VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
DEPARTMENT OF INTELLIGENT SYSTEMS

FRAMEWORK PRO PLACENOU INTERNETOVOU SLUŽBU

FRAMEWORK FOR A WEB INTERNET SERVICE

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — **FILIP HÁJEK**
AUTHOR

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

BRNO 2022

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav inteligentních systémů (UITS) Akademický rok 2021/2022

Zadání bakalářské práce

Student: **Hájek Filip**
Program: Informační technologie
Název: **Framework pro placenou internetovou službu**
Framework for a Web Internet Service
Kategorie: Web

Zadání:

1. Prostudujte kontejnerizaci programů. Prostudujte programování webových aplikací a serverů založených na REST API.
2. Navrhněte webové/REST API rozhraní pro uživatelské spouštění kontejnerů infrastruktury docker. Navrhněte systém administrace uživatelů služby a způsoby zpoplatnění za výpočetní služby.
3. Implementujte službu s demonstračním kontejnerem. Nasaďte službu na veřejném cloudovém systému.
4. Testujte provoz služby simulovanými přístupy uživatelů.

Literatura:

- Dokumentace Docker.

Pro udělení zápočtu za první semestr je požadováno:

- První dva body zadání.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce: **Hrubý Martin, Ing., Ph.D.**
Vedoucí ústavu: Hanáček Petr, doc. Dr. Ing.
Datum zadání: 1. listopadu 2021
Datum odevzdání: 11. května 2022
Datum schválení: 3. listopadu 2021

Abstrakt

Cílem této práce je navrhnutí a implementování kontejnerizované internetové služby, která poskytuje uživatelům možnost spouštět úlohy. Úlohou se myslí pouze abstrakce nad nějakou konkrétní implementací výpočtu. Úloha se spouští jako Docker kontejner. V práci se dále implementuje administrace uživatelů a metoda zpoplatnění spuštěných úloh. Architektura systému se skládá z REST API webového serveru a uživatelského webového prostředí. Nezbytnou komponentou systému jsou worker aplikace, které vykonávají uživatelské úlohy přidělené serverem. Celý systém se podařilo implementovat pomocí frameworku .NET 6 v programovacím jazyce C#. Provoz služby se podařilo nasimulovat v prostředí Dockeru. Služba integruje existující služby Auth0 a Google Cloud Storage. Výsledkem je internetová služba nasazená na veřejném cloud serveru.

Abstract

The goal of this thesis is to design and implement a containerized web service, which provides the ability to create and run users' tasks. The task is only an abstraction over some particular implementation of computation. The task runs as a Docker container. The thesis also implements user management and the method of running task charging. The system architecture consists of a REST API web server and a user web environment. Worker applications are an essential component of the system. They start user tasks assigned by the server. The system was implemented using the .NET 6 framework in programming language C#. The run of the service has been simulated in a Docker environment. The service integrates existing Auth0 and Google Cloud Storage services. The result is an internet service deployed on a public cloud server.

Klíčová slova

Kontejnerizace, Docker, Webový server, REST API, Autorizace, Azure, Auth0, Uživatelská administrace, C#, ASP.NET Core

Keywords

Containerization, Docker, Web server, REST API, Authorization, Azure, Auth0, User administration, C#, ASP.NET Core

Citace

HÁJEK, Filip. *Framework pro placenou internetovou službu.* Brno, 2022. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Martin Hrubý, Ph.D.

Framework pro placenou internetovou službu

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Martina Hrubého, Ph.D. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

.....
Filip Hájek
4. května 2022

Poděkování

Tímto bych chtěl poděkovat panu Ing. Martinovi Hrubému, Ph.D za vedení této práce, za cenné rady a další odbornou pomoc.

Obsah

Kapitola 1

Úvod

Bakalářská práce se zabývá návrhem a vývojem webové služby[1]. Ta vytváří pro uživatele jednoduché rozhraní umožňující zakládání a spouštění výpočetních úloh. Za každou spuštěnou úlohu musí uživatel zaplatit virtuální měnou. Úlohy se v systému řadí do fronty podle času a priority. Ve službě existují uživatelé, kteří mají VIP status. Jejich úlohy mají přednost před ostatními úlohami. Systém postupně odebírá úlohy z fronty. Ty se následně zpracovávají na více počítačích. Uživatel může sledovat stav vytvořených úloh. Po skončení úlohy si může výsledek stáhnout ve formě souboru.

Cílem této práce je detailně promyslet architekturu této služby, vybrat nejideálnější řešení a to implementovat. Implementovaný systém by se měl následně nasadit na veřejný cloud server.

V práci se využívá moderních technologií, jako je Docker, který byl vydán v roce 2013. Využívá nových metod usnadňujících vývoj a nasazování webových aplikací. Na této technologii staví nejmodernější systémy a webové služby. Právě Docker byl důvodem, proč jsem si vybral toto téma. Existuje celá řada dalších nástrojů, které jsou postaveny na Dockeru a na principech, které využívá. Tyto nástroje se čím dál více používají v produkci. Je dobré se chytit nejnovějších trendů a naučit se základy těchto principů a technologií.

Práce začíná teoretickou částí. V 2. kapitole je vysvětlena kontejnerizace. Kapitola následně popisuje platformu pro spouštění kontejnerů Docker. Serverová aplikace využívá architektury REST. Programování těchto serverů se rozebírá v kapitole 3. Kapitola se dále zaměřuje na směrování dotazů, autorizaci a základy frameworku ASP.NET Core 6, který jsem vybral pro implementaci této práce. Poté se přechází na praktickou část. Popis architektury a detailní návrhy částí systému jsou uvedeny v kapitole 4. V 5. kapitole se ukazují některé zajímavé detaily implementace. Proces nasazení celé služby je popsán v kapitole 6. Simulace provozu se řeší v kapitole 7. Závěrečná kapitola 8 shrnuje výsledky práce a nabízí náměty na další pokračování.

[1] Služba je v období obhajob nasazena na https://tasklauncher.azurewebsites.net/

Kapitola 2

Kontejnerizace programů

V této kapitole se popisuje proces zabalování aplikací a programů do tzv. kontejnerů. Popisuje se, jaké problémy kontejnery řeší a jaké výhody přinášejí. Dále se krátce zmiňuje Docker Engine, který poskytuje jednotné rozhraní pro vytváření a spouštění kontejnerizovaných aplikací.

