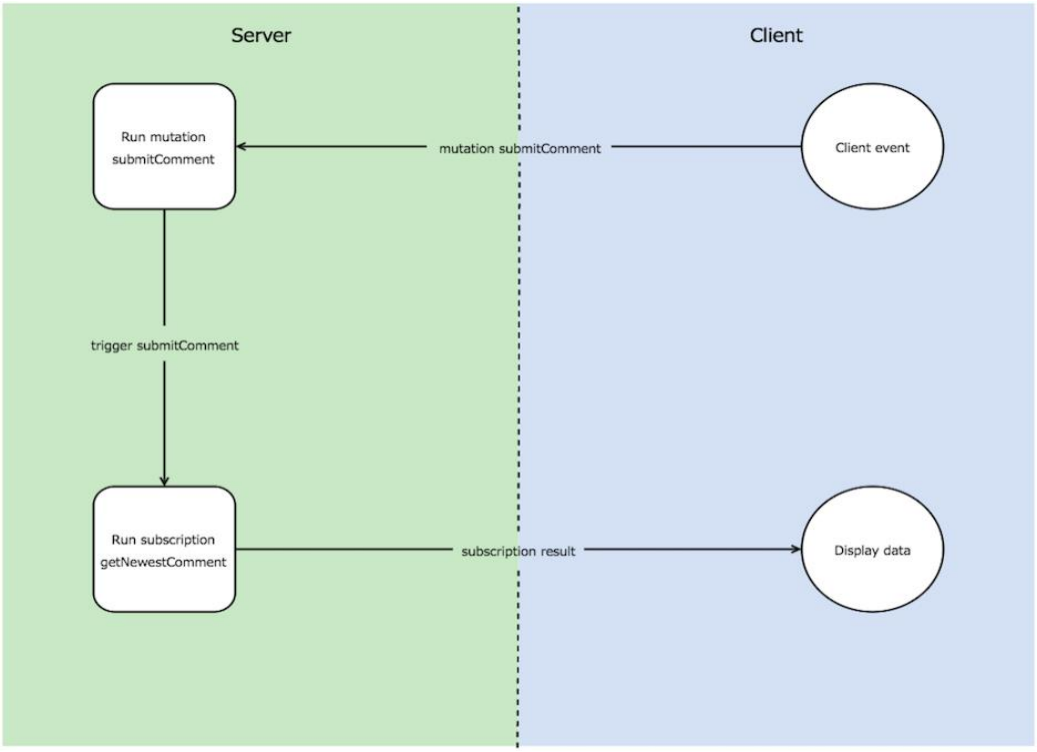
2.3 GraphQL

GraphQL ger mer flexibilitet åt klienten genom att ge mer kontroll över data som skickas mellan en server och en applikation. GraphQL möjliggör att servern endast behöver anropas en gång för att komma åt alla nödvändiga data. Detta betyder att man kan minska användningen av bandbredd och data som skickas mellan servern och klienten, vilket i sin tur resulterad i applikationer med högre prestanda: ”GraphQL is a query language for API:s and a runtime for fulfilling those queries with your existing data. GraphQL provides a complete and understandable description of the data in your API, gives clients the power to ask for exactly what they need and nothing more, makes it easier to evolve APIs over time, and enables powerful developer tools” (graphql, 2022).

2.3.1 Operations

På samma sätt som i en REST API, så finns det olika sorters metoder för att hantera data inom GraphQL. Dessa metoder, eller *operations*, som man kallar det inom GraphQL är *queries*, *mutations* och *subscriptions:*

- **Queries** är till för att hämta data från en server till en klient. Queries fungerar på ett likadant sätt som en GET -begäran ,med tanken att man kan hämta data från servern till klienten. Fördelen med queries till GET -begäran är att man på klienten kan specificera exakt vilken data och vilka fält som man vill att skall returneras från servern till klienten.
- **Mutations** är till för att skicka data från klienten till servern. Mutations kan ta emot olika argument och fungerar på samma sätt som en DELETE, PUT eller PATCH begäran: ”If queries are the GraphQL equivalent to GET calls in REST, then mutations represent the state-changing methods in REST (like DELETE, PUT, PATCH, etc.)” (Stemmler Khalil, 2021).
- **Subscriptions** är tillför att skicka realtidsdata från servern till klienten och tvärtom med hjälp av Websockets. Med en vanlig *query* så begär klienten data från en server endast då ett *event* har skett, medan med subscriptions så öppnar klienten en långvarig anslutning mellan en server och en klient. Varje gång som det sker ett nytt *event* på servern, som till exempel att ett nytt meddelande har skapats, så lyssnar klienten efter det och hämtar det nya meddelandet till klienten. Subscriptions är gjorda för situationer där applikationer kräver data i realtid, som till exempel i spel eller chatt-system.



Figur 6. Data flödet för en subscription. (Amanda Liu. 2016)

