BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INFORMATION SYSTEMS
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FRAMEWORK FOR DEVELOPMENT AND OPERATION OF CLOUD SERVICES

RÁMEC PRO VÝVOJ A PROVOZ CLOUDOVÝCH SLUŽEB

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR **PETER HAMRAN**
AUTOR PRÁCE

SUPERVISOR **doc. Ing. RADEK BURGET, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2022

Brno University of Technology
Faculty of Information Technology

Department of Information Systems (DIFS) Academic year 2021/2022

Master's Thesis Specification

Student: **Hamran Peter, Bc.**
Programme: Information Technology and Artificial Intelligence
Specialization: Intelligent Systems
Title: **Framework for Development and Operation of Cloud Services**
Category: Information Systems
Assignment:

1. Learn about the major global cloud providers and the type and range of services available.
2. Explore the current technologies for implementing both the server and client side of web applications, with a focus on running in the cloud.
3. With supervisor's agreement, select a cloud platform and design the architecture of a framework solution for application development that includes user management, client component integration, and other parts.
4. Implement the proposed solution using appropriate technologies.
5. Test the developed solution on a suitable demonstration application.
6. Evaluate the achieved results.

Recommended literature:

- Kavis, M. J.: Architecting the Cloud: Design Decisions for Cloud Computing Service Models (SaaS, PaaS, and IaaS), Wiley, 2014, ISBN: 978-1-118-61761-8

Requirements for the semestral defence:

- Items 1 to 3.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor: **Burget Radek, doc. Ing., Ph.D.**
Head of Department: Kolář Dušan, doc. Dr. Ing.
Beginning of work: November 1, 2021
Submission deadline: May 18, 2022
Approval date: January 28, 2022

Abstract

The complexity of modern application development is rising. In this thesis, my efforts aim at designing and demonstrating a framework consisting of cloud-native services that target common business-critical issues. I explore the available cloud service providers on the market and current technologies for implementing both server and client-side web applications running in cloud environments. I have developed the services of this framework following microservice architecture principles with a working demo application that utilizes this framework.

Abstrakt

Zložitosť vývoja moderných aplikácií postupne narastá. V tejto práci sa snažím navrhnúť a demonštrovať aplikačný rámec skladajúci sa zo služieb navrhnutých pre cloudové prostredie, ktorý rieši problémy dôležité pre podnikanie. Vytvoril som súhrn existujúcich poskytovateľov cloudových služieb spolu s prehľadom technológií dôležitých pre implementáciu serverovej a klientskej strany aplikácií so zameraním na prevádzku v cloude. Aplikačný rámec je navrhnutý s ohľadom na princípy architektúry mikroslužieb ako distribuovaný systém služieb spolu s fungujúcou demo aplikáciou, ktorá ich využíva.

Keywords

Cloud, Framework, Framework Design, Microservice Architecture, Platform as a Service, Python Django, React, Google Cloud Platform

Kľúčové slová

Cloud, Aplikačný Rámec, Návrh Aplikačného Rámca, Architektúra Microservice, Platforma ako Služba, Python Django, React, Google Cloud Platform

Reference

HAMRAN, Peter. *Framework for Development and Operation of Cloud Services*. Brno, 2022. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor doc. Ing. Radek Burget, Ph.D.

Rozšírený abstrakt

Moderný prístup k vývoju aplikácií sa s príchodom inovácií v sektore cloudových výpočtov stal viac dynamickým. Prístup k výpočtovým zdrojom je jednoduchší ako kedykoľvek predtým a súčasne sa vývojár môže spoľahnúť na stabilné pripojenie k internetu. Avšak tento stav technologického vývoja zvyšuje tlak na individuálnych developerov a zapríčiňuje nárast požiadaviek na finálny produkt. Tento fakt má dopad a limituje lukrativitu individuálneho podnikania pri narastajúcom trende freelancingu.

Problémom je, že developer sa prostredníctvom svojej aplikácie snaží vyriešiť určitú prekážku, ktorú vo svojom okolí identifikoval. Aplikácia však musí pokrývať technické aj podnikateľské výzvy. To pre developera prezentuje značnú počiatočnú investíciu. Aj keď technologické riešenia, ktoré ponúkajú poskytovatelia cloudových služieb sú veľmi dostupné, ich porozumenie a implementácia vôbec nie je zanedbateľné kritérium. Pre vývoj produktu, s ktorým sa dá prezentovať na trhu je ale potrebné pokryť aj aspekty ako napríklad licencovanie alebo platby. Aplikačný rámec prezentuje extrakciu takýchto služieb na vyššiu úroveň, pričom využíva znalosti, ktoré priemerný developer nemusí mať. Vďaka tomu dokáže pri vývoji poskytnúť služby ako napríklad licencovanie, ale aj základňu užívateľov, čo následne prispieva k nárastu záujmu o aplikácie.

Mojím cieľom v tejto práci bolo navrhnúť aplikačný rámec, ktorý podporuje vývoj aplikácií v cloudovom prostredí. V takto navrhnutom informačnom systéme som identifikoval dvoch aktérov, jedná sa o užívateľov a developerov.

- Developer je osoba, ktorá aktívne využíva služby navrhnuté v aplikačnom rámci na vývoj aplikácií.
- Užívateľ je osoba, ktorá následne konzumuje tieto aplikácie.

Preto takýto systém vystupuje ako platforma ako služba pre developerov a ponúka softvér ako službu pre užívateľov.

V kapitole 2 predstavujem najväčších poskytovateľov cloudových služieb na trhu. Keďže počet cloudových poskytovateľov je obrovský, rozhodol som sa zamerať na trojicu najväčších a to Amazon Web Services, Microsoft Azure a Google Cloud Platform. Služby, ktoré poskytujú, sa čiastočne prekrývajú z dôvodu, že sa snažia pokrývať značnú časť trhu, avšak aj napriek tomu tu existujú rozdiely.

Kapitola 3 zahŕňa teóriu potrebnú pre pochopenie a návrh škálovateľných webových aplikácií. V tejto kapitole vysvetlujem prečo a kedy využívať výhody architektúry mikroslužieb v takýchto aplikáciách oproti zaužívanému monolitu. Taktiež sa snažím poukázať na isté problémy, ktoré z toho rozhodnutia plynú.

Technológie na vývoj aplikácií v cloude sú predstavené v kapitole 4. Demonštruje technológie nevyhnutné alebo užitočné pri návrhu a implementácií cloudových aplikácií, ako napríklad aplikačné rozhrania (API), jazyky, ktoré majú dobrú podporu v cloudovom prostredí, ako aj rámce použiteľné na vývoj aplikácií.

Kapitola 5 obsahuje samotný návrh a implementačné detaily aplikačného rámca. Do práce som sa snažil zhrnúť zaujímavé adôležité fakty a poukázať na procesy, ktoré som pri práci využil. Taktiež tu predstavujem návrh a využitie demo aplikácie, ktorá využívate aplikačný rámec v praxi.

Nakoniec kapitola 6 sumarizuje problémy, s ktorými som sa pri riešení uvedenej problematiky stretol a musel vysporiadať. Taktiež tu spomínam zaujímavosti, na ktoré som pri vypracovávaní tejto témy narazil.

Framework for Development and Operation of Cloud Services

Declaration

I hereby declare that this Master's thesis was prepared as an original work by the author under the supervision of doc. Ing. Radek Burget Ph.D. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

.....
Peter Hamran
May 18, 2022

Acknowledgements

First and foremost, I would like to thank my professor Ing. Radek Burget, Ph.D for allowing me to pursue the topic of this thesis. I also want to thank my friends and family for their moral support. Namely, I want to thank my friends and colleagues Kamil Pšenák and Tomáš Daniš for our brainstorming sessions and their technological expertise and also Nikita Nikolaenko for his great expertise in cloud technology. This thesis would not be possible without the support of the people I have around me.

Contents

1 Introduction **3**

2 State of the art Cloud Computing **5**
2.1 Worldwide cloud service providers 5
2.2 The *aaS World 6
2.2.1 Infrastructure as a Service 7
2.2.2 Platform as a Service 7
2.2.3 Software as a Service 7
2.2.4 Serverless Computing 8
2.3 Containerization 8
2.4 Amazon Web Services 10
2.5 Microsoft Azure 13
2.6 Google Cloud Platform 15
2.6.1 Google Compute Engine 16
2.6.2 Cloud Run 16
2.6.3 Cloud Storage 17
2.6.4 Spanner 18
2.6.5 Firebase 18

3 Web Application Architecture **20**
3.1 Monolithic Architecture 20
3.2 Microservice Architecture 22
3.3 Distributed Transactions 25

4 Cloud Native Tech Stack **29**
4.1 Gateway to the Cloud 29
4.2 HTTP Messaging 30
4.3 REST-ful API 31
4.4 GraphQL API 32
4.5 Pub-Sub Messaging 34
4.6 API Gateway 36
4.7 OpenAPI Specification 38
4.8 Technology Stack for Application Development in Cloud 42
4.9 Java and Jakarta EE 42
4.10 Java Spring Framework 43
4.11 C# ASP.NET Framework 44
4.12 Node.js Express Framework 44
4.13 Python Flask 45

4.14 Python Django Framework . . . 45
4.15 Typescript . . . 46
4.16 React . . . 46
4.17 Angular . . . 48

5 Technological Design **49**
5.1 Domain Model . . . 49
5.2 Infrastructure . . . 52
5.3 API Design . . . 53
5.4 Transactional Consistency . . . 54
5.5 Component Design . . . 54
5.6 User Authentication . . . 55
5.7 Application State Management . . . 55
5.8 Demo Application . . . 56

6 Project Takeaways **59**
6.1 Cloud Runtime . . . 59
6.2 Automatic Scaling . . . 59
6.3 Framework Authentication . . . 60
6.4 Asynchronous Frontend . . . 60
6.5 Service Provisioning . . . 61

7 Conclusion and Future Work **62**

Bibliography **63**

Chapter 1

Introduction

With innovations in the cloud computing sector, modern application development has shifted to a more dynamic model. Access to computational resources is easier than ever, and the developer can rely on the stability of the internet connection. However, this state of technological development puts pressure on individual developers where their solutions have the potential to be more thorough with the available resources. It strains the rising trend of freelancing in application development and IT in general.

The problem is that a developer can have a specific business problem in mind with an application designed to solve it. The application has to solve both technical and business challenges. Therefore, it carries an initial cost to the developer. The technological solutions offered by cloud providers might be more accessible, but they become more complex. It requires upfront research when choosing the right services to consume to prevent the project from failing in the beginning. The business challenges are harder to get right with no previous experience. The framework tries to leverage the knowledge of business processes and an existing user base of its platform to provide tools while accelerating the application adoption by users.

My goal in this thesis is to design an application framework that aims at aiding the developers in application development in the cloud environment. The implementation of this framework is a Platform as a Service solution. It acts as a development portal providing services that target business-critical concerns like licensing and billing while providing an interface for user authentication and application management. It also aims to provide a platform for consumers where they can potentially find solutions that fit their needs. The platform offers Software as a Service solutions to the consumer.

In chapter 2, I introduce the biggest worldwide providers of cloud services. There are many cloud service providers on the market. The holy trinity of Amazon Web Services, Microsoft Azure, and Google Cloud Platform is the staple of cloud computing in the modern era. The overlap of offered services is significant, yet the specific implementations frequently differ.

Chapter 3 introduces the theory needed to understand and design scalable web-oriented applications. I go into why and when using microservice architecture benefits the application compared to the monolithic design. I also outline some challenges that emerge from this transition.

I explore the technologies necessary for cloud-native application development in chapter 4. It contains technologies like APIs, cloud-friendly languages and language-specific frameworks.

Chapter 5 outlines the framework design choices and their implementation. I tried to tackle all the important topics and thought processes that went into designing the framework. I also introduce the design of the demo application that aims to put the services into practice.

Last but not least, chapter 6 summarizes the lessons learned and challenges encountered during the design and implementation parts of this thesis.

Chapter 2

State of the art Cloud Computing

The National Institute for Standards and Technology (NIST) defined cloud computing as:

Cloud computing is a model for enabling convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction[17].

Cloud computing is a market response to the inevitable expansion of the internet. As the internet infrastructure grows, more and more people gain access to online services. Yet, the world is still unevenly covered with internet connectivity ranging from over 97% coverage in Northern Europe to some regions in Africa having below 10% of coverage[16]. Experts predict that those regions will generate new users exponentially[4].

Another huge consumer of online services is IoT devices. In the last five years, the amount of IoT devices have grown larger than non-IoT users of the internet[24]. And this amount of devices requires resilient and secure architecture to run reliably.

Existing cloud computing covers the needs of both groups by providing a spectrum of products like infrastructure, platform or software as a service. But also raw computing resources, disc storage or pre-built databases. There are many ways that a business can benefit from using cloud computing services, and I will attempt to outline some of them in this chapter. This approach is often adopted to improve user experience by bringing the services geographically closer to the end-user.

Worldwide cloud coverage roughly follows the internet coverage trends. We can see a high density of cloud providers in more developed regions to lesser developed ones.

2.1 Worldwide cloud service providers

Cloud computing is the new go-to model for globally deployed services. In the last years, it has gained traction as companies prioritize as-a-service providers over the traditional hosting vendors. One of the reasons is how easy it is to start using these services out of the box. All the major cloud computing providers guarantee high levels of security, compliance with data retention standards and availability. And businesses have to consider these aspects when building their infrastructure.

Another reason for considering cloud over local hosting providers is its elasticity. Cloud elasticity is one of the cloud functions that allow it to scale up or scale down its resources. This functionality can be automatic or manual based on the nature of the scaled technology. It provides businesses with an option to meet occasional spikes in demand that usually

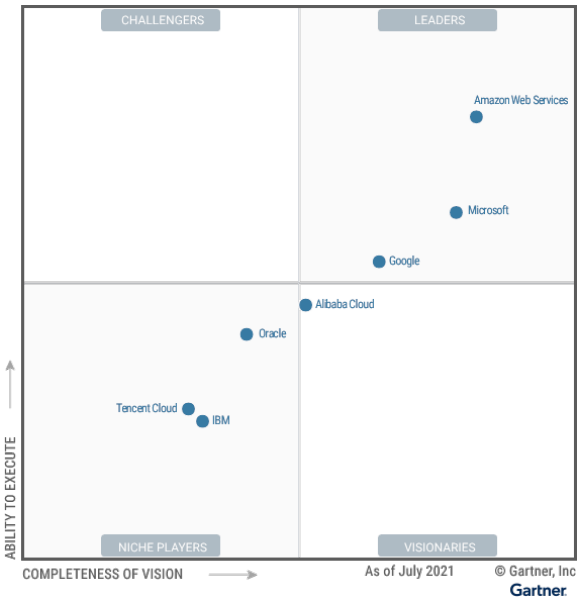

Figure 2.1: Magic Quadrant for Cloud Infrastructure and Platform Services[21]

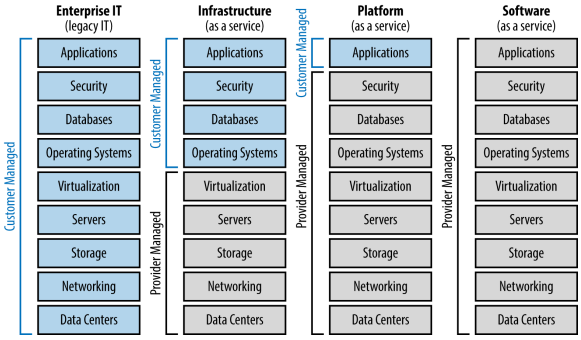

Figure 2.2: Cloud provider versus customer roles for managing cloud services[6]

happen in rush hours or at launches of new products. Another non-technical aspect of cloud elasticity is that businesses don't need to pay the price for the resources they are not using. This way, cloud providers attempt to optimize the load across their hardware.

2.2 The *aaS World

The cloud ecosystem has become a complex, ever-changing pool of providers, technologies, products and services. The general public recognizes it as a service pool due to many other business aspects that come alongside (e.g. SLAs, Help Desk, guides). Therefore, the market has adopted the naming convention of as a Service. Some of the frequently used services are infrastructure, platform or software, but there is no limitation to what can clouds provide as a Service 2.2.

2.2.1 Infrastructure as a Service

Infrastructure as a service (IaaS) is the most elementary service modern cloud providers offer. It provides the consumer with options for managing the resources required for building an IT infrastructure. It abstracts the consumer from problems that come from owning on-premise hardware. The consumer accesses the hardware components using a web-based management console where he has direct control over a set of services. This abstraction from a hardware component to service enables cloud service providers to shield the customer from the underlying structure.

There are many benefits to using IaaS instead of an on-prem solution. The cloud service provider is managing the security of your infrastructure for you. There are many measures in place to protect your data from being stolen or altered. Some of them are end-to-end encryption and encryption at rest. Increased resilience and disaster recovery are other benefits provided by cloud service providers. With guaranteed data backup, your infrastructure will run even if there is a failure. And this is all covered by the service level agreements that cloud providers offer.

2.2.2 Platform as a Service

Platform as a service (PaaS) is the next abstraction on the cloud stack. In PaaS, the consumers rely heavily on the service provider for development tools, infrastructure and operating systems. It focuses on the development, runtime and growth of applications. Provides high-level services like caching, asynchronous communication and computing and storage services while keeping the underlying infrastructure hidden together with patch management, capacity planning or resource procurement.

Platform as a service works best for small businesses and individuals thanks to its abstractions. It allows more focus on the specifics of your application while you do not need to worry about maintenance. It results in time and cost-saving and increased speed to market ratio. PaaS also provides options for dynamic scaling of your applications.