2.1 Úvod do kontejnerizace

Kontejnerizace je proces zabalení softwaru do spustitelné jednotky nazývané kontejner. Kontejner obsahuje veškeré závislosti, systémové knihovny, framework a další komponenty, které jsou nezbytné pro spuštění softwaru.[6]

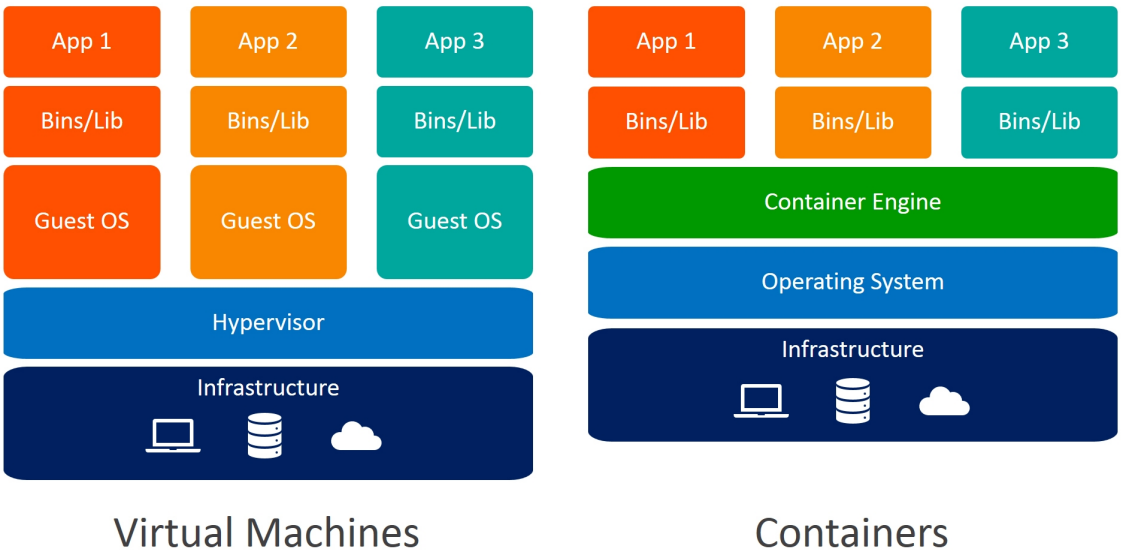
Kontejner je zcela nezávislý na okolí, je přenositelný a funguje spolehlivě ve všech prostředí bez ohledu na operační systém. Kontejnerizaci lze považovat za virtualizaci na úrovni operačního systému. Může se spouštět několik kontejnerů na stejném stroji, kde kontejnery izolovaně přistupují ke stejnému kernelu. Kontejnery fungují v jakémkoli prostředí.[25]

Tuto technologii poprvé prosadila firma Docker, která v roce 2013 uvedla na trh software Docker Engine. Docker nastavil standard pro vytváření a spouštění kontejnerů (OCI – open container initiative).[6]

2.2 Výhody kontejnerizace

Virtualizace je abstrakcí fyzického hardwaru. Hypervisor sice umožňuje spustit několik virtuálních strojů na jednom stroji, nicméně každá instance obsahuje celou kopii operačního systému. Virtuální stroj tak zabírá spoustu místa, vyžaduje více zdrojů a bývá obecně pomalejší. Kontejnery jsou naopak jednoduché, rychlé a neobsahují celý operační systém jako virtuální stroj. Kontejner vytváří abstrakci pouze nad operačním systémem a obsahuje pouze nezbytné komponenty. Velikost kontejneru se tak pohybuje v jednotkách MB, velikosti virtuálního stroje v jednotkách GB.[5]

Rozdíly mezi virtualizací a kontejnerizací jsou zobrazeny na obrázku 2.1.



Obrázek 2.1: Rozdíl mezi virtualizací a kontejnerizací. Převzato z [10].

Vývojáři většinou programují ve svém oblíbeném prostředí (Linux, MacOs, Windows). Problém nastává v momentě, kdy se vývojář rozhodne nasadit software nebo změnit vývojové prostředí. Původní systém například obsahuje systémové knihovny, které nejsou dostupné na jiných systémech. Software tak nelze jednoduše přesunout. Pomocí kontejnerizace se tento problém dá vyřešit. Stačí zabalit software do kontejneru. Software jako Docker pak dokáže spustit stejné kontejnery prakticky na každém operačním systému.[6]

Během vývoje se často používá CI/CD (průběžné integrování a nasazování). Kontejnerizace aplikací se dá velmi jednoduše automatizovat a integrovat do CI/CD pipeline. Kontejnery se mohou použít k automatizovanému testování softwaru a nasazení softwaru. Lze například snadno vytvořit CI pipeline, která po nahrání nové funkcionality do repositáře, spustí testy ověřující funkčnost aplikace. Kontejnery a nástroje pro jejich spouštění jsou tedy kompatibilní pro DevOps procesy pro rychlý vývoj a integrování nových funkcionalit.[6]

