FRAMEWORK FOR A WEB INTERNET SERVICE IMPLEMENTED IN GOOGLE CLOUD PLATFORM

FRAMEWORK PRO PLACENOU INTERNETOVOU SLUŽBU V PROSTŘEDÍ GOOGLE CLOUD

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **ANDREI ROSHKA**
AUTOR PRÁCE

SUPERVISOR **Ing. MARTIN HRUBÝ, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2022

Department of Intelligent Systems (DITS) Academic year 2021/2022

Bachelor's Thesis Specification

25187

Student:	**Roshka Andrei**
Programme:	Information Technology
Title:	**Framework for a Web Internet Service Implemented in Google Cloud Platform**
Category:	Web

Assignment:

1. Study literature about program containerization. Learn about programming of web applications. Learn about Google Cloud Platform (GCP).
2. Design a web interface for remote execution of containers (from Docker) as an Internet service. Design a framework for administration of users of this service and ways how to monetize the service. All design must be compliant to GCP, i. e. authentization of users, deployment via Cloud Run, Kubernetes etc.
3. Implement the service with a demonstration container. Deploy the service on GCP.
4. Test the service using a simulated traffic.

Recommended literature:

- Documentation of Docker.
- Documentation of Google Cloud Platform.

Requirements for the first semester:

- First two points of the assignment.

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

Supervisor:	**Hrubý Martin, Ing., Ph.D.**
Head of Department:	Hanáček Petr, doc. Dr. Ing.
Beginning of work:	November 1, 2021
Submission deadline:	May 11, 2022
Approval date:	April 13, 2022

Abstract

This thesis aims to design and implement a framework for a paid internet service and deploy it to the cloud services provided by the Google Cloud Platform with the lowest operation price. The resulting architecture is serverless, so it uses Firebase services such as Firestore NoSQL database, Firebase Storage, and Firebase Authentication. The solution is written in TypeScript and consists of two main parts – the front-end application and the worker, which processes computational tasks. React is used to build component-based UI for the front-end, forming a solid foundation with Redux for centralized application state management. As a server for the front-end, Nginx is used. The worker is based on the Node.js back-end JavaScript runtime environment. The application is successfully deployed to the Cloud Run as a set of Docker containers using a CI/CD pipeline built on Cloud Build.

Abstrakt

Cílem této bakalářské práce je navrhnout a naimplementovat rámcové řešení pro placenou internetovou službu a nasadit ho na cloudové služby poskytované Google Cloud Platform s nejnižší provozní cenou. Výsledná architektura je serverless a používá služby Firebase jako Firestore NoSQL databáze, Firebase Storage a Firebase Authentication. Řešení je psáno v jazyce TypeScript a skládá se ze dvou hlavních části – front-end aplikace a pracovního procesu, který zpracovává výpočetní úlohy. React se používá k vytvoření uživatelského rozhraní založeného na komponentách spolu s Reduxem pro centralizovanou správu stavu aplikací. Jako server pro front-end se používá Nginx. Pracovní proces je postaven na Node.js – serverovém prostředí pro JavaScript. Aplikace je úspěšně nasazena do servisy Cloud Run jako sada Docker kontejneru za použitím CI/CD pipeliny postavené na Cloud Build.

Keywords

Web, React, Node.js, Nginx, framework, cloud-based web application, Google Cloud Platform, Cloud Run, Firebase, Firestore, Cloud Storage, Docker, on-demand computing.

Klíčová slova

Web, React, Node.js, Nginx, framework, cloudová webová aplikace, Google Cloud Platform, Cloud Run, Firebase, Firestore, Cloud Storage, Docker, on-demand computing.

Reference

ROSHKA, Andrei. *Framework for a Web Internet Service Implemented in Google Cloud Platform.* Brno, 2022. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Martin Hrubý, Ph.D.

Rozšířený abstrakt

Tato bakalářská práce se zabývá návrhem a implementací rámcového řešení pro placenou internetovou službu do cloudového prostředí Google Cloud Platform se zaměřením na nejnižší cenu provozu. Dnes stále více aplikací migruje na cloudová řešení. Aplikace přecházejí z monolitické architektury na mikroslužby, protože je to snazší na rozdělení práce mezi týmy a udržování. Takové mikroslužby obvykle běží uvnitř kontejnerů Docker. Vznik kontejnerů Docker v roce 2013 byl obrovským krokem v technologiích kontejnerizace. Zmírnil křivku učení kontejnerizace a přinesl ji širší veřejnosti. Kromě toho, Docker kontejnery představily významná vylepšení ve srovnání s možnostmi Linux Containers (LXC), což je zabalení aplikace a všech její závislostí ve virtuálním kontejneru, který může běžet na mnoha různých platformách.

Aplikace musí být někde hostování a zde dochází k otázce, jak zefektivnit proces hostování. Aplikace může být samozřejmě hostování klasickým způsobem, když běží na fyzickém nebo virtuálním soukromém serveru, ale to vyžaduje spoustu ruční konfigurace a údržby. Často je dokonce nutné mít inženýra nebo dva, kteří se zaměřují pouze na údržbu a nasazení aplikace do takového prostředí.

Poskytovatele cloudových služeb neboli Cloud Service Providers nabízejí několik možností pro jejích nasazení. Jedním z nejvýznamnějších hráčů na cloudovém trhu je Google se svou Google Cloud Platform. Poskytuje mnoho služeb na jiné úrovni abstrakce od základní infrastruktury, jako jsou virtuální stroje v Cloud Compute, klastry v Kubernetes Engine, plně zvládnutelné a automaticky škálovatelné výpočetní platformy Cloud Run pro spuštění kontejnerových služeb a cloudové funkce. Nabízí také mnoho cloudových řešení pro komponenty aplikační infrastruktury, jako jsou databáze, úložiště, monitorování, logování a další. Ale některé cloudové služby mohou být těžko pochopitelné pro průměrného uživatele, který chce provést nějaký výpočetní úkol náročný na zdroje na tomto stroji.

Proto některé Internetové služby jsou vyvinuty pro vytvoření další abstrakční vrstvy nad cloudovými službami k vyřešení problému popsaného výš. Obvykle je to nějaká SaaS (Software as a Service) služba. Základní struktura placených internetových služeb pro provoz kontejnerů Docker může být vypracována předem, tím pádem zjednoduší implementaci takové služby.

V rámci této práci vnikl základní kód pro vývoj, testování a nasazení takové služby. Zahrnuje knihovnu, která zjednodušuje komunikaci front-end klienta a back-end pracovního procesu s Firestore databází. Knihovna pro pracovní proces navíc poskytuje mechanismus pro uzamčení zdroje, který momentálně zpracovává, aby se zabránilo současnému zpracování jinými možnými instancemi. Díky tomuto mechanismu je možné aplikaci bezpečně škálovat. Dále se práce zabývá bezpečnostními aspekty implementace služby s použitím Google Cloud a Firebase produktu, který je postaven na Google Cloud. Tyhle BaaS (Backend as a Service) služby umožňují postavit serverless aplikace s minimálním úsilím. Rámcové řešení bude používat tyhle BaaS služby v plné míře a poskytne přehled o postavení aplikace na nich.

Řešení je psáno v jazyce TypeScript a skládá se ze dvou hlavních části – front-end aplikace a pracovního procesu, který zpracovává výpočetní úlohy. React se používá k vytvoření uživatelského rozhraní založeného na komponentách spolu s Reduxem pro centralizovanou správu stavu aplikací. Jako server pro front-end se používá Nginx. Pracovní proces je postaven na Node.js – serverovém prostředí pro JavaScript. Aplikace je úspěšně nasazena do servisy Cloud Run jako sada Docker kontejneru za použitím CI/CD pipeliny postavené na Cloud Build.

Framework for a Web Internet Service Implemented in Google Cloud Platform

Declaration

I hereby declare that this Bachelor's thesis was prepared as an original work by the author under the supervision of Mr. Martin Hrubý. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

....
Andrei Roshka
May 11, 2022

Acknowledgements

I would like to acknowledge my supervisor, Ing. Martin Hrubý, Ph.D., for his assistance and guidance that contributed to the creation of this thesis. Throughout my entire studies at the Brno University of Technology, I have gained valuable knowledge and experience, for which I thank every teacher I have met during this period.

Finally, my family has been supportive and kind over these three years. I am very grateful for this.

Contents

Chapter 1

Introduction

Today, more and more web applications migrate to cloud-based solutions. Applications are being refactored from a monolithic architecture to microservices because it is easier to maintain and split between different development teams. Such microservices are usually running inside the Docker containers. Docker containers appearance back in 2013 was a massive step in containerization technologies. It eased the containerization learning curve and brought it to a wider public. Moreover, Docker containers introduced significant improvements compared to Linux Containers capabilities, allowing for the packaging of an application and all its dependencies in a virtual container that can run on many different platforms. The application has to be hosted somewhere, and here it comes to the question of how to make the hosting process more efficient and effortless. Of course, the application can be hosted in a classical way when it is running on a physical or a virtual private server, but this requires tons of manual configuration and maintenance. It is often even necessary to have an engineer or two who focuses only on maintaining and deploying the application to such an environment.

Cloud Service Providers (CSP) are coming to the rescue bringing the Infrastructure as a Service, Platfrom as a Service, and serverless solutions for the businesses. It allows deploying an application to the cloud while not being concerned with the server's security, OS maintenance, and software updates. One of the most prominent players in the cloud market is Google with its Google Cloud Platform. It provides many services on a different level of abstraction from the underlying infrastructure, such as virtual machines in Cloud Compute, clusters in Kubernetes Engine, fully manageable and automatically scalable compute platform Cloud Run for running containerized services, and Cloud Functions. It also offers many cloud solutions for application infrastructure components such as databases, storage, monitoring, logging, and more.

Cloud Services might be hard to understand for an average user who wants to perform some resource-demanding computational task on a machine capable of its demand. Some additional paid internet services are developed to build an abstraction layer above Cloud Services to solve this problem. The underlying structure of paid internet services for running Docker containers can be worked out in advance. It will simplify the implementation of such a service. This thesis aims to implement a framework for a paid internet service and deploy it to the Google Cloud Platform with the lowest operation price. It will provide a boilerplate code for developing, testing, and deploying the service. It includes the library, which simplifies the communication of the front-end client and the back-end worker with the database. Moreover, the library for the worker provides a mechanism for locking the resource that the worker is processing in order to prevent the simultaneous processing by other possible worker instances. Thanks to this mechanism, it is possible to scale the application safely. Also, the thesis will cover the security aspects of implementing the service on top of Google Cloud and Firebase product, which is built on top of Google Cloud.

As a result, I will describe the common aspects of the solution that I have tried to develop and describe an implementation of a demonstration application. First of all, in Chapter 2, I will research the key technologies in web development today. Then in Chapter 3, there is an overview of the tools used to implement the application. In Chapter 4 I will describe the developed architecture model for such an application. Then in Chapter 5, I will highlight the main aspects of the framework implementation. Then in Chapter 6, I will provide an overview of my experience in deploying the application. At the end of Chapter 7, I will test a service with simulated traffic.

Chapter 2

Key Technologies

This chapter provides a brief history of Web application programming, an understanding of the client-server architecture, highlights the best practices and design patterns in Web application development, and an overview of the most popular frameworks for creating Web applications today. Also, it introduces the idea of containerization and shows the differences between containers and virtual machines. Finally, this chapter familiarizes with the cloud computing concept.

2.1 Web Applications Programming

Nowadays, Web technologies help to build many modern applications. People do not need to install software on their computers locally anymore. They can create, communicate, and collaborate online. Even sophisticated programs such as spreadsheets[1], 2D graphical editors[2], 3D modeling software[3], and even games[4] can run in the web browser today.

The Web

CERN introduced the World Wide Web (Web) in the 1990s as the instrument that allowed publishing, sharing, accessing, and linking vast amounts of data. Scientists primarily generated this data. It uses the HyperText Transfer Protocol (HTTP) as the primary protocol for transmitting hypermedia documents, primarily written in HyperText Markup Language (HTML). The HTTP is a simple request-reply protocol designed to make a document from the server computer available on the client computer. Moreover, The Web introduced a method of naming and referring to the documents that are called Uniformed Resource Locator (URL), thanks to which it is possible to address the documents using names such as `example.com` instead of their IP address.

In the beginning, the Web consisted of HTML documents with static content, and these documents could also be statically linked to each other. These days, this Web is often referred to as Web 1.0 2.1. Soon it was clear that the server could execute sophisticated programs and return their results to the client. That is how the Web evolved into the so-called Web 2.0. Compared to Web 1.0, Web 2.0 2.2 applications are interactive and can react to user requests. The Common Gateway Interface (CGI) helped to achieve this

[1]Google Spreadsheets `https://docs.google.com/spreadsheets`
[2]2D editor Figma `https://figma.com`
[3]3D editor Vectary `https://www.vectary.com/`
[4]JavaScript DOS Emulator `https://js-dos.com/games/`

interactivity. It allowed web servers to execute a script to process user requests. Thanks to CGI, a developer can configure a server, so it will execute a script on a server when a URL is requested. The result is usually inserted directly into the requested document. These programs were first created in the PERL interpretive language and then in PHP. [11]

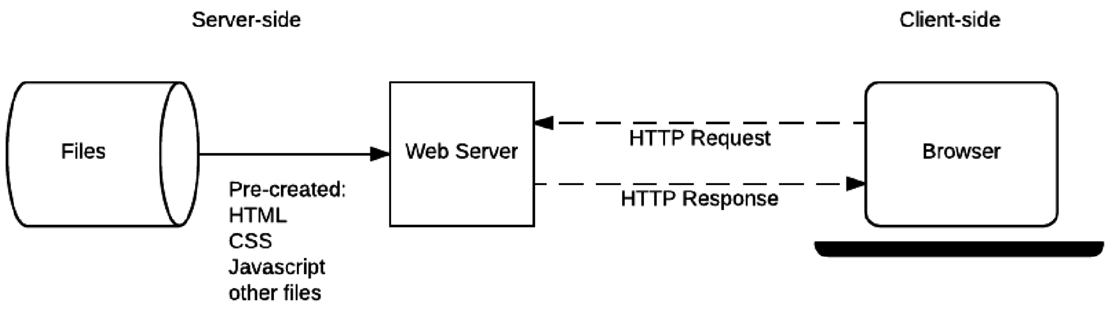


Figure 2.1: Basic static application server diagram https://developer.mozilla.org/en-US/docs/Learn/Server-side/First_steps/Client-Server_overview

Client-server architecture

A classical Web application today consists of both the client-side and the server-side. [15] A client-side is essentially a document displayed to a user that is an HTML page filled with some data. A user may interact with the document in the browser due to scripts written in the JavaScript language. These scripts are embedded into an HTML document. Thanks to the scripts, an application may interactively react to different events such as user clicks, mouse movements, and inputs from the keyboard. Also, it is possible to modify the document right in the browser. Every node in the document, such as a heading, paragraph, or text inside an HTML tag, is parsed into a Document Object Model (DOM) – object representation of an HTML page – that an embedded script can modify. The server-side is a script that processes a request. Depending on the specifics of an application, the server-side may perform authentication and authorization, store and retrieve information from a database, and perform any tasks that usual software can perform.