On the other hand, the platform as a service introduces a vendor lock-in. When we build our application with the functions provided by a vendor platform, we essentially intertwine our technology stack with their platform-specific services. It can become an issue when the vendor substantially changes the product offering or when the price increases and the solution is no longer viable.

2.2.3 Software as a Service

Software as a service (SaaS) is at the top of a cloud stack structure. It is a complete product supported by the service provider that requires only application-specific configurations and user management. The consumers of SaaS are usually the end-users of cloud-native applications. The service provider manages everything from the business logic of a SaaS application to its delivery. Common examples of SaaS applications are customer relationship management (CRM), accounting and other business application. But also, a web-based email client can be considered a SaaS application.

Applications built as software as a service are inherently multitenant applications to provide services to many customers. It is the case because the underlying infrastructure and business logic are the same for all consumers.

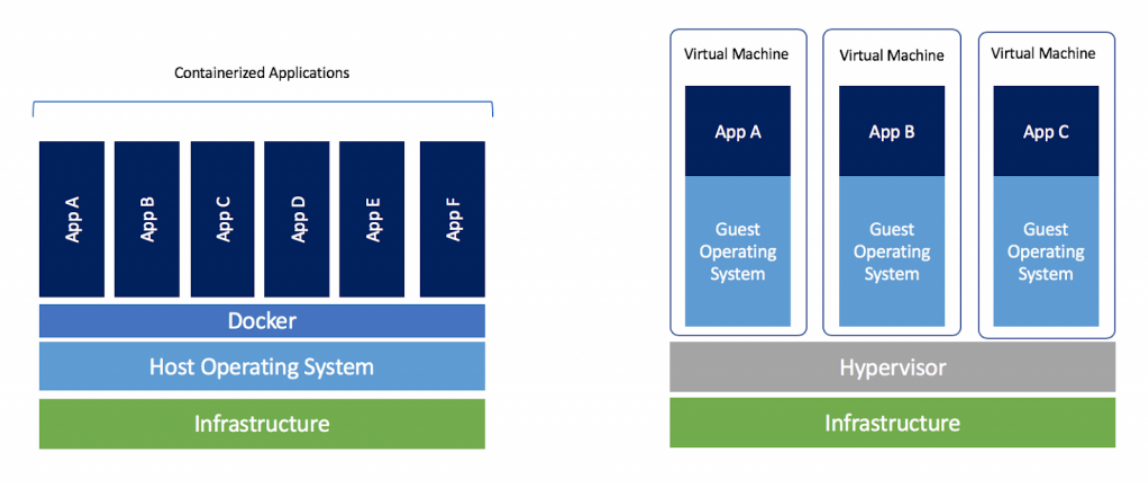

Figure 2.3: Containers versus Virtual Machines[8]

2.2.4 Serverless Computing

Serverless computing is a very similar concept to the platform as a service. The service provider allows the consumer to rent out backend services on the go.

The difference between serverless computing and platform as a service is better scalability. The applications based on serverless technology scale instantly and automatically with any additional configuration. Serverless also allows applications to scale down to complete inactivity when there is no traffic.

2.3 Containerization

Containerization in cloud computing is packaging the code together with its related dependencies. Therefore, the application has everything it needs and can run smoother in an isolated environment. It also helps the deployment process because containers are predefined and create the same environment wherever we deploy them. It is very similar to virtualization. While both technologies allow the packaging of applications into an environment, containers do this more efficiently.

The main difference between containers and virtualization is resource allocation. While both technologies provide a segregated environment for our applications, virtualization takes it up a notch and creates guest operating systems to run the application. While containers run directly on the host operating system, they share the host OS kernel and libraries. It results in much lightweight and portable architecture.

The first thing when working with containers is to have an application you want to containerize. You want to put it in an isolated environment together with all the dependencies it needs. We define the outlines of our container by creating a docker file. Docker files serve as container manifests. Once we have a docker file, we build a docker imagine based on the rules we have established in it. When we have our image built, it can be pushed to a registry and create a final container out of it.

Another feature of containers is that they heavily rely on resource sharing. When deploying multiple containers on the same machine, they reuse the already existing resources they share and only contain the bare minimum they require to run in isolation.

What is Docker?

Docker is fairly popular nowadays and is often used interchangeably with containers. Docker is a software framework for building, running and managing containers. It is one of many tools developers use for containerization on servers and clouds.

Linux containers have facilitated a massive shift in high-availability computing [2]. There are many toolsets to help run services or even entire operating systems in containers. The Open Container Initiative (OCI) is an industry standards organization that encourages innovation while avoiding the danger of vendor lock-in. Thanks to the OCI, there is a choice when choosing a container toolchain, including Docker, CRI-O, Podman, LXC, and others.

The Docker engine is a useful tool for lone developers as it is lightweight, clean environment for testing, but without a need for complex orchestration. It introduces its own terminology and structure that is found in every container engine available in some form. I will go into further details on these structural elements in the upcoming sections.

Dockerfile

The dockerfile serves as an instruction manifest for docker. You can think of it as a classic shell script. The file contains commands a user would manually enter on the command line to create an image manually. But this way, the file contains commands that specify how to build a docker image.

Dockerfiles are just regular text files with a special syntax. Docker defines a set of supported instructions, and I will try to outline the most common ones. You can find the complete dockerfile instruction documentation on the docker docs website.

As an example demonstration of a dockerfile, I create a dockerfile that builds an image with python and flask.

```
1  # syntax=docker/dockerfile:1
2
3  FROM python:3.8-slim-buster
4
5  WORKDIR /app
6
7  COPY requirements.txt requirements.txt
8  RUN pip3 install -r requirements.txt
9
10 COPY . .
11
12 CMD [ "python3", "-m" , "flask", "run", "--host=0.0.0.0"]
```

Every docker file has to start with the `'FROM'` command to specify a starting point to build the image. You can use the 'FROM scratch' to explicitly initialize with an empty docker image, but usually, you will use an existing docker image to start. I have initialized my docker image from 'python:3.8-slim-buster' with the first command in the example.

Additionally, you can specify a parser directive in your dockerfile on the first line. The parser directive is not considered being a command and is optional. It is similar to using a shebang at the beginning of a script file or specifying a schema specification at the start of JSON schema.

The next set of commands sets our working directory with the `'WORKDIR'` command to *'/app'*. This way, we can have an easier time with paths. We need to copy everything we need to set up our environment. In this case, it will be our *'requirements.txt'* file with the

'COPY' command. The `'RUN'` command executes standard bash scripts and binaries like you would on any Linux OS. In this case, `'RUN pip3 install -r requirements.txt'` installs dependencies from *'requirements.txt'*. Then the `'COPY . .'` command copies all the files located in the current directory and copies them into the docker image. The last command from the example runs the flask application once the docker image executes inside a container.

Every command we issue in a docker file acts as a separate layer. You can understand layers as image snapshots. When we introduce a change in a layer, the engine will only rebuild the image from that layer, and everything before will stay the same. It is called layer caching. You can disable the layer caching when the functionality is not desirable by using the `-no-cache=true` option.

Docker Image

The docker image is an intermediate immutable object between the dockerfile manifest and a container. It contains the application code, libraries, tools and other dependencies required by the application.

A hierarchy of base images containing different base technologies like programming languages or tools developers can base on their docker images.

The naming in the official docker images follows a simple principle. First, the name starts with a technology the docker image contains. In the example from the Dockerfiles section, we have used the 'python:3.8-slim-buster' image as a source image. The binaries for the python environment are in this docker image. What follows after the colon is called a docker tag. This tag contains various information about the docker image itself. For example, '3.8' is the version of python binaries, 'slim' makes the image smaller by installing only the necessary packages to run python, 'buster' is a Debian release in case the application has compatibility requirements.

Container

The docker image becomes a container at runtime. Docker images are immutable and therefore can not change their state. The state inside a docker image can only change before being built or running.

The best practice for creating containers is that each container should take responsibility for one aspect of the project. Containers are also stateless. Once the container is destroyed, it starts from a clean slate, just as designed in the dockerfile.

2.4 Amazon Web Services

Amazon Web Services (AWS) was the first provider of cloud computing services out of the big trio of AWS, Microsoft Azure and Google Cloud Platform. Many organisations use the services and products offered by AWS as they hold over 30% market share. One of AWS's most used services is Amazon EC2, which lets customers create virtual machines for their strategic projects while spending less time on maintaining servers. Another service is Amazon Simple Storage Service (S3), which offers a secure file storage service. In addition, Amazon also provides security, website infrastructure management, and identity and access management solutions.

Amazon Web Services (AWS) operates in regions in the United States, South America, Europe and the Asia Pacific. Each region contains between two and five availability zones that are geographically separate from one another.

Amazon Elastic Compute Cloud

The Amazon Elastic Compute Cloud is also known as EC2. It is the lowest form of abstraction that AWS offers and the service is considered to be IaaS 2.2.1. It offers the users access to the creation and management of virtual machines based on predefined images from the predefined Amazon Machine Image (AMI) or from a custom AMI image. The EC2 also enables the import of existing on-premise virtual machine images to the cloud.

The benefit of EC2 comes from the basis that it is an IaaS and therefore offers computing capability for rent instead of the need for purchasing the underlying hardware. In addition to general-purpose instances, AWS offers an instance type for computing, memory, accelerated computing, and storage-optimized workloads that can all be deployed under the EC2 service.

The EC2 service allows for complete control of instances which makes the operations as simple as on an owned machine. The underlying secure connectivity to other cloud services makes it a secure and easy solution for computing, query processing and cloud storage use-cases. The downside of operating in the cloud at such a low abstraction poses challenges in resource utilization. EC2 service is paid for instance per hour of runtime. This means that the developer must manage the number of instances needed for the task at hand to avoid long and costly runtimes. Also, the developer is responsible for the management of AMI instances in case of custom configurations.

The most common use-cases[1][3] for EC2 instances are:

- Running enterprise applications
- Running high-performance computational applications
- Training and deployment of machine learning applications
- Creating environments for development and testing

Amazon Simple Storage Service

The Amazon Simple Storage Service is also known as Amazon S3. It is a scalable and high-speed cloud storage PaaS 2.2.2 for online backups and data archiving. It differs from regular storage models as it is an object storage service. Object storage is an abstraction where the developer does not have to manage the data as files or blocks. The data are represented by an object ID, which the developers and applications can use to access the stored object.

The S3 service offers developers a set of storage classes. Each storage class is optimized for different use-cases. There is a standard storage class designed for the most common type of use where the data stored are frequently accessed with low latency and high throughput. Then the S3 service offers multiple classes with so-called intelligent tiering. These classes expect that data access needs will be changing or are generally unknown to the developer. It has four different access tiers and can be adapted on the fly, the tiers are frequent access, infrequent access, archive and deep archive. Last but not least I will mention the Glacier storage class. As one would expect this storage class is optimized for infrequent access and is a good solution for archiving data.

The storage itself is organized in units called buckets. The bucket can be viewed as a logical container or a namespace to store objects. There is no limit to the number of objects that can be stored in a bucket, but AWS poses a quota of 100 buckets per account.

Amazon DynamoDB

Amazon DynamoDB is a NoSQL database hosted on the AWS platform. More specifically it is a fully managed, serverless, key-value NoSQL database designed for high-performance queries. The storage for DynamoDB instances is created using arrays of solid-state drives. It creates a suitable environment for high I/O performance and fast handling of high-scale requests.

The DynamoDB infrastructure enforces replication across at least three availability zones for high availability and durability. An availability zone is an isolated location within the data centre region.

Using the Amazon DynamoDB service is a good fit for:

- Extensive integration with AWS Lambda makes DynamoDB a great fit for building serverless applications. The integration enables Lambda functions to directly interact with the DynamoDB and respond to data changes without the need for a lot of custom logic.
- DynamoDB simple key-value access patterns make it a fast and reliable solution for generating and serving recommendations to the users of a client application.
- Applications working with large amounts of data can experience latency problems when using standard SQL queries and joins over massive databases. DynamoDB guarantees predictable latency for queries of any size.

AWS Lambda

AWS Lambda service is a function based, self-contained environment. It classifies as a serverless computing service 2.2.4 that offers a set of supported languages and runtimes for developers to use for their applications. However, AWS Lambda does not support applications in itself. The code running under this service is considered to be a function. The Lambda functions can perform any kind of computing task, from serving web pages and processing streams of data to calling APIs and integrating with other AWS services.

The Lambda functions run in separate containers. When the function is created, the Lambda service packages it into a new container based on the selected language and environment and then executes the container once triggered. One of the main architectural features of AWS Lambda is that many instances of the same function can be created and executed concurrently without the need for complicated configuration. Once the function is done with the work is has been assigned it is shut down. It enables the system to scale down to nothing when the services are not used.

AWS Lambda functions are a great fit for tasks that run for a short period of time, are generally self-contained and have the potential to experience spikes in usage. Some of the most common use-cases for AWS Lambda are scalable APIs. They perfectly fit the profile of being simple requests that are self-contained and with potential spikes in usage. This profile of execution greatly benefits the microservice architecture 3.2. With its event-driven model, AWS Lambda is a great fit for data processing. A good example of this is to have

a Lambda function do some work every time a new record is submitted to the database, therefore, creating a notification for this change.

2.5 Microsoft Azure

Microsoft Azure is one of the two most dominating cloud computing platforms on the market. Microsoft provides various IT solutions ranging from desktop applications to enterprise solutions. When talking about cloud computing, they offer the highest data centre coverage. It allows the deployment of solutions to different locations around the world and ensures high accessibility of resources. Consumers of Azure services can rely on the triple replication guaranteed by most service level agreements.

Thanks to Microsoft having previous experiences with infrastructure and enterprise systems, they offer a well-integrated set of services like Azure Active Directory as an identity provider or services to manage virtual machines, among other things. Access to all the resources is through the Azure portal.

Azure Virtual Machines

Azure Virtual Machines service is typically the IaaS2.2.1 developers choose when they need more control over the computing environment. As it can be understood from its name, this service offers the lowest abstraction in the cloud environment available in form of virtual machines. Azure Virtual Machines are scalable, on-demand computing resources offered by Microsoft.

At its core, it is similar to Amazon's Elastic Compute Cloud. It offers the flexibility of creating a cloud infrastructure without having to buy and own any hardware.

Azure Virtual Machines uses virtual hard disks (VHDs) as storage for OS and data. Azure provides many images for use with various versions of the Windows Server operating system in the marketplace.

The key component of Azure Virtual Machines architecture is the Azure Fabric controller. Independent of any operational intervention, it governs the patching, provisioning and scaling of cloud nodes.

Azure Virtual Machines is often used as:

- Development and test environments that can be scaled up in computing power as necessary and easily duplicated.
- Running an application on a virtual machine in Azure caters to the unpredictability in demand for an application. It avoids making a big investment into on-premise components and instead allows better and faster scalability.
- On-premise infrastructure extension. The Azure cloud has great native intra-operability with running on-premise infrastructure based on Windows Server operating system. This enables developers easy integrations with already existing cloud-native and on-premise services.

App Service

Azure App Service is a Platform-as-a-Service 2.2.2 that is suitable for web applications, REST APIs and mobile backends. It supports a variety of programming languages and

application environments. The main support for operating systems on this platform are Windows and Linux operating systems, but there is also fully supported Docker containerization and therefore, any development environment that can be created using containers.

The application running on App Service itself is restricted by an App Service plan. An App Service plan defines a set of computing resources available to a single application or an application pool. Therefore, an App Service plan can be viewed as a billing model as well as the feature set that is available to the applications running on the given plan.

An interesting idea an App Service promotes is the concept of deployment slots. Deployment slots can be created for any given App Service and they are used to run an instance of the application. It enables developers to run multiple versions of the application. For example, a developer can decide to release a newer version of the application. The deployment itself is done using a deployment slot while the application is running in production. Once the deployment is done the Azure environment can be reconfigured so that the service redirects users to an instance running in the newer deployment slot.

Azure Cosmos DB

Azure Cosmos DB is a fully managed NoSQL serverless database for app development. The main idea behind Cosmos DB is to build a horizontally scalable and globally distributed database service. The big selling point from Azure for Cosmos DB is low latency accessibility and high availability.

Cosmos DB provides support for different APIs. Based on the data model used in the application, a developer can use a specific API to interact with the Cosmos DB database service. The NoSQL types that Cosmos DB supports are:

- Key-value persistent dictionary
- Column, wide-column, or column-family for the organization of related data into columns
- Document storage that allows persisting JSON objects
- Graph for the storage and navigation of complex relationships

When provisioning a Cosmos DB database the developer has an option to choose what APIs the database supports. This determines what types of NoSQL databases will be created in the background. Cosmos DB supports the creation of:

- MongoDB for document storage
- Cassandra for wide-column storage
- Azure Table for key-value storage
- Gremlin for graph storage

Azure Blob Storage

Azure Blob Storage is a cloud storage service for storing unstructured data. Unstructured data is data that doesn't fit a particular data model or definition, such as text or binary data. This unstructured piece of data is often called a binary large object or blob. The

Azure Blob Storage is designed for storing and serving documents, images and videos from anywhere on the internet.

Blob storage offers three types of resources:

- The storage accounts
- Containers in the storage account
- Blobs in the container

The storage account can be viewed as a unique namespace. It contains all the data objects and exposes them through the Blob Storage API. A container serves as a folder to better organize the blobs inside of the namespace. Azure Storage supports three types of blobs:

- Block blobs to store text and binary data
- Append blobs that are made of block blobs and optimized for append operations
- Page blobs to store virtual hard drive files and mostly used with Azure Virtual Machines

Azure Functions

Azure Functions is a serverless service provided by Microsoft Azure. It is a service that runs the code for you without the need for provisioning infrastructure. The code running under this service is often regarded as a function instead of an application and is usually triggered by an HTTP or timed event. Azure functions are by design stateless, which means once the function finishes running all its data will be deleted together with the runtime.