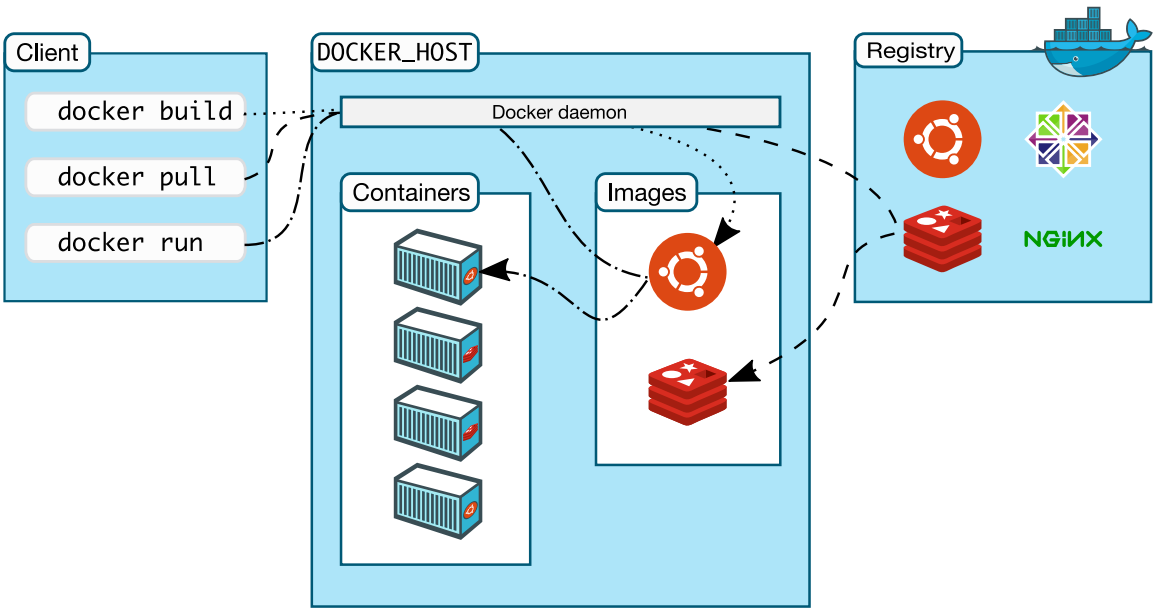
Kontejnery lze jednoduše spravovat a konfigurovat. To umožňuje tzv. orchestrátorům škálovat, nasazovat, monitorovat a spravovat kontejnerizované aplikace.[6]

2.3 Docker Engine

Tato sekce je převzata z oficiální dokumentace [8, 9].

Docker Engine je open source platforma pro vytváření a spouštění kontejnerizovaných aplikací. Docker používá architekturu server-klient, která je vidět na obrázku 2.2. Serverem je Docker démon nazývaný dockerd. Ten vykonává všechny operace jako spouštění a vytváření kontejnerů. Klientské aplikace komunikují s démonem přes Docker Engine API, na kterém démon naslouchá všem požadavkům. Klientská aplikace může být spuštěna na stejném systému jako démon. S démonem lze komunikovat i vzdáleně. Klientskou aplikací je buď oficiální GUI aplikace Docker Desktop nebo CLI klient docker. Jelikož je Docker Engine API klasické REST API, může vývojář vytvořit vlastní klientskou aplikaci nebo nástroj, který bude uzpůsoben jeho požadavkům. Poslední částí je Docker registr (například Docker Hub). Registr je v podstatě repositář, kam se ukládají Docker image.

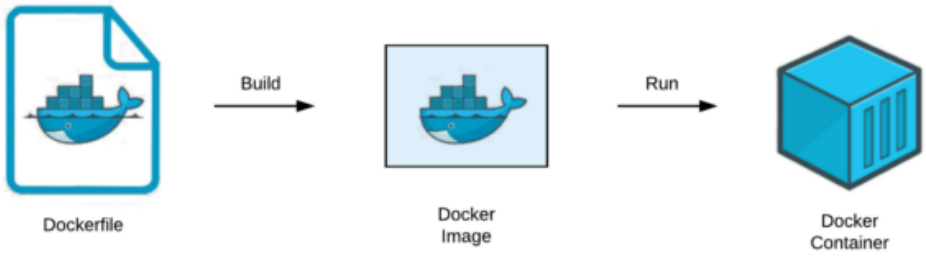
Docker není jediné prostředí pro spouštění kontejnerů, alternativou je například Podman.



Obrázek 2.2: Architektura Dockeru. Pomocí klienta se ovládá démon, který generuje kontejnery spuštěním image z nějakého registru. Převzato z [9].

Image

Image si lze představit jako šablonu, ze které se vytváří kontejnery. Image je pouze pro čtení, nelze ho nijak modifikovat. Lze ho pouze spustit, čímž vznikne kontejner. Image se ukládá lokálně nebo do registru, jako je například zmíněný Docker Hub. Image je definován pomocí jednoduché syntaxe v souboru Dockerfile. Tento soubor obsahuje několik instrukcí (příkazů) a kroků, které definují, jak bude výsledný image sestaven. Sestavování provádí démon a může trvat i několik minut. Každá instrukce ze souboru Dockerfile vytváří novou vrstvu. Pokud uživatel zmodifikuje soubor Dockerfile a image znovu sestaví, sestaví se pouze změněné vrstvy. Jakmile dochází ke spuštění image, vygeneruje se kontejner. Tímto se přidá nová vrstva, která se nazývá vrstva kontejneru. Tato vrstva je již modifikovatelná. Lze v ní například vytvářet, upravovat a mazat soubory. Proces vytvoření kontejneru je znázorněn na obrázku 2.3.



Obrázek 2.3: Postup vytvoření Docker kontejneru. Převzato z [19].

Kontejner

Pokud je image šablona pro vytváření kontejnerů, kontejner je instance image. Kontejner vzniká spuštěním image (při každém spuštění image vznikne nový kontejner). Kontejner se chová jako normální aplikace. Můžeme ho zastavit, znovu spustit nebo smazat. Tyto operace opět provádí zmíněný démon dockerd, kterého informují přes Docker Engine API klientské aplikace. Kontejner se různě konfiguruje. Typicky se musí nakonfigurovat port, na kterém se bude naslouchat příchozím požadavkům, dále se mohou měnit proměnné prostředí, které definuje image. Poté se dá nastavit perzistentní úložiště a aby Docker kontejner mohl komunikovat s jinými kontejnery, může se nakonfigurovat Docker network.

Perzistence dat

Docker kontejner může vytvářet soubory, ukládat cenná aplikační data a podobně. Jakmile ale dojde ke smazání kontejneru, smaže se nenávratně i veškerý obsah vytvořený kontejnerem (dochází ke smazání vrstvy kontejneru, kterou lze modifikovat). Existují především dva způsoby, které řeší persistenci dat.