The MVC design pattern

From the software engineering point of view, Web applications have the same concept. An interactive user interface presents some data to a user and allows them to operate over this data. In 1979, Trygve Reenskaug introduced the design pattern Model-View-Controller (MVC). [22] Over time, that pattern has transformed into different forms. Such as Model-View-Presenter (MVP) [16], Model-View-ViewModel (MVVM) [24], which is an MVC pattern with two-way data binding between View and ViewModel which allows automatically propagating changes made in the user interface. However, the main idea has remained the same, decouple application logic and divide it into several parts, as shown in figure 2.3.

The view is responsible for presentational logic, i.e., how to present given data to the user. Its primary purpose is to suppress some parts of the data, which are unnecessary in the specific case, and highlight the others. The Model is the source of all the data for the view. Usually, it is a layer between a view and some source, e.g., a database, and

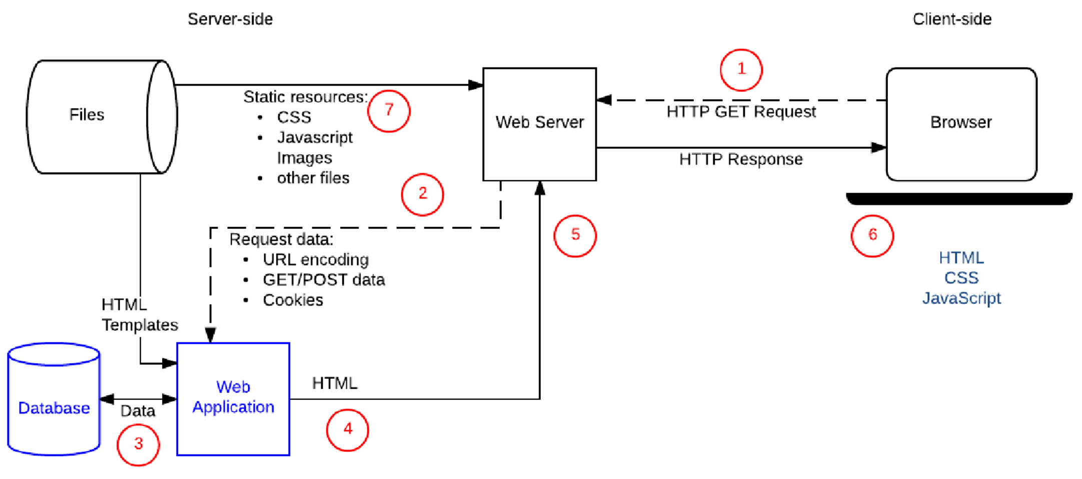


Figure 2.2: Basic dynamic application server diagram https://developer.mozilla.org/en-US/docs/Learn/Server-side/First_steps/Client-Server_overview

often provides View-specific data. The Controller is responsible for processing the incoming requests. It is a part that connects the Model and the View. Usually, it passes the data through the Model and then passes the results to the view.

Application frameworks

There are plenty of frameworks for both client-side and server-side that provide an abstraction layer that makes following best practices and using quality patterns easier. Almost any language allows writing a server-side back-end. Among them are Ruby, PHP, Java, Python, and even JavaScript, thanks to the well-known back-end runtime environment Node.js discussed later in Chapter 3. It is usual to follow the MVC pattern for such frameworks, as mentioned before. Server-side frameworks provide a convenient way to work with requests, e.g., fill particular structures with data collected from a request, authenticate and authorize users, and perform routing. There are also Object-Relational Mapping (ORM) frameworks that provide a better experience working with databases. Using such a framework improves code readability by converting relational data from the database with the entity objects. It also reduces the potential errors in an SQL code and the risk of SQL injections.

As for client applications, there are also many frameworks for implementing user interfaces, such as Angular, React, Svetle, and Vue.

AJAX

Asynchronous JavaScript and XML or AJAX is a model of updating the content in web applications without the need to reload the whole page. AJAX makes the application faster and more responsive to user actions. It uses the XMLHttpRequest object to communicate with the server. Various formats such as HTML, XML, JSON, and plain text can be used to communicate. Two things that define AJAX are the ability to make requests to the server without reloading the page and receive the data from the server to work with it.

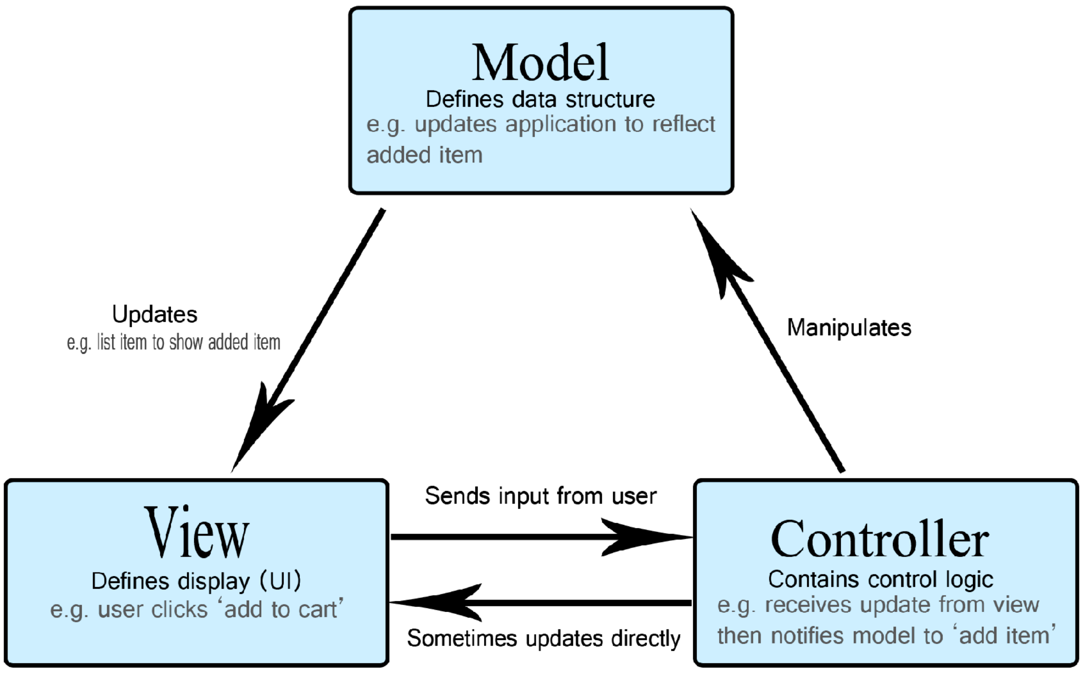

Model
Defines data structure
e.g. updates application to reflect added item
Updates
e.g. list item to show added item
Manipulates
View
Defines display (UI)
e.g. user clicks 'add to cart'
Sends input from user
Sometimes updates directly
Controller
Contains control logic
e.g. receives update from view then notifies model to 'add item'

Figure 2.3: Model-View-Controller diagram https://developer.mozilla.org/en-US/docs/Glossary/MVC

Release web application

In the case of a classical desktop application, it is necessary to notify users about the new version of the software and distribute it. In the case of a Web application, there is only one copy of the application on the server. Users access it through the browser every time they get the newest version of the software. Unlike the classical desktop application, which is released once every several months or even years, it is common to deliver new versions of Web applications several times a day. [11]

Deploy web application

In order to run, a Web application must be deployed on a server machine running some server software. The most common options are Apache Server or NGINX Server. Both of them are open source. As for a database, standard options are open-source relational database management systems MySQL and PostgreSQL and document-oriented MongoDB. [11]

2.2 Relational and Non-relational Databases

Applications of every size require some persistent data storage. It is necessary to choose between two main types: relational (SQL) and Non-relational (NoSQL or not only SQL) databases. The classical and still relevant approach is to use a relational database model, which has a quite rigid schema. That schema must be created ahead of time before data is uploaded. Changing the schema of databases is complex, especially when the database is partitioned between numerous servers. Oppositely, a non-relational database „addresses several issues that the relational model is not designed to address, like large volumes of structured, semi-structured structured and unstructured data, agile sprints, quick iteration, frequent code pushes, object-oriented programming, efficiency, monolithic architecture and so on“ [9].

Data is stored in columns and rows according to the table schema in a relational database. In a non-relational database, data is stored either in documents, graphs, column family stores, or key-value pairs. Of course, it introduces additional code overhead with data checking, but on the other hand, document-based databases are more flexible and fast. [9]

2.3 Virtual Machines and Containerization

Even though Web applications have many benefits, it is still a piece of software that demands configured executive environment and dependencies to run. It is a common problem when an application is developed on a local machine without problems during its execution. However, errors occur due to incorrect configuration or libraries' absence when the application is pushed to a production environment. That is why there is a requirement for a technology that will help avoid these problems, and software engineers will be able to predict how the software will behave on any machine. Virtualization and containerization can help to achieve this. This subchapter has been adopted from [26].

Virtual machines

There are several ways in that applications can run. The most traditional way is to have a dedicated physical server that will run server software. A server is an entire physical machine reserved for running one particular application in such a case. That is not very efficient since the machine resources are underutilized at every moment. Moreover, if this machine is used to run more applications, some may need different versions of the same libraries. It is possible to achieve, but such a setup will be complex in maintenance and updating.

Another option is to run the application on the Virtual Machine (VM). VM is software that emulates a process of an actual computer with an Operating System (OS) on it. [1] It is run on a real machine, and one such machine can have several VM instances on it. The only limiter here is machine resources which are distributed between VM instances. Also, VM instances may be distributed between different machines thanks to Hypervisor, which creates and runs VMs. In the case of VMs, a whole OS comes packed with the software. It requires more resources and memory than needed for running the software.

Containerization

Containers is a lightweight virtual machine alternative. It removes performance overheads of VMs by sharing the same OS kernel. The first idea lying behind containers is the UNIX chroot, which appeared in 1979. The chroot changes a root directory for a process and its children. Such a process cannot refer to the directories outside the specified root, which isolates software, and introduces abilities for better dependencies control and security. Then in 2000, FreeBSD jails were introduced. It is a mechanism that allows separating OS into several virtual environments sharing the same kernel. It introduced another level of security and allowed the creation of a superuser in each jail, in contrast, to chroot that was quickly breakable by the user with privileged access. In 2005 Solaris company developed Solaris Containers „to provide OS and hardware-level abstractions to isolate applications from PM in terms of applications, device paths, and network interface names." The following important virtualization technology was introduced in 2008: Linux Containers (LXC). It brought two essential mechanisms to the world of containers. The first one, cgroups, allows partitioning groups of processes and limits memory, CPU, block input/output, and network resources for each group, i.e., container. The second one, namespaces, allows customizing view of the system resources for namespaces with which processes are associated.

In oppose to VM, in the container, the software is packed into a so-called image with its executives, libraries, and dependencies ready to start. The main difference between containers and VMs is that containers do not contain the entire OS inside and run directly above the host machine OS with the help of the containerization system engine, as shown in the figure 2.4.

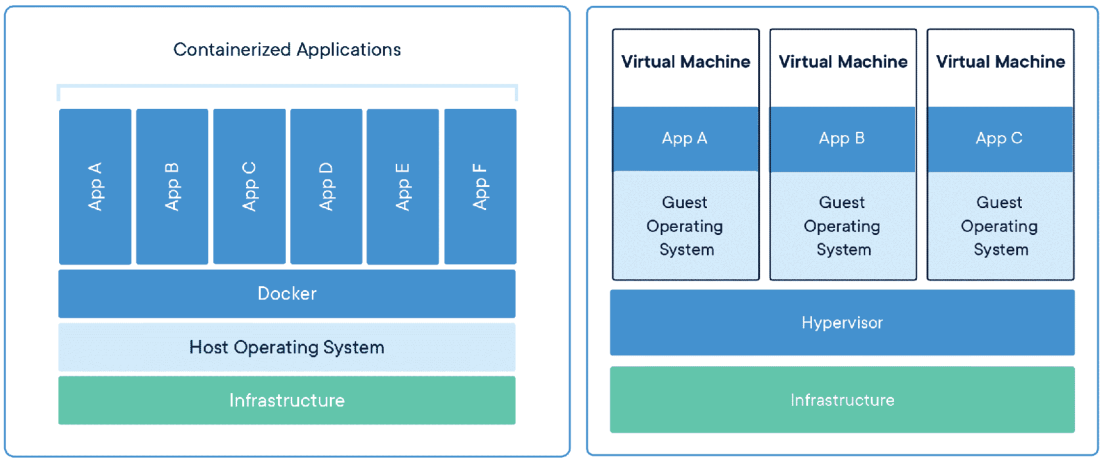


Figure 2.4: Comparison of Virtual Machine and Containers https://www.docker.com/resources/what-container/

2.4 Cloud Computing

Today there is no need to own an actual machine to host a server or database, run VM on it, or organize data storage. This can be done on the remote machines provisioned and maintained by the Cloud Service Provider (CSP). Cloud computing is a model of on-demand access to computing resources remotely over the Internet. These resources are storage, servers, services, networking and security tools, and applications. The resources are dynamically utilized from the shared pool of resources. CSP provides the facilities for utilizing the resources to the customer.

The distinctive feature of cloud computing is billing on a per-usage basis. Such billing implies measuring the usage of the resources. For instance, the count of requests to a server, transmitted bytes, or the number of retrieved documents from the storage. This subchapter has been adopted from [18].

There are several computing models provided by CSPs:

Infrastructure as a Service (IaaS)

In this model, virtualized computer resources such as CPU, RAM, OS, and Application software are provided in the cloud. It can be dynamically provisioned for the customer, released, and scaled according to their needs at any time. One of the main benefits is that the customer gains access to enterprise-grade IT resources and infrastructure. Examples of the IaaS are Amazon Web Services (AWS), Google Compute Engine (GCE), and Microsoft Azure.

Platform as a Service (PaaS)

PaaS is a more advanced cloud computing service. CSP provides and maintains both OS and the software necessary to run customers' applications in this model. Customers are charged for access to the platforms where they can host their applications. CSP undertakes the duty of upgrading and maintaining the software. This way, customers are not obliged to own either hardware or software. Examples are AWS Elastic Beanstalk, Google App Engine, and Heroku.