Azure functions implement an extension called „Durable Functions“. It lets the developers design stateful functions in a serverless environment, and define workflows in code. The Azure Functions environment automatically checkpoints the progress whenever the function awaits and makes sure that the state is not lost if the process restarts or the underlying infrastructure reboots.

The most common use-cases for Azure Functions are:

- Reminders and notifications
- Scheduled tasks and messages
- Data or data streams processing
- Running background backup tasks
- Prototyping and MVPs

2.6 Google Cloud Platform

Google Cloud Platform (GCP) covers the infrastructure and platform sides of the Google Cloud. G-Suite covers the software side where Google provides software like Google Sheets, Google Documents etc. But I will focus on the GCP part of Google Cloud and leave the G-Suite out as it is irrelevant for the contents of this thesis. There are too many services to cover as a subsection in this thesis, so I will focus on the most useful ones to use the GCP for application development.

GCP global architecture offers resources in 24 locations globally. The location consists of regions, and each provides one or more availability zones, which are isolated from a single point of failure. Some resources like HTTP load balancers are global, which means they are not regionally dependent. Other resources such as storage or computation are regional and must be deployed per region.

2.6.1 Google Compute Engine

Google Compute Engine (GCE) is an Infrastructure as a Service offering that allows clients to run workloads on Google's physical hardware. Using virtual machines for application infrastructure poses additional challenges where developers have to manage the underlying infrastructure. While being shielded from hardware components, it falls on the developers to handle instance updates, patches and environmental management. GCE offers the lowest abstraction of all the cloud services offered by the Google Cloud Platform.

GCE offers multiple pre-sets of virtual machines to handle different requirements. They vary in the number of virtual CPUs, memory, and memory types. Some applications can be performing heavy computations on GPUs. These demands are covered by the following types:

- General-purpose machines offer a good balance between the price and computational power. They are often used for databases or testing environments.
- Scale-out types are optimized for tasks that are expected to rapidly scale-out like web servers or microservices. They are based on the new family of virtual machines called TAU VMs.
- Ultra-high memory or Compute-intensive types that offer specialized types of components to handle the respective tasks at hand more efficiently.

Management of GCE's instances is done via a RESTful API, Google SDK command-line interface or the cloud console.

2.6.2 Cloud Run

Google Cloud Run takes the concept of serverless and merges it with containers to provide a seamless alternative for developers. Cloud Run utilizes the portability and flexibility of containerization to deploy and scale applications to meet traffic demands. The service supports auto-scaling options which enable developers to fully utilize this feature without the need for changes in the underlying technology.

Cloud Run focuses on resource management and accessibility benefits while supporting any development environment that fits inside a container. It runs on the open-source Google-backed project Knative [20] to enable portability across platforms. The developers have multiple options when deploying Cloud Run projects. Thanks to the container encapsulation the project is deployable under the Cloud Run service itself as a pay-per-use project or a custom Kubernetes cluster.

Developers can use Cloud Run to deploy anything from small, function-like API endpoints to monolithic web applications as long as those workloads comply with a few basic rules:

- They must listen for requests on the port defined by the PORT environment variable.

- They must be stateless, meaning they cannot rely on a persistent local state.
- They must not perform background activities outside the scope of request handling.

While Cloud Run sets up a subdomain to help access your services, it also offers custom domain support. The developer can power an entire web application within a Cloud Run service without touching any additional infrastructure.

The resources needed to run the instances in the Cloud Run environment are automatically determined by the service based on the load it handles. The auto-scaling feature is the selling point of Cloud Run service as it can the application by running new instances of a provided container and handling the load balancing in the background. It can also scale down to zero instances when there is no workload present to save costs and computing resources. Settings for maximum and minimum instances are present to avoid infinite scaling in case of an error. The minimum quota specifies the number of instances to be kept idle at any time in the system. It prevents the initial delay of starting up the first instances.

2.6.3 Cloud Storage

Google Cloud Storage is a public cloud storage platform for unstructured data sets. Unstructured data is information that is not arranged according to a pre-set data model or schema, and therefore cannot be stored in a traditional way. The common examples of unstructured data are text documents and multimedia.

The service stores the data close to the chosen geological location for faster response times. It provides unified object storage in the cloud for storing live or archival data. The objects stored in Cloud Storage are organized into buckets. A bucket is a container within the cloud architecture that can be assigned to a storage class.

The Google Cloud Platform offers four storage classes to developers:

- Multi-regional storage stores data in data centres across the globe. It is suitable for use-cases where data needed to be accessed frequently. This storage class ensures the replication of data to at least two separate locations, which improves the availability.
- Regional storage class stores data in a single region instead of spreading the data. It works the best when the storage and compute resources are in the same region.
- The nearline storage class is optimized for long-term storage of data with infrequent access.
- Coldline storage class aims at storing data that are accessed very rarely. It is the cheapest option out of all but this storage class comes with a fee for data retrieval.

One of the challenges of properly leveraging storage classes is that the same type of data might require different handling over the lifecycle. For example, let's imagine logging data generated by an application. In the beginning, the data needs to be accessed regularly for debugging and monitoring purposes. Later in the project, the data become less frequently accessed and in the latest stage the data become archival and we want to keep them for compliance reasons, therefore being rarely ever accessed. For this use-case, there is a service that automatically manages storage lifecycle management rules. It is a built-in feature of Cloud Storage that enables developers to define logic rules over data objects stored in buckets.

2.6.4 Spanner

Google Cloud Spanner is a distributed relational database service that runs on Google Cloud. It supports global deployment, SQL semantics and transactional consistency and is horizontally scalable.

Google Cloud Spanner's strengths are in the ability to provide both availability and consistency. These traits are usually contradicting each other, with data designers typically deciding whether to emphasize either availability or consistency. The trade-off has been outlined by the CAP Theorem, which initiated a general move to NoSQL databases for availability and scalability in web and cloud distributed systems. In pursuing both system availability and data consistency, Google Cloud Spanner combines SQL and NoSQL traits.

Google Cloud Spanner as its name indicates is a database that can possibly span over multiple data centres while still keeping the data consistent. It supports distributed SQL queries (as well as query restarts). This is all possible due to the fact that Cloud Spanner utilizes TrueTime, a Google Cloud clock synchronization service that uses a combination of atomic clocks and GPS. An atomic clock is the most accurate type of time measuring device in the world. Cloud Spanner uses this functionality to assign system consistent timestamps to transactions and ensures linearizability.

2.6.5 Firebase

Google Firebase is a Google-backed application development platform. It provides tools for developing web and mobile applications. A developer can use services offered on the Firebase platform as a generic backend for his application directly or utilize them as tools in his own infrastructure. The services are hosted in the cloud environment and are designed with scaling in mind. Services such as analytics, authentication, databases, messaging or file storage are just a few to mention of the tool-set provided by the Google Firebase platform.

The most commonly used tools that the Google Firebase platform offers:

- Firebase Authentication enables developers to quickly and easily build secure systems with enhanced sign-in experience for their users. Firebase itself is not an identity platform as it does not offer functionality like multi-factor or SAML authentication. But it offers complete support for email and password accounts as well as provides easy access to well-known identity providers such as Facebook, Microsoft and others.
- Firebase Realtime Database and Cloud Firestore are cloud-hosted NoSQL databases that enable data to be stored and synced between users in real-time. The data are synced across all clients in real-time and are still available when an app goes offline.
- Firebase Cloud Messaging is a cross-platform messaging service that enables developers to reliably receive and deliver messages on iOS, Android and the web at no cost.
- Firebase offers full support of Google Analytics to collect and present data about user behaviour and enable better decision-making about application performance and marketing strategies.

Firebase is considered to be a Backend as a Service (BaaS) by many. BaaS is conceptually very similar to PaaS. The main difference is that while PaaS focuses on reducing the load of infrastructure management on the developer BaaS offers concrete out of the box

tools to aid in the application development while generalizing the aspects PaaS helps developers to solve. Firebase services are exposed to the developers either directly by calling the respective RESTful APIs or by using a language-specific SDK.

Both Realtime Database and Cloud Firestore offer realtime data updates through the Firebase SDK. It is done by registering a webhook that triggers a notification when certain data are updated. It allows an application to present an up to date state to users without the need to regularly poll user data.

Chapter 3

Web Application Architecture

When building any piece of software, whether it is a small or big project, it is always a good practice to have a plan. Without it, we can forget to include a feature or find out too late that we underestimated the complexity. What it contains is usually based on the scope and complexity of the problem at hand.

The application architecture and software design patterns help developers in the first stages of designing an application. They are condensed knowledge from many previous projects built by clever and diligent people. Each architecture comes with an ideology and design patterns enforcing it.

Many concerns go into choosing the architecture for your application, like whether it is an application that needs to interact with users or just a background service. To have the ability to rapidly provide new functionality and services to consumers and fast feature development. The need for a database access layer and much more.

3.1 Monolithic Architecture

It is hard to upsell the advantages of a microservices architecture without first introducing its predecessor, the monolithic architecture. A monolithic architecture is a traditional unified model for designing various software products. It is successfully used in smaller projects as it has a straightforward structure. But with the rise of cloud computing, it lacks the ability to scale with demand.

A monolithic application is self-contained with interconnected and dependant components by design. More often than not, it consists of a single code base and is deployable as a unit. The architecture is decomposable into individual layers, namely Presentation Layer, Logic Layer and Data Layer. This architecture pattern is also known as a three-tier architecture, but we will refer to it as a monolithic in this thesis.

A simple yet perfect example of a monolithic application would be a personal event planner accessible online. Let's say that the use-case of this application will require accessibility from various devices. The application can have single or multiple users and store the events permanently. I will go more into detail on each tier in the separate section. And then showcase the strengths and shortcomings of monolithic applications.

Presentation Tier

The presentation tier is an application layer or a separate application that conveys information from an application state to the user and takes user inputs to alter the application state.

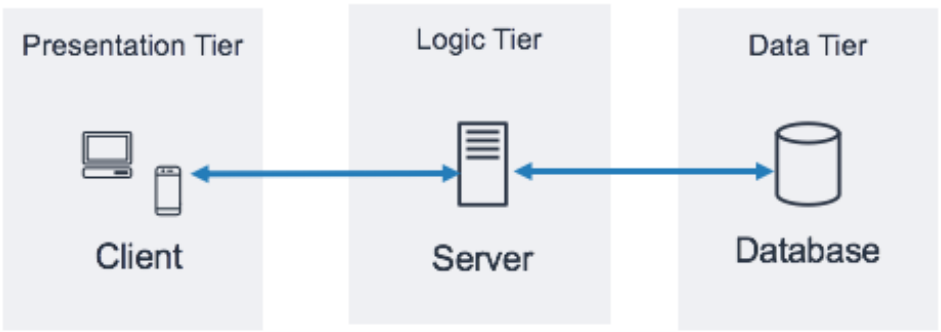

Figure 3.1: Three tier architecture

The sole purpose of this tier is to present and collect information from the user. Nowadays, web applications function as a presentation layer for most modern applications. But we can consider any user interface connected with some business logic to be a presentation tier of an application.

We could build a web application as a web-based user interface for our event planner application. Then the running of an application inside any web browser supported by selected technology is possible. This tier can, at least to some extent, validate the input user data before sending them to the application core.

Logic Tier

Also known as an Application Tier handles the logic and computations of an application designed as monolithic. Information collected in the presentation tier is sent here and processed using business logic, a specific set of functions realizing processes, transactions and queries.

Here you can imagine everything that goes into building a proper application back-end in any language of your liking. All the logic creating, deleting and updating events will be placed in the logic tier in separate modules. The application uses a lightweight communication protocol to connect with the presentation tier.

Data Tier

The final tier of a three-tier architecture is responsible for data persistence. This tier communicates exquisitely with the logic tier via a query system. Typically a relational database management system such as PostgreSQL or MySQL stores the data. But in some cases, more modern technologies such as NoSQL databases are more efficient.

The data tier for the event planner application consists of a database scheme and a script for table creation. It has to store our events with information like date, time, place and potentially some notes.

Strengths and Shortcomings of Monolithic Architecture

I have already talked a bit about the usefulness of developing monolithic applications. But with the technological trends advancing, a set of flaws emerged. Both strengths and shortcomings of monoliths depend on the problem definition.

The strengths of monolithic architecture lie in the straightforward approach that developers take when designing a monolith. And these strengths shine the most at the beginning of a project. While the codebase is clean and small and functional complexity is low monolithic applications are easy to test as all the dependencies are always present. The deployment and scaling of a small monolithic application are relatively simple. The two most common approaches to scaling a monolithic application are adding a more powerful machine or running more of them with a load balancer. This type of scaling is also called horizontal scaling. When it comes to production, developers look for options on handling cross-cutting concerns like logging, monitoring and configuration. It is easy to solve in monolithic applications by introducing specialized modules.

The shortcomings of monolithic architecture start showing once our monolith grows in complexity and userbase. Monolith requires a long-term commitment to a specific technology stack. But once the application becomes large, changing a single module becomes challenging. Performing an update of a monolithic application includes completely redeploying all its instances, which can take a lot of time and effort. It becomes a problem once we adopt the modern approach to continuous deployment. We can not forget about the human factor in every project, and large monolithic applications pose a significant challenge to onboarding new developers. Hence, reducing the agility of large monolithic projects.

3.2 Microservice Architecture

As the application grows, some aspects of the project get complicated. These complications can then take their toll on the user experience or the final cost of the service. Many internet giants like Amazon[14], Netflix[15] and Instagram[13] are pioneering the microservice architecture in their products. And its success can be seen in the continuous delivery and quality of service. Many companies are following this trend and implementing their products using microservice architecture[23].

The microservice architecture is a modern adaptation of the Service-Oriented Architecture, shortly SOA. SOA tried to break down monolithic applications into more agile components, with communication realised through a lightweight communication protocol. It still only used a single data storage layer. Therefore, SOA is still monolithic due to having a single database schema. This approach is challenging when caching over a vast amount of data with a broad userbase.

What is a Microservice?

Building an application using microservice architecture requires a breakdown of business logic into individual services. Each of these services offers a subset of the overall business logic. Individual microservices have isolated codebase and use whatever technology independently.

Furthermore, individual services should have a separate database layer containing only the information relevant to the given microservice. It contributes to why microservices are great at solving scalability and deployment issues of monolithic applications. This

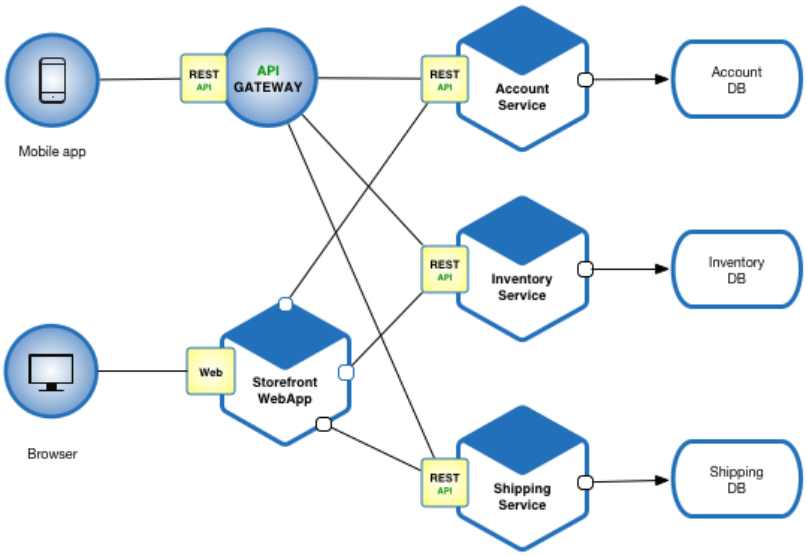

Figure 3.2: Microservice architecture[22]

approach to data separation is also called Domain-Driven Design. The database layer is exposed externally through an application interface. It should only be accessible this way.

A microservice can then be easily deployed, scaled and tested independently. It follows a single responsibility principle and fulfils only one function. It allows teams to choose a development language per service and treat it as a separate project. The services can scale independently without scaling the whole application because a single feature faces a higher load.

If I take a look back at the event planner example, individual microservices could be

- user management service,
- event service,
- calendar service and a
- service that is managing holidays based on user location.

Domain-Driven Design

Domain-Driven Design centres the whole software development approach around understanding processes a business domain needs. The name originates from a book by Eric Evans that carries the same name[7]. A domain-driven design process is not mechanical and does not guarantee the „right“ result. It promotes thinking and provides tools to communicate the design to others in the process.

This approach has two iterative phases. The first phase is a strategic domain-driven design, and it helps keep the architecture focused on the business domain structure. The central pattern in this phase is creating a set of bounded contexts. This pattern divides a large domain into smaller subdomains based on the data decoupling. Eric Evans, the originator of domain-driven design, portrays this process as the creation of a ubiquitous language[9]. And this ubiquity should be understood as present across all stages of software development.