- **Bind mount** je jednoduchý způsob, jak zachovat data kontejneru. Do kontejneru se přimontuje adresář nebo soubor ze systému hosta. Tento způsob závisí na souborovém systému hostitelského stroje a není spravován Docker démonem.
- **Volumes** jsou dalším způsobem, jak uchovávat data. Volume je plně spravovaný Docker démonem a je izolován od operačního systému hostitelského stroje. Tento způsob je tak preferovanější než bind mount. S volume se pracuje snadněji a rychleji. Volume může být přimontován k více kontejnerům. Díky tomu se dá docílit sdílení souborů mezi kontejnery. Největší výhodou je, že data mohou být uložena na jiném stroji nebo u některého cloud providera. Data se ve výchozím nastavení ukládají do souborového systému hosta.

Komunikace mezi kontejnery

Komunikaci mezi kontejnery lze zajistit tak, že budou sdílet nějaké soubory (adresář), přes které si budou vyměňovat data. Tento způsob je pouze vhodný, pokud si kontejnery vyměňují soubory nebo spolu komunikují velmi ojediněle. Nejčastější způsob sdílení souborů mezi kontejnery je zmíněný volume. Pokud spolu aplikace komunikují neustále nebo jsou

založené na architektuře REST, server-klient a dalších, musí spolu komunikovat přes síť. V Dockeru lze vytvořit několik sítí (tzv. Docker network). K docílení komunikace přes síť mezi danými kontejnery stačí spustit kontejnery se stejnou sítí.

Dockerfile

Syntax souboru Dockerfile je velmi jednoduchá. První slovo instrukce je jméno příkazu, po kterém následují parametry. Dockerfile vždy začíná příkazem `FROM`. Tento příkaz znamená, že image, který se aktuálně definuje, bude založen na nějakém existujícím image. Za příkazem `FROM` se píše název Docker image. Docker image se najde na lokálním stroji nebo se hledá v Docker Hub registru. Pokud se image najde, Docker ho stáhne, uloží a vytvoří vrstvu, která je založena na staženém image. Jako parametr této instrukce je většinou název oficiálního image frameworku, který vývojář používá. Příkazů `FROM` může být v Dockerfile souboru více. Typicky je potřeba, aby se aplikace sestavila. Na to je například potřeba celé SDK používaného frameworku. Přitom ke spuštění aplikace je pouze potřeba runtime frameworku, který bývá signifikantně menší velikosti. Pomocí příkazů `FROM` se dá lehce rozfázovat sestavování výsledného image. Ukázka 2.1 obsahuje příklad.

Nejjednodušším příkazem je příkaz `RUN`. Ten vykonává příkaz v shellu. Do parametru se zadává příkaz, který se má vykonat. Příkaz `WORKDIR` slouží pro nastavení pracovního adresáře pro další příkazy. Příkaz `EXPOSE` nastavuje, na jakém portu bude naslouchat kontejner. Příkaz vyloženě nezveřejňuje port, pouze slouží jako dokumentace pro spuštění kontejneru, kde se teprve port zveřejní. Příkazem `COPY` se kopírují soubory z aktuálního adresáře na aktuálním systému do souborového systémů kontejneru. Podporuje se kopírování souborů i z jiného image. Počáteční bod spuštění kontejneru se určuje pomocí instrukce `ENTRYPOINT`. Jakmile se spustí tento image, veškeré argumenty dodané za příkazem `docker run` se přidají za argumenty příkazu `ENTRYPOINT`. Tyto instrukce lze vidět na první ukázce 2.1.

```
# Image označený jako base bude založený na .NET runtime image
FROM mcr.microsoft.com/dotnet/runtime:6.0 AS base
# Nastavení pracovního adresáře
WORKDIR /app

# Image s .NET SDK se využije pro fázi sestavení aplikace
FROM mcr.microsoft.com/dotnet/sdk:6.0 AS build
WORKDIR /src
COPY ["TestImage/TestImage.csproj", "TestImage/"]
COPY . .
WORKDIR "/src/TestImage"
# Spuštění příkazu dotnet build pro sestavení aplikace
RUN dotnet build "TestImage.csproj" -c Release -o /app/build

# Vytvoření fáze publikování aplikace založené na fázi sestavení aplikace
FROM build AS publish
RUN dotnet publish "TestImage.csproj" -c Release -o /app/publish

# Vytvoření konečného image pojmenovaného jako final založeného na base image
FROM base AS final
WORKDIR /app
# Kopírování výsledného adresáře s aplikací z fáze publish do final image
COPY --from=publish /app/publish .
# Nastavení počátečního bodu spuštění
ENTRYPOINT ["dotnet", "TestImage.dll"]
```

Výpis 2.1: Ukázka souboru Dockerfile s komentáři.

2.4 Orchestrace

Při vývoji softwarového systému se často nepracuje pouze s jednou aplikací. Celý systém je obvykle rozdělen na několik částí například: frontend webová stránka, REST API server, backend služba a databáze. Neboli existuje systém, kde figurují čtyři aplikace komunikující mezi sebou. Každá část může být popsána souborem Dockerfile. Vytvoří se image. Ty se spustí v prostředí Docker s nějakými konfiguracemi a spustí se tak celý systém. Jakmile musí být zároveň vytvořeno několik kontejnerů, každý s jinou konfigurací, začne být velmi složité spouštět a nasazovat takový systém. Proto existují tzv. orchestrátoři.

Orchestrátor je nástroj, který je schopen spouštět, udržovat, spravovat a škálovat několik kontejnerů. To vše dokáže orchestrátor zajistit díky přenositelnosti a replikovatelnosti kontejnerů. Nejjednodušším orchestrátorem je Docker Compose (další Docker klient). Je definován pomocí YAML souboru. Hlavním účelem je definovat a nakonfigurovat veškeré části systému a spustit jej pomocí jednoho příkazu. V Docker Compose lze definovat tzv. služby. Službou se myslí kontejner. Služba (kontejner) se pojmenuje, určí se, z jakého image se vytvoří nebo se přímo určí cesta k souboru Dockerfile, který se použije k sestavení image. Následně se nakonfigurují proměnné prostředí a port, na kterém bude služba naslouchat. Konfigurují se sítě a volume, které se pak přiřadí k daným službám (kontejnerům). Docker Compose se velmi používá při lokálním vývoji kontejnerizovaného systému. Umožňuje spustit celý systém jedním příkazem s jedním konfiguračním souborem.[8]