Software as a Service (SaaS)

In this model, CSP develops, runs, and maintains application software. It provides access to the application through a web-based interface, i.e., customers can use its services through the browser from any device. The model advantage is that customer does not need to buy a license, upgrade, or maintain the software. The SaaS services are efficient, easily configurable, and scalable. Examples of SaaS solutions are Gmail, Dropbox, and Zoom.

2.5 Serverless

According to the source, [23], "Serverless is a cloud-native development model that allows developers to build and run applications without having to manage servers". It is another abstraction layer above the PaaS model, in which the CSP undertakes not only resource provisioning and maintaining the software but also scaling the server infrastructure. In contrast to the IaaS cloud computing model, serverless infrastructure is running only when the application is active, e.g., got the HTTP request. It is the CSP's responsibility to scale the infrastructure when the application is under a higher load.

Back-end as a Service (BaaS)

BaaS provides access to third-party services such as authentication services, cloud-based databases, push notifications, and hosting. Developers do not really know how the service works under the hood but only use the public API to build an application.

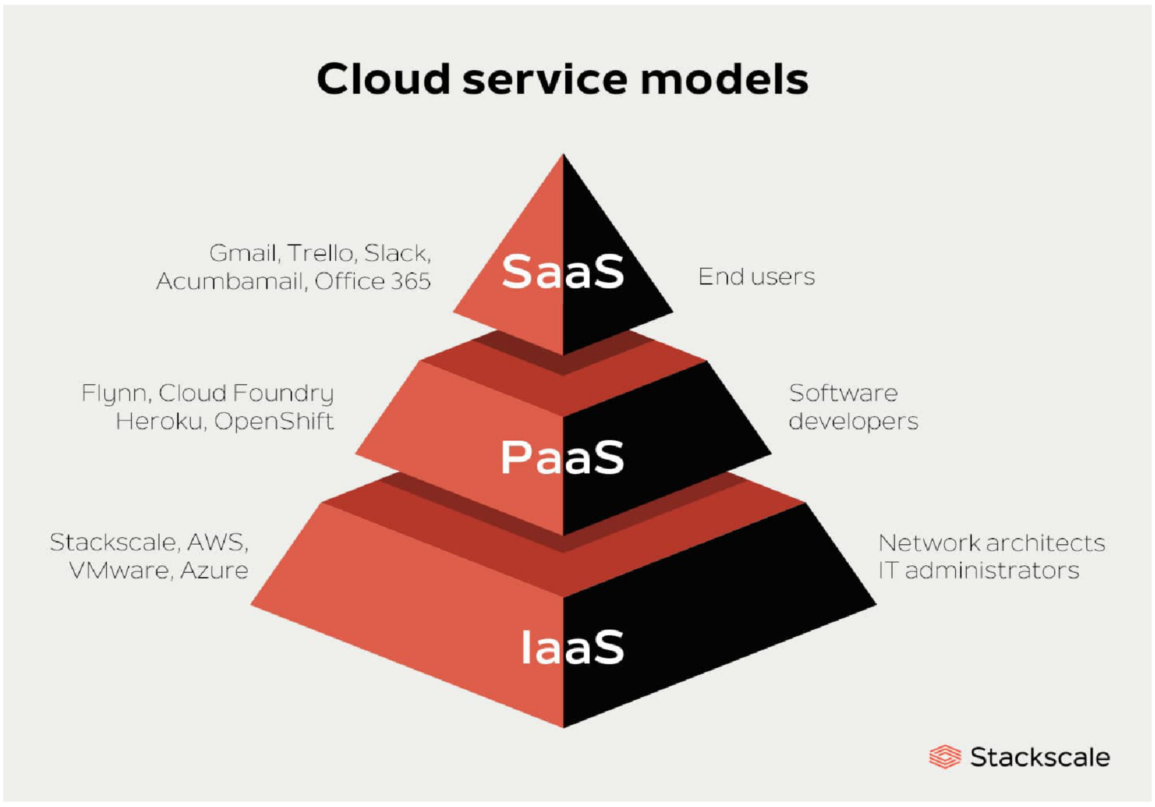

Cloud service models
Gmail, Trello, Slack, Acumbamail, Office 365
SaaS
End users
Flynn, Cloud Foundry Heroku, OpenShift
PaaS
Software developers
Stackscale, AWS, VMware, Azure
IaaS
Network architects IT administrators
Stackscale

Figure 2.5: Hierarchy of cloud service models https://www.stackscale.com/blog/cloud-service-models/

Function as a Service (FaaS)

Unlike BaaS, FaaS gives the developer much more control over the logic of an application. Developers write the code for functions that will run in the containers fully managed by a cloud provider. FaaS is an event-driven cloud computing model, so it is executed on demand.

Chapter 3

Tooling

This chapter will describe the tools I used to create the framework for the paid internet service.

3.1 Docker

Docker is written in the Go programming language and uses benefits provided by LXC (a reference to LXC) in particular namespaces and cgroups, to create an isolated workspace. Docker is an open platform for developing, shipping, and running applications. It allows bundling and executing a program with dependencies in a container, a loosely separated environment. Thanks to isolation and security, it is possible to run several containers on the same host simultaneously. Containers are lightweight and include everything needed for running an application, so users do not have to rely on what is already installed on the host. [4]

Architecture

The heart of Docker is Docker Engine, which uses the client-server architecture to work. There is a daemon called `dockerd` (server), which is a long-run process that manages containers. A user can communicate with it through Application Programming Interface (API) using `docker` (client) Command Line Interface (CLI).

Docker daemon

Docker daemon is a long-running process that manages the containers. Users interact with it primarily using Docker CLI. Docker daemon does all the hard work, meaning building, running, and pushing containers to the registry. It listens for Docker API requests and manages images, containers, networks, and volumes. To manage Docker services, it can also communicate with other Daemons.

Docker CLI

Docker client is the primary way to interact with the Docker daemon. When a user executes commands for building, running, pushing, and performing other operations over containers, it sends commands to the Docker daemon.

Registry

Docker registry is the storage for Docker images. For example, there is an official Docker registry called Docker Hub.

Images

A Docker image is a template with the instructions for creating a container. It may be based on other images and customized as needed. Users can assemble their images and use other publicly available images. Most popular software usually has its images provided by official developers through Docker Hub.

In order to build an image, build commands should be described in a Dockerfile with a simple syntax. Every instruction creates an additional cached layer. The rebuild of the layer occurs only when necessary.

Containers

A container is a sandboxed process, i.e., isolated from other processes on the host machine. It is a runnable instance of an image. It is possible to connect it to a network and attach storage. Also, a developer can create an image from its current state. Initially, containers are strongly isolated from the host machine and other containers, but it is easily configurable. More on containers can be found in Containerization 2.3 section.

3.2 TypeScript

Typescript is an open-source superset of the JavaScript language that extends it allowing static type checking. TypeScript increases code quality and readability, allowing scale applications more efficiently with all benefits of both statically-typed and dynamically-typed languages.

Type checking is done using variable type annotations. It can be skipped sometimes, for example, when the TypeScript can infer a variable type from a value assigned to it using its inferring mechanism. TypeScript compiles to the plain JavaScript for a production run. TypeScript can catch type errors at runtime and during the compilation process. [25]

3.3 Node.js

Node.js is an open-source, cross-platform JavaScript runtime designed to create network applications. It is built on Chrome's V8 engine and can be executed at the back-end. Node.js is asynchronous and event-driven, so it is easier to write Node.js applications than thread-based applications because no thread management is needed. Asynchronous applications use callbacks to notify about task execution results. Except for synchronous operations from the standard library, most operations are not blocked. [13]

Node Package Manager

Node Package Manager (npm) is a JavaScript package manager with millions of open-source packages. Many packages contain Node modules and can be easily used in Node application development. A typical npm package consists of the source code and metadata about the package stored in the file called `package.json`. It represents a module and stores information about its dependencies, entry file, and other metadata. [14]

3.4 Used packages

React

React is an open-source JavaScript framework for building User Interfaces (UI) created and maintained by Meta[1] and its community. The framework uses a component-based approach. Thus it is highly scalable and allows to keep the codebase logically divided. Since React is declarative, it drives all the work of updating and re-rendering the components, making the implementation of the interfaces much more effortless. [19]

JavaScript syntax extension

Moreover, React provides an extension called JavaScript Syntax Extension (JSX) that makes writing components more straightforward and more convenient. This extension allows inserting components HTML right in the JavaScript or TypeScript code. As a result, the code written on plain JS looks like this: [19]

```
const element = React.createElement("h1", null, "Hello, world!");
```

With JSX can be written as:

```
const element = <h1>Hello, world!</h1>;
```

React Router

React Router is a library for both client-side and server-side routing. It provides a convenient way to manage application routes on the client-side. In my thesis, it is used for client-side routing. [20]

Redux Toolkit

Redux is a state container for JavaScript applications. It brings the centralized state to the application, which allows to manage the data in the application conveniently. Redux Toolkit provides a layer above the Redux with the good defaults and the most commonly used Redux addons such as asynchronous thunk middleware built-in. [21]

Webpack

Webpack is a highly configurable module bundler. Initially, it bundles only JavaScript modules, but with plenty of plugins and loaders, it may also bundle the assets such as HTML, CSS, and the images.

Mocha

Mocha is a simple test library providing convenient tools for asynchronous application testing. It provides an ability to define test suites, hooks, and individual tests.

[1]Former Facebook.

3.5 Google Cloud Platform

Google Cloud Platform (GCP) is a set of cloud computing services supplied by Google. It is built on the same infrastructure that Google uses for its own SaaS 2.4 products such as Google Mail, Google Docs, Google Drive, and others. Google Cloud Platform provides IaaS, PaaS, and serverless computing opportunities.

Resources are assigned on a per-project basis. Every account on GCP can have multiple projects under control. Users can add editors and give them relevant, granular privileges: whether it is a project resource control permissions or access to the billing of the particular VM. Also, it allows the creation of service accounts, a special kind of account used by applications rather than people. Its permissions are also can be configured similarly. [18]

Almost all Google Cloud products are bound to particular regions and zone within the region. The zone is the so-called regional failure domain. It represents an underlying structure of physical resources, and zonal outages can affect some or all of the resources in the zone. A developer can choose where to locate services. Service price, latency, and availability depend on the location. In the work `europe` region was used to have minimal latency in Europe. [8]

Google Cloud Run

Google Cloud Run is a GCP service for running containerized applications. It provides a convenient Web user interface for setting up and deploying the container images in a few clicks.

Google Artifact Registry

In order to deploy a container image, it needs to be uploaded to a registry. Google provides the Artifact Registry service, which allows the creation of repositories for different artifacts and using them inside GCP infrastructure. Among available Artifact repository formats are Docker containers, Maven artifacts, npm, and Python packages.

3.6 Firebase

Firebase is an app development platform developed by Google. It provides tools such as authentication, non-relational databases, file storage, hosting, serverless cloud functions, and machine learning.

Firebase CLI

Firebase command-line interface provides tools for managing, configuring, and deploying Firebase projects. It also introduces an ability to download and run emulators of the almost entire Firebase infrastructure, including services such as Firestore, Storage, Realtime Database, Authentication, Functions, and more. [5]

Authentication

Firebase Authentication provides back-end services for user authentication. Authentication is performed with minimal effort from the developer side. It is enough to use the convenient SDK provided by Firebase. Users can be identified using a password, phone number, or multiple popular identity providers such as Google, Facebook, and Twitter. Moreover, Firebase Authentication seamlessly integrates with other Firebase services.

Firestore Database

Firestore Database is a real-time, non-relational (NoSQL) 2.2 cloud database. Unlike classical relational databases, data in this database is not stored in columns and rows but rather in the documents. Documents have a structure similar to JavaScript Object Notation (JSON) format with some extensions, e.g., in the case of Firestore Database, these extensions are map, timestamp, geopoint, and reference data types. The top-level data in Firestore Database is organized into collections. The collection itself does not hold the actual data, but it consists of documents. Documents contain fields mapping to values. The values can be either usual data types or nested collections. Data from the database is retrieved using the flexible querying system provided by SDK.

Storage

Firebase Storage is a cloud storage solution based on Google Cloud Storage. It is an object storage service that enables securely storing files in the cloud. Storage uses buckets for organizing and controlling access to data. A bucket is a primary container that holds files and directories but not other buckets. Files in the Firebase Storage are immutable, meaning that uploaded files cannot be modified later but can be replaced. Underneath, it uses Google Cloud Storage, which makes the buckets accessible from either of the services. Objects are individual pieces of data accompanied by object metadata specified by the user. Metadata is just a collection of key-value pairs.

3.7 Nginx

Nginx is an open-source web server that is one of the most popular on the market. It is very lightweight and fast. Also, it offers the functionality of reverse proxy, load balancer, mail proxy, and HTTP cache. [12]

Chapter 4

Architecture Design

This chapter describes service architecture design. First of all, it justifies the chosen technologies according to the thesis purpose. The purpose is to deploy an application with as low a cost as possible of running and storing the necessary resources. Then, it reviews the pricing of the chosen technologies and configuration considerations. It also covers user interaction with the service.

4.1 Service Architecture

Considering the service will run on the GCP, there is a broad functionality of the cloud services available. I analyzed possible solutions for running such a service in the cloud. The serverless architecture 2.5 approach was used. I identified cloud services that will be used in the solution.

Database

For a database NoSQL solution Cloud Firestore 3.6 was chosen. Its flexibility allows the creation of applications to be easier and faster. Also, it has natively implemented real-time updates notifications, which not only allows to provide a good user experience but also serves as an essential feature on top of which the workers could be implemented.

Storage

As for storage, object Firebase Storage 3.6 was picked. It offers convenient data access and different storage classes which could be potentially used for archiving older data, which minimizes costs.

Authentication

Firebase Authentication is a perfect cloud-based solution for the purpose of the application. It allows authenticating a user via multiple methods, as was mentioned in the section 3.6, catching up quickly with the rest of the services provided by Firebase. For example, it is actively used when defining the security rules for the Firestore and Firebase Storage.

Container images deployment

Then I had to decide which service to use to run the containers on the GCP. In general, there are several options:

- Compute Engine VM instance
- Google Kubernetes Engine (GKE)
- Cloud Run

Since the aim is to deploy a loosely coupled, scalable serverless application, Compute Engine VM instance was not a good option. The biggest con is that Compute Engine VM requires much manual configuration of the OS to run the containers. For example, it is essential to install Docker on it and implement a monitoring system for the containers or integrate it into them. When the new version of the application is out, updating the images on the VM is complicated. It is necessary to SSH into it, pull the latest images or changes from a repository with the source code and build it, and then manually start the updated versions.

Instead, GCP provides more convenient ways to deploy and run the containers such as GKE and Cloud Run. GKE is a powerful tool for deploying Kubernetes clusters to the cloud. Kubernetes is an orchestration tool for containerized applications. Unlike Compute Engine VM, in GKE, developers do not need to maintain the VMs used by Kubernetes and the software itself. Kubernetes seemed like a convenient choice for the application since it does all the job of deploying changes to production without downtime, managing and scaling the application containers and clusters, resource balancing, networking, and traffic management. Nevertheless, GKE clusters require a lot of configuration and maintenance, and it is not worth using them for this small application.