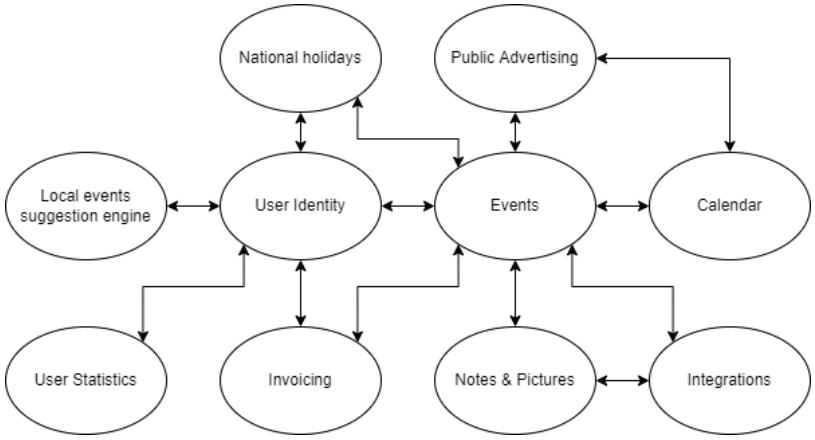

Figure 3.3: Event planner domain analysis

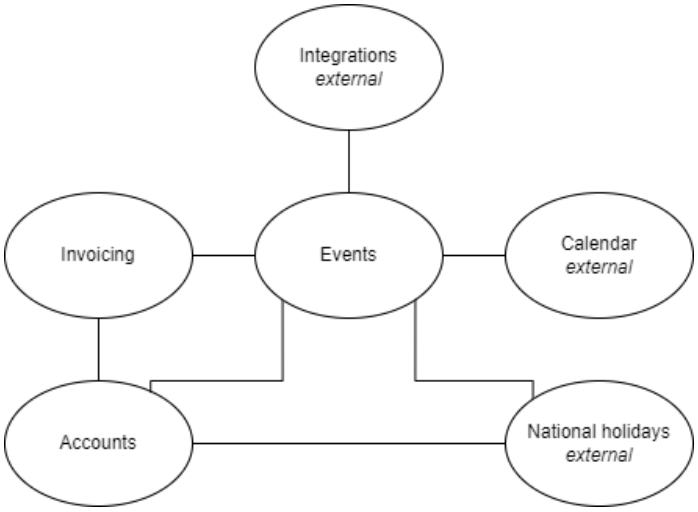

Figure 3.4: Event planner bounded concepts

The second phase is a tactical domain-driven design, which provides a set of design patterns to create the domain model. It works within a single bounded context and applies tactical design patterns. When using domain-driven design for designing microservices, the entity and aggregate patterns are what we need.

An entity is a unique persistent object with an identity that can span multiple bounded contexts. It has an identifier that enables us to retrieve it from a database. The purpose of an aggregate is to model transactional invariants[10]. It consists of one or multiple entities. There are some challenges this approach poses, and I will address them later on.

Decentralization

Centralisation in microservice architecture almost does not exist. Microservices use lightweight communication protocols to communicate over the internet or message brokers to communicate with each other. This separation of concerns helps drive the decision-making closer to the problem. It enables the developers to use more fine-grained technology stacks to solve specific problems and use tools better suited for them.

Strengths and Shortcomings of Microservice Architecture

Microservices solve some of the concerns I have outlined in the monolithic architecture. These concerns are mostly related to the modern state of application development, where applications have to be accessible anytime, and many companies focus on UX.

Component independence helps us to tackle scaling and unit testing. While also improving readability. It enables individual developer teams to choose and implement the technologies they like and want. Individual components are easier to understand compared to a huge monolithic application with fewer dependencies floating around and much cleaner infrastructure.

But not everything is simplified by implementing a microservice architecture. It brings added complexity when looking at the system as a whole. While microservices make the testing and understanding of individual components simpler. The resulting infrastructure is intricate and requires a deeper understanding of business processes. Dealing with cross-cutting concerns like logging, monitoring and configuration become challenging. Testing the whole system once there are many microservices becomes difficult.

3.3 Distributed Transactions

Transactions are an essential part of applications. Without them, it would be impossible to maintain data consistency. Transactions must be atomic, consistent, isolated, and durable (ACID).

- Atomicity means that each statement in a transaction (read, write, update or delete data) is treated as a single unit. Either the entire statement is executed, or none of it is executed.
- Consistency ensures that transactions only make changes to tables in predefined, predictable ways.
- Isolation happens when multiple users are reading and writing from the same table all at once, isolation of their transactions ensures that the concurrent transactions do not interfere with or affect one another.
- Durability ensures that changes to your data made by successfully executed transactions will be saved, even in the event of system failure.

Transactions within a single service are ACID, but cross-service data consistency requires a cross-service transaction management strategy. A database-per-microservice model provides many benefits for microservices architectures. Encapsulating domain data lets each service use its best datastore type and schema, scale its data store as necessary, and be insulated from other services failures. However, ensuring data consistency across service-specific databases poses challenges.

When we start sharing our data through distributed systems, we can no longer guarantee data consistency, availability and partition tolerance. Computer scientist Eric Brewer put forward the CAP theorem. It states that a distributed system can not guarantee all three aspects (consistency, availability and partition tolerance) at all times.

- A consistent system always returns the same information no matter what node we query.

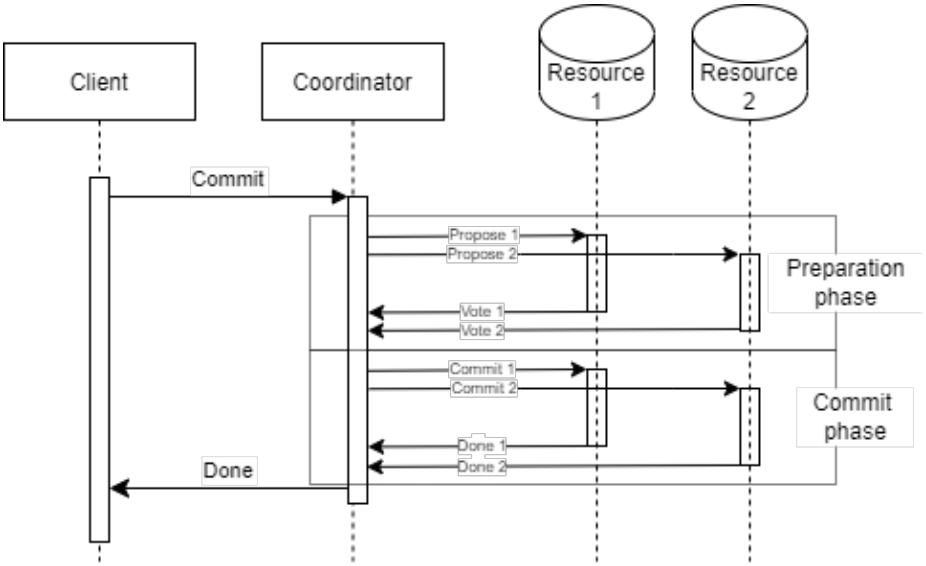

Figure 3.5: Sequential diagram for successful 2PC

- An available system gives every read or write request an appropriate successful response.
- And partition tolerance refers to the ability of a system to function normally in case of a network failure.

Two-Phase Commit

The Two-Phase Commit (2PC) protocol is an atomic protocol for transaction coordination. It consists of two main components, the coordinator and the nodes.

The two phases of 2PC are the preparation phase and the commit phase. In the preparation phase, the nodes participating in the process acquire resources needed for the second phase. It includes placing locks on resources across the system. Once a node has all the necessary resources, it confirms its commitment to the coordinator. Once all the nodes confirm, the coordinator proceeds to initialize a distributed commit. If any node is unable to promise a commitment to the transaction, the coordinator initializes a rollback for the transaction.

The problems with 2PC are:

- There is a single point of failure in the form of a centralized coordinator.
- The throughput of the system is dependent on the slowest node.
- In a complicated system, 2PC locks all the resources it is working with, and they become unavailable until the process finishes.
- NoSQL databases do not support the 2PC protocol.

The Sagas Pattern

The sagas were proposed as a solution to a Long-Lived Transaction (LLT) in a single relational database in the original paper[11]. A single LLT like a financial aggregation

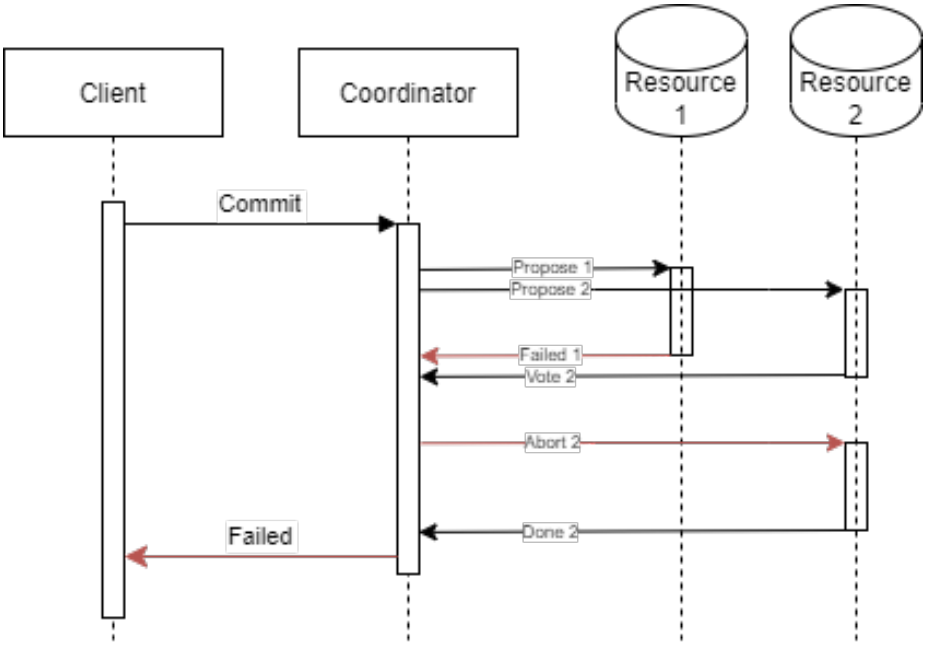

Figure 3.6: Sequential diagram for 2PC with failure

would bottleneck a system for an extended period. The paper proposes a breakdown of LLTs into a sequence of independent smaller transactions that can be interleaved. These transactions can support ACID on a single database.

The sequence of transactions is either completed successfully, and the operation is considered finished or compensating transactions are run to amend the partial execution. A compensating transaction semantically undoes a transaction but does not necessarily return the system to the original state. Some transactions may be irreversible, and therefore a compensating transaction performs a set of steps to counteract the previous actions.

There are two common saga implementation approaches, choreography and orchestration. Each approach has its own set of challenges and technologies to coordinate the transaction flow.

The choreography sagas is a way to coordinate an exchange of events between nodes without a centralized point of control. It promotes further decentralization of the system and does not introduce a single point of failure as with the two-phase commit. Further, it does not require an additional service for operation.

But the drawbacks of implementing choreography sagas are that each service has to implement the routing logic for requests. It can be hard to interpret and can introduce cyclic dependencies between nodes. Also, this approach introduces tight coupling between dependent services.

The orchestration-based sagas introduce a centralized service that orchestrates the nodes. This coordinator service executes saga requests, stores and interprets the states of each task in the saga log, and handles failure recovery with compensating transactions. It introduces a centralized element into the system, but it is stateless and can be restarted at any point without consequences.

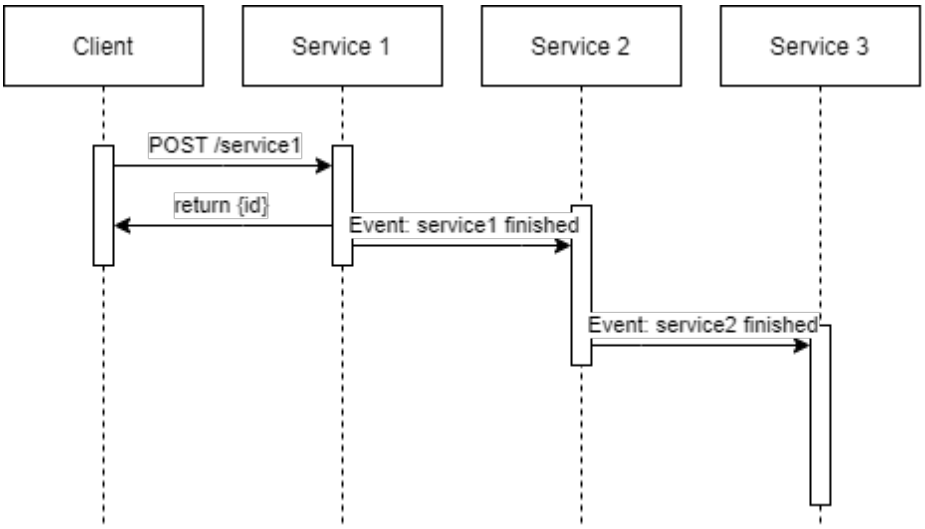

Figure 3.7: Sequential diagram for Choreography-based saga

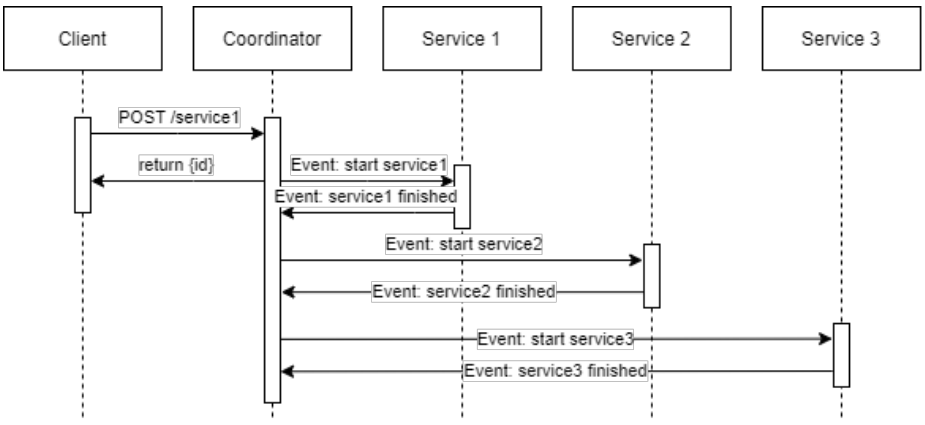

Figure 3.8: Sequential diagram for Orchestration-based saga

Chapter 4

Cloud Native Tech Stack

4.1 Gateway to the Cloud

When developing applications in the cloud, we have to figure out a way to reach them so that our clients or other applications can use and communicate with them. Application programming interface (API) is the most common pattern used for enabling communication flows. Modern APIs usually communicate through HTTP and expose application business logic and underlying data layer, while HTTPS provides an out-of-the-box security layer for the requests. It is a perfect package able to cover a wide range of scenarios.

We can break the application programming interface (API) can down into two separate elements:

- Procedures - are the functions the underlying software provides
- Protocols - the formats used to communicate the data between applications

The separation helps with the design process of endpoints and data structures. Frequent data formats used for APIs are CSV, XML and JSON. The JSON data format is the most common one used in modern applications.

The concept of APIs is loosely defined in the scope and structure of services provided. First commercial APIs emerged on the break of the millennium by tech companies like Salesforce and eBay and revolutionized the commercial use of the web [reference]. Time has shown that consistency for this type of service is necessary. The answer for this need is architectural styles like REST and GraphQL that propose a set of rules to follow when designing an API.

APIs are a good start when developing a set of intercommunicating services, but once the network grows, the demands on the infrastructure get overwhelming. When many clients are trying to reach different services, it is easy to imagine how this complexity scales with adding a client or a service. And a client can be simply another service in the system as well. This issue does not only pose a challenge from a networking perspective but also a development one. It is hard to keep track of all the messages going through a system like this. It is time to consider asynchronous communication, where services publish and react to events reliably delivered by a pub-sub messaging system.

Direct client-to-microservice communication means exposing the APIs for each microservice. However, the granularity of microservice APIs can be different to the client's demands. More outstanding issues are API changes, authentication, monitoring, throttling and other aspects of any API that would have to be developed independently for each microservice,

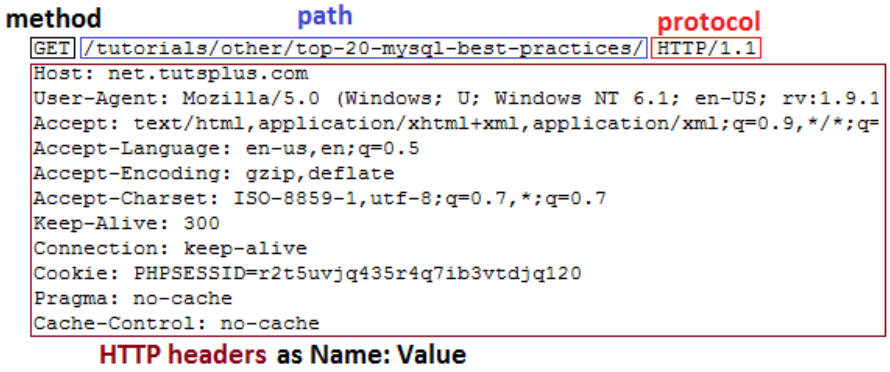

Figure 4.1: HTTP Request Header Example[12]

potentially in different programming languages. To mitigate the outlined issues, I will introduce an API Gateway pattern and its challenges.

4.2 HTTP Messaging

The Hypertext Transfer Protocol (HTTP) message is the most common carrier of information in client-server communication. It uses the HTTP protocol to carry the data over the internet to its destination. The message is either a request or a response, depending on the flow of the communication. HTTP messages are composed of textual information encoded in ASCII. HTTP is a protocol and therefore is language and framework agnostic.

HTTP is stateless and every request is fully independent and can be compared to SQL transactions in a way. Therefore, in case we need information about a state we need to handle them externally. For example when a user is browsing through our webpage, we can use tools contained in the web browser like IndexedDB Storage and Cookies to help us create a context. HTTP allows us to send various information inside its headers and body. We can use this to our advantage and create our own context.