2.3.2 Resolvers

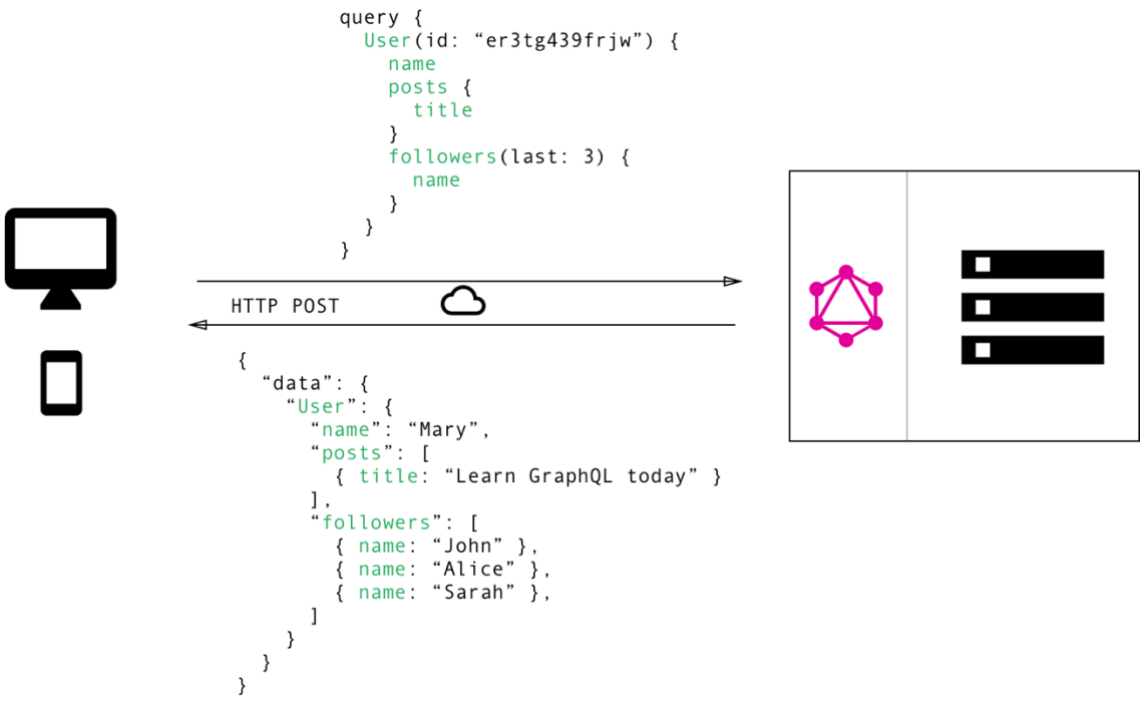
En *resolver* är en samling av diverse resolver-funktioner. Dessa resolver-funktioner kan hantera olika operationer som *query*, *mutation* eller *subscriptions*. En resolver-funktion kan på samma sätt som en vanlig funktion ta emot olika argument för att kunna till exempel hantera *pagination* eller filtrering. En *resolver* kan även använda sig av diverse *middleware*-funktioner. Middleware-funktioner är återanvändbara funktioner som körs innan själva funktionen blir körd för att kunna kolla ifall klienten eller en server har tillstånd att köra en annan funktion. Exempelvis kan man kolla ifall en användare är inloggad för att sedan kunna hämta en viss sorts data som är anknuten till den inloggade användaren.

2.3.3 Single Source of Truth

För att komma åt all data inom en GraphQL-applikation, är man tvungen att definiera ett GraphQL-schema. Schemat innehåller alla de olika *resolvers* som blivit skapade och via schemat kommer klienten med endast ett anrop åt alla definierade resolver-funktioner, som i sin tur returnerar all data som klienten behöver.

Hos klienten kan man specificera en Query-string där man specificerar vilka resolver-funktioner man vill anropa. Inom Query-stringen kan en utvecklar på klienten specificera kapslade (*nested*) fält som returnerar den exakta datan som behövs för den specifika situationen.

I figuren nedan begär klienten en specifik användare, genom att definiera en resolver-funktion som heter *User*, med argumentet *id*, som refererar till en specifik användare med *id* ”er3tg439frjw”. *User* resolver-funktionen returnerar också alla diverse inlägg med endast dess rubriker. Via funktionen kommer man även åt alla de olika följare som användaren har och i detta exempel namnen av användarens 3 äldsta följare.



Figur 7. Data Fetching with REST vs GraphQL. (howtographql.com. 2022)

2.3.4 Fördelar

Schemat samlar alla resolver-funktioner på en och samma centrala plats, vilket gör det lätt för klienten att komma åt alla resolver-funktioner och deras data med endast ett anrop från en och samma plats. Jämför man detta med ett REST-gränssnitt, så är klienten tvungen att anropa servern ett flertal gånger för att komma åt all behövlig data.

GraphQL löser problemet med over- och under-fetching. En utvecklare kan på klienten definiera de fält som ett användargränssnitt behöver, och hämta exakt den data man behöver. Detta resulterar i minskad användning av bandbredd, pga. att man inte behöver hämta data som inte behövs eller används, eller göra flera nätverksförfrågningar till olika resurser.

GraphQL-scheman använder sig av ett schemadefinitionsspråk som heter SDL eller *schema definition language* för att minska möjligheterna för buggar på grund av typfel. SDL tvingar utvecklare på serversidan att definiera varje typ av fält som en resolver-funktion är menad att returnera. Detta resulterar i betydligt mindre mängder fel inom utvecklingen av applikationen både på serversidan och klienten. Minskningen av fel beror på att både klienten och API:er returnerar fel ifall någondera försöker returnera ett fält som inte existerar inom den specifika resolver-funktionen (Ashita Gopalakrishnan, 2021).

2.3.5 Nackdelar

Cachelagring är svårare att implementera inom GraphQL-applikationer än med resursbaserade API:er. När en klient hämtar data från en resurs, så blir resursens adress samt data sparad inom ett cachelager. Ifall samma resurs blir anropad på nytt, så kan klienten kolla ifall data redan existerar inom cachen. Ifall data existerar så behövs det i många fall inte ett nytt anrop till API, vilket resulterar i bättre prestanda och minskad användning av bandbredd. Problemet som uppstår med GraphQL, är att det endast finns en resurs från vilken man hämtar alla data, alltså leder det till att klienten inte kan skilja på vilka data som har blivit hämtade tidigare. För att hjälpa till med begränsningen så har det utvecklats bibliotek till GraphQL som stöder implementeringen av cachelagring på klienten, som till exempel Apollo Client eller URQL. Biblioteken kräver dock extra konfigurering och leder också till större ”bundle sizes”.