Ukázka 2.2 obsahuje YAML soubor pro Docker Compose. Definuje se služba s volume. Mění se některé proměnné a nastavuje se cesta k Dockerfile souboru pro sestavení image.

```yaml
version: '3.5'

# Definice sítě localdev
networks:
  localdev:
    name: localdev

# Služba tasklauncher.worker
services:
  tasklauncher.worker:
    image: tasklauncher.worker
    container_name: tasklauncher.worker
    # Odkud se bude sestavovat image
    build:
      context: .
      dockerfile: TaskLauncher.Worker/Dockerfile
    # Přepsání konfiguračních proměnných
    environment:
      - ServiceAddresses__WebApiAddress=https://tasklauncher.app.server:443
      - ServiceAddresses__HubAddress=https://tasklauncher.app.server:443/WorkerHub
    volumes:
      - worker-vol:/app/tmp
    depends_on:
      - tasklauncher.app.server
    networks:
      - localdev

# Vytvoření volume
volumes:
  worker-vol:
```

Výpis 2.2: Ukázka YAML souboru pro Docker Compose s komentáři.

Nicméně pokud se nasazuje služba na veřejný server, musí se zajistit dostupnost a spolehlivost služby. Služba by se také měla škálovat podle její vytíženosti. Tyto požadavky Docker Compose nezajišťuje. Existují proto více robustnější orchestrátoři jako je Kubernetes nebo Docker Swarm, kteří zajišťují výše zmíněné požadavky. Tyto nástroje umožňují také automatizovanou údržbu aplikací, nasazování aplikací, výměnu zhavarovaných kontejnerů a podobně. [8, 6]

Kapitola 3

Programování webových aplikací a serverů založených na REST API

Kapitola se zaměřuje na programování serverových aplikací založených na REST. Popisují se základy REST architektury. Následně se píše o komunikaci přes protokol HTTP a směrování HTTP dotazů na webových serverech. Jsou zmíněné způsoby autentizace a autorizace. Na závěr se uvádí framework ASP.NET Core, ve kterém byla implementována tato práce.

3.1 REST API

REST (Representational state transfer) je architektonický styl, který představil a definoval Roy Fielding ve své disertační práci [31]. REST není protokol ani standard, je to pouze sada doporučení a omezení, které by měl výsledný systém implementovat. Jak takový systém implementovat je zcela na vývojáři. REST klade důraz na několik následujících omezení. Ty jsou převzaty z [11].

- **Klient-server architektura** – První omezení je použití klient-server architektury. Nejdůležitější myšlenkou je oddělení závislostí. Díky tomu lze vyvíjet aplikace nebo části systému nezávisle. Typicky se odděluje uživatelské rozhraní od logiky aplikace. Klient poskytuje rozhraní pro uživatele a odesílá požadavky na server, kde jsou vykonány. Zlepšuje se škálovatelnost serveru a přenositelnost uživatelského rozhraní mezi platformami.
- **Bezstavová komunikace** – Komunikace mezi klientem a serverem musí být bezstavová. Server je implementován tak, že neuchovává žádný stav klienta. Požadavek, který dorazí na server, tak musí obsahovat všechny potřebné informace. Stav uchovává pouze sám klient. Na rozdíl od tradičních stavových serverů, se nemusí dbát na udržování stavových informací. Díky tomu jsou implementace bezstavových serverů jednodušší. Lépe se monitorují, protože požadavek má ve svém těle všechny potřebné informace. Jsou spolehlivější a nemusí se starat o výpadek klienta. Servery jsou také škálovatelnější, bez nutnosti ukládat stav se po vykonání požadavku mohou uvolnit použité prostředky. Nevýhodou je zvýšení provozu na síti, pokud se požadavky často opakují. Server ztrácí kontrolu nad chováním aplikace, to musí správně implementovat klient.

- **Kešování** – Řeší problém se zvýšeným provozem na síti. Některá data se mohou ukládat do paměti cache. Kešování zvyšuje rychlost vrácení odpovědi, protože se nemusí nutně přistupovat do databáze nebo k jiným zdrojům.

- **Jednotné rozhraní** – Hlavním rysem REST je jednotné rozhraní, které zjednodušuje a zpřehledňuje celkovou architekturu. Server poskytuje své služby přes jasně definované rozhraní, na které se může napojit jakýkoli klient z jakékoli platformy.

API (Application Programming Interface) je soubor definic a protokolů. API může představovat volání funkce systémové knihovny. Do funkce se předají parametry, funkce se vykoná a vrátí se návratová hodnota. Webové API se chová prakticky stejně. Pošle se požadavek na server, kde se vykoná a vrátí se odpověď. API přesně definuje komunikaci mezi tím, kdo poskytuje nějakou službu a tím, kdo ji volá. U webového API se tak definují informace, které potřebuje server vědět, aby mohl vykonat daný požadavek. Zároveň je definována odpověď, kterou server vrací. REST API je takové API, které splňuje omezení architektonického stylu REST. Takovému rozhraní se říká RESTful API.[31]

3.2 HTTP komunikace a směrování HTTP dotazů

Implementace REST serverů využívají pro přenos dat především protokol HTTP. REST sice HTTP přímo nevyžaduje, nicméně HTTP je velmi používané a splňuje kritéria tohoto architektonického stylu [11].