HTTP itself is not secured in any way and all the information inside the messages are clear text. This was acommon problem in many cases and therefore Hyper Text Transfer Protocol Secure was created. Where data sent are encrypted by Secure Socket Layer (SSL) or (Transport Layer Security) TLS. All data that are sent over the internet should be secured at least by using HTTPS.

Common HTTP methods:

- GET - Retrieves data from the server
- POST - Submits (adds) data to the server
- PUT - Updates data that are already on the server
- DELETE - Deletes data from the server

With each request and response message comes a header and a body. The body typically contains the requested data or submitted data. Both of the headers contain the protocol used to send the message. The request header 4.1 then contains the HTTP method used and the path while the response header 4.2 comes with an HTTP status code. The rest of the header is a list of key-value pairs.

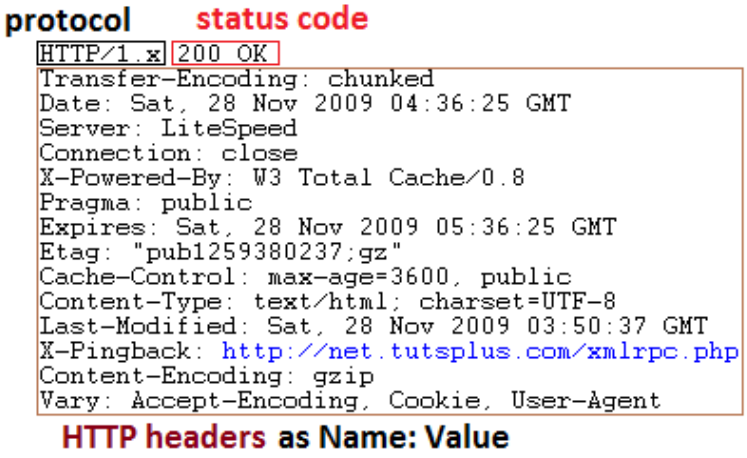

Figure 4.2: HTTP Response Header Example[12]

HTTP status code groups:

- 1xx - Informational
- 2xx - Success (200 - OK, 201 - OK created)
- 3xx - Redirect (301 - Moved to new URL)
- 4xx - Client Error (400 - Bad request, 401 - Unauthorized, 404 - Not found)
- 5xx - Server Error (500 Internal server error)

4.3 REST-ful API

REST is an acronym for Representational State Transfer, and REST-ful generally means that the service implements REST methodology. It is not a protocol or a standard but rather a set of architectural constraints.

When a client requests information through the REST API, it transfers the representation of the state of the resource to him. The service can return the state representation in different formats, and it depends on the implementation of the service, whether it supports multiple or not. The most common resource formats are JSON and XML.

For an API to be considered REST-ful, it needs to implement the following:

- A client-server architecture of clients, servers, and resources, with requests managed through HTTP.
- Stateless, client-server communication where all requests are separate and unconnected, and the server does not store information about a client between the GET requests.
- Data are cacheable to streamline the communication between the server and a client.
- A uniform interface between components to transfer the data in a standardised form.
- REST allows for an architecture composed of multiple layers of servers.

- More often than not a REST API will return a static representation of an object as a JSON or XML. However, when necessary, servers can send executable code to the client.

It can be challenging to adhere to all these demands for implementing a full RESTful API. Thankfully, an API does not need to implement everything at all costs, as these architectural constraints are just a guideline. However, other aspects can cause problems when designing a REST API:

- The endpoint paths should be consistent, following the web standard.
- Endpoint URLs should not be invalidated when used internally or in other applications, introducing an API versioning.
- The amount of data can increase in time and cause prolonged response times.
- The security aspects including HTTPS, URL validation, fraud prevention, request logging and failure investigation.
- Choosing and deploying an authentication model (basic authentication, API keys, JWT, OAuth 2.0).
- API testing and automated API testing.
- Defining error codes and messages.

4.4 GraphQL API

GraphQL API is a powerful alternative to the REST API. GraphQL is a query language for the API and a server-side runtime to process user define queries over the data. It is a newer API standard than REST, and it enables declarative data fetching, where a client can specify exactly the data it needs from an API. Compared to REST, GraphQL only exposes a single API endpoint and responds with precisely the data that a client has requested.

GraphQL Schema

GraphQL API uses a schema to describe all the data accessible via the API to the client through the service. An API designer creates a GraphQL schema, defines the object types the schema is made up of and defines the kinds of objects requestable and their fields. GraphQL has its type system that is used to define the schema of API. The syntax for writing schemas is The Schema Definition Language (SDL). SDL allows us to define custom types composed of elementary types like int or string, but also introduces relationships between these types.

GraphQL schema is one of the most important concepts when working with GraphQL API

- the schema defines the capabilities of the API
- represents a contract between a client and the server
- is a collection of GraphQL types with special root types

The root types of a schema define the entry points for the GraphQL API. The root types are Query, Mutation and Subscription.

As an example, I will use a simple system where users can create posts and retrieve them. The schema for this system would have to cover two custom entities User and Post. The relationship between them is that a User can have any number of posts (one to N) and a Post has to be posted by a single User (one to one).

```
type User {                          type Post {
    id: ID!                              title: String!
    name: String!                        content: String!
    age: Int!                            author: Person!
    posts: [Post!]!                  }
}
```

Listing 4.1: Example of GraphQL entities schema

Then the root types could look like this.

```
type Query {
    allUsers(last: Int): [User!]!
}

type Mutation {
    createUser(name: String!, age: String!): Person!
}

type Subscription {
    newPost: Post!
}
```

Listing 4.2: Example of GraphQL root types schema

Functional capacity of this example is limited and does not have full CRUD (Create, Read, Update, Delete) support. But I believe that as an example it displays the important details of GraphQL schema.

While REST API 4.3 follows the principles of HTTP Message 4.2 when requesting or submitting data. GraphQL uses Queries to request the data and mutations to submit or update them.

GraphQL Query

GraphQL API typically exposes only a single endpoint for data access contrary to REST API with many endpoints. It works because the structure of data returned by the GraphQL endpoint is not fixed as with the REST endpoints. The data are flexible based on the client's needs. It also means that the server requires more information from the client to understand and cover its data needs. This information is called a query.

As an example of a GraphQL query in our system of Users and Posts, let's retrieve all the user records from the database and the posts they posted. The first query returns an array of users with only the „name“ field included. The second query demonstrates the strength of GraphQL, requesting nested objects through relationships. Furthermore, the GraphQL query enables a client to only request the subset of all the data by adding filter parameters.

```
{                                    {
    allUsers {                           allUsers {
        name                                 name
```

```
        }                                   posts {
    }                                           title
                                            }
                                        }
                                    }
```

Listing 4.3: Example of GraphQL Query schema

GraphQL Mutation

Only requesting information from the application would not cover all the needs functional solution needs. To create, update and delete information on the application server GraphQL uses a mutation system. The GraphQL mutations always have to start with the „mutation“ keyword. To submit data mutations use the aforementioned field arguments. A niche feature that GraphQL mutations provide is that the client can submit a query request together with the mutation to query certain aspects of the object that are generated by the server upon creation such as object identifier.

```
    mutation {
        createUser(name: "Peter", age: 25) {
            id
        }
    }
```

Listing 4.4: Example of GraphQL Mutation schema

GraphQL Subscription

Subscriptions are a GraphQL feature that allows the server to send data to the client when a specific event occurs on the backend. For this functionality to work a client needs to maintain a long-lived connection with the server. The client initially opens the long-lived connection by sending a subscription query which specifies what events is the client interested in. Every time this event the client is subscribed to occurs, the server uses the connection to push the notification about this change to the client.

GraphQL subscription in the example system could enable a client to subscribe to an event of post creation. Then the client would get notified once a post is posted and show it to the user in form of a notification.

```
    subscription {
        newPost {
            title
        }
    }
```

Listing 4.5: Example of GraphQL Subscription schema

4.5 Pub-Sub Messaging

In sections 4.2 and 4.3 I have outlined how REST and GraphQL API can be applied to client-server communication. However, this type of communication is not the most efficient at conveying information in a distributed system. Both REST and GraphQL are forms of synchronous communication (with exception of GraphQL subscriptions). Over-reliance on

the use of synchronous patterns has negative consequences that apply to the communication between microservices and in some cases are at odds with the principles of microservice architecture.

- Tight coupling - Some level of coupling between services will always be present (specifically around the data structures) but regular API services assume that the message will only be delivered to a single client. For each new component in the structure, there has to be a new message to a new endpoint. This way a simple microservice will become an orchestrator and this breaks the „single purpose“ attribute of microservices.
- Blocking - When invoking a synchronous service, the invoking application thread is blocked waiting for a response. This behaviour might seem innocent but becomes an issue once scaling is considered.
- Error Handling - The underlying protocol for REST and GraphQL is HTTP and it was designed for the web. It does not offer a good retry logic in case of failure. This responsibility lies on the client itself and is not a clean-cut process. Depends on the type of error and data the client is requesting therefore binding it even more tightly to the server.

The issues are addressed in GraphQL subscriptions, yet they are still client-server oriented in their use-case. The server tries to push each message based on the subscription and this functionality remotely resembles pub-sub architecture.

Basics of Pub-Sub Messaging

Any pub-sub model consists of the four core components:

- The topic is an intermediary channel that maintains a list of subscribers to relay messages that are received from publishers. Topics allow pub-sub messages to be broadcasted asynchronously across multiple sections of the applications.
- A message in the pub-sub model can be any serialized piece of information sent to a topic by a publisher.
- The publisher is the application that owns or creates data subscribers are interested in getting. A publisher does not know anything about subscribers.
- A subscriber is an application that registers itself with the desire to receive the messages of a specific topic with no knowledge of where these messages originate.

This separation of concerns makes it possible to create event-driven services without the constant need for querying a message queue for updates. It also aids developers in the creation of decoupled services using the same data that can be provided to multiple subscribers at the same time.

Wins and Losses of Pub-Sub Messaging

Pub-sub systems are robust messaging services that bring solutions to problems in distributed systems like microservices. A well designed pub-sub messaging enriches the system by introducing:

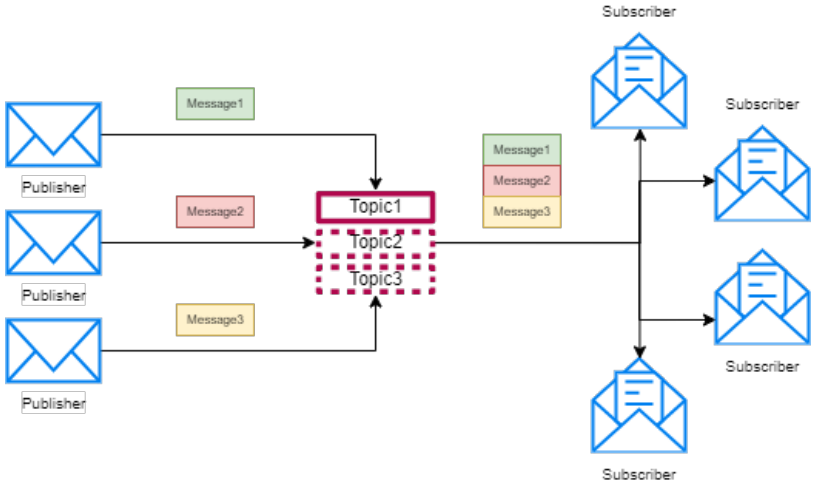

Figure 4.3: Pub-sub model structure

- Loose coupling between system components by decoupling the communication logic from business logic.
- A better view of the system-wide Workflow.
- Enables faster integration as it is language-agnostic, which allows disparate components of a system to be integrated faster.
- Promotes scalability by not allowing the recipients to talk back to the senders.

On the other hand, every design pattern has limitations and trade-offs and pub-sub is not an exception. It is not a silver bullet when it comes to communication between services and it is for these reasons:

- Simpler systems that are unlikely to scale up do not benefit as much from the pub-sub messaging. On the contrary, it can introduce unwanted complexity to the system.
- It is not suitable for media streaming systems. Media streaming systems have nuanced requirements and pub-sub messaging can not provide a steady connection to the client.
- When dealing with periodic tasks it is important to keep in mind that pub-sub messaging is asynchronous, therefore not suitable for systems that run periodic tasks in the background.

4.6 API Gateway

When designing a microservice architecture one of the problems we need to deal with is how to make the services available and accessible from the internet. The number of services varies based on the purpose of the solution and certain technologies can pose challenges when not managed with abstractions. In this section, I will outline how API Gateway can solve the availability issues together with authentication and other API specific aspects.

The concept of an API Gateway can be viewed from different perspectives. One is the perspective of a client, where the API Gateway serves as a single-entry point for the

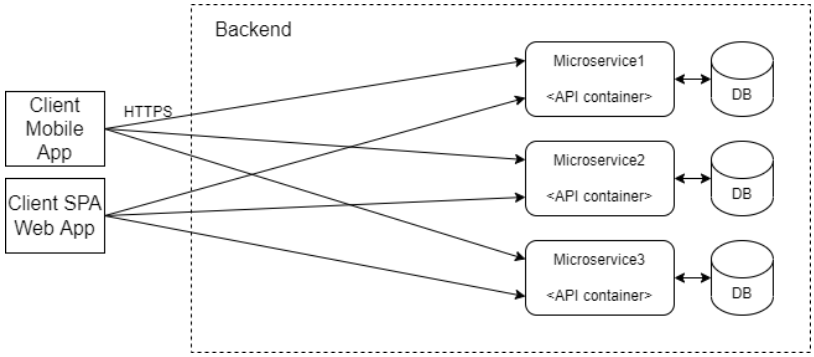

Figure 4.4: Simplified microservice-based architecture without an API Gateway

client application. The developers view the API Gateway as a reverse proxy that routes the traffic from the clients to services. Therefore, the API Gateway sits between the client applications and the microservices.

The API Gateway is located between the client application and the backend microservices. It works as a reverse proxy, routing the requests from the clients to the services and returning the responses back to the clients. This routing is often done based on an OpenAPI specification file that is provided to the gateway and serves as a configuration and documentation at the same time. The API Gateway is often used to accommodate cross-cutting concerns like authentication, SSL, throttling and cache.

Microservice-based architecture without an API Gateway

APIs over the internet are nothing new. A distributed system design with microservice architecture contains at least several services that a client application needs to communicate with.

Designing such a system without an API Gateway means that the client application needs to access the backend services directly over HTTP. This approach has several obvious problems for both the client and the developer.

It's often impractical for a client to perform API composition over the internet. The granularity of the service APIs provided by microservices is often different from what a client needs. The APIs are fine-grained as the microservice architecture enforces single-purpose services and this forces a client to interact with multiple services to finish a single request. Another problem for the client is when the underlying service infrastructure changes. For example, when the developers split a single service into two due to the design changes. This forces the client to reimplement change how they query the information from the backend and is generally considered a bad practice.

On the other hand, making a distributed system of services without an API Gateway creates an additional load on the development. The developer has to be concerned with the API authentication and user authorization on the service level. This approach violates the DRY principle as well as breaks the single-purpose rule on the service itself. Additional challenges might be caused by specifics of a service implementation such that the service can be using a different type of communication than over an HTTP protocol.

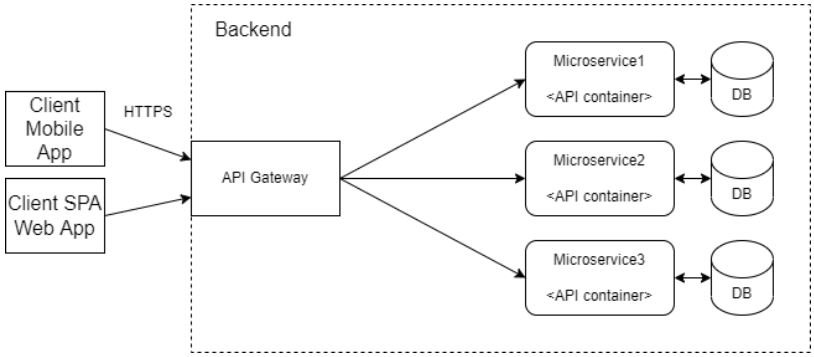

Figure 4.5: API Gateway in a simplified microservice-based architecture

Microservice-based architecture with an API Gateway

Usually, an API Gateway is used as a single-entry point to the set of backend services. The API Gateway then serves as a reverse proxy service for the client application. It also reduces the number of messages the client needs to send to get all the information it needs. As a result of using a single-entry point, the client is shielded from infrastructure changes.

One of the key functions that API Gateway handles is request routing. The gateway itself does not process the request in-depth but forwards them to the respective services and then aggregates the result into a single response. The request routing itself can be a simple one-to-one mapping of endpoints or a more complicated composition.

There might be several client applications running on different platforms connected to a single API Gateway. Supporting multiple platforms such as desktop, mobile or web the requirements on the gateway increase as well. Supporting platforms can result in the need of adding more business logic to the gateway and therefore increases complexity. This can escalate into creating a single point of failure in the architecture.

Backends For Frontends

Rather than provide a one-size-fits-all style API, the API gateway can expose a different API for each client. The requirements of mobile, desktop and web applications can differ significantly. This can cause an API Gateway to become bloated with different calls and data structures which goes against the single-purpose per service principle.

The Backend For Frontend architecture of API Gateways addresses the separation of concerns between different types of application platforms and serves the data an application needs. The front-end developer can then focus on the gateway dedicated to supporting data for a specific platform.