En annan nackdel inom GraphQL-applikationer är att det är svårare att implementera felhantering jämfört med REST-API:er. Varje gång ett GraphQL API anropas så returnerar API:t en HTTP-kod av 200, även om det skulle innehålla ett fel. Detta kan manuellt korrigeras på serversidan genom att ge specifika HTTP-koder på basen av definierad logik, men i jämförelse med REST-API:er var rätt HTTP-koder returneras

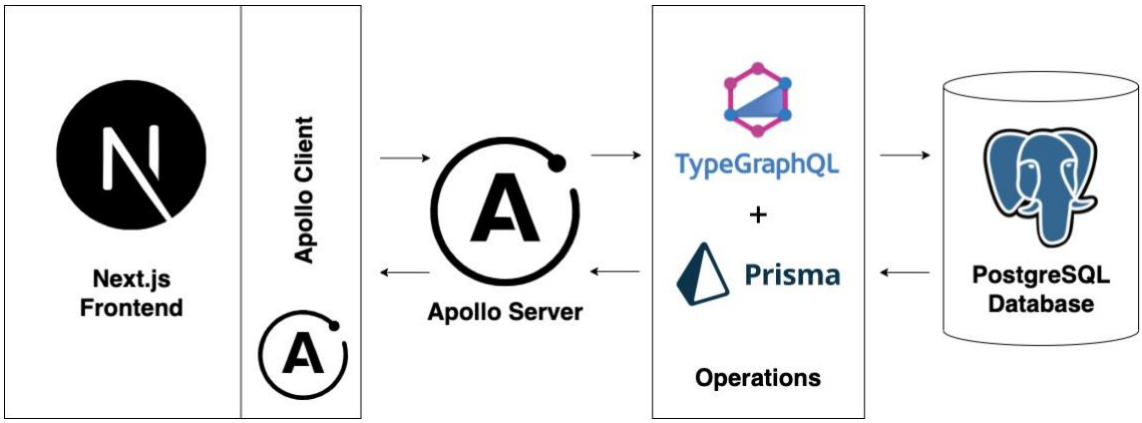
automatiskt, så är handlar det om mycket mer kod och arbete (Gopalakrishnan Ashita, 2021).

3 UTVECKLANDET AV EN GRAPHQL-APPLIKATION

På basen av det tidigare kapitlet borde läsaren ha fått en uppfattning om vad GrahpQL är och varför GraphQL används inom flera stora applikationer. Som nästa steg kommer arbetet fortsätta utforska diverse funktionalitet inom GraphQL och, med hjälp av Collabor8 -webbapplikationen, ge praktiska exempel på hur programmeringsspråken och biblioteken inom GraphQL använts.

3.1 Teknikstacken

Detta delkapitel kommer kortfattat behandla de viktigaste programmeringsspråken samt biblioteken som blivit använda inom Collabor8 -projektet. Detta för att ge läsaren en översikt över hur de olika teknikerna samspelar med varandra och med GraphQL.



Figur 8. Dataflödet inom Collabor8 applikationen.

3.1.1 NodeJS & NPM

Node.js tillåter utvecklare programmera JavaScript på serversidan. Node.js använder sig av ”V8 JavaScript Runtime Environment” som konverterar JavaScript-kod till maskinkod: ”Both your browser JavaScript and Node.js run on the V8 JavaScript runtime engine. This engine takes your JavaScript code and converts it into a faster machine code. Machine code is low-level code which the computer can run without needing to first interpret it” (Patel Priyesh, 2018).

NPM eller ”Node Package Manager” är den största samlingen av JavaScript-baserade bibliotek med öppen källkod, som omfattar bland annat Next.js, GraphQL och Apollo. NPM gör det enkelt för utvecklare att installera och använda sig av dessa bibliotek.

3.1.2 TypeScript

TypeScript hjälper utvecklare att skriva mera förutsägbar JavaScript -kod. TypeScript hjälper utvecklare att fånga upp fel i utvecklingsprocessen genom att använda sig av TypeScript-transpilern som även konverterar koden till så kallad Vanilla JavaScript. Med Vanilla JavaScript, som alltså är JavaScript utan något ramverk eller typsystem, så måste själva filen som innehåller JavaScript koden köras ifall man vill veta ifall koden är giltig eller inte. Med TypeScript så kompileras däremot koden, och en lista av fel uppstår ifall transpilern hittat olika syntaxfel. Detta hjälper utvecklare att hitta fel innan skriptet körs. TypeScript kräver att utvecklare definierar vilken typ ett värde har: ifall de är till exempel *strings*, *numbers* eller *boolean*. Detta resulterar i mer skalbara applikationer och hjälper utvecklare fånga fel innan applikationen blivit distribuerad för allmänheten. (typescript.com, 2022)