Komunikaci začíná klient, který vytváří HTTP dotaz. Dotaz se skládá z tzv. request line, HTTP hlaviček a těla. V request line se specifikuje, jaká HTTP metoda se použije. Vyplňuje se zde cesta nebo celá URI adresa, na kterou se zasílá požadavek. Poslední informací je verze protokolu HTTP (nejčastěji je to verze HTTP/1.1 nebo aktuálně nejnovější HTTP/2.0). Povinná hlavička v protokolu HTTP/1.1 je hlavička Host. Ta obsahuje cílové doménové jméno a port, kam se bude zasílat dotaz. Pokud se neuvede, použije se výchozí port 80 pro HTTP a pro HTTPS 443. Mezi další hlavičky patří například Content-Type. Tato hlavička určuje, jaký typ dat se posílá nebo přijímá [12]. Pokud klient posílá dotaz z prohlížeče, vyplní se další hlavičky jako Accept. Prohlížeč vyplňuje hlavičku Cookie, pokud tak server nastavil v jedné z odpovědí přes hlavičku Set-Cookie [30]. Server odesílá odpověď, která se liší od dotazu tím, že místo request line je status line. Zde se nachází verze HTTP protokolu, status kód a reason phrase (krátký popis status kódu).[13]

Důležitou součástí komunikace je směrování dotazů. Směrováním se určuje, kdo zpracuje a obslouží dotaz (aplikace nebo nějaký koncový bod). Koncové body (cesty, kam se mohou směrovat dotazy) se registrují předtím, než se spustí webový server [23]. Vykonávají obsluhu dotazu, který se na bod přesměroval. Existují především tři způsoby směrovaní HTTP dotazů [18].

- **Směrování podle doménového jména** – Nejtradičnější způsob je směrování podle doménového jména aplikace. Tento způsob umožňuje na jedné IP adrese hostovat více webových aplikací. Tento způsob používají například load balancer služby (služby vyrovnávající zátěž provozu).[18]

- **Směrování podle URI** – Dalším typem je směrování podle cesty v URI adrese. Cesta se většinou spojuje do dvojice s HTTP metodou. Podle této dvojice se následně určí, kdo obslouží dotaz (většinou to je funkce, obsluha, nějaký kód, který je namapovaný na tuto dvojici).[23]

- **Směrování pomocí hlaviček** – Směrování lze docílit i pomocí hlaviček dotazu. Technicky je směrování podle doménového jména docíleno také pomocí hlaviček (využívá hlavičku Host). Směrování pomocí hlaviček se používá například v proxy serverech.[18]

3.3 Autentizace a autorizace

Autentizace je proces přihlašování uživatele. Uživatel zadává své údaje, aby se mohl přihlásit ke službě. Autorizace ověřuje, zda přihlášený uživatel má přístup k danému zdroji. Každý webový server by měl zajišťovat autentizaci a autorizaci.

Základní metody

Mezi základní HTTP autentizační metody patří Basic a Digest Access autentizace. Basic autentizace pouze kóduje přihlašovací údaje do formátu base64, musí se tak používat šifrované spojení. Údaje se následně posílají s každým požadavkem v hlavičce Authorization. Digest Access je robustnější řešení. Princip spočívá v tom, že klient spočítá MD5 kontrolní součet přihlašovacích údajů a nějakého řetězce, které zaslal server. Součet se pošle společně s uživatelským jménem na server. Server si dohledá podle jména heslo, spočítá MD5 součet a porovná ho se zaslaným součtem.[14]

Jelikož existuje šifrovaný protokol HTTPS, autentizace se často řeší pouze jako šifrovaný HTTP POST dotaz. V těle dotazu jsou uvedeny přihlašovací údaje. Řešení autentizace a autorizace se tak může realizovat přes cookie nebo token autentizaci.

Cookie autentizace

Cookie je tradiční nástroj umožňující implementaci autentizace ve webových prohlížečích. Po úspěšném ověření přihlašovacích údajů na serveru se zašle zpátky odpověď s hlavičkou Set-Cookie. Tím prohlížeč uloží cookie na počítači a v každém dalším požadavku se data uvedou v hlavičce Cookie. Na serveru proběhne autorizace, ověření, zda uživatel má přístup ke zdroji a vrátí se příslušná odpověď.[30]

Existují dvě implementace využívající cookie pro autentizaci: stavové cookie a bezstavové cookie[1]. Stavové cookie vyžadují uložení stavu na serveru. Musí tedy existovat databáze, kde jsou uloženy relace všech přihlášených uživatelů. Do této databáze se mohou ukládat data spojená s danou relací. V cookie je uloženo pouze ID relace, se kterým se přistupuje do databáze. Server má veškerou kontrolu nad uživateli. Tato metoda se považuje za jednu z nejbezpečnějších. Musí se ale implementovat další opatření, například proti útokům CSRF[2].[7]

Bezstavové cookie nevyžadují ukládat stav na serveru, veškeré autorizační informace společně s dalšími daty související se stavem a relací jsou uloženy přímo v cookie. Díky tomu, že je vše uloženo v cookie, se odesílá více dat. Nelze jednoduše odhlásit uživatele jako u stavových cookie, ale nevyžaduje databázi pro ukládání relací.[7]

[1] Spousta publikací používá různé názvy a termíny pro cookie autentizaci. Vždy se však myslí stejný princip implementace. Stavové cookie se také říká session cookie, někdy se používá session autentizace. U bezstavové cookie se kolikrát vyskytuje pouze cookie autentizace.

[2] CSRF nebo XSRF – Cross Site Request Forgery https://owasp.org/www-community/attacks/csrf

Token autentizace

Token autentizace (Bearer autentizace) je další způsob řešící HTTP autentizaci. Tato metoda byla vytvořena pro protokol OAuth 2.0. Lze ji používat i bez tohoto protokolu. Funguje podobně jako Basic autentizace, kdy se do hlavičky Authorization přidávají zakódované přihlašovací údaje. Na rozdíl od Basic autentizace hlavička obsahuje token bez přihlašovacích údajů (metoda by se stejně měla používat pouze přes HTTPS).[4]

Jakmile klient zašle správné přihlašovací údaje, server vytvoří podepsaný token, který vrátí klientovi. Ten ho s každým dalším požadavkem posílá na server v hlavičce Authorization.