In this case, an ideal tool to use is Cloud Run. It is a fully managed serverless platform that allows creating services and deploying container images quickly and easily. Also, it provides configurations to control service scalability. It is enough to specify the desired image from Artifact Registry and perform basic configuration.

As a result, I came up with the model shown in figure 4.1.

Client application server

In the figure 4.1 on the right side, there is a service that represents the client application server. The client-server is very simple and is only serving the static client files since routing can be done on the client-side using React Router 3.4. The client application will be sent to the user's browser and perform calls to the different Firebase APIs. By virtue of serverless architecture, all other checks and validations usually performed by a server will happen in the cloud, and therefore no additional API exposed to the application is needed. The client service will be deployed as a separate service to the Cloud Run.

Worker

In the figure 4.1 in the bottom, there is a stack of workers. A worker is an application that will be listening to the database updates. It is on the worker which tasks it is able to process. It is assumed that each worker has responsibility for performing only one type of task, so it could be potentially scaled in the cloud when the CSP detects a spike in CPU

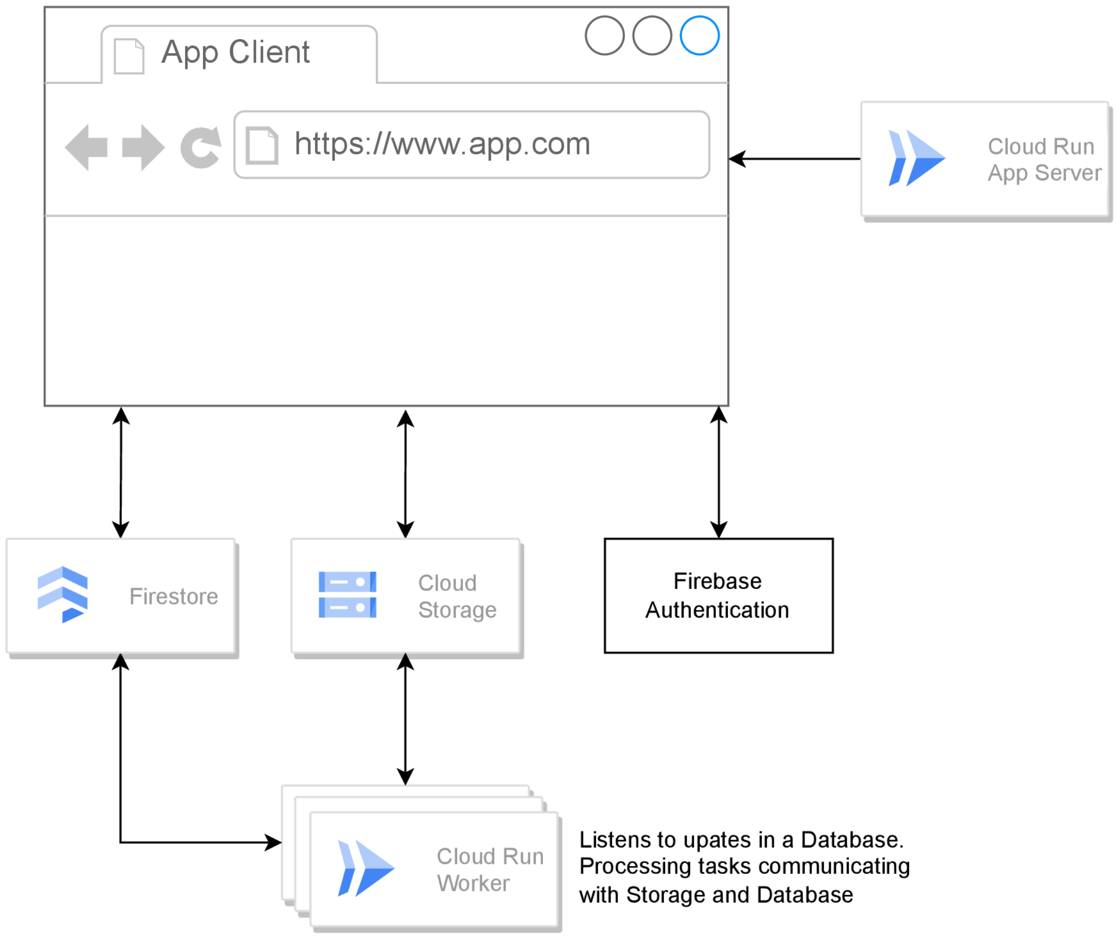


Figure 4.1: Service architecture

and memory consumption. The type of tasks it is processing will fetch the tasks from the database and/or storage to process them. Since several instances of the worker might be running simultaneously, the worker must lock the task and only then process it. Otherwise, other instances can start to process it too, and as a result, the task will be processed twice. The desired lock is similar to the mutex in Linux, which is used to lock shared resources in multithreaded programming to avoid inconsistency. The aspects of the implementation of the lock will be covered in the section 5.5.

4.2 Pricing conciderations

Artifact Registry

To at-rest stored images, Cloud Storage pricing is applied in the repository. Nonetheless, Artifact pricing also consists of network egress. While ingress is free, egress pricing is based on the premium network tier[1]. Egress to other GCP services is free if data moves within

[1]Network Service Tiers pricing https://cloud.google.com/network-tiers/pricing#premium-pricing

the same location or from regional to multi-regional GCP service within the same continent and vice-versa.

Cloud Run

Cloud Run charges for used resources such as CPU, memory, requests, and networking. CPU usage is tracked in vCPU-second units, memory in GiB-second units. Each resource has a free tier and a paid tier that depends on CPU allocation configuration and region. The free tier resource usage is shared between projects within one billing account and resets every month. There are two CPU allocation strategies: only during request processing and constant, so the CPU is always allocated. The first strategy offers smaller free-tier quotes and higher paid-tier costs. Oppositely, the second strategy offers bigger free-tier quotes and lower paid-tier costs. Also, it does not count requests. [2]

The containerized server that serves the front-end application introduced in this thesis only needs to respond to user requests. It is the same for the worker application, which listens to updates from the Firestore database and processes the tasks. Hence, for this small application, it is cheaper to use CPU allocation only during requests. The free tier for the chosen CPU allocation strategy includes:

- First 180,000 vCPU-seconds
- First 360,000 GiB-seconds
- 2 million requests
- 1 GiB free egress within North America per month

Resources used beyond these limits will be billed according to Cloud Run paid-tier pricing [2].

Firebase Storage

Firebase Storage is using Cloud Storage under the hood. As a result, Cloud Storage pricing is applied. It builds up from storage location, data processing, and network usage. There are different classes of storage, divided by access frequency. Among them are standard (the most expensive in storage but the cheapest in operations), nearline, coldline, and archive (the cheapest storage but the most expensive in operations). The data may be stored either in the region, dual-region, or multi-region storage. Storage buckets listing, data insert, read, update, and others are between data processing operations. The processing operations are also divided into classes with different operational fees. Network usage forms from egress, i.e., data read from Cloud Storage, and ingress, i.e., data written to Cloud Storage. [3]

However, Firebase has its own pricing for Cloud Storage which is different. The default bucket usage fees apply according to Cloud App Engine pricing [2], and only additional buckets are billed according to Cloud Storage pricing [3]. Firebase also offers a no-cost tier for its paid products. In the case of Firebase Cloud Storage, it includes 5GB of total storage and 1GB bandwidth per day.

[2] App Engine pricing https://cloud.google.com/appengine/pricing

Firestore

Firestore Database offers 1 GB stored in total, 10 GB bandwidth per month, 20,000 writes, 20,000 deletes, and 50,000 reads per day for free. For exceeded stored data, Firebase pricing [6] is applied, but for network egress and read, write, delete, operations, Google Cloud pricing[3] is applied.

Google Cloud provides a convenient pricing calculator[4], which allows estimating the price of selected services. It is possible to perform detailed configurations and calculate different services estimate altogether.

4.3 Database Structure

Tasks collection

To create a framework for task processing, it is necessary to abstract from what particular workers will perform. I have highlighted the following noticeable characteristics of tasks:

- Type – to divide the tasks between different workers
- Creation date – to process tasks queue
- Owner id – to bound a task to the owner account
- Status – so a worker could work on the task, and a user could see progress
- Processing time – to bill the user accordingly
- Task identifier

This data should be present in every task in the database, no matter which task type is. Now and later, I will call it service data. Even though an identifier is present in the service data, it is not stored in the service data but only added to the data during task processing to be available for convenience. Firestore will generate a task identifier as part of a task document metadata. To make the service data extensible in the future, a developer using the framework passes the desired name of the service field to the client function for tasks creating, which is described in section 5.3. This will help to avoid collisions in the naming inside the same document, and service data will have its place in the document as so:

```
// document /tasks/{taskId}
{
    serviceFieldName: {
        creationDate: timestamp,
        ownerUid: string,
        processTimeInMs: number,
        status: string,
        type: string
    },
    ...
    otherFields
```

[3]Google Cloud Firestore pricing https://firebase.google.com/pricing
[4]Google Cloud Pricing Calculator https://cloud.google.com/products/calculator

```
    ...
}
```

Users collection

It is not necessary to store much data about users authenticated via Google. All essential information, such as display user name, email address, profile picture, and user identifier, is defined by the provider and available through the Auth SDK. Thus, in the user document, primarily billing-related information will be stored. For demonstration purposes, there will be only two fields:

- User balance
- Invoices collection

Invoices collection in a collection of invoice data defined by the following fields:

- Creation date
- Payment rate defined in dollars per millisecond
- Total amount calculated using payment rate

The resulting document will look like follows:

```
// document /users/{userId}
{
    balanceInCents: number,
    invoice: { // document /invoices/{invoiceId}
        creationDate: timestamp,
        dollarsPerMs: number,
        total: number
    }
}
```

Security rules requirements

One of the most critical things in the application is its security. Unlike relational databases, which have strictly defined type constraints for the columns, the user authorization, and inserted values validation is performed on the server. It is different in a serverless Firebase application since Firestore has its mechanism for securing and validating data right in the cloud. A configuration file called Firestore rules allows applying constraints to the document writing and reading. The rules can be very flexible, and it is possible to apply the rules to different collections separately. I followed the principle of least privileged, which means a user has privileges only necessary to use the application and no more. This principle is implemented bottom-up: a user has no permissions initially. Then, when it is required for a user to have some privileges, they are granted.

In the case of tasks, it is necessary to ensure a user can:

- Create task only if the user is authorized

- Create task containing only service field
- Service field discussed earlier 4.3 must contain only part of the service data, such as
  1. Owner identifier
  2. Type
  3. Status
  4. Creation date
- Service data fields must have the corresponding types
- Owner user identifier must correspond to the one in the request
- Created task must be in the `queued` state

As for the users' collection, users should not be able to define their balance and invoices. A payment system callback will invoke a cloud function, which will update the values in the database. The implementation of such a function is not a part of this thesis.

4.4 Payment System

Considering the usage of GCP services for running containers, it should be possible to estimate the pricing of the services relying on the Cloud Run pricing system described in the section 4.2. Thus it is possible to hardcode price for computations or evaluate it dynamically depending on the worker configuration. When performing a task, it is possible to measure used resources and store this information along with task data and the configuration of an instance used to process a task. Then, the processed task price can be calculated, and the user can be billed accordingly.

Chapter 5

Implementation

This chapter covers how the demo client application, worker, and framework libraries were developed. The framework covers some routines in processing tasks by providing client and server libraries. Also, the demo application shows how security rules are applied to the database in order to secure the application based on the framework. The entire solution is written in TypeScript using the Node.js framework, UI libraries, and SDK libraries for communication with Firebase services.

For demonstration purposes, the worker is accepting a file submitted by the user with the integers separated by commas. The worker extracts the numbers, computes their sum, and writes the result to the output file. This also allows demonstrating the work with the storage, which is usually a part of almost every application.

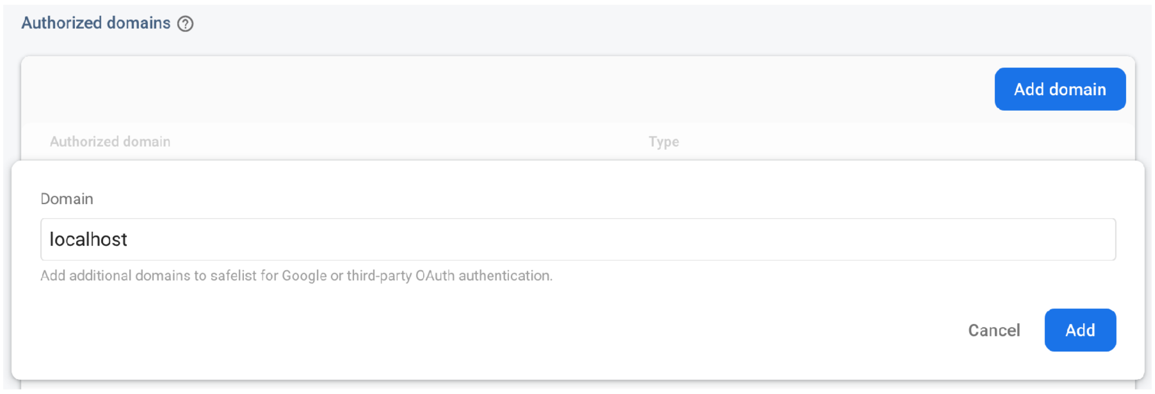
5.1 Preparing Firebase

Google Cloud projects are the foundation for creating, managing and using Google Cloud services. Projects also make per-project billing possible, adding collaborators, governing services permissions, and sharing resources. For this thesis, the project with the `vut-bachelor-thesis` identifier was created. It will appear in the different URLs and service settings, so it is good to keep it in mind while working on a project.

There are several ways to start with Firebase. The desired project from GCP is imported to the Firebase or created directly in the Firebase. When a project is created, it forms the Firebase config, which is used by the client application for communication with the Firebase services. After that, activation of all required features such as Authentication, Firestore, and Storage becomes available. The connection of the client application will be discussed later in the section 5.2.

Firebase Authentication

The domain must be added to the Firebase authorized domains list to perform authentication on either the development or production domain. Development domain, i.e., `localhost` should be added to the authorized domains in the Firebase Authentication as shown in figure 5.1.



Domain

localhost

Add additional domains to safelist for Google or third-party OAuth authentication.