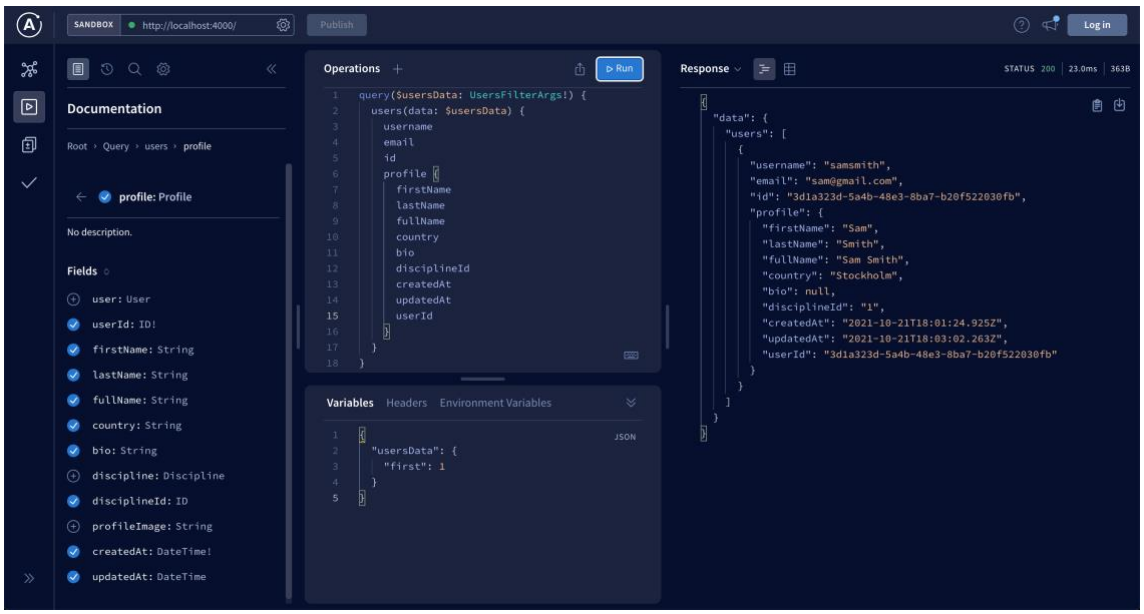
3.1.3 TypeGraphQL

TypeGraphQL hjälper till att minska mängden kod i vår GraphQL + Typescript-applikation, genom att definiera återanvändbara klasser: ”The biggest problem is the redundancy in our codebase, which makes difficult to keep this things in sync. To add a new field to our entity, we have to jump through all the files modify an entity class, then modify part of the schema, as well as the interface. The same goes with inputs or arguments. It’s easy to forget to update one piece or make a mistake with a single type” (Lytek Michał, 2018).

Med TypeGraphQL-klasser så är huvudtanken att vi en endast har en *source of truth*. Detta innebär att vi endast behöver definiera våra klasser, som vi sedan kan importera in i diverse filer, och använda inom våra scheman, resolver-funktioner och olika inputs. Ifall det sker ändringar inom en klass, så reflekteras detta i hela applikationen där klasserna används. TypeGraphQL -resolvers använder sig av en lite annorlunda syntax om man jämför med vanlig GraphQL -kod, men funktionaliteten förblir densamma.

3.1.4 Apollo Server

Apollo Server gör det enkelt för utvecklare att skapa en GraphQL-server. GraphQL tar hand om själva funktionaliteten, medan Apollo Server tar hand om servern och dess konfiguration. Till detta tillhör byggandet av schemat med dess resolver-funktioner, hanteringen av CORS (*Cross-Origin Resource Sharing*) och själva körandet av servern. Apollo server inkluderar även ett verktyg som heter Apollo studio, där utvecklare kan inspektera och testa sina GraphQL-operationer.



Figur 9. Användningen av Apollo studio.

3.1.5 PostgreSQL

PostgreSQL är en av de mest använda relationsdatabaserna vid sidan av Microsoft SQL Server, Oracle och MySQL: (Chand Mahesh, 2022) ”PostgreSQL is an advanced, enterprise class open source relational database that supports both SQL (relational) and JSON (non-relational) querying. It is a highly stable database management system, backed by more than 20 years of community development which has contributed to its high levels of resilience, integrity, and correctness”. (aws.amazon.com, 2022)

3.1.6 Prisma

Prisma ORM är ett bibliotek för att hämta och manipulera data mellan en databas och en server. ORM eller Object-Relational Mapping är ett sätt för utvecklare att hämta samt manipulera data på ett objektorienterat sätt. Problemet som oftast uppstår med alternativa metoder som SQL är dess komplexitet och problematik med att felsöka. För att komma i

gång med Prisma så måste man skapa en fil som oftast kallas för schema.prisma. Inom filen kopplas databasen med Prisma client (sköter all begäran till databasen) och inom samma fil så definieras strukturen på databasen (olika tabeller och dess fält). Prisma ORM fungerar med många olika SQL och NoSQL-baserade databaser som PostgreSQL, MySQL och MongoDB. Prisma var ett utmärkt alternativ till Collabor8 projektet eftersom kodbasen strävade efter så mycket typsäkerhet som möjligt. Exemplet nedan visualiserar skillnaden mellan en SQL och Prisma query.

```
// SQL query för att hämta en användare
SELECT * FROM users WHERE id="100";

// Prisma query för att hämta en användare
const user = await prisma.user.findUnique({
  where: {
    id: "100",
  },
});
```

Figur 10. Skillnaden mellan en SQL och Prisma query.

3.1.7 NextJS

Next.js är ett så kallat meta-ramverk, alltså är det byggt ovanpå React.js ramverket. React.js hjälper utvecklare bygga användargränssnitt genom att dela upp applikationen i diverse återanvändbara komponenter och genom att fungera som en single-pageapplikation: ”Single page application is a web application or website that interacts with the user by dynamically rewriting the current page, rather than loading entire new pages from the server” (Lawson Katie, 2022). En viktig funktionalitet inom React.js är *state* och *props*, vilka möjliggör delandet av data mellan olika komponenter. Huvudsyftet med React är att utveckla mera skalbara applikationer med tydligare struktur. Next.js utökar funktionaliteten ovanpå React.js, utan ytterligare konfiguration: ”Next.js gives you the best developer experience with all the features you need for production: hybrid static & server rendering, TypeScript support, smart bundling, route pre-fetching, and more. No config needed” (nextjs, 2022).

3.1.8 Apollo Client

Apollo Client hjälper utvecklare att hantera data samt olika tillstånd på klienten, genom att hämta eller uppdatera data från en Apollo server. Apollo Client gör detta via olika GraphQL-operationer som *query, mutation och subscriptions*. Apollo Client stöder också cachelagring utan extra konfiguration vilket GraphQL på egen hand inte gör.

Apollo Client fungerar mycket bra med React-baserade applikationer, genom att använda sig av olika React-metoder såsom hooks (används ofta för att lagra diverse tillstånd inom React-applikationer). Allt detta möjliggör en centraliserad plats för tillståndslagring, vars data sedan kan användas tvärs över hela React-applikationen, utan att behöva använda ett ”state management” bibliotek som Redux eller Context API. Detta resulterar i ökad prestanda, eftersom klienten inte behöver anropa API, varje gång som applikationen behöver data som redan har hämtats (apollographql.com, 2022).