Token je většinou implementován pomocí JWT[3]. JWT je standard definující kompaktní způsob bezpečného přenosu informací ve formátu JSON. JWT je rozdělen na tři části: hlavička, tělo a podpis. V těle se ukládá obsah spojený s relací a tzv. claims (nároky). Podepisuje se algoritmem uvedeným v hlavičce (typicky HMAC-SHA256). Obsah se kóduje pomocí Base64Url. Informace z těla tokenu sice jdou dekódovat a přečíst, ale nelze je modifikovat. Jakmile token někdo změní, server to díky podpisu odhalí.[32]

Tokeny se používají všude tam, kde nelze použít cookie nebo kde se cookie nehodí (aplikace neběžící v prohlížeči, komunikace mezi serverovými aplikacemi). Autorizace pomocí tokenů se obecně nedoporučuje používat v prohlížečích. Na rozdíl od cookie musí být token uložen přímo do prohlížeče, kde může být přečten jinou webovou aplikací. Cookie nelze přečíst z žádné webové aplikace (HTTP-only cookie).[27]

OAuth2 a OpenIdConnect

Na zmíněných metodách je postaven autorizační protokol OAuth2. Slouží pro získání přístupu k aplikaci třetí strany. Například pokud nějaká aplikace požaduje přístup ke Google Drive, aby mohla uložit soubor. Jelikož OAuth2 neřeší autentizaci, byl vytvořen protokol OpenIdConnect (OIDC). Ten staví na OAuth2 protokolu. Umožňuje například přihlašování uživatele do aplikace přes poskytovatele OIDC (například přihlášení přes Facebook, Google a další).[22]

3.4 Programování REST API ve frameworku ASP.NET Core

Klasický server v architektuře server-klient funguje v jedné neustále běžící smyčce. Vždy, když klient naváže nové TCP spojení, začne jeho obsluha. Komunikace a obsluha probíhají obvykle podle nějakého protokolu, který bývá stavový, tudíž server musí uchovávat stav. Webový server založený na REST architektuře funguje obdobně, jen neuchovává žádný stav. Pouze přijme dotaz, který přesměruje na koncový bod, kde se vykoná daná obsluha a vrátí se odpověď. Implementace prototypu takového serveru je snadná, nicméně REST klade důraz na spoustu omezení zmíněných v sekci 3.1. Vývojáři si tak většinou volí framework, který už implementuje spoustu funkcionalit pro snadnější vývoj REST architektury. Existuje celá řada frameworků zaměřujících se na vývoj webových aplikací založených na REST architektuře. Mezi některé známé frameworky patří například Spring Boot pro jazyk Java, Rails pro Ruby, Fast API pro Python, Node.js pro Javascript a ASP.NET Core pro C#. V této práci jsem vybral poslední zmiňovaný framework.

ASP.NET Core 6 je open source webový framework od firmy Microsoft založený na technologickém stacku .NET 6. Programovací jazyk tohoto frameworku je jazyk C# (lze

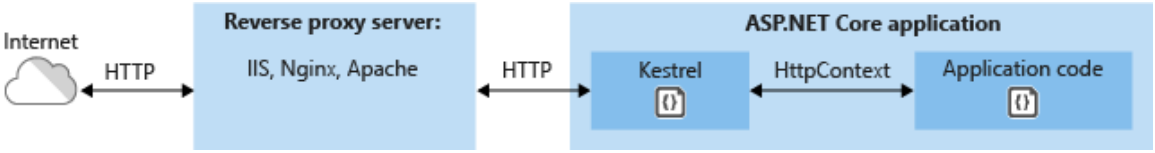
[3] JWT – Json Web Token

použít i jazyky jako F#, protože se všechny překládají do jazyka CIL[4]). Framework nabízí šablony MVC, Razor Pages a Blazor Server pro vývoj serverových aplikací s UI. Pro vývoj SPA[5] existuje Blazor WebAssembly. Pro vývoj serverů s REST API se používá šablona Web Api. Všechny tyto šablony lze kombinovat. To znamená, že například UI aplikace v MVC může zároveň poskytovat veřejné REST API.[26]

Tento framework jsem vybral na základě svých zkušeností. Dalším důvodem je, že celý projekt a všechny jeho části mohou být napsány pomocí jednoho frameworku a jednoho programovacího jazyka. .NET navíc funguje nativně jak na Windows, tak Linux operačním systému a podporuje kontejnerizaci.

ASP.NET Core aplikace

Aplikace běží v procesu implementace HTTP serveru. Server naslouchá všem HTTP požadavkům a předává je do aplikace jako objekt třídy `HttpContext`. Tato třída kompletně zapouzdřuje celý požadavek společně se všemi hlavičkami. Obsahuje i HTTP odpověď, která se v době zpracování požadavku modifikuje. Výchozí implementace HTTP serveru je server Kestrel. Oproti jiným možnostem jako IIS, IIS Express nebo HTTP.sys je kompatibilní napříč všemi platformami a je ze všech nejrychlejší. Splňuje všechny požadavky pro vytváření REST API serverů.[2]



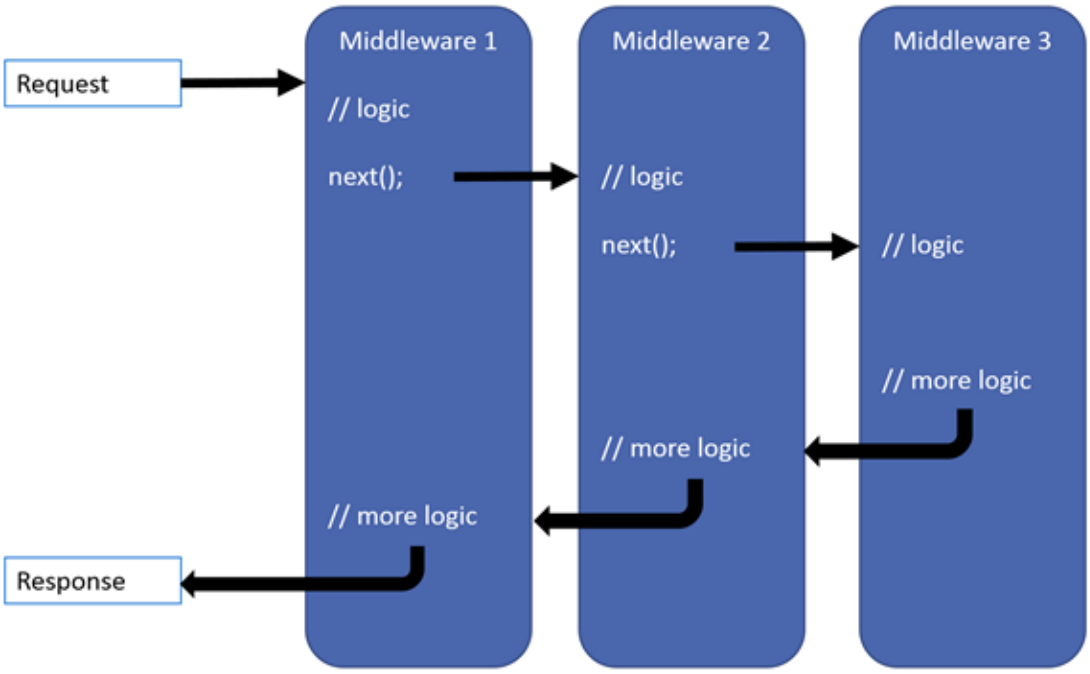
Obrázek 3.1: Architektura nasazení ASP.NET Core aplikace. Převzato z [2].