Figure 5.1: Authorizing Firebase Auth domain

Emulated development environment

Since this thesis aims to introduce the cheapest way to run the service in the cloud, the solution also includes a fully prepared development environment. Firebase CLI, among other features, allows for installing and running emulated Firebase services locally. Using an emulator can reduce the costs of two following aspects during application development. Firstly, it allows a developer not to be concerned about wasting too many free or paid requests to Firestore. Also, it leaves out uploading unnecessary data or data above the free limit to storage. Secondly, it prevents the things mentioned above while testing an application. Most importantly, it eliminates the possibility of breaking a production database when tests go wrong and occasionally wipes all the data in the database. Thus, the following instructions for building images, including the Firebase emulator, were created:

```
FROM node:14-alpine
WORKDIR /app

RUN apk add --no-cache --update openjdk11-jre bash libintl && \
    apk add --virtual build_deps gettext && \
    cp /usr/bin/envsubst /usr/local/bin/envsubst && \
    apk del build_deps && \
    npm i -g firebase-tools

CMD ["./docker-entrypoint.sh"]
```

/test/Dockerfile.dev

It will install the platform for running Java, which is required to run Firebase CLI, the Firebase CLI, and `envsubst` – utility for populating file templates with environment variables. At container startup, it will run script `/test/docker-entrypoint.sh`, thus substitute the variables from `/test/firebase.JSON.template`, install all emulators specified there and run them on the custom ports defined in the `.env`:

```
...
FIREBASE_EMULATOR_UI_PORT=7777
FIRESTORE_EMULATOR_PORT=7000
FIREBASE_STORAGE_EMULATOR_PORT=7001
FIREBASE_AUTH_EMULATOR_PORT=7002
```

```
FIREBASE_EMULATOR_HUB_PORT=4400
FIREBASE_EMULATOR_LOG_PORT=4500
...
```

/variables.env

The emulator also must get the `FIREBASE_TOKEN` variable containing the authorization token. Firebase authorization token can be obtained using `firebase login` and `firebase login:ci` consequently. This variable should be added to `.env`. This file is not included in appendices files since it contains sensitive credential information.

5.2 Client Application

Client application requirements

A typical user should be able to:

- Log in
- Submit a computational task
- List their previously submitted tasks
- List invoices for the processed

The client application will cover this functionality. As already mentioned, internet services use a browser as an application client. So the client should provide a GUI to the user and the ability to submit the computational task to the service.

User interface

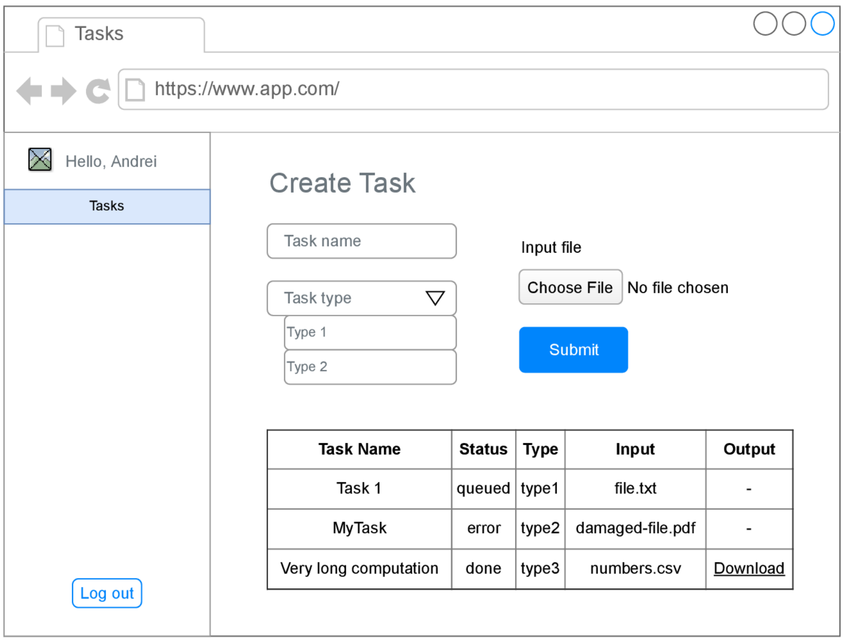
Firstly, according to the requirements, the client UI was mocked. The viewport is divided into a sidebar and content area. The sidebar consists of information about the logged-in user and the navigation menu. The content area holds information relevant to the current page.

Figure 5.2 shows a page with the form for submitting tasks to the database. It covers all necessary service fields discussed in section 4.3, which user should specify, i.e., only task type. The other fields, such as user identifier, initial task status, and creation date, could be derived by the client library itself. Also, it introduces a couple of demo-specific inputs, such as the name of the task and file input.

Also, there is a table that will hold information about previously submitted tasks and their statuses. If a worker has already started working on a task or the task is already done, information will be automatically updated and displayed here without the need to reload the page. As a part of the payment system concept, the mockup of the invoices page was implemented. It shows the user their balance and processed tasks with its price and invoice status.

Routes

The implementation of the server only for the sake of routing was left out in this serverless application. Thanks to the React Router package, routing can be conveniently performed



Task Name	Status	Type	Input	Output
Task 1	queued	type1	file.txt	-
MyTask	error	type2	damaged-file.pdf	-
Very long computation	done	type3	numbers.csv	Download

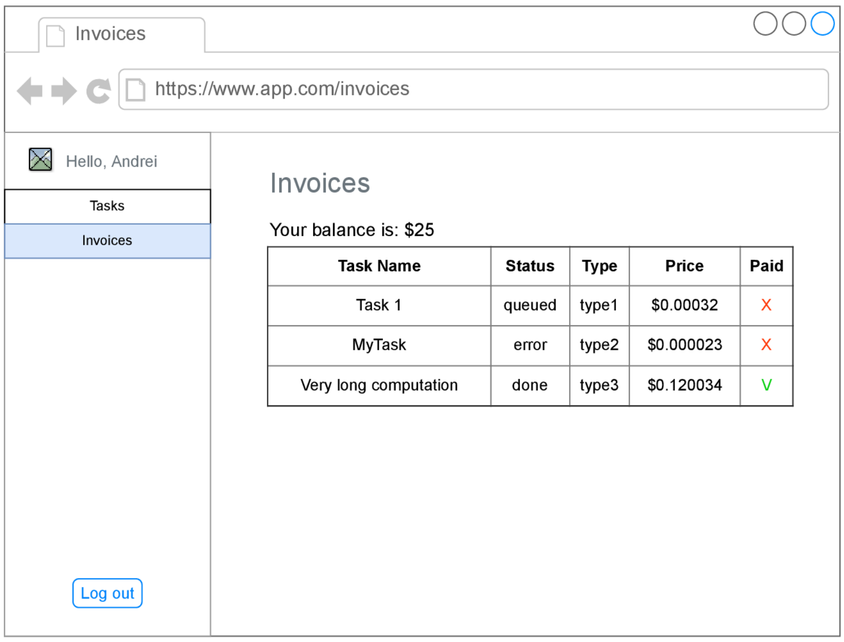
Figure 5.2: Mockup of the Login and Tasks pages

right on the client-side, making it easy to create a single-page application (SPA). SPA loads just a single HTML page file and updates the content of that page dynamically using JavaScript. JavaScript fetches content parts by using. Thus, users do not need to load the whole page every time they redirect to another site page server. The application has several routes:

- `/login` – login form for unauthorized users
- `/` – root containing tasks dashboard along with the form if user is authorized
- `/invoices` – invoices overview page

Which are defined in the applicatio in the following way:

```
...
<Routes>
    <Route
        path="/" element={
            <RequireAuth>
                <Page title={'Tasks'} component={<TasksPage />} />
            </RequireAuth>
        }
    />
    <Route
        path="/invoices"
        element={
            <RequireAuth>
                <Page title={'Billing'} component={<InvoicesPage/>}/>
```



Task Name	Status	Type	Price	Paid
Task 1	queued	type1	$0.00032	X
MyTask	error	type2	$0.000023	X
Very long computation	done	type3	$0.120034	V

Figure 5.3: Mockup of the Invoices page

```
            </RequireAuth>
        }
    />
    <Route path="/billing"
        element={
            <RequireAuth>
                <Page title={'Billing'} component={<TasksPage />} />
            </RequireAuth>
        }
    />
    <Route path="/login" element={<LoginPage/>}/>
</Routes>
...
```

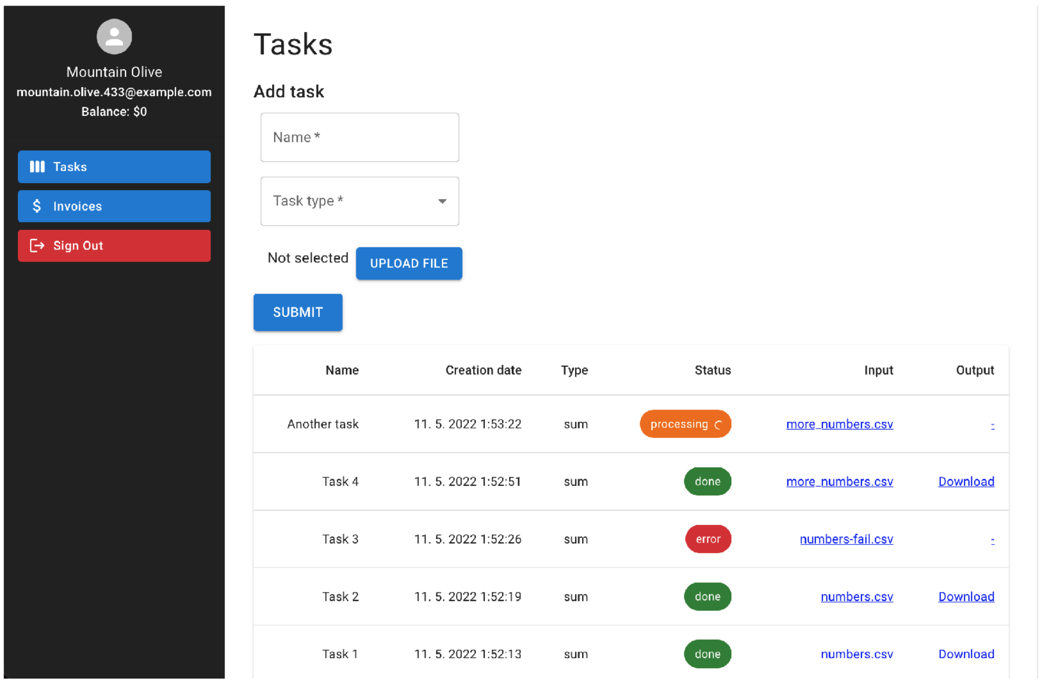
/app/src/App.tsx

The resulting user interface of tasks and invoices pages is shown in Figure 5.4 and Figure 5.5 respectively.

Connecting application to Firebase

Firebase config have to be passed to the Firebase SDK application initializer. Initialization of the Firebase application is performed as follows:

```
import { initializeApp } from 'firebase/app'
import { getFirestore, connectFirestoreEmulator } from 'firebase/firestore'
import { getStorage, connectStorageEmulator } from 'firebase/storage'
import { getAuth, connectAuthEmulator } from 'firebase/auth'
import config from './config'
```



Name	Creation date	Type	Status	Input	Output
Another task	11. 5. 2022 1:53:22	sum	processing	more_numbers.csv	-
Task 4	11. 5. 2022 1:52:51	sum	done	more_numbers.csv	Download
Task 3	11. 5. 2022 1:52:26	sum	error	numbers-fail.csv	-
Task 2	11. 5. 2022 1:52:19	sum	done	numbers.csv	Download
Task 1	11. 5. 2022 1:52:13	sum	done	numbers.csv	Download

Figure 5.4: UI of the tasks dashboard

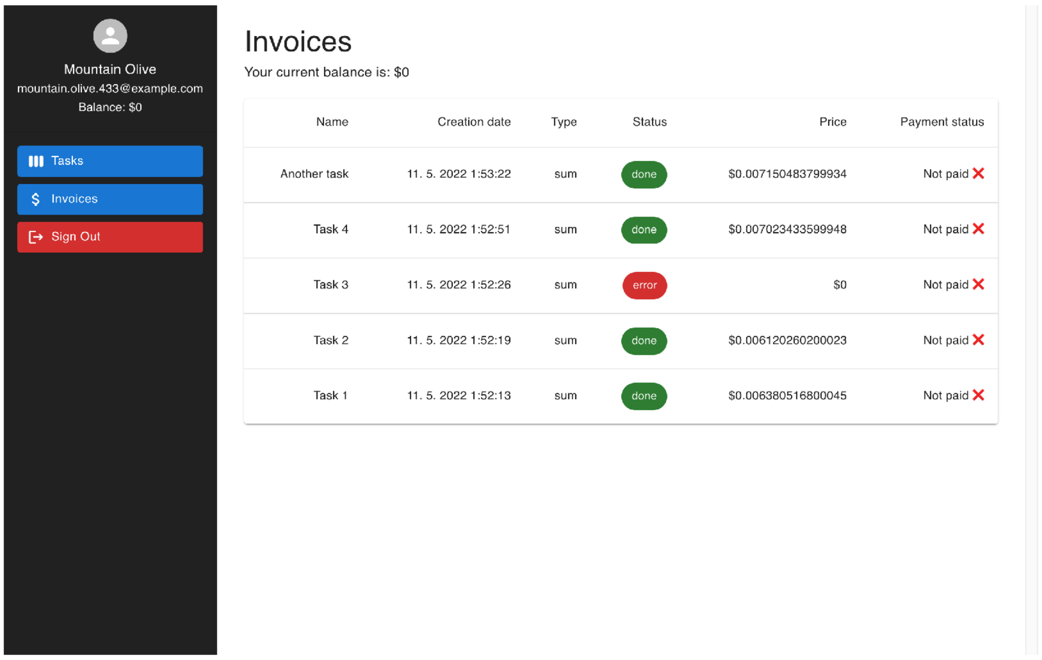
```
export const firebaseConfig = {
    apiKey: 'AIzaSyBS3WhhMEQs2sjEqA94BAsnECEoqe-1l1Q',
    authDomain: 'vut-bachelor-thesis.firebaseapp.com',
    projectId: 'vut-bachelor-thesis',
    storageBucket: 'vut-bachelor-thesis.appspot.com',
    messagingSenderId: '328686553325',
    appId: '1:328686553325:web:1863d1693dfd89048124e4',
    measurementId: 'G-W7ZG3NPH5Y',
}

export const app = initializeApp(firebaseConfig)
export const auth = getAuth(app)
export const db = getFirestore(app)
export const storage = getStorage(app)
...
```

/app/src/utils/firebase.ts

After that, all necessary instances are ready to work with. In order to make the instances work with emulator instead of production Firebase services, special functions from SDK `connectAuthEmulator`, `connectFirestoreEmulator`, and `connectStorageEmulator` should be called as shown below:

```
...
if (config.isDev) {
    connectAuthEmulator(
        auth,
```