Exemplet nedan visar hur Collabor8 applikationen har implementerat Apollo Client inom en Next.js applikation:

Figur 11. Användningen av Apollo Client inom en Next.js komponent.

3.2 GraphQL API:er

Detta delkapitel kommer att hantera hur man implementerar teknologierna som TypeGraphQL, PostgreSQL och Prisma på serversidan. Delkapitlet kommer även att gå igenom ett exempelscenario som använts inom Collabor8 -applikationen.

Målsättningen med detta exempel är att kunna visa läsaren ett praktiskt exempel där ovannämnda teknologier används tillsammans i samband med att användare ska både kunna skapa ett meddelande och sedan hämta eller lyssna efter the inkommande meddelandet. Genom exemplet kommer läsaren ges en uppfattning om hur de olika GraphQL-operationerna, som *queries*, *mutations* och *subscriptions*, används i praktiken på serversidan.

3.2.1 ChatResolver

I ett tidigare skede beskrevs det vad en resolver och vad de olika resolver-funktioner är till för. Varje resolver innehåller en samling av diverse resolver-funktioner som alla hanterar någon sorts funktionalitet. Arbetet kommer nu att fördjupa sig mera i de olika resolver-funktionerna, samt visa diverse praktiska exempel på hur dessa använts i Collabor8-applikationen. I exemplet har vi en resolver som är döpt till ChatResolver, som inkluderar bland annat dessa resolver-funktioner: contactAddMessage, contactMessages och newMessage

Namn	Operation	Beskrivning
contactAddMessage	Mutation	Skapar ett nytt meddelande.
contactMessages	Query	Hämtar alla meddelande.
newMessage	Subscription	Lyssnar efter nya meddelande.

Tabell 1.ChatResolvers olika resolver funktioner

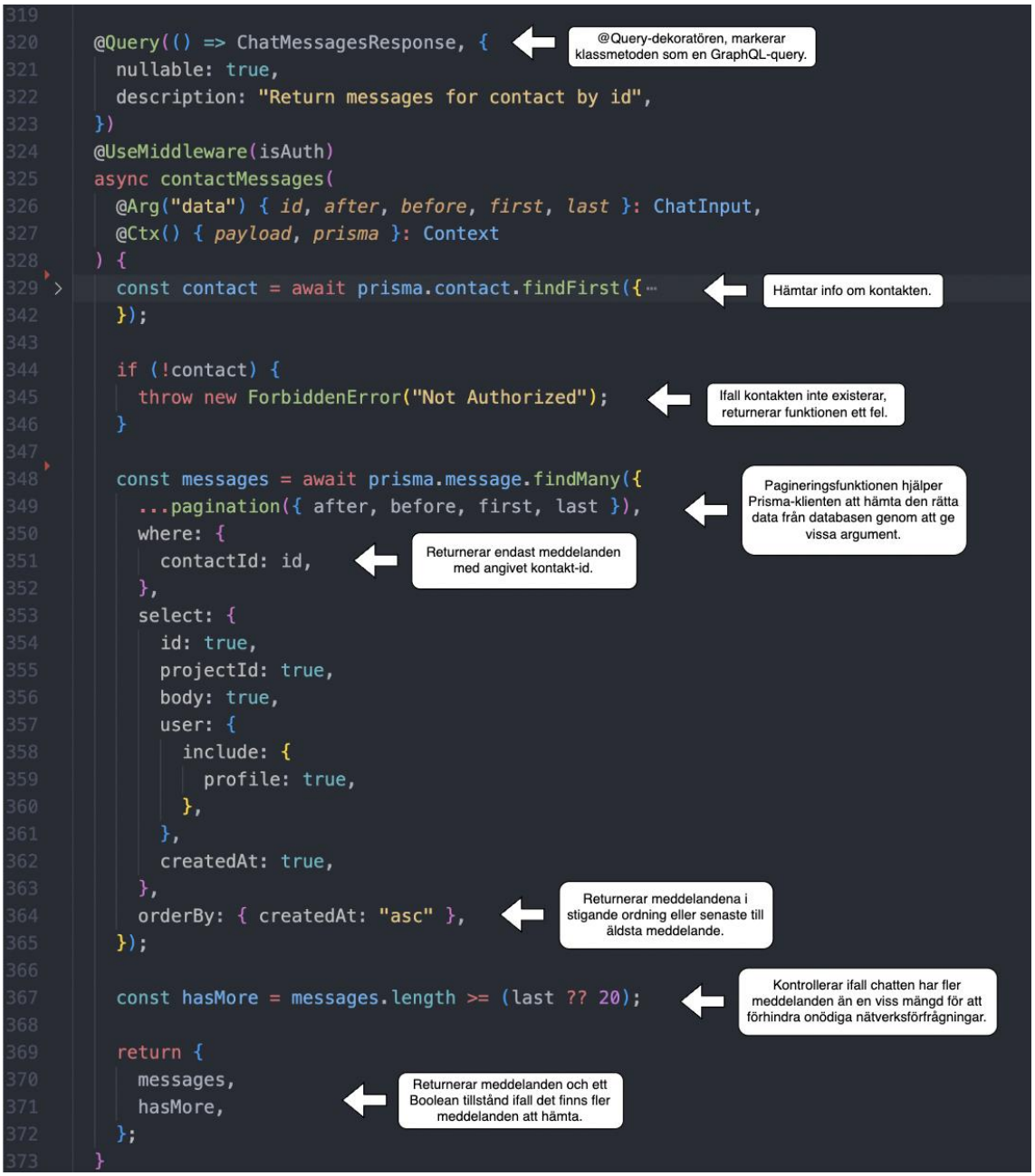
3.2.2 Mutation

I figuren nedan har vi ett exempel på hur resolver-funktionen contactAddMessage är uppställd, samt de olika argument funktionen kan ta emot innan själva logiken inom funktionen körs. Anpassade middleware-funktioner som isAuth middleware-funktionen ger utvecklare återanvändbara sätt att kontrollera ifall användare är inloggade eller inte. Uppställningen och återanvändningen av isAuth middleware funktionen gör så att kodbasen är betydligt mer skalbar och hjälper utvecklare att följa viktiga programmerings principer som DRY eller ”Don’t repeat yourself”. Anpassade typen CreateMessageInput gör funktionen mera typsäker genom att kräva att contactAddMessage måste ha två parametrar av id och body:

Figur 12. ContactAddMessage resolver funktion.