4.7 OpenAPI Specification

An API is just a set of protocols that allow different applications to communicate with each other. It can be imagined as a data channel between a client application and a server. The client generates a request and sends it over the data channel to the server, and the server sends back a response. I have already explained the details of how this process works in section 4.2. But how does the client know what protocols to use and what endpoints to call? How does he know what to expect in the response? In the previous sections, the API

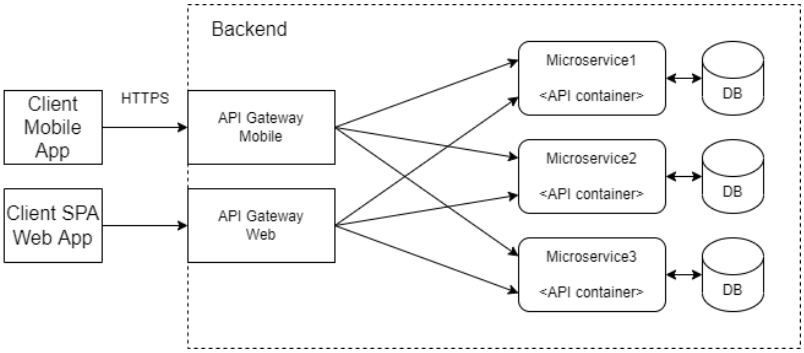

Figure 4.6: API Gateway used as Backend For Frontend

was just a black box or the user that just worked. In this section, I will explore the tools needed to properly document and maintain an API.

In API design, the specification is meant to standardize the exchange of data between the client and the server. Developers rely on the specification to understand how exactly the API should behave. There are many types of API specifications that developers can use to describe their APIs. The description formats that were used in the past are Service Object Access Protocol (SOAP) and Web API Description Language (WADL). Nowadays OpenAPI Specification has become the „industry standard“ for describing APIs.

The noteworthy characteristics of OpenAPI Specification that distinguish it from other specifications are:

- OpenAPI Specification offers a standardized and language-agnostic interface for defining APIs
- It is in both machine and human readable format
- It conveys the capabilities of the underlying service in a comprehensible way to both human and machine consumers without direct access to the source code, network traffic or other documentation

OpenAPI Root Document

The OpenAPI specification comes in a form of a text document called a root document. This file is usually in either JSON or YAML format and is commonly called *openapi.json* or *openapi.yaml* respectively. The official principles of these document formats apply to the specification definition.

The full OpenAPI specification file structure is too complex to go through and is out of the scope of this thesis. Therefore, I will outline what could be considered a minimal document structure containing a set of must-have fields. The document basically defines a top-level root Element that is called OpenAPI Object [18] and then further defines fields such as `openapi`, `info`, `paths` and `components` are required.

- `openapi` - indicates the version of OpenAPI specification this document is using, similar to using a `$schema` field in JSON schema definition.
- `info` - provides general information about the API like title, version and description.

- `paths` - this field describes all the endpoints an API has, including their parameters and server responses.
- `components` - often the API definitions share some common parameters or return the same structure. Components are used to avoid code duplication.

The OpenAPI document defined in YAML could look something like the following:

This part of an OpenAPI document defines the version of the OpenAPI specification used. Sets the title, version and description. There is nothing tricky or complicated as it is just the header of the document.

```
openapi: 3.1.0
info:
  title: Portal API definition
  description: |
    This API allows application developers to use the developed
    toolset to handle common functionality across applications
    to enhance the functionality and speed up development.
  version: 1.0.0
```

Listing 4.6: Example of OpenAPI specification header

The next part is responsible for endpoint definitions. You start by defining the path item first by specifying the endpoint location. Each of the path items defined here can contain operations that are available on this object such as HTTP methods for example. To display how this all looks I will define the `/users` endpoint that implements the GET HTTP method and on success returns a list of users.

```
paths:
  /users:
    get:
      summary: Get the user object
        responses:
          200:
            description: OK
              content:
                application/json:
                  schema:
                    type: array
                    minItems: 1
                    maxItems: 10
                    items:
                      type: object
                      properties:
                        userId:
                          type: number
                        userName:
                          type: string
                        userRole:
                          type: string
                          enum: ["ADMIN", "USER", "DEVELOPER"]
```

Listing 4.7: Example of OpenAPI endpoint specification

I believe it is clear how to define a simple API endpoint using OpenAPI specification. Now if we wanted to query for a specific user using the „userId“ URL parameter we would have to copy and paste the definition of the user resource. This is where OpenAPI components come to assist.

```
components:
  User:
    title: User
    type: object
    properties:
      userId:
        type: number
      userName:
        type: string
      userRole:
        type: string
        enum: ["ADMIN", "USER", "DEVELOPER"]
```

Listing 4.8: Example of OpenAPI component

And last but not least let us define a detailed view of a user and demonstrate how to use the User component. We have to define an additional field named parameters that dictates what parameters are supported by the endpoint, how are they supposed to be provided and whether they are mandatory.

```
paths:
  /users/{id}:
    get:
      summary: Get the user object
      parameters:
      - name: id
        in: path
        required: true
        responses:
          200:
            description: OK
              content:
                application/json:
                  schema:
                  $ref: '#/components/schemas/User'
```

Listing 4.9: Example of using OpenAPI components

The Benefits and Concerns of OpenAPI Specification

Apart from the API documentation, the OpenAPI specification comes with several tools for accelerated development. Tools such as:

- Auto generators that take the OpenAPI specification and turn it into the code or vice versa.
- Documentation tools to generate HTML pages out of OpenAPI specification.
- Mock servers that take the description document as input and then handle the routing of incoming HTTP traffic or generate example responses.

OpenAPI specification promotes a design-first approach where an API designer defines the endpoints and data structure ahead of the implementation. This allows for issues to be spotted ahead of time and avoided, therefore saving time during the implementation process. Later in the development process, the OpenAPI specification serves multiple purposes. At

first, it can be used to auto-generate resource classes and API interfaces. It also increases the chance of a more stable implementation. Once the API is implemented the OpenAPI specification serves as a reliable source of truth for clients and testing in form of standardized documentation.

On the other hand, OpenAPI has a learning curve for any new developer who has not interacted with the standard.

4.8 Technology Stack for Application Development in Cloud

Developing cloud-based applications has become almost a silver bullet in modern application development. Cloud-based applications are not much different from their regular counterpart, the difference being that the cloud-based applications consume or utilize some cloud service. Developers working on mobile and web applications have adopted cloud technologies into the development methodology as well as the business logic. With a reliable internet connection and high internet speeds, the application development has shifted from building monolithic systems that handle everything to building more distributed systems with API interfaces and relying on communication and data sharing.

Application development in the cloud era has shifted from the previously enclosed system with infrequent updates to the modern approach. The modern approach enables developers to deliver new features and updates to their applications on daily basis. Cloud services also play a major role in delivering much needed computational power and infrastructure to enable even the smallest of development teams. The development teams that choose to utilize cloud environments for their applications gain very powerful tools that help them on every step. Tools like Containers as outlined in section 2.3, CI/CD pipelines for DevOps and many services are provided by the modern cloud providers that I went into detail in capter 2.

Backend application development has moved away from robust web servers serving HTML pages while performing the application logic and more towards building small and agile APIs. APIs that deliver the queried data they retrieve from a local database and offer some logic have become very common practice and they have the potential to utilize cloud environments. When developing backend services there is not much difference between locally deployable code and a cloud application. The approach that has been promoted by cloud providers leverages local-like environments that are deployable in the cloud environment in the form of containers.

For some services, it is okay to only have the option of delivering raw data when queried. For other applications that require user interaction or have a need to present the data in a human-understandable format, the developer has to consider implementing a front-end part of the application.

In this section, I will go over the languages and frameworks that are popular choices when it comes to application development in clouds. Furthermore, I have chosen to go with the Python Django framework when developing the backend part of the cloud development framework and Typescript with React framework combination for the frontend part.

4.9 Java and Jakarta EE

Java is one of the world's most used programming languages and the basis of some of the world's biggest software projects. But standalone Java might not be enough for running and

supporting more complex software projects. You need to use it with Jakarta EE, formerly called Java EE. Jakarta EE extends the popular Java SE with specifications for developing and running scalable, reliable, and secure applications. Jakarta EE has been formerly known as Java EE before when it was still developed by Oracle. Nowadays, the whole project has been taken up by the Eclipse Foundation software organisation and Jakarta EE has been made open-source.

Jakarta EE is in the simplest terms, a collection of APIs and a framework for creating new ones. Therefore, Jakarta EE is a relevant piece of technology in backend and server-side development. With clouds, distributed environments like microservice systems and containers became increasingly popular. Applications developed in distributed environments are required to communicate with other applications and systems.

The Jakarta EE specifications are designed to work with a Jakarta EE compatible runtime. A runtime is a program which runs the application and handles the HTTP requests that connect it to its users on the internet. Larger runtimes help developers with introducing additional features to handle common application concerns like security, configuration and logging. Jakarta EE is designed to work with these larger runtimes.

Jakarta EE server provides underlying services in the form of a container for every component type. Containers are the interface between a component and the low-level, platform-specific functionality.

4.10 Java Spring Framework

The Spring Framework (Spring) is an open-source application framework that provides infrastructure support for developing Java applications. It is one of the most popular application development frameworks in Java. Spring is considered to be a secure, low-cost and flexible framework. Spring improves coding efficiency and reduces overall application development time because it is lightweight. Spring handles the infrastructure so developers can focus on the application.

Spring Framework includes a number of third-party library integrations and offers custom dependency injection (DI) and inversion of control (IoC) components. Developers can utilize the DI and IoC to build loosely coupled applications that are scalable and easier to unit test. Developers have access to interfaces such as Dispatcher Servlet, ModelAndView and ViewResolver to decouple application objects and further simplify development.

The Spring Boot framework module enables developers to create stand-alone applications that can run immediately as self-contained deployment units. Moreover, developers can create various configuration profiles for different development environments and easily differentiate parts of their application configuration.

Spring Cloud builds on the concepts of Spring Boot to solve some of the problems that developers encounter when building microservices. Spring Cloud incorporates both Spring Framework's unified programming model and Spring Boot's rapid application development capabilities. Spring Cloud essentially provides a variety of design patterns and services such as registry and discovery support, therefore, avoiding a need for static hostnames. On the downside, Spring is unable to interchange technology stacks, libraries and languages. Moreover, a developer must make sure the Spring Cloud Config Server is up and running every time in order to run a single microservice.

4.11 C# ASP.NET Framework

ASP.Net is an open-source web development platform provided by Microsoft. It is used for the development of fast and secure web-based applications. It is an extension of the .Net platform for cross-platform application development. With support for multiple programming languages, I will primarily focus on the C language as it is by far the most popular one.

ASP.NET offers three frameworks for creating web applications: Web Forms, ASP.NET MVC, and ASP.NET Web Pages. Each framework targets a different development style. ASP.NET MVC gives you a patterns-based way to build web applications that enables a clean separation of concerns. ASP.NET Web Pages and the Razor syntax provide a fast way to combine the backend, and server-side code with HTML elements to create dynamic web content. The three frameworks are not independent and choosing one does not exclude using another. ASP.NET Web API is a framework that makes it easy to build HTTP services. ASP.NET Web API is an ideal platform for building RESTful applications on the .NET Framework.

The .Net framework offers access to many official and custom packages through the Nugget package management system. Nugget helps developers search and download potentially helpful libraries for their projects while also managing the versions and dependencies. It is a powerful tool that can be used with public or private Nugget repositories.

4.12 Node.js Express Framework

Node.js is an open-source, cross-platform runtime environment that allows developers to build server-side tools and applications in JavaScript. However, the runtime is expected to be running directly on the operating system instead of running in a browser environment like standard JavaScript. As such, the environment omits browser-specific JavaScript APIs and adds support for more traditional OS APIs including HTTP and file system libraries. It has become the standard server framework for node.js. Express is the backend part of something known as the MEAN stack. The MEAN is a free and open-source JavaScript software stack consisting of MongoDB, Express, Angular and Node.js.

Express comes with an express-generator tool that helps developers quickly set up an Express application skeleton by running a simple command via the command-line interface. Express does not have any built-in ORM systems. It utilizes a rich package ecosystem to connect to different types of databases. Adding the capability to connect databases to Express apps is just a matter of loading an appropriate Node.js driver for the database in the application.

Express is a routing and middleware web framework that has minimal functionality of its own. An Express application is essentially a series of middleware function calls. Middleware functions have access to the request and response objects and the next middleware function in the application's request-response cycle.

The node package manager (NPM) provides access to hundreds of thousands of reusable packages. It also has best-in-class dependency resolution and can also be used to automate most of the build toolchain.

4.13 Python Flask

Flask is a Python-based microframework used for the development of web applications. It joins two solutions together to create a library capable of building web applications. The aforementioned solutions are Wekzeug, a web server framework and Jinja2, a templating library. It does not depend on external libraries to perform the tasks of a framework. All the tools are readily available for developers to support the functionality of a web application.

The "micro" in microframework means Flask aims to keep the core simple but extensible. Flask is not an opinionated framework and therefore abstains from making technological decisions for the developer and the decisions made, like for example the templating engine can be easily changed. By default, Flask does not include a database abstraction layer or form validation but instead promotes the usage of already existing libraries. It supports extensions to add such functionality to the application as if it was implemented in Flask itself.

Flask framework requires a certain level of experience in designing applications to get right. With no enforced project structure all the decision making is directed at the developer. This approach creates certain freedom in design, but sometimes too much freedom can hurt. Flask does not offer any support for database systems and Object-Relation Mapping (ORM) out of the box. Therefore, while being a lightweight microframework, the developer has to solve a lot of problems at the beginning of the project before he even starts developing the application. Concerns like administration tools, ORM, security and more has to be solved in advance and add to the total time of the project. This in my opinion renders Flask unfriendly towards the development of Minimum Viable Product applications.

4.14 Python Django Framework

Django is a high-level open-source Python web framework that encourages rapid development and clean, pragmatic design. It takes care of much of the hassle of web development so the developer can focus on writing the business logic. Django offers a big collection of modules which can be used in web application development.

Django is considered to be an opinionated framework. An opinionated framework is one which is designed in such a way that its users will experience the least friction with that framework when the framework is used in a way that does not violate the assumptions made by the framework designer. It means that the framework itself offers the developer sensible defaults that enable rapid development. Django offers a clear project structure with Django projects and Django apps. Django project is a python package that represents the whole application and can contain multiple Django apps. However, the Django app is just a Python module inside a project containing business logic that can either be shared or contained inside the application.

Django implements a Model-View-Controller (MVC) pattern in its own way. It uses the principles of the MVC pattern but introduces its own terminology and calls it Model-Template-View (MTV). Django uses the term Template for the views and View for the controllers. The templates represent the HTML code enriched by the Django template language.

Django framework provides a very powerful Object-Relation Mapper (ORM). ORM is a service or a tool that enables the developer to interact with the application database. Django's ORM is just a pythonic way to create SQL to query and manipulate the database and get results in a pythonic fashion. Django uses Python classes that subclass from a

Django Model class to map SQL data onto Python objects. It allows Django ORM to provide developers with automatically generated database access API. This approach to data access is also called the Active Record pattern.

Django REST Framework

Django REST framework (DRF) is an open-source and well supported Python library for Django Framework that helps developers with building REST-ful APIs. It provides developers with a fully-featured toolkit for the development of both turn-key and complicated REST APIs.

DRF allows developers to define URL structure and not rely on an auto-generated one based on a conversion from Django models to REST endpoints. Web API developed with DRF is rich and web browsable and supports a wide range of media types, authentication and permission policies out of the box.

DRF introduces the concept of model serializers. DRF serializer is a class that subclasses the ModelSerializer class from the DRF module. DRF's Serializers convert model instances to Python dictionaries, which can then be rendered in various API appropriate formats like JSON or XML.

4.15 Typescript

TypeScript is a superset of JavaScript. It builds on top of the JavaScript base functionality and syntax and introduces additional aspects to the language. TypeScript uses a special compiler that converts TypeScript code into JavaScript while checking the type compatibility during the compilation. By being a superset, any JavaScript program that is valid is also a TypeScript program. However, most TypeScript compilers enforce a rule against type inference where the type can not be inferred based on the assigned value. Therefore, typescript would infer type any.

TypeScript is intended to be used when developing complex applications in JavasScript. The consistent use of TypeScript in a project initially increases the skill level requirement of all developers. But over the course of a project, this initial effort can pay off in many areas like better code readability and avoiding runtime errors due to incorrect types.

4.16 React

React is a JavaScript-based UI development library. However, React is not a framework, it is indeed specified as a library. The explanation for this is that React only deals with rendering the UI components and reserves many things at the discretion of individual projects. A component is a mixture of HTML and JavaScript that captures all the logic required to display a small portion of the UI. Components can be nested in each other thus creating a tree. This tree is then roughly transformed into a representation of a DOM.

React embraces the fact that rendering and UI logic are inherently intertwined. Instead of separating the technologies handling events and data display, React components contain both technologies. React uses a syntax extension called JSX (or TSX if using typescript) to describe what the UI should look like. It is a markup language that allows developers to mix HTML with JavaScript expressions. After compilation JSX becomes a regular JavaScript function that calls and evaluates to JavaScript objects.

The simplest way to define a component is to write a JavaScript function. The javaScript function is a valid React component when it accepts the „props“ argument and returns a React element. Functional components can be then used instead of the regular HTML tags using the HTML syntax when used with JSX. It allows developers to compose component elements out of other user-defined or HTML components.

React provides developers with a declarative API to abstract component rendering from the application logic. To make this possible React needs to implement a reconciliation algorithm. React uses a render function to generate a tree of React elements. Whenever an application state or component property is changed, React needs to figure out how to update UI to match the potentially affected tree. The state of the art algorithms [5] for figuring out the tree transformations have a complexity in the order of $O(n^3)$ where n is the number of elements in the tree. This would be problematic and therefore React implements a heuristic algorithm with the complexity in the order of $O(n)$ based on assumptions that:

- Two elements of different types will produce different trees.
- The developer can hint at which child elements may be stable across different renders with a key prop.