Invoices

Your current balance is: $0

Name	Creation date	Type	Status	Price	Payment status
Another task	11. 5. 2022 1:53:22	sum	done	$0.007150483799934	Not paid ✗
Task 4	11. 5. 2022 1:52:51	sum	done	$0.007023433599948	Not paid ✗
Task 3	11. 5. 2022 1:52:26	sum	error	$0	Not paid ✗
Task 2	11. 5. 2022 1:52:19	sum	done	$0.006120260200023	Not paid ✗
Task 1	11. 5. 2022 1:52:13	sum	done	$0.006380516800045	Not paid ✗

Figure 5.5: UI of the invoices dashboard

```
            `http://localhost:${config.firebaseAuthEmulatorPort}`,
            {
                disableWarnings: true
            }
        )
        connectFirestoreEmulator(
            db,
            'localhost',
            config.firestoreEmulatorPort,
        )
        connectStorageEmulator(
            storage,
            'localhost',
            config.firebaseStorageEmulatorPort,
        )
}
..
```

/app/src/utils/firebase.ts

Assuming the following `config`, filled with environment variables which was discussed earlier in the section 5.1:

```
...
const config = {
    isDev: process.env.NODE_ENV === 'development',
    serviceField: '_taskerService',
    taskCollectionPath: 'tasks',
    firestoreEmulatorPort: parseInt(process.env.FIRESTORE_EMULATOR_PORT ||
        '7000', 10),
    firebaseStorageEmulatorPort: parseInt(process.env.
        FIREBASE_STORAGE_EMULATOR_PORT || '7001', 10),
    firebaseAuthEmulatorPort: parseInt(process.env.
        FIREBASE_AUTH_EMULATOR_PORT || '7002', 10),
}
..
```

/app/src/utils/config.ts

Since the browser does not have the `process` variable because it is defined by Node.js when running scripts on the back-end, the Webpack plugin is used to inject it to the client during the build as so:

```
...
    plugins: [
        ...
        new DefinePlugin({
            'process.env.FIRESTORE_EMULATOR_PORT': JSON.stringify(process.
                env.FIRESTORE_EMULATOR_PORT),
            'process.env.FIREBASE_STORAGE_EMULATOR_PORT': JSON.stringify(
                process.env.FIREBASE_STORAGE_EMULATOR_PORT),
            'process.env.FIREBASE_AUTH_EMULATOR_PORT': JSON.stringify(
                process.env.FIREBASE_AUTH_EMULATOR_PORT),
        }),
    ],
...
```

/app/webpack.common.js

The client application production Dockerfile consists of two simple parts. Firstly, it will bundle the application TypeScript using Webpack, transpiling it to the plain JavaScript using `node` image as a builder:

```
FROM node:14-alpine as builder

# create app directory
WORKDIR /app

# install dependencies
COPY package*.json ./
RUN npm i
COPY . .
RUN npm run build
```

/app/Dockerfile

Then it copies the production ready code and nginx config to the `nginx` image:

```
FROM nginx:alpine
COPY --from=builder /app/dist /usr/share/nginx/html
```

```
# copy nginx config template
COPY ./nginx/nginx.conf.template /etc/nginx/templates/nginx.conf.template
RUN rm /etc/nginx/conf.d/default.conf

EXPOSE $PORT
```

/app/Dockerfile

The Nginx config defines the root location, which will always fall back to `index.html`, in order to client-side navigation may work properly:

```
server {
  listen ${PORT};

  add_header Access-Control-Allow-Origin *;

  location / {
    root /usr/share/nginx/html;
    index index.html index.htm;
    try_files $uri $uri/ /index.html;
  }

  error_page 500 502 503 504 /50x.html;

  location = /50x.html {
    root /usr/share/nginx/html;
  }
}
```

/app/nginx/nginx.config.template

5.3 Client library

The client library consists of two main classes. The `TaskerClient` class represents a client, and a `Task` class, which has not been used directly and always will be filled by `TaskerClient`. Once the application is initialized, the `TaskerClient` is instantiated in the following way:

```
...
constructor(auth: Auth, serviceField: string) {
    this.auth = auth
    this.serviceField = serviceField
}
...
```

/app/src/utils/tasker/TaskerClient.ts

It accepts two parameters:

1. Application Firebase Auth instance which was obtained in the section 5.2
2. Chosen name of the service field

After initialization, the client is ready to create and list tasks in the Firestore.

Creating a task

The function `createTask` can be called on the `TaskerClient` instance in order to create a task. Here is its signature:

```
...
async createTask<T extends DocumentData>(
    type: string,
    collectionRef: CollectionReference,
    data: T,
): Promise<Task<T>>
...
```

/app/src/utils/tasker/TaskerClient.ts

The parameters are:

1. Task type to create
2. Reference to the tasks collection
3. User defined data to store in the task

When creating the task document, the Tasker under the hood merges all necessary service data to the provided data and wraps the data into `Task` instance.

Getting tasks

Task client is used for getting tasks as follows:

```
...
async getTasks<T>(
    collectionRef: CollectionReference,
    ...queryConstraints: QueryConstraint[]
): Promise<Task<T>[]>
...
```

/app/src/utils/tasker/TaskerClient.ts

As a parameters it accepts:

1. Reference to the tasks collection
2. Query constraints like `orderBy`, `where`, `limit`, and others supported by Firestore SDK

At this moment, under the hood, the tasker will check if the current user is authorized on the client-side, add a query constraint `where` for the field `ownerUid` with the comparison between the current user id and the user id stored in the document. Then it returns an array of docs wrapped into `Task` instances.

As for getting one instance, everything is way easier:

```
async getTask<T>(taskRef: DocumentReference<T>): Promise<Task<T>>
```

/app/src/utils/tasker/TaskerClient.ts

It accepts task document reference, gets it, and wraps it into `Taks` instance as always.

5.4 Worker

Each worker type should only implement the function that will handle whatever input will be provided from the task document. As was already mentioned, in this case, for demonstration purposes, it will process files uploaded by users.

Connecting worker to Firebase

Unlike the client Firebase SDK, `firebase-admin` does not accept Firebase application configuration. Instead, it accepts a Google Cloud service account credentials, which has an administrator role for the Firebase services. Service account information is stored in the JSON file, and this file is passed to the initializer. In this case, the storage bucket name is also provided to the initializer, as this demonstration worker will need to access Firebase storage.

By default, the worker is connected to the production Firebase services. In order to change that, special environmental variables must be present. These variables are defined in the `variables.env` discussed earlier:

```
...
FIRESTORE_EMULATOR_PORT=7000
FIREBASE_STORAGE_EMULATOR_PORT=7001
...
FIRESTORE_EMULATOR_HOST=firebase_emulator:$FIRESTORE_EMULATOR_PORT
FIREBASE_STORAGE_EMULATOR_HOST=firebase_emulator:
    $FIREBASE_STORAGE_EMULATOR_PORT
...
```

/variables.env

If these variables are present, the worker will be connected to the local emulator.

Even though the worker is supposed to receive events by subscribing to the Firestore updates, it is also running the server to receive Cloud Run health check HTTP requests. Otherwise, it would not be possible for Cloud Run to know if the instance needs to be restarted.

The worker container definition is the following. In the first step, the builder container builds the bundle in the environment with the development dependencies:

```
FROM node:14-alpine as builder

WORKDIR /app

# install dependencies
COPY ./package*.json ./
RUN npm i
# bundle app source
COPY . .
RUN npm run build
```

/sum-worker/Dockerfile

Then it transfers the resulting bundle to the clean `node:14-alpine` image and installs the production dependencies. When run, the container will execute the bundle using node:

```
FROM node:14-alpine
WORKDIR /app

COPY ./package*.json ./
RUN npm ci --only=production
COPY --from=builder /app/dist /app/dist

CMD ["node", "./dist/bundle.js"]
```
/sum-worker/Dockerfile

5.5 Worker library

The worker library, as well as the client library, consists of two classes. These classes are `Tasker` for the tasks handling, and `Task` for wrapping the individual documents for convenience as it does `TaskerClient`. The implementation differs from the client library, since the `firebase-admin` SDK is used instead of `firebase` client SDK. Also, while `TaskerClient` for the front-end has the purpose of convenient creation and retrieval of tasks, `Tasker` for the back-end worker does not do that. Instead, it is focused on subscription to the updates in the databases, obtaining a lock on the document so that other workers do not start to process it, measuring execution time, and providing the ability to execute different lifecycle callbacks.

Before processing tasks, it is necessary to instantiate `Tasker` as well as on the client. In the case of the worker library constructor awaits the following:

```
constructor(firestore: Firestore, type: string, serviceField: string,
    handler: TaskHandler<T>)
```
/sum-worker/src/tasker/Tasker.ts

1. Firestore instance
2. Type of the tasks this worker will process
3. Service field name
4. Handler function. The result of the handler function is the document data that should be merged into the document

Then `Tasker` is ready to handle the query. To listen on the particular queue, the function `listenQueue` is used:

```
async listenQueue(query: Query<T>): Promise<Function>
```
/sum-worker/src/tasker/Tasker.ts

The query here is library user-defined query including constraints such as `where`, `orderBy` and others defined by the SDK. The `listenQueue` function works as follows:

1. Subscribes on the updates of the given collection by applying the following conditions:
    - Task status should be equal to the `queued`

    - Task type should correspond to the worker type
    - Limit to the 1 document only

2. When the event is fired, a document lock is obtained by creating a document in `locks` collection
3. The document wrapped into `Task` passed to the handler function
4. If the task is successfully processed, then the result is merged to the document with status `done`
5. Otherwise, document status is set to `error`, and it is not processed
6. In both cases document representing the lock is deleted from the database

The most important thing here is obtaining the lock using the property of Firebase `create` operation. When create operation is performed on the existing document, it fails. Therefore it indicates whether the task is already in processing or not. If the document is already in processing by another worker, Tasker will skip it. If the document is not locked, Tasker will lock it and set its status to `processing`.

Also, Tasker supports several lifecycle callbacks. Among them are `before`, `after` and `onError`. The callbacks are fired right before a task is passed to the handler function, right after, and if an error occurred while processed by the handler function respectively. The task wrapped into `Task` object is passed to every callback. In the case of `onError` callback, the error is passed as a second argument, so the worker can do something according to the error type.

5.6 Firebase

Defining Security Rules

The security rules requirements discussed in the section 4.3 can be deployed from the `/test` project folder using Firebase CLI using `firebase deploy` command. Alternatively, it can be written directly on the Firebase web. Following mentioned earlier least privileged principle, the rules for the `tasks` collection should be defined as follows:

```
...
    match /tasks/{taskId} {
      allow create: if
        request.auth != null &&
        request.auth.uid == request.resource.data._taskerService.ownerUid &&

        // field keys
        request.resource.data.keys().hasOnly(['_taskerService', '
            inputFilePath', 'name']) &&
        request.resource.data._taskerService.keys().hasOnly(['ownerUid', '
            type', 'status', 'creationDate']) &&

        // service data types
        request.resource.data._taskerService.status is string &&
        request.resource.data._taskerService.creationDate is timestamp &&
        request.resource.data._taskerService.type is string &&

        // service values
      request.resource.data._taskerService.status == 'queued' &&

        // app types
        request.resource.data.inputFilePath is string &&
        request.resource.data.name is string;

      allow read: if
        request.auth != null &&
        request.auth.uid == resource.data._taskerService.ownerUid;
    }
...
```

/test/firestore.rules

And the rules for the `users` collection:

```
match /users/{userId} {
      allow create: if
        request.auth != null &&
        request.auth.uid == request.resource.data.uid;

      allow read: if
        request.auth != null &&
        request.auth.uid == userId;

      match /invoices/{invoiceId} {
        allow read: if
          request.auth != null &&
          request.auth.uid == userId;
      }
    }
```

/test/firestore.rules

Now, users' privileges fully correspond with the requirements, and the application can be securely exposed to the actual users. Also, user needs privileges to upload and download files from the Storage. The rules for the storage are defined as follows:

```
rules_version = '2';
service firebase.storage {
  match /b/{bucket}/o {
    match /tasks/{userId}/{taskInputFile} {
      allow create: if
        request.auth != null &&
        request.auth.uid == userId &&
        request.resource.contentType.matches('text/csv');

      allow read: if
```

```
        request.auth != null &&
        request.auth.uid == userId;
    }
  }
}
```

/test/storage.rules

Chapter 6

Deploying Demo Services to the Google Cloud Platform

This chapter provides an overview of experience deploying a dockerized application to the Google Cloud Platform services. Firstly, it covers uploading the images to the Artifact Registry. Then it explains the uploaded image deployment to the Cloud Run. Also, it introduces Continuous Integration and Continuous Development (CI/CD) with the Cloud Build.

GCP provides highly configurable services, and there are several ways to interact with them. The most convenient way is to use the Google Cloud Console GUI and specify all settings using it. Most of the time, it is convenient to use Cloud Console. However, sometimes it is valuable or even necessary to use Google Cloud Command Line Interface (CLI) `gcloud`. For example, when creating a CI/CD pipeline, the building steps are using Google Cloud CLI. [7]

6.1 Storing Containers in Artifact Registry

After the application is dockerized, it is ready to be deployed to the GCP. To deploy an application to the GCP, it is necessary to push the images to the GCP using Artifact Registry or to Container Registry, which provides only a subset of Artifact Registry features. Artifact Registry allows storing not only Docker images 3.5. The first step is to create a Docker repository in the Artifact Registry. I have chosen `ibp` name for the repository, location `europe-west4 (Netherlands)` and Google-managed encryption key.