3.2.3 Query

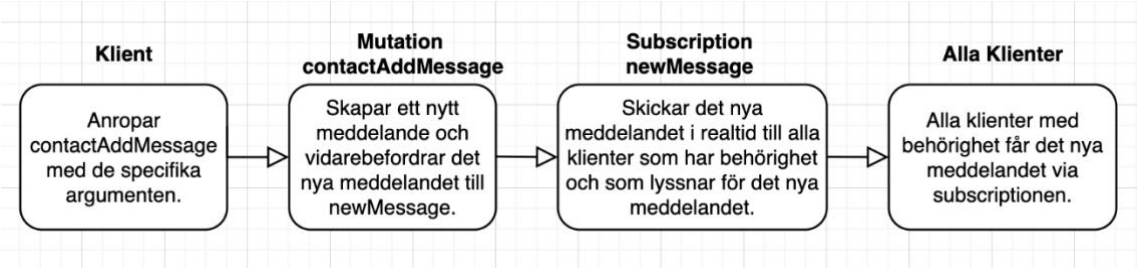
I figuren nedan så hämtar vår query-resolver contactMessages en lista över de senaste meddelanden mellan de två kontakterna. Denna resolver-funktion konsumerar flera argument för att hantera funktionalitet som infinite-scroll för att se till att när en användare på klienten rullar till toppen av chatt-lådan så skickas en ny begäran till GraphQL API, vilket resulterar i att en viss mängd av tidigare meddelande skickas till klienten:



Figur 13. ContactMessages resolver funktion.

3.2.4 Subscription

Subscriptions gör det möjligt för klienter att ta emot meddelanden i realtid. När en användare på klienten anropar contactAddMessage -mutationen, med de definierade argumenten, så skickas ett meddelande från klienten till serversidan. Mutationen sparar sedan det nya meddelandet i en databas och vidarebefordrar meddelandet till newMessage -subscription. Denna subscription skickar sedan meddelandet i realtid till alla klienter som lyssnar efter det (eller subscribear). Klienten måste dock bevisa att användaren har tillstånd att ta emot meddelandet, genom att kolla ifall användarens ID är giltigt.



Figur 14. Subscription flöde inom Collabor8 applikationen.

Figuren nedan beskriver hur newMessage subscription resolver-funktionen fungerar.

Figur 15. newMessage resolver funktion.

3.3 GraphQL & Apollo Client

Detta delkapitel kommer att gå igenom hur man implementerar GraphQL-API:er på klienten med hjälpen av Apollo Client. Delkapitlet kommer även förklara hur man kan generera *types* och färdiga React-hooks via ett GraphQL-schema med hjälp av GraphQL Code Generator eller Codegen.

3.3.1 Codegen

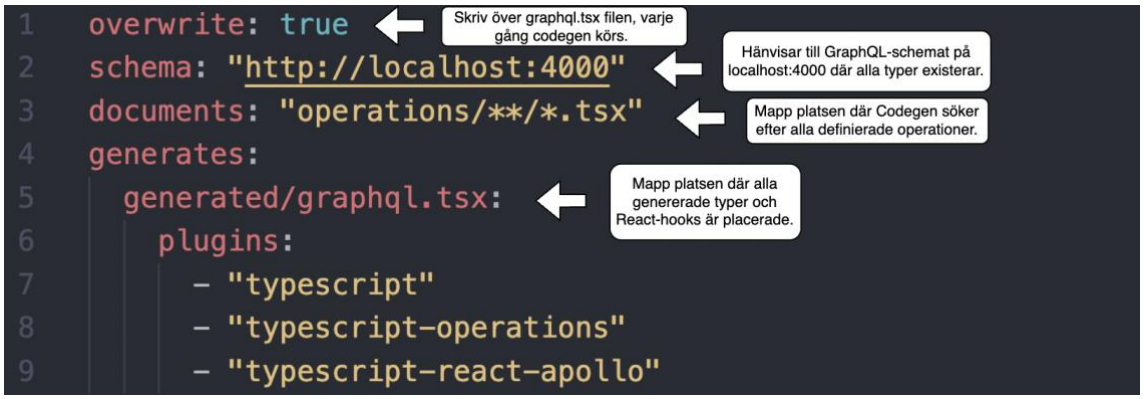
Codegen genererar React-hooks och typer som kan användas med hjälpen av TypeScript för att fånga fel i utvecklingsprocessen inom React-applikationen Collabor8. Dessa hooks är funktioner inom React som oftast returnerar någon slags tillstånd inom en komponent,

utan att själva komponenten behöver vara en klass-baserad komponent. Genom att använda dessa React-hooks och typer, så kommer klienten alltid att vara synkroniserad med de olika typerna som används inom vår GraphQL API. Genom att specificera olika typer inom vår kod så får vi en mycket mindre felbenägen och mer förutsägbar applikation. Codegen kan också användas på serversidan, men användes denna gång inte. För att använda Codegen inuti en React-applikation, måste projektet konfigureras och vissa paket installeras på klienten:

```
npm install -D @graphql-codegen/cli @graphql-codegen/typescript
@graphql-codegen/typescript-operations @graphql-codegen/ typscript-react-apollo
```

Figur 16. NPM paket som måste installeras för att kunna använda sig av Codegen.