Sometimes it is unavoidable that a component needs to return a list of children. For this case React implements Fragments and their sole purpose is to map a collection of components to a fragment.

React faced many issues regarding readability and reusability. Functional components were mainly used as UI components due to the fact they could not manage the state by themselves. Some client libraries tried to solve this issue for developers by creating a shareable context between components. Ever since the React version 16.8, this all has been changed. React developers released hooks to address a number of problems.

React faced many issues regarding readability and reusability. Functional components were mainly used as UI components due to the fact they could not manage the state by themselves. Some client libraries tried to solve this issue for developers by creating a shareable context between components. Ever since the React version 16.8, this all has been changed. React developers released hooks to address a number of problems.

React hooks are functions that let developers interact with the state and lifecycle features inside functional components. There are a number of hooks built-in the React library but there are options to build custom hooks also. Just to name a few:

- Effect hook adds the ability to trigger side effects from a functional component. It automatically triggers when a given component is mounted or when a property is updated.
- State hook declares a state variable inside a functional component. State variables are preserved by React between the function calls and allow developers to pass a state between components. When a state variable is updated, all components that are dependent on this variable are re-rendered.
- The memo hook will only recalculate the memoized value when there is a change in its dependencies. It is mainly used in optimization and should not contain any side effects as the function provided runs during the component's render.

4.17 Angular

Angular (also referred to as Angular2+, do not mix with AngularJs) is an open-source, JavaScript-based front-end framework written in TypeScript. Google maintains it, and its primary purpose is to develop single-page applications. Angular is still the second most used JavaScript front-end framework. However, user satisfaction has dropped over the past few years which shows the complexity of the framework. As a framework, Angular has clear advantages while also providing a standard structure for developers to work with.

Angular utilizes a component-based architecture where a large application is broken down into logical components. Developer designed components are then organized into NgModules. NgModule is the basic building block of the Angular framework and an Angular application is defined by a set of NgModules. Angular components define view sets of screen elements and use services, which can provide specific functionality to the component.

Modules, components and services are classes that use decorators. These decorators mark their type and provide metadata that tells Angular how to use them. The metadata for a component class associates it with a template that defines a view. A template combines ordinary HTML with Angular directives and binding markup that allow Angular to modify the HTML before rendering it for display. The metadata for a service class provides the information Angular needs to make it available to components through dependency injection.

Due to a layered architecture angular can end up being a slightly difficult framework to debug sometimes and people who are not used to n-tier architectures can find some of the concepts complicated. The concepts of dependency injection and inversion of control are both great tools in development but they can be very challenging in more complex systems. In many frameworks that use dependency injections, the injection itself happens at the bean or configuration level. Angular uses an injector associated with a NgModule that is responsible for dependency creation and injection based on the module metadata.

Chapter 5

Technological Design

In this section, I will go over the design process and thoughts on the framework together with implementation insights. I will introduce the domain model as a monolith and then split it into individual services. Then I will design a system infrastructure built on the Google Cloud Platform and go over the API endpoint design. In the end, I introduce the demo application and how I intend to demonstrate the functionality of the framework.

5.1 Domain Model

In a system design with microservice architecture, there is no single point of truth. Every service owns a specific set of data and knows either nothing or very little about its surroundings. It is a common practice when developing a monolithic application to design a robust database where tables reflect real objects as outlined in the section 3.1. This principle is not applicable when designing a distributed system as every service is only concerned about a specific part of the global functionality. But a good start is to design a familiar schema and then iteratively split it into separated domains.

Users Table

The user table in the database represents a system user entity. Records in the user table represent the system-wide user identity, roles and permissions. The user entity implements the following fields:

- User's name as a part of user identity in the system. This field contains a string with the user's full name.
- Email address is a unique field that carries the user's identity in the system. The assumption is made that the user account binds to a unique address as only one user should be entitled to this unique address.
- Role in the system. Users can be assigned multiple roles based on their registration. It is an enumerable value as the roles are defined by the system itself.

Applications Table

Applications play an important role in this system. Therefore, they inevitably carry important data that needs to be persisted and shared. Under the application entity, you can

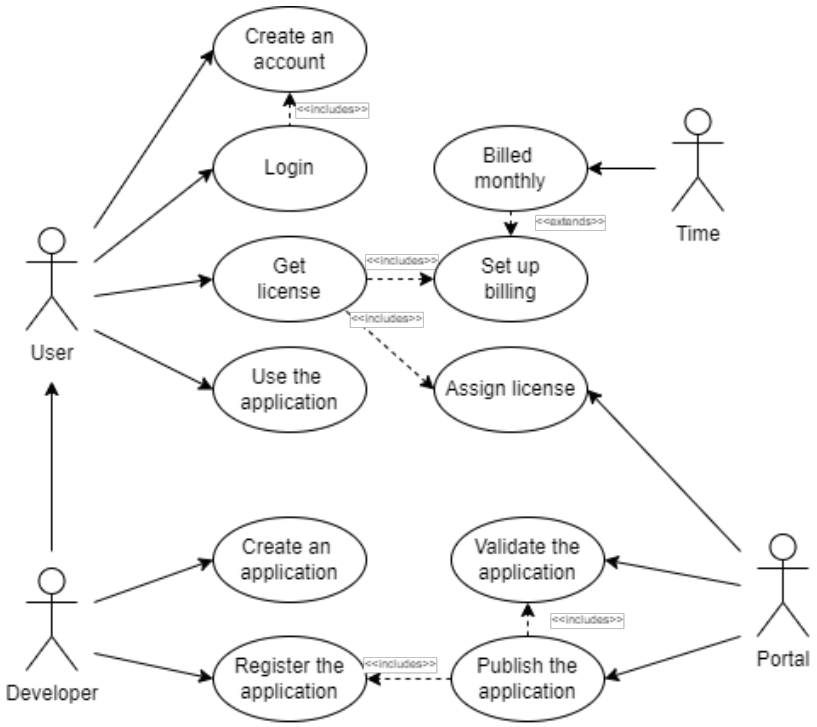

Figure 5.1: Portal use-case diagram

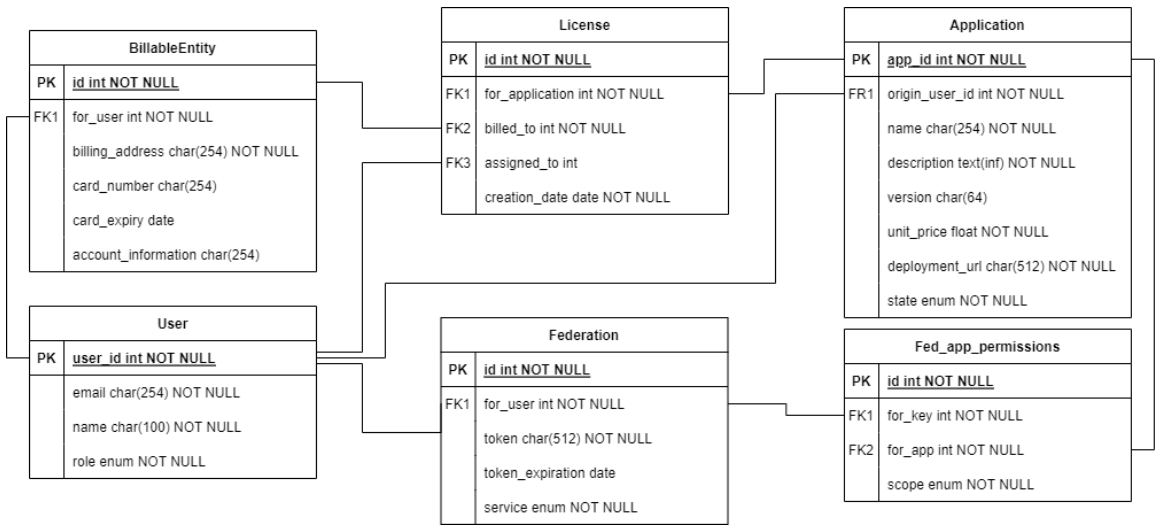

Figure 5.2: Monolithic Portal ERD diagram

imagine an application that a user can subscribe to and use in this system. The application entity implements the following fields:

- Deployment URL field which is important for user redirects. Once the application is deployed on the platform this field keeps track of the URL the application is running on and accessible to the public.
- Application version field. The application can have multiple versions deployed and this field acts as a versioning element.
- Application state field to represent what state the application is in. Whether the application is deployed, running or having issues.
- Application metadata fields like application name, description, pricing etc.

Licenses Table

Computational resources are not free. Therefore, the system needs to keep track of who uses what applications. The licenses table represents a relationship between the user and the application entities. The license entity implements the necessary fields to map the applications to a specific user.

Billing Table

Bulling is an integral part of monetization and subscription. Every user entity has a corresponding billable entity created in the system. The billing entity is created atomically together with the user entity upon registration. It stores important billing information about the user that is used by the system. Information such as:

- User's billing address field for the legal documents and invoices.
- User's card information for potential processing of subscription payments.
- Developer's account information for earnings.

Federation Table

The Federation table aims to store information necessary for system interactions with third-party service providers. It is a common demand to provide integration with already existing solutions through APIs. The Federation table enables developers to integrate their solutions with existing services on the user's behalf by querying for a stored access token. The federation entity implements the following fields:

- Token field storing the encoded access token.
- Field with the token expiration date.
- A reference to the origin service.

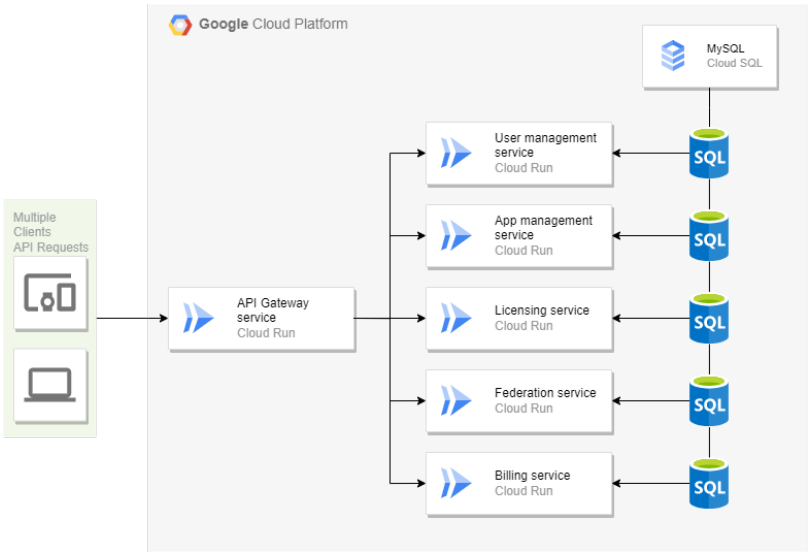

Figure 5.3: Google Cloud Platform infrastructure diagram

Microservice Architecture Adoption

The breakdown of monolithic architecture into individual services requires splitting the domain model into isolated subdomains. I have explained in the section 3.2 why it is a good idea to separate a single monolithic database into dedicated isolated databases. In the case of portal design, I will go with the separation of tables into individual databases. Where each database will serve a dedicated microservice.

The standard foreign keys will still exist but they will no longer point to a specific record in the database. For the service and the database itself, they will act as external identifiers. The responsibility of keeping the database consistent is shifted to the developer instead. The values will still correspond to the identifiers of existing entities therefore, they can be queried from a different microservice when needed.

5.2 Infrastructure

I have chosen the full infrastructure for this thesis to be located in the public cloud environment. There are multiple reasons why I chose this approach. At first, I wanted to explore the options and solutions provided by cloud providers. Also, some direct and indirect experiences played a role where the solution created was not prepared for the traffic it experienced and this effect caused distress for a lot of people.

Cloud infrastructure offers easy to opt-in technologies where the developer is not required to own the underlying hardware when taking advantage of computational resources. This fact can also reduce initial costs for startup projects. Some technologies are even offered for free until a certain threshold is reached. Furthermore, modern cloud solutions usually offer some form of service scaling out of the box.

I have chosen to build my project infrastructure on the Google Cloud Platform specifically utilizing their Cloud Run service. Cloud Run offers strong support for containerization and as I have outlined in the section 2.3, containers are capable of packing the project dependencies into a sort-of executable package. This package is then pushed to the cloud

repository, and by utilizing the power of serverless technology as mentioned in section 2.2.4, deployed.

I am using Docker containers to deploy individual services to the Cloud Run service. Each of the microservices implements an API that is exposed to the internet. Google Cloud Platform offers a setting, where the endpoint is secured by a service account access. Therefore, the APIs are secured and can not be accessed by anyone except the API Gateway service. More about the API Gateway in the next chapter.

The architecture contains a single SQL Server running on the Google Cloud. I have decided to go with a MySQL version as the data of the core services seems to be relational and the data location can be regionally based on the user's location. The server runs multiple databases specifically one for each microservice.

5.3 API Design

REST-ful APIs are the golden standard of the internet. The ease of development and understanding of the structure of REST endpoints made it very popular amongst developer communities. For this project, I have chosen to implement all the service APIs using the REST principles outlined in section 4.3.

Each microservice offers a specific set of endpoints granting access to its data. Endpoints are secured by a Google Cloud Platform configuration, requiring service account credentials for authentication. I have made this design decision to restrict access to the service endpoints. The endpoints are not meant to be accessed outside the platform architecture. It is primarily done to avoid the need for implementing features like authentication and request caching on every service. Therefore, any request would pass unauthenticated and bypass any permission settings.

As I mentioned, the services themselves are not the way to access their endpoints. I am using an API Gateway service to aggregate all the endpoints into a single service alongside the authentication, routing and other API features. The API Gateway first serves the purpose of a request proxy. The problem with running multiple microservices, other than the development challenges, is that each of the services runs under a different URL. It makes it challenging for consumers especially as they act as a single system. Therefore, the API Gateway exposes its own API endpoints with intention of proxying the requests made to a single URL to the rest of the system. API endpoints are then only known by the Gateway service and this configuration can be performed during the deployment of the service.

The API Gateway service implements a basic user authentication using the username and password to grant access. There are better modern approaches to user authentication, for example, OAuth2. I have chosen to implement only the basic auth as it demonstrates the position and functionality of API authentication while having very little architectural demands. OAuth2 authentication requires the deployment of a custom identity server. Another option for user authentication is using one of the many identity providers on the market like Facebook, Google or Microsoft. The decision to not utilize these technologies was to make the core of the framework as isolated as possible.

5.4 Transactional Consistency

Supporting a transactional consistency in a distributed system is a reoccurring problem. Different solutions lean towards different approaches to solving it. For example, as I have explained in the section 2.6 Google Spanner tackled this issue by implementing a transaction synchronisation that spans the globe. A more common approach is by introducing Sagas to the infrastructure.

There are two approaches to implementing the Sagas pattern in the microservice architecture as I have outlined in the section 3.3. I have decided to implement the Sagas pattern in this thesis very similarly to the choreography approach instead of the orchestrator. The reason behind this decision was that the implementation of an orchestrator seemed too complicated. The orchestrator needs to have the support of a service discovery microservice. Also, the orchestrator should be able to restart all the sagas that were in execution once they failed. My approach has the downsides of not being language-agnostic and introduces a service level responsibility for maintaining the saga itself.

The implementation works on a basis, that the service uses a saga wrapper for methods that need to maintain consistency across multiple services or with 3rd party solutions. The method then registers a set of rollback callbacks that undo the actions performed by its execution in case something fails.

```
    class SubscriptionManager():
        @is_saga
        def create_subscription(self, saga):
            subscription = Subscription.create(
                customer=customer_id,
                items=sub_items,
            )
            saga.register_rollback(lambda: Subscription.delete(subscription.id))

            return self.create(sub_id=subscription.id)
```

Listing 5.1: Usage example of Sagas in Python

5.5 Component Design

The frontend application is divided into a couple of root components. Two of those components are dedicated to handling the users who are not signed into the system. I am using two components to handle sing in and sign up flows. The last top-level component is restricted to signed-in users and provides the application overlay and routing for the rest of the application components.

```
    <BrowserRouter>
      <Routes>
        <Route path="login" element={<Login />}/>
        <Route path="register" element={<Register />}/>
        <Route path="/" element={<AuthRoute><Home /></AuthRoute>}>
          <Route path="dashboard" element={<Dashboard />}/>
          <Route path="profile" element={<Profile />}/>
          <Route path="profile/:userId" element={<Profile />}/>
          <Route path="solutions" element={<Solutions />}/>
          <Route path="solutions/:solutionId" element={<SolutionDetail />}/>
          <Route index element={<Marketplace />} />
        </Route>
```

```
13        </Routes>
14      </BrowserRouter>
```

Listing 5.2: Application URL routing using react-router library

I am using the `react-router` library for serving UI components based on the URL path. The structure is designed so that the overlay component is always rendered as a parent component and the content is served separately. `React-router` provides the developer with an `Outlet` component. The `Outlet` component acts as a component injector based on the application routing.

5.6 User Authentication

Portal authentication is a key component as it serves multiple purposes. At first, it verifies the user identity and role in the system. However, it also serves as a user identity context for the application. Once the user logs in, the application stores information about him and provides them to the other components. I have achieved this functionality by implementing a custom authentication hook.