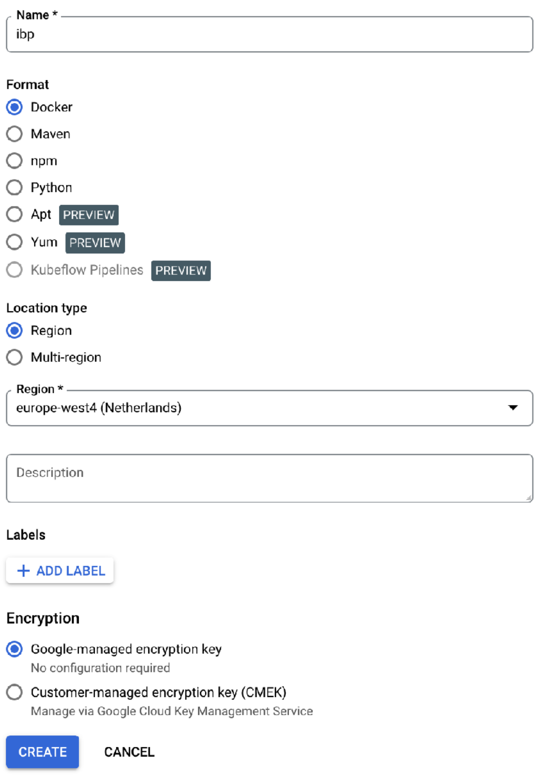
Then the repository is ready for uploading the docker images. A repository address template can be represented as a template:

```
LOCATION-docker.pkg.dev/PROJECT/REPOSITORY_NAME
```

Where `LOCATION` is a repository location, `PROJECT` is a project name, and `REPOSITORY_NAME` is the name of a repository. So my repository has an address `europe-west4-docker.pkg.dev/vut-bachelor-thesis/ibp`. To push the images from the local machine, I used the option of configuring `gcloud` as a credential helper for the repository.

```
$ gcloud auth configure-docker europe-west4-docker.pkg.dev
```

Then it becomes possible to push an image to the repository using `docker push` command. In order to do that, there should be a local image on the machine tagged in the following way `europe-west4-docker.pkg.dev/vut-bachelor-thesis/ibp/IMAGE:TAG` where


Name *
ibp

Format
[✓ Docker]
Maven
npm
Python
Apt PREVIEW
Yum PREVIEW
Kubeflow Pipelines PREVIEW

Location type
[✓ Region]
Multi-region

Region *
europe-west4 (Netherlands)

Description

Labels

Encryption
[✓ Google-managed encryption key]
No configuration required
Customer-managed encryption key (CMEK)
Manage via Google Cloud Key Management Service

Figure 6.1: Creating an Artifact Registry Repository

`IMAGE` is the image name and `TAG` is a version tag of the image. Omitting the `:TAG` part will automatically tag it as `latest`, as it normally occurs when pushing Docker containers. I have decided to use names `app` for the demo application and `sum-worker` for the worker. [17]

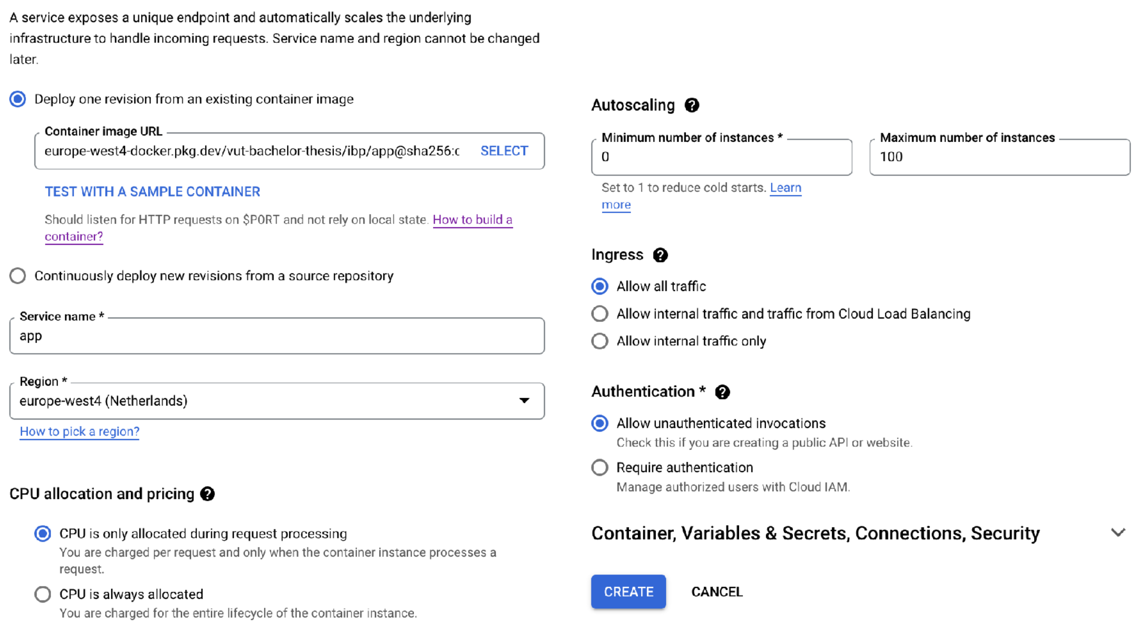
6.2 Deploying Services to Cloud Run

For a start, I have chosen to deploy the latest image revisions of the `app` application front-end and `sum-worker` worker to the `europe-west4` region, allocating CPU only during request processing. I left the default Autoscaling setting from 0 to a maximum number of 100 instances on both of the services.

As for ingress, settings for `app` and `sum-worker` are different. `app` must allow all unauthenticated traffic in order to get the incoming requests and serve the client front-end. `sum-worker` on the contrary, should not allow any incoming traffic since it is only a worker, and it will listen to changes in the Cloud Firestore database and react to it.

The worker must have specified environmental variable `STORAGE_BUCKET` to indicate a Firebase Storage bucket it works with. Also, it must have access to the `firebase-admin` SDK through the service account created earlier. So I specified all needed variables and secrets in the corresponding tab as shown in figure 6.3.

As soon as a container is deployed to the service successfully, it gets a unique URL. Services are automatically served through HTTPS with a redirect from HTTP to HTTPS. Their URLs have the following structure:



A service exposes a unique endpoint and automatically scales the underlying infrastructure to handle incoming requests. Service name and region cannot be changed later.

[✓ Deploy one revision from an existing container image]

Container image URL: europe-west4-docker.pkg.dev/vut-bachelor-thesis/ibp/app@sha256:c SELECT

TEST WITH A SAMPLE CONTAINER

Should listen for HTTP requests on $PORT and not rely on local state. How to build a container?

Continuously deploy new revisions from a source repository

Service name *: app

Region *: europe-west4 (Netherlands)

How to pick a region?

CPU allocation and pricing

[✓ CPU is only allocated during request processing]
You are charged per request and only when the container instance processes a request.

CPU is always allocated
You are charged for the entire lifecycle of the container instance.

Autoscaling

Minimum number of instances *: 0

Maximum number of instances: 100

Set to 1 to reduce cold starts. Learn more

Ingress

[✓ Allow all traffic]

Allow internal traffic and traffic from Cloud Load Balancing

Allow internal traffic only

Authentication *

[✓ Allow unauthenticated invocations]
Check this if you are creating a public API or website.

Require authentication
Manage authorized users with Cloud IAM.

Container, Variables & Secrets, Connections, Security

CREATE CANCEL

Figure 6.2: Creating a Service in Cloud Run

`https://APP-PROJ_HASH-REGION_SHORTCUT.run.app`

In the case of the `app` service it got the `https://app-gkeo4mmi5a-ez.a.run.app/` URL. As when the app was developed on the `localhost`, I allowed the new domain in the Firebase Auth configuration panel. After these steps the `app` and `sum-worker` are deployed to the cloud and fully functioning.

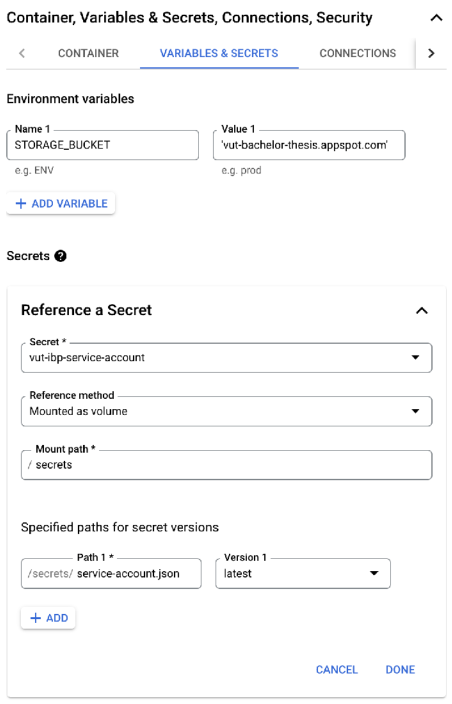
6.3 Configuring CI/CD with Cloud Build

The Google Cloud Build is a serverless CI/CD solution for automating project build and deployment. Source code may be imported via Cloud Storage, Cloud Resource Repositories, Github, and Bitbucket. Cloud Build runs the build as a sequence of user-configured steps, each running in its own Docker container. [10]

Builds are based on build configuration `cloudbuild.yml`. It can be present either in the repository or added manually in the Cloud Console. To prepare a repository for build automation, a Cloud Build trigger should be created. The trigger is associated with a source code repository, and it allows to choose an event trigger that will invoke the build. It can be a pull, push, or pull request to any branch, webhook, or manual invocation. Also, it allows specifying the substitution variables for the `cloudbuild.yml` file, which work similarly to environment variables.

At first sight, it is cheaper not to use Cloud Build for deploying the containers because it is also billed. Although there are 120 minutes of free builds, it can be many builds if used right, and it makes sense to use it at least before 120 minutes is spent.

In this demonstration example, two services need to be built and deployed – the demonstration client application and the worker. It is convenient to create two different configs and two different triggers because it will help eliminate unnecessary builds when nothing is changed. Also, it will help not to waste build time if one of the builds is failed because


Container, Variables & Secrets, Connections, Security

Environment variables

Name 1: STORAGE_BUCKET (e.g. ENV)

Value 1: 'vut-bachelor-thesis.appspot.com' (e.g. prod)

Secrets

Reference a Secret

Secret *: vut-ibp-service-account

Reference method: Mounted as volume

Mount path *: / secrets

Specified paths for secret versions

Path 1 *: /secrets/ service-account.json

Version 1: latest

Figure 6.3: Defining Variables and Secrets for the Container

then the whole build is terminated, and nothing is deployed. So they could be deployed separately, two similar build configurations were created:

```
steps:
  - name: 'gcr.io/cloud-builders/docker'
    args: [ 'build', '-t', '${_LOCATION}-docker.pkg.dev/$PROJECT_ID/${
        _REPOSITORY}/${_IMAGE}', './app' ]
  - name: 'gcr.io/google.com/cloudsdktool/cloud-sdk'
    entrypoint: gcloud
    args:
      - 'run'
      - 'deploy'
      - '${_SERVICE}'
      - '--image'
      - '${_LOCATION}-docker.pkg.dev/$PROJECT_ID/${_REPOSITORY}/${_IMAGE}'
      - '--region'
      - '${_LOCATION}'
images:
  - '${_LOCATION}-docker.pkg.dev/$PROJECT_ID/${_REPOSITORY}/${_IMAGE}'
```

/cloudbuild.app.yml

The cloudbuild config shown above consists of two steps. First, it will build the application image in the `./app` folder context and upload it to Artifact Registry to the the repository `${_REPOSITORY}` with name `${_IMAGE}`. The variables starting with an underscore are substitution variables, and they can be provided to the cloudbuild from a trigger. Second, it will deploy the resulting image to the `${_SERVICE}`.

Chapter 7

Testing

Thanks to the emulator installed in Section 5.1, it is easy to test the Firebase application. I used the Mocha 3.4 test framework, which provides convenient tools for asynchronous testing. Firebase, in turn, provides its library for unit-testing called `@firebase/rules-unit-testing`. It allows to easily create a testing environment to simulate the requests from both unauthenticated and authenticated users. I divided tests into two main parts: tests of the security rules and the correct workers functioning, which will indicate that the worker library 5.5 is working as expected.

Test suits for both parts were created, thus allowing to run each suite separately. Mocha is very descriptive, and a typical test case looks like the following:

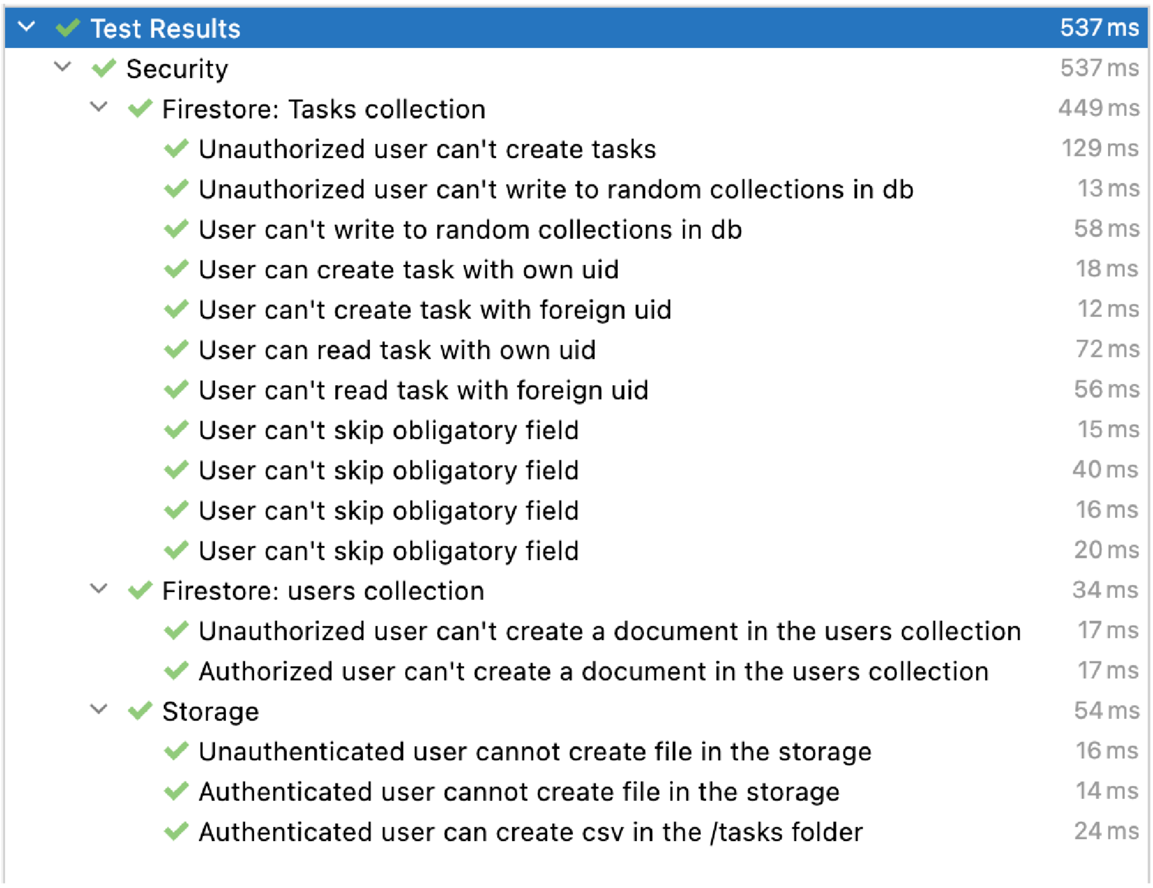
```
...
describe('Security', () => {
    describe('Firestore: Tasks collection', () => {
        it(`Unauthorized user can't create tasks`, async () => {
            const testDoc = firestoreUnauthed.doc('tasks/testDoc')
            const docData = {
                [SERVICE_FIELD]: {
                    ownerUid: `anon`,
                    type: 'sum',
                    status: 'queued',
                    creationDate: new Date(),
                },
            }

            await assertFails(testDoc.set(docData))
        })
...
```

/test/test.js

Firstly, I tested the security rules. In an attempt to cover most possible scenarios, tests shown in figure 7.1 were created.