Filen codegen.yml anger bland annat katalogen där olika operationer är definierade. Operationer innehåller en specifikation på den data som vi vill att klienten skall hämta från en server. Vanligtvis finns dessa individuella operationer inom sina egna filer, med bra namngivning, i en mapp som kallas för operations. Detta håller kodens struktur ren, så att ifall ändringar krävs kommer det inte att vara så svårt att hitta dessa specifika operationer och till exempel definiera ytterligare fält.



Figur 17. Konfigurationen inom codegen.yml filen.

Som tidigare nämnts så är själva operationerna definierade av klienten i sina egna separata filer inom operations-mappen, med en egen separat struktur var datan som skall returneras till klienten definierats. Med operationerna kan Codegen generera de rätta typerna och React-hooks till klienten. I figuren nedan har applikationen definierat en query string contactMessages som innehåller en contactMessages query resolver-funktion, för att kunna returnera en mängd tidigare meddelanden samt ett tillstånd på ifall det finns flera meddelanden att hämtas:

Figur 18. Query resolver contactMessages, som hämtar tidigare meddelande.

God praxis är att alltid köra kommandot "npm run codegen" för att hålla klienten synkroniserad när GraphQL API:er förändras. Detta kan helt enkelt göras genom att lägga till ett skript som heter "codegen" till filen package.json inom objektet ”scripts”:

```
"codegen": "graphql-codegen --config codegen.yml"
```

Figur 19. Codegen skript inom filen package.json.

Vid det här skedet borde all konfiguration vara klar. För att Codegen skall kunna köras, så måste servern lyssna på en adress samt port: i det här fallet den givna adressen inom filen codegen.yml, alltså localhost:4000. Efter att det är gjort så kan "npm run codegen" köras för att generera ”typer” och React-hooks från GraphQL API till filen generated/graphql.tsx.

3.3.2 Query inom React-applikation

Nu när ”typerna” och React-hooks har genererats, kommer React-komponenterna att kunna importera och använda sig av dem. I figuren nedan har vi en komponent som är döpt till Contact. Komponenten använder sig av useContactMessagesQuery-hook som nyligen blivit genererad av Codegen. Hooken anropar contactMessages query resolvern inom GraphQL API. Ifall inget gått fel hämtas en lista på de senaste meddelanden mellan kontakterna. Hooken returnerar ett objekt som innehåller olika egenskaper som bland

annat *data*, *loading* och *error* som kan användas för att visualisera olika tillstånd i användargränssnittet. *Data-egenskapen* innehåller i detta fall alla meddelanden, loading tillståndet visar ifall useContactMessagesQuery-hook fortfarande laddar efter data och error visar ifall något gick fel:

Figur 20. Codegen genererade React-hook useContactMessagesQuery.

Figur 21. Resultatet returnerat av useContactMessagesQuery.

3.2.3 Mutation inom React-applikation

För att en användare ska kunna skicka ett meddelande till en annan användare måste en *mutation* utföras med vissa specifika värden, såsom meddelandetexten, som skapas med hjälp av ett formulär och chatt-id som kommer ifrån länkens parametrar. GraphQL Codegen generade tidigare en hook (useContactAddMessageMutation), som gör det enkelt för React-klienten att anropa denna *mutation* på GraphQL API. Denna *mutation*

kommer inte att hantera cacheuppdateringar, eftersom vår subscription hook i exemplet nedan kommer att ta hand om det. När denna hook har utförts återställs formuläret och ett nytt meddelande kan skickas i väg:

Figur 22. Användning av useContactAddMessageMutation.

3.2.4 Subscription inom React-applikation

För att ta emot nya meddelanden i realtid måste klienten öppna en WebSocket-anslutning mellan klienten och servern. Den tidigare useContactMessagesQuery returnerade också en subscribeToMore-funktion, vars syfte är att öppna en WebSocket-anslutning till newMessage-subscriptionen inom GraphQL API för att sedan kunna lyssna efter nya meddelanden och mer specifikt, meddelanden för den nuvarande chatten. Funktionen hanterar cachelagring så att när ett nytt meddelande skapas, så läggs det till som det senaste meddelandet i den nuvarande chatten på klienten:

Figur 23. Användning av subscribeToMore

4 RESULTAT

Inom de tidigare kapitlen har arbetet gått igenom skillnaderna mellan GraphQL och REST-API:er och även gått igenom hur man själv kan implementera GraphQL både på server- och klient-sidan. Baserat på utlägget från tidigare kapitel av arbetet kan man dra flera slutsatser varför GraphQL kan vara ett bra alternativ för att hantera data inom webbapplikationer i jämförelse med en standard REST API.

Den största fördelen med GraphQL är att man ger mer flexibilitet åt klienten att kunna definiera vilka data man behöver och att man kommer åt den med endast en anropning. Detta resulterar i minskad användning av bandbredd, pga. att man inte behöver hämta data som inte behövs eller används, eller göra flera nätverksförfrågningar till olika resurser, och därmed skapar effektivare applikationer.

Apollo Client gör det enkelt för klienter att distribuera data som redan har hämtats mellan flera olika komponenter, utan att varje gång hamna göra separata förfrågningar till servern

eller genom att skicka data till en tillståndshanterare som Redux. Subscriptions i sin tur gör det enkelt att skicka data i realtid mellan servern och klienten och Apollo Client möjliggör det att enkelt kunna spara data inom cachelagret.

Däremot kräver GraphQL-applikationer relativt mycket konfiguration i jämförelse med en standard REST API som utvecklats med till exempel Express.js. Inom GraphQL måste man importera varje resolver till ett schema för att klienten skall ha tillgång till dess data, medan med en REST API kommer man direkt åt datan genom att anropa en resurs. Detta kan dock leda till komplikationer för klienten på grund av att man är tvungen att anropa flera olika resursers för att komma åt all behövlig data. Det bör även tilläggas att eftersom att REST-API:er inte använder sig av ett schemadefinitionsspråk som SDL kommer klienten inte kunna generera typer direkt från API:er. Utvecklare måste själv skriva dem manuellt, vilket i sin tur kan leda till en högre felbenägenhet hos klienten.