The authentication hook gets distributed between the components by using a context provider as a parent component. The `AuthProvider` component creates a context provider that enables children components to access and use the authentication hook. I have achieved this by using AuthProvider as a parent component to the BrowserRouter.

```
1     <AuthProvider>
2         <BrowserRouter>
3           ...
4         </BrowserRouter>
5      </AuthProvider>
```

Listing 5.3: Authentication provider placement in the application tree

Application components then can call the `useAuth()` hook to accept the context object containing authentication functions and the user object. There exists only one instance of the shared context and therefore all the applications that get the context from the same `AuthProvider` will have access to the same instance of the user object.

5.7 Application State Management

The application state is inherently outdated. What I mean by saying that is that when an application is separated into a frontend and backend, the frontend is usually responsible for obtaining and displaying data stored on the backend. However, once the data are pulled there is usually no backwards link to keep the data updated. Therefore, the data could have been changed the moment after they were pulled, and the frontend application would not know about it.

The problem with application state consistency can be solved in multiple ways. I have decided to solve this issue by implementing a query strategy for the data. I am using a `react-query` library that provides tools to implement data fetching, caching, synchronizing and updating server state asynchronously.

`React-query` library provides two key hooks for managing data. First is the `useQuery` hook. Semantically it represents the `HTTP GET` method with additional functionality. `React-query` does not perform the fetching itself, it just provides a wrapper to the fetch function

that uses the function to make the asynchronous request while adding additional logic as to when to perform the fetch and what to do with the data. Therefore, a function that implements the data acquisition and returns a Promise object has to be provided in the form of a callback. For `HTTP POST` calls a `useMutation` hook is used with similar principles to the `useQuery`hook.

`React-query` manages query caching based on query keys. Query keys can be as simple as a string, or as complex as an array of many strings and nested objects. As long as the query key is serializable, and unique to the query's data. `React-query` will trigger data refetch automatically whenever the query key changes. This is particularly useful when working with filters.

As I have mentioned at the beginning of this section, data become stale when pulled from the backend server to the frontend almost immediately. The developer can make assumptions about his data based on the knowledge he has about the system. But there is another guaranteed way to avoid unnecessary data fetching. When the user is not looking at the data, it does not matter if they are up to date or not. `React-query` library implements a few refetch strategies out of the box:

- `refetchOnMount` - this strategy refetches data whenever a new component that calls `useQuery` mounts.
- `refetchOnWindowFocus` - this strategy refetches data whenever the focus returns to the browser tab.
- `refetchOnReconnect` - this strategy triggers a refetch whenever the application comes back online after losing connection to the network.

5.8 Demo Application

I have implemented a Fitness tracker application as a demonstration of how the framework can benefit both the developer and the end-user. The intention of the application is to serve as a SaaS to the end-user. The developer can leverage the framework as a platform that offers some common services for users. This includes services such as login and billing but also extends to non-technical services like marketing or user experience (UX). In this section, I will outline the development steps a developer has to go through to implement any type of cloud-native SaaS application using the framework.

Application Design

The Fitness tracker application allows its users to track their progress between training sessions and compete with friends. The application calculates and displays the progress of the user's body-mass index (BMI) as he logs in his progress in weight gain and loss. This metric is projected to the user by a line chart on the main page of the application.

The application also allows the user to create different profiles (very similar to the concept of Netflix profiles). Profiles are the objects that the application uses to organize its data. Each profile is intended to be potentially a different person. Each profile has a separate tracking for weight and exercise progression.

This brings me to the last interesting design feature the application has. It is a feature that enables a competition mode. Data from all profiles will be shown on the user's exercise graph. This way the user can compare his achievements with their friends.

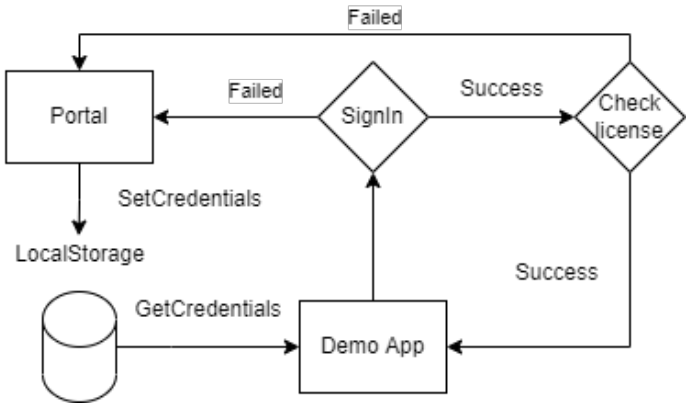

Figure 5.4: Shared authentication schema

Shared Authentication and Licensing

The idea is for users to only have one identity when jumping between different applications utilizing the framework services. The application-level access is then determined by the combination of user identity and a license object for the given application. The application first verifies that the user trying to access is a part of the platform and then validates his license. The user's license plays a role of a system permission object to an application.

Current implementation stores the user identification as an object in the indexedDB storage inside the web browser. Different applications can leverage this storage to check if the user identification token is present and validate its legitimacy with the platform.

The approach of shared authentication is very common among modern technological solutions. Often this aspect is implemented using single-sign-on that basically works as outlined above.

Leveraging Backend as a Service

I have decided to leverage the Firebase platform 2.6.5 as a backend for my demo application. Mostly to demonstrate the full power of services provided by cloud providers nowadays. Also because I have identified a great fit for the idea I came up with.

The functionality of the Fitness tracker application can easily be handled by the frontend components designed in React and Material UI therefore, I only needed to handle the data persistence. I have decided to use Firestore document storage for this purpose. The application uses a „users collection“ to store the necessary documents. Each document is representing the data of a single user.

Firebase project provides an SDK for JavaScript applications with additional functions to manage the access to Firestore collections. This was not enough for my use case as it only allowed me to create, read and delete the documents I needed to work with. This posed a challenge for data storage capabilities as it did not allow modification of nested structures. I have implemented a custom API that the demo application uses that provides additional features by additionally modifying the data structure.

Benefits of Using the Framework

Applications developed with the framework in mind gain significant technological and non-technological benefits. I believe they all are equally important. An application that is capable of fulfilling its purpose is worthless without the userbase.

Demo application primarily leverages the technological benefits to support its users and provide seamless access. It uses the authentication and licensing endpoints to identify its users and determine their permissions.

Chapter 6

Project Takeaways

Designing a framework for the development of cloud-native applications was a big challenge. Understanding the underlying technologies needed to deploy and run applications in a cloud environment is a never-ending story as there is too much to unpack. Modern cloud environments provide a near unlimited set of options for application hosting and monitoring. The technological spectrum supported by these options is also huge where a developer can decide to use multiple languages, deployment options, and frameworks in a single project. Cloud billing is also an aspect worth considering when designing a sturdy system with an intention to last a long time.

6.1 Cloud Runtime

In this project, I have focused on using computational resources provided by the Google Cloud Platform (GCP) 2.6. I have designed the infrastructure to mostly run on the Cloud Run service, which offers great support for containerization and automatic scaling. As I mentioned in section 2.6.2, Cloud Run natively support scaling down to zero instances. It means that when the project is not generating any traffic it also does not cost anything.

However, Cloud Run is a serverless service which generally costs more for the same runtime. When compared with a GCP Compute Engine, which I have described in section 2.6.1, it is more expensive to run a steady service on a Cloud Run instance than on the Compute Engine ones. GCP Compute Engine also grants price discounts for long term commitment to using their services. However, Compute Engine does not scale as quickly as Cloud Run instances can and also can not scale to zero. But when considering a significant steady demand for the services running there Compute Engine comes on top.

A technological strategy can possibly be devised out of this observation where once an average traffic threshold is reached it makes sense to change the cloud infrastructure to avoid higher costs. However, for the projects where the demand and traffic generated are unknown, I believe it is a good choice to go with a Cloud Run as a default option.

6.2 Automatic Scaling

Automatic scaling is a good servant but a bad master. It is great when you have a great spike in traffic and service unavailability is undesirable. For example, when in 2020 the Czech government released web eDalnice together with a system for buying electronic vignettes. The system crashed shortly after being released to the public [19]. However,

Figure 6.1: Request count recorded by the Cloud Run environment

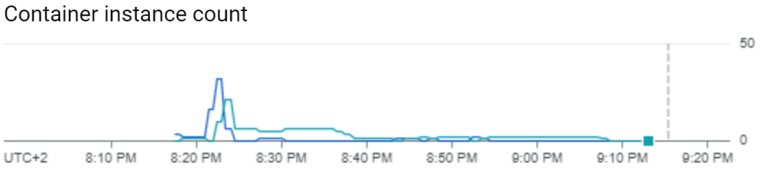

Figure 6.2: Container instance count recorded by the Cloud Run environment

automatic scaling in clouds protects the system from crashing or unavailability when experiencing heavy traffic by increasing the number of computational resources available in the infrastructure.

On the other hand, it can drastically increase the costs of a project when managed poorly. The system might get under attack, where attackers send many requests to your services, and the natural response of a system with automatic scaling is to scale up. I learned this the hard way when I made a mistake in the business logic of a frontend implementation. I have started generating 60 to 80 requests a second and sending them to my backend implementation. This resulted in the backend scaling up from a couple of instances to 50.

6.3 Framework Authentication

During the process of deciding on the technology stack for the framework, I ended up with basic authentication. I believe it demonstrates the authentication process well enough when it comes to APIs while sacrificing security for ease. I have implemented the authentication on the API Gateway service to authenticate the users when they try to access endpoints. Also, I have implemented an AuthProvider webhook on the frontend to handle and validate the user credentials.

However, when I started working on the implementation of the demo application. I have realized how impractical basic authentication is when dealing with single-sign-on-like behaviour. I wanted to implement a system to automatically perform the user authentication in the demo application if he was previously signed in on the platform. I had too many dependencies at this point, so I have defaulted to storing the user credentials as an authentication object inside the indexedDB storage. It is a trend adopted from Firebase Authentication. Firebase Authentication stores the user's JWT as a value inside the browser's index storage.

6.4 Asynchronous Frontend

A frontend application is one way to serve user data and allow them to interact with them. It uses API calls to operate with the data (CRUD operations), but this communication is usually only initiated by the frontend application. It means that anytime the state

changes on the backend, the frontend application has to request the new data. But how is it supposed to know?

One way to solve this issue is by implementing webhooks on the backend service. However, this approach is expensive and not practical for the majority of systems. The more common method is implementing a refetching strategy in the frontend application.

When I implemented a refetching strategy into the frontend application in this project. It instantly became livelier, and the data showed kept refreshing without the need for a refresh button or a user invoked page refresh. However, refetching can be challenging and tricky. I have decided to use automatic refetching only when the user focus came back to the page and when a user got reconnected. I have also invoked a manual refetch upon data update. This helped to prevent situations like in section 6.2.

6.5 Service Provisioning

When I designed the framework as I am presenting it in this thesis, there were many services I have considered. I would classify the service types in this system into two categories, platform-oriented and application-oriented. Platform-oriented services provide the much needed infrastructural support to the framework like billing. On the other hand, application-oriented services provide much-needed support for application functionality like authentication or licensing.

The framework aims at eliminating the need for having to implement business-critical services for your application. But it should not provide a full range of services that can be generally used. I will use the example of the Google Cloud Platform. It provides a range of base services like databases and computational power as services. The framework should never degrade to such granularity as to start exposing its underlying structure to the user or the developer. It would result in becoming another cloud service provider.

Another motivation for implementing additional services is the potential integration with third-party systems. The federation service aims to address this issue partially.

Chapter 7

Conclusion and Future Work

My thesis aimed to design and implement a framework for the development and operation of cloud services. At first, I had to learn about major cloud service providers and understand the range of offered services. Then devise a technology stack with a focus on developing applications in the cloud environment.

The framework was implemented as distributed system following the microservice architecture principles. Backend services were implemented using the Python Django framework completed by the Django Rest Framework extension to implement a set of REST API endpoints. The frontend application of this project was implemented using React JavaScript library utilizing the Material UI components.

I have designed multiple independent services such as licensing, billing and application management. An important backend component binding the system together is an API Gateway implementation that serves as a reverse proxy server and enforces user authentication. The frontend part of implementation consists of two applications. One is the portal application that serves as a framework hub for users and developers. The second application is a demo application utilizing the framework services.

I firmly believe that a framework like this has potential when it comes to the current application development market. Mainly as the current professional sector is experiencing a shift where many developers and other professionals are moving from big corporates to smaller teams or deciding to work on their own terms as freelancers. This framework is trying to provide the necessary tools to accommodate this talent and expand on it. The bigger picture is to allow people to sell their talents and services with ease while helping someone else in the process.

However, I feel like I have only scratched the top of the issue. There are certainly more services that could be offered under this framework. As the part of the framework serves the actual users of the applications I believe a huge step could be made in aspects of user experience. Including services like product recommendation and data analysis and improving the user experience of frontend applications. Another idea would be to introduce a static analysis for submitted applications. The current process requires someone to review the application source code before deployment to ensure the legitimacy of its intentions.

Bibliography

[1] *What are some Amazon EC2 use cases?* [online]. [cit. 2022-04-26]. Available at: https://www.awsforbusiness.com/amazon-ec2-use-cases/.

[2] *What is Docker?* [online]. Available at: https://www.gartner.com/doc/reprints?id=1-271OE4VR&ct=210802&st=sb.

[3] *Amazon EC2* [online]. 2022 [cit. 2022-04-26]. Available at: https://aws.amazon.com/ec2/.

[4] Abiodun, B., Oneyibo, O., Nwokoma, C., Kolawole, O. and Mogoli, P. *By 2025 internet penetration will increase by 130% in Sub-Saharan africa.* Mar 2018. Available at: https://techpoint.africa/2018/03/08/global-internet-users-to-hit-5billion-by-2025/.

[5] Bille, P. A survey on tree edit distance and related problems. *Theoretical Computer Science*. 2005, vol. 337, no. 1, p. 217–239. DOI: https://doi.org/10.1016/j.tcs.2004.12.030. ISSN 0304-3975. Available at: https://www.sciencedirect.com/science/article/pii/S0304397505000174.

[6] Bond, J. *The enterprise cloud.* O'Reilly Media, Inc. Available at: https://www.oreilly.com/library/view/the-enterprise-cloud/9781491907832/ch01.html.

[7] Evans. *Domain-Driven Design: Tacking Complexity In the Heart of Software.* USA: Addison-Wesley Longman Publishing Co., Inc., 2003. ISBN 0321125215.

[8] Fong, J. *Are containers replacing virtual machines?* 2018. Available at: https://www.docker.com/blog/containers-replacing-virtual-machines/.

[9] Fowler, M. *UbiquitousLanguage* [online]. 2006 [cit. 2022-01-22]. Available at: https://martinfowler.com/bliki/UbiquitousLanguage.html.

[10] Fowler, M. *DDD Aggregate* [online]. 2013 [cit. 2022-01-22]. Available at: https://martinfowler.com/bliki/DDD_Aggregate.html.

[11] Garcia Molina, H. and Salem, K. Sagas. *SIGMOD Rec.* New York, NY, USA: Association for Computing Machinery. dec 1987, vol. 16, no. 3, p. 249–259. DOI: 10.1145/38714.38742. ISSN 0163-5808. Available at: https://doi.org/10.1145/38714.38742.

[12] Guzel, B. *Záhlaví HTTP pro nechápavé* [online]. 2021 [cit. 2022-03-20]. Available at: https://code.tutsplus.com/cs/tutorials/http-headers-for-dummies--net-8039.

[13] Kavis, M. J. *Architecting the cloud: design decisions for cloud computing service models (SaaS, PaaS, and IaaS)*. John Wiley & Sons, 2014. ISBN 9781118617618.

[14] Kramer, S. The biggest thing amazon got right: The platform. *Retrieved August*. 2011, vol. 16, p. 2019.

[15] Mauro, T. Adopting microservices at netflix: Lessons for architectural design. *Recuperado de https://www. nginx. com/blog/microservices-at-netflix-architectural-best-practices*. 2015.

[16] Max Roser, H. R. and Ortiz Ospina, E. Internet. *Our World in Data*. 2015. https://ourworldindata.org/internet.

[17] Mell, P. M. and Grance, T. *SP 800-145. The NIST Definition of Cloud Computing*. Gaithersburg, MD, USA, 2011.

[18] Miller, D., Whitlock, J., Gardiner, M., Ralphson, M., Ratovsky, R. et al. *OpenAPI Specification v3.1.0* [online]. 2021 [cit. 2022-04-25]. Available at: https://spec.openapis.org/oas/v3.1.0#openapi-object.

[19] Neufus, O. Web k elektronickým vinětám nefunguje. *Garáž.cz*. 2020. Available at: https://www.garaz.cz/clanek/web-k-elektronickym-vinetam-nefunguje-21005230.

[20] Pariseau, B. Knative serverless Kubernetes bypasses FaaS to revive PaaS. *Techtarget*. 2019. Available at: https://www.techtarget.com/searchitoperations/news/252469607/Knative-serverless-Kubernetes-bypasses-FaaS-to-revive-PaaS.

[21] Raj Bala, D. S. D. W. K. J. Magic Quadrant for Cloud Infrastructure and Platform Services. *Gartner*. 2021. Available at: https://www.gartner.com/doc/reprints?id=1-271OE4VR&ct=210802&st=sb.

[22] Richardson, C. *What are microservices?* [online]. 2021 [cit. 2022-01-18]. Available at: https://microservices.io/.

[23] Swoyer, S. and Loukides, M. *Microservices adoption in 2020*. O'Reilly Media, 2020. Available at: https://www.oreilly.com/radar/microservices-adoption-in-2020/.

[24] Vailshery, L. S. *Global IOT and non-IoT Connections 2010-2025*. Mar 2021. Available at: https://www.statista.com/statistics/1101442/iot-number-of-connected-devices-worldwide/.