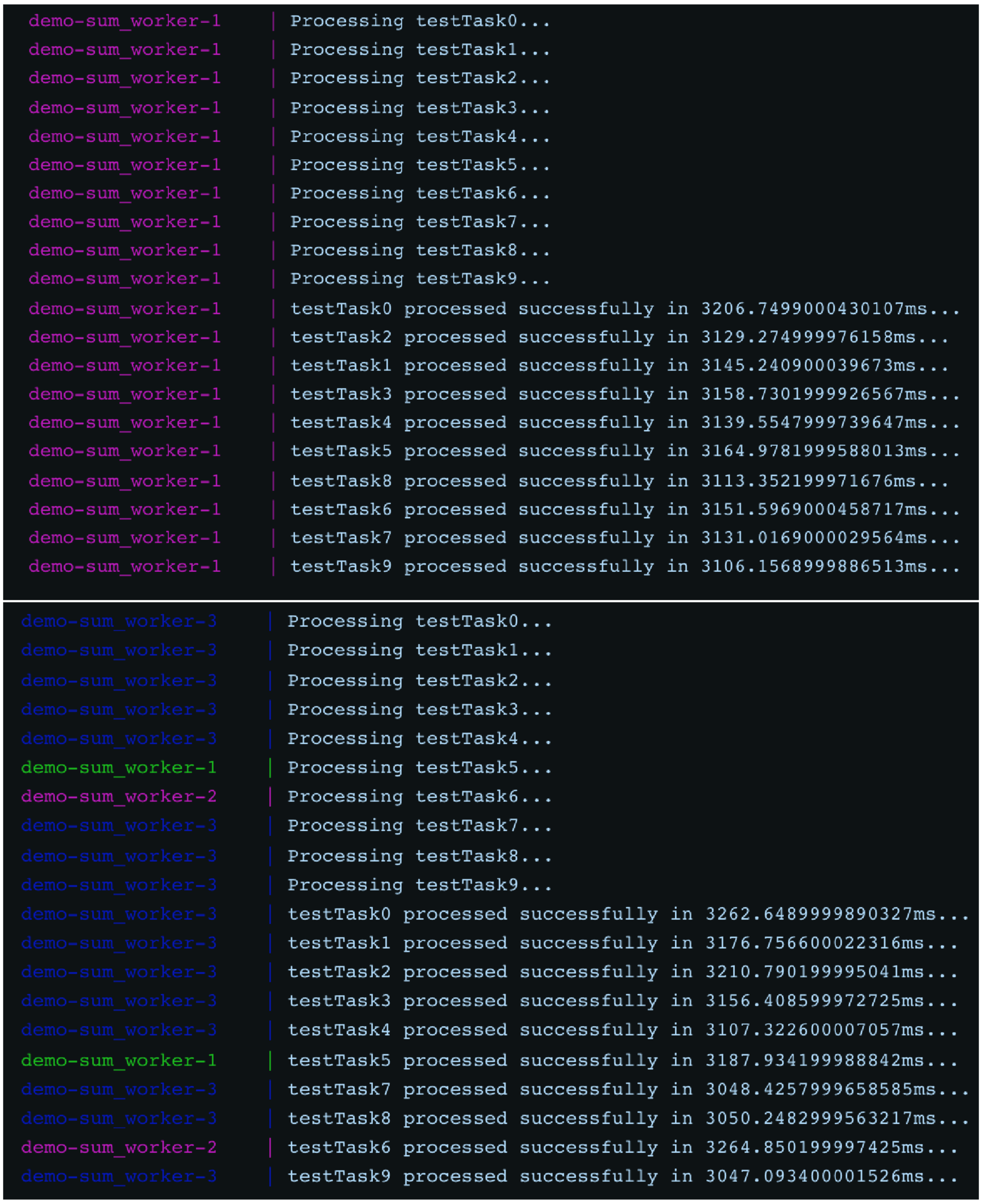
All the rules were working as expected. Then I tested the demonstration worker in two setups. In the first case, there was a single instance of the worker running, which is shown on the first screenshot of figure 7.2. Next, I scaled the worker, allowing it to run in three instances simultaneously using the command `docker-compose up -d -scale sum_worker=3`. This scenario result is shown in the second screenshot.



```
✓ Test Results                                                              537 ms
  ✓ Security                                                                537 ms
    ✓ Firestore: Tasks collection                                           449 ms
      ✓ Unauthorized user can't create tasks                                129 ms
      ✓ Unauthorized user can't write to random collections in db            13 ms
      ✓ User can't write to random collections in db                         58 ms
      ✓ User can create task with own uid                                    18 ms
      ✓ User can't create task with foreign uid                              12 ms
      ✓ User can read task with own uid                                      72 ms
      ✓ User can't read task with foreign uid                                56 ms
      ✓ User can't skip obligatory field                                     15 ms
      ✓ User can't skip obligatory field                                     40 ms
      ✓ User can't skip obligatory field                                     16 ms
      ✓ User can't skip obligatory field                                     20 ms
    ✓ Firestore: users collection                                            34 ms
      ✓ Unauthorized user can't create a document in the users collection    17 ms
      ✓ Authorized user can't create a document in the users collection      17 ms
    ✓ Storage                                                                54 ms
      ✓ Unauthenticated user cannot create file in the storage               16 ms
      ✓ Authenticated user cannot create file in the storage                 14 ms
      ✓ Authenticated user can create csv in the /tasks folder               24 ms
```

Figure 7.1: Security tests result

As expected, both single and multiple instances started processing tasks in order of creation date. Moreover, multiple instances were correctly skipping already locked tasks.



```
demo-sum_worker-1     | Processing testTask0...
demo-sum_worker-1     | Processing testTask1...
demo-sum_worker-1     | Processing testTask2...
demo-sum_worker-1     | Processing testTask3...
demo-sum_worker-1     | Processing testTask4...
demo-sum_worker-1     | Processing testTask5...
demo-sum_worker-1     | Processing testTask6...
demo-sum_worker-1     | Processing testTask7...
demo-sum_worker-1     | Processing testTask8...
demo-sum_worker-1     | Processing testTask9...
demo-sum_worker-1     | testTask0 processed successfully in 3206.7499000430107ms...
demo-sum_worker-1     | testTask2 processed successfully in 3129.274999976158ms...
demo-sum_worker-1     | testTask1 processed successfully in 3145.240900039673ms...
demo-sum_worker-1     | testTask3 processed successfully in 3158.7301999926567ms...
demo-sum_worker-1     | testTask4 processed successfully in 3139.5547999739647ms...
demo-sum_worker-1     | testTask5 processed successfully in 3164.9781999588013ms...
demo-sum_worker-1     | testTask8 processed successfully in 3113.352199971676ms...
demo-sum_worker-1     | testTask6 processed successfully in 3151.5969000458717ms...
demo-sum_worker-1     | testTask7 processed successfully in 3131.0169000029564ms...
demo-sum_worker-1     | testTask9 processed successfully in 3106.1568999886513ms...
```

```
demo-sum_worker-3     | Processing testTask0...
demo-sum_worker-3     | Processing testTask1...
demo-sum_worker-3     | Processing testTask2...
demo-sum_worker-3     | Processing testTask3...
demo-sum_worker-3     | Processing testTask4...
demo-sum_worker-1     | Processing testTask5...
demo-sum_worker-2     | Processing testTask6...
demo-sum_worker-3     | Processing testTask7...
demo-sum_worker-3     | Processing testTask8...
demo-sum_worker-3     | Processing testTask9...
demo-sum_worker-3     | testTask0 processed successfully in 3262.6489999890327ms...
demo-sum_worker-3     | testTask1 processed successfully in 3176.756600022316ms...
demo-sum_worker-3     | testTask2 processed successfully in 3210.790199995041ms...
demo-sum_worker-3     | testTask3 processed successfully in 3156.408599972725ms...
demo-sum_worker-3     | testTask4 processed successfully in 3107.322600007057ms...
demo-sum_worker-1     | testTask5 processed successfully in 3187.934199988842ms...
demo-sum_worker-3     | testTask7 processed successfully in 3048.4257999658585ms...
demo-sum_worker-3     | testTask8 processed successfully in 3050.2482999563217ms...
demo-sum_worker-2     | testTask6 processed successfully in 3264.850199997425ms...
demo-sum_worker-3     | testTask9 processed successfully in 3047.093400001526ms...
```

Figure 7.2: Worker tests. One worker instance on the top, three worker instances on the bottom

Chapter 8

Conclusion

As a result, the development of web applications, from classical to the constantly growing cloud-based serverless solutions, was researched. I developed the framework for a specific purpose: running computational tasks in the cloud using Docker containers. Even though the solution is written using the TypeScript language and Node.js environment, thanks to its highly decoupled architecture, the different workers can be written in any language, providing the ability to run any software for task processing.

The client-side demonstration application was developed using the ReactJS library to build the component-based UI with a powerful Redux for managing the centralized application state. The MUI user interface components library was used for the styling. As of logic, the `TaskerClient` library was developed, which makes working with the tasks easier by wrapping database documents into `Task` objects. It abstracts some aspects of task creation, making the work with Firestore easier and leaving the SDK flexible.

The worker was written using pure TypeScript capabilities, using Firestore features such as a subscription to the real-time updates from the Firestore Database. The work with the database in terms of the tasks' state management and locking the processed resource was also abstracted into class `Tasker`. This solution allows focusing on the task processing instead of writing logic for the queue and provides an excellent point to start with a similar service.

The billing solution is shallow but provides an excellent point to start. The users can be billed according to the processing time recorded in the service data by the Tasker. The demonstration solution shows an example of how this information can be utilized.

The application is deployed to the Cloud Run, which is a fully manageable platform for running Docker containers. As for essential web application components, it uses the Firestore NoSQL database and Google Cloud Storage. An Artifacts Repository is used in order to store the images. Also, the CI/CI pipeline was leveraged using Cloud Build CI/CD.

Bibliography

[1] Cervone, H. F. An overview of virtual and cloud computing. *OCLC Systems & Services: International digital library perspectives*. Emerald Group Publishing Limited. 2010. ISSN 1065-075X.

[2] *Cloud Run pricing* [online]. Google [cit. 2022-04-28]. Available at: https://cloud.google.com/run/pricing.

[3] *Cloud Storage pricing* [online]. Google [cit. 2022-04-29]. Available at: https://cloud.google.com/storage/pricing.

[4] *Docker Documentation* [online]. Docker [cit. 2022-05-01]. Available at: https://docs.docker.com/.

[5] *Firebase CLI reference* [online]. Firebase [cit. 2022-05-11]. Available at: https://firebase.google.com/docs/cli.

[6] *Firebase Pricing* [online]. Firebase [cit. 2022-04-28]. Available at: https://firebase.google.com/pricing.

[7] *Gcloud CLI overview* [online]. Google [cit. 2022-04-25]. Available at: https://cloud.google.com/sdk/gcloud.

[8] *Geography and regions* [online]. Google [cit. 2022-04-25]. Available at: https://cloud.google.com/docs/geography-and-regions.

[9] Gyorödi, C., Gyorödi, R. and Sotoc, R. A comparative study of relational and non-relational database models in a Web-based application. *International Journal of Advanced Computer Science and Applications*. Citeseer. 2015, vol. 6, no. 11, p. 78–83.

[10] *Integrating with Cloud Build* [online]. Google [cit. 2022-05-01]. Available at: https://cloud.google.com/artifact-registry/docs/configure-cloud-build.

[11] Jazayeri, M. Some trends in web application development. In: IEEE. *Future of Software Engineering (FOSE'07)*. 2007, p. 199–213. ISBN 0-7695-2829-5.

[12] *Nginx news* [online]. NGINX [cit. 2022-05-11]. Available at: http://nginx.org/.

[13] *Node.js* [online]. Node.js [cit. 2022-05-01]. Available at: https://nodejs.org/.

[14] *Npm* [online]. npm [cit. 2022-05-01]. Available at: https://www.npmjs.com/.

[15] Oluwatosin, H. S. Client-server model. *IOSRJ Comput. Eng.* 2014, vol. 16, no. 1, p. 2278–8727. ISSN 2278-0661.

[16] Potel, M. MVP: Model-View-Presenter the Taligent programming model for C++ and Java. *Taligent Inc.* 1996, vol. 20.

[17] *Pushing and pulling images* [online]. Google [cit. 2022-05-10]. Available at: https://cloud.google.com/artifact-registry/docs/docker/pushing-and-pulling.

[18] Rashid, A. and Chaturvedi, A. Cloud computing characteristics and services: a brief review. *International Journal of Computer Sciences and Engineering*. 2019, vol. 7, no. 2, p. 421–426. ISSN 2347-2693.

[19] *React – A JavaScript library for building user interfaces* [online]. Meta Platforms, Inc [cit. 2022-05-03]. Available at: https://reactjs.org/.

[20] *Zoom* [online]. Remix [cit. 2022-05-03]. Available at: https://reactrouter.com/.

[21] *Redux Toolkit* [online]. Dan Abramov [cit. 2022-05-11]. Available at: https://redux-toolkit.js.org/.

[22] Reenskaug, T. M. H. The original MVC reports. 1979.

[23] *What is serverless?* [online]. RedHat, 2017 [cit. 2022-05-05]. Available at: https://www.redhat.com/en/topics/cloud-native-apps/what-is-serverless.

[24] Sorensen, E. and Mikailesc, M. Model-view-ViewModel (MVVM) design pattern using Windows Presentation Foundation (WPF) technology. *MegaByte Journal*. 2010, vol. 9, no. 4, p. 1–19.

[25] *TypeScript: JavaScript With Syntax For Types.* [online]. Microsoft [cit. 2022-05-03]. Available at: https://www.typescriptlang.org/.

[26] Watada, J., Roy, A., Kadikar, R., Pham, H. and Xu, B. Emerging trends, techniques and open issues of containerization: a review. *IEEE Access*. IEEE. 2019, vol. 7, p. 152443–152472. DOI: 10.1109/ACCESS.2019.2945930.

Appendix A

Contents of the included storage media

```
.
|-- app/ -- demo front-end application
| |-- nginx/ -- nginx configuration
| |-- src/
| | |-- components/
| | |-- pages/
| | | |-- InvoicesPage/
| | | |-- LoginPage/
| | | `-- TasksPage/
| | `-- utils/
| | |-- hoc/
| | |-- redux/
| | `-- tasker/ -- client library
| `-- static/
|-- sum-worker/ -- demo worker application
| `-- src/
| |-- tasker/ -- worker library
| `-- utils/
`-- test/ -- emulator and tests
```