Både GraphQL och REST API:s har sina för och nackdelar: GraphQL möjliggör skalbarhet, återanvändbarhet, effektivitet och har inbyggd funktionalitet som gör den mer robust vilket gör att den lämpar sig utmärkt för medelstora till stora applikationsprojekt där klienten kräver en stor mängd data och var det inte är önskvärt att behöva uppdatera API:er varje gång datakraven ändras (ändringar eller tillägg i funktionalitet inom applikationen/applikationerna). REST API:s kan däremot vara snabbare och enklare att komma i gång med och kräver mindre arbete att upprätthålla alltså kan det fungera som ett utmärkt alternativ för både små och stora applikationer där backend-utvecklare känner till klientens datakrav och klienten i sig inte har för många olika databehov.

Att lära sig och komma i gång med GraphQL kan kännas svårare än med ett REST-API, men när GraphQL-koncept som konfiguration, scheman och resolvers väl förstås kan utvecklare spara tid genom att använda sig av GraphQL. Frontend-utvecklare kommer att ha mer kontroll över den data som blir skickad till klienten och är inte lika beroende av en backend-utvecklare på att göra en viss ändring för att uppfylla klientens behov. Mindre beroendeförhållanden leder oftast till en högre effektivitet och skalbarhet i sig självt.

4.1 Slutdiskussion

Jag har utvecklat API:er med REST vid flera olika tillfällen innan jag tog mig an att utveckla Collabor8-webapplikationen med GraphQL. Att konfigurera allting tog mig en relativt lång tid, både på server- och klient-sidan, särskilt i relation med samma arbete i jämförelse med en REST API. Däremot när jag väl hade konfigurerat allting tog det inte länge att inse fördelarna med GraphQL och hur GraphQL kan hjälpa till med att snabba till utvecklingsprocessen av vissa applikationer, särskilt om det finns stora krav på data eller om kraven på data ändras ofta eller snabbt.

Både GraphQL och REST API:s har sina fördelar och nackdelar, och efter att ha arbetat med båda så tror jag att det är fördelaktigt för varje utvecklare att först bekanta sig med REST-API:er innan de lär sig om GraphQL och dess fördelar då REST-API:er utgör grunden för det mesta inom GraphQL. Att jag hade grundläggande kunskaper inom REST API:s gjorde att jag, med alla mått mätt, relativt snabbt kunde komma i gång med GraphQL och i sin förlängning kunde utveckla webbapplikationen Collabor8.

KÄLLOR

Apollographql, 2022, Subscriptions. Tillgänglig: https://www.apollographql.com/docs/react/data/subscriptions/ Hämtad: 27.8.2022

Apollographql.com, 2022, Why Apollo Client?. Tillgänglig: https://www.apollographql.com/docs/react/why-apollo/ Hämtad: 8.1.2022

Aws, 2021, What is PostgreSQL? Tillgänglig: https://aws.amazon.com/rds/postgresql/what-is-postgresql/ Hämtad: 8.1.2022

Budibase, 2021, What is a CRUD app and how to build one | Ultimate guide. Tillgänglig: https://budibase.com/blog/crud-app/ Hämtad: 27.8.2022

Chand Mahesh, 2022, Most Popular Databases In The World. Tillgänglig: https://www.c-sharpcorner.com/article/what-is-the-most-popular-database-in-the-world/ Hämtad: 8.1.2022

Facebook Engineering, 2015, GraphQL: A data query language. Tillgänglig: https://engineering.fb.com/2015/09/14/core-data/graphql-a-data-query-language/ Hämtad: 27.12.2022

Gopalakrishnan Ashita, 2021, GraphQL vs REST API. Tillgänglig: https://bejamas.io/blog/graphql-vs-rest-api/ Hämtad: 27.8.2022.

GraphQL, 2022, A query language for your API. Tillgänglig: https://graphql.org/ Hämtad: 2.1.2022

Lawson Katie, 2022, What Is a Single Page Application and Why Do People Like Them so Much? Tillgänglig: https://www.bloomreach.com/en/blog/2018/07/what-is-a-single-page-application.html Hämtad: 9.1.2022

Lytek Michał, 2018, GraphQL + TypeScript = TypeGraphQL. Tillgänglig: https://medium.com/@MichalLytek/graphql-typescript-typegraphql-ba0225cb4bed Hämtad: 8.1.2022

Mozilla, 2022, The WebSocket API (WebSockets). Tillgänglig: https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API Hämtad: 27.8.2022

Newseed, 2021, REST API – vad betyder det? Tillgänglig: https://newseed.se/fakta/rest-api-vad-betyder-det Hämtad: 30.12.2021

Nextjs, 2022, The React Framework for Production. Tillgänglig: https://nextjs.org/ Hämtad: 9.1.2022

Patel Priyesh, 2018, What exactly is Node.js? Tillgänglig: https://www.freecodecamp.org/news/what-exactly-is-node-js-ae36e97449f5/ Hämtad: 8.1.2022

Postman, 2022, A Guide to the Most Popular APIs: REST, SOAP, GraphQL, gRPC, and WebSockets. Tillgänglig: https://blog.postman.com/guide-to-apis-rest-soap-graphql-grpc-websockets/ Hämtad: 30.12.2021

Stemmler Khalil, 2021, GraphQL Mutation vs Query – When to use a GraphQL Mutation. Tillgänglig: https://www.apollographql.com/blog/graphql/basics/mutation-vs-query-when-to-use-graphql-mutation/ Hämtad: 4.1.2022

Typescript, 2021, What is TypeScript?. Tillgänglig: https://www.typescriptlang.org/ Hämtad: 27.8.2022.

Yachaka, 2017, What is Over-fetching or Under-fetching? Tillgänglig: https://stackoverflow.com/questions/44564905/what-is-over-fetching-or-under-fetching Hämtad: 31.12.2021