Celá aplikace může být nasazena jako server, který přímo přijímá dotazy z internetu. Takový server je však většinou nasazen za nějakým proxy serverem (Apache, Nginx, IIS), protože Kestrel neimplementuje několik funkcionalit. Nasazení je zachyceno na obrázku 3.1. Kestrel neumožňuje sdílet jednu IP adresu mezi více procesy a nefunguje směrování podle hlavičky Host. Servery jako Apache nebo Nginx toto implementují a v tomto případě se chovají jako reverzní proxy servery. Proxy server zároveň slouží jako další vrstva zabezpečení nebo jako load balancer.[2]

Middleware v ASP.NET Core

Příchozí požadavek prochází tzv. aplikační pipeline. Pipeline je tvořena několika middlewary. Procházení middlewary lze vidět na obrázku 3.2. Middleware je část kódu, který má jedinou funkci (například směrování). Framework poskytuje několik vestavěných middlewarů pro autorizaci, směrování, serializaci dat, poskytování statických souborů a další. Každý middleware může upravit `HttpContext` a buď vrátit odpověď, nebo poslat požadavek na další middleware k dalšímu zpracování.[1]

[4]CIL – Common Intermediate Language
[5]SPA – Single Page Application



Obrázek 3.2: ASP.NET Core aplikace obsahuje pipeline middlewarů, kterou prochází požadavky. Middleware je část kódu, která má na starost jednu konkrétní věc. Převzato z [1].

Směrování v ASP.NET Core

Tato sekce je převzata z oficiální dokumentace [21, 28].

Směrování implementuje vestavěný middleware. Cílem směrování je předat příchozí HTTP dotaz ke spustitelnému koncovému bodu. Koncový bod je spustitelná jednotka kódu, která vrací HTTP odpověď. Koncové body jsou konfigurovány při spouštění aplikace. Směrování mohou konfigurovat kontrolery, Razor Pages, SignalR, gRPC a různí delegáti. Pro REST API se využívá kontrolerů a delegátů. Kontroler je třída dědící z ControllerBase, kde jsou koncové body definovány metodami třídy. Delegát je metoda, která se registruje do aplikace jako koncový bod. Pro konfiguraci směrování se používají šablony cest. Každý REST API koncový bod je v aplikaci registrován pod nějakou cestou a HTTP metodou. Pokud URL dotazu odpovídá šabloně cesty a HTTP metodě, přesměruje se na daný koncový bod. Existuje spousta způsobů, jak nakonfigurovat směrování na daný koncový bod. Zmíněny jsou jen ty nejdůležitější.

První způsob se nazývá konvenční směrování. Je doporučeným směrováním pro MVC aplikace (serverové aplikace s UI). Koncové body kontrolerů se registrují pod nějakou šablonou, například `controller=Home/action=Index/id?`. Pokud dotaz obsahuje cestu `/`, bude přesměrován na kontroler `HomeController` na metodu `Index`. Stejně se přesměrují dotazy s cestou `/Home` a `/Home/Index`. Dotaz s cestou `/User/Picture` bude přesměrován na kontroler `UserController` do metody `Picture`. Pokud kontroler s metodou neexistují, implicitně se přesměruje na `HomeController`. Id je volitelné. Pokud se v cestě uvede, bude předán jako parametr do patřičné metody.

Ve Web Api se koncové body častěji konfigurují přes atributy dekorující třídy a metody. Tento způsob je uveden v ukázce 3.2. Cestu na koncový bod konfiguruje i metoda registrující

delegáta. Tento případ je zobrazen na ukázce 3.1. V obou případech se společně s šablonou cesty nakonfiguruje i zvolená HTTP metoda. Vytvoří se dvojice šablony cesty a HTTP metody, na které se přesměrovávají odpovídající dotazy. Následují příklady vytvořených dvojic.

- **POST api/test** – Na takto nakonfigurovaný bod se může přesměrovat pouze dotaz s cestou `/api/test` s metodou POST.
- **GET api/[controller]/{id:int}** – Tato šablona použita například v kontroleru `HomeController` dosadí do cesty jméno kontroleru a vznikne `api/home/id:int`. Dotaz se sem přesměruje s cestou `api/home/5`. Poslední segment musí být číslo.
- **GET api/test?id=12&next=5** – Parametry dotazu se v šablonách nespecifikují. Pokud však nějaká metoda představující koncový bod obsahuje v parametrech nějaký primitivní datový typ jako `int`, bude se na něj mapovat parametr z URL adresy. Podobný parametr se objevuje v ukázce 3.2.

Jednoduchá implementace REST API v ASP.NET Core

Na následující ukázce kódu 3.1 je naprogramováno jednoduché REST API způsobem Minimal API[6]. Využívá zmíněných delegátů. Pomocí jednoduchých metod se zaregistrují koncové body, které jsou namapovány na určitou kombinaci cesty a HTTP metody. Framework se stará o serializaci dat z těla dotazu do parametru delegáta. Z parametrů zároveň rozpozná třídy, které jsou zaregistrované v DI (dependency injection) kontejneru. Takovou třídu pak podle konfigurace vytvoří a předá do koncového bodu.

```
1  var builder = WebApplication.CreateBuilder(args);
2  builder.Services.AddScoped<ITaskService, TaskService>();
3  var app = builder.Build();
4
5  //GET metoda namapována na http://{host:port}/api/test/b57098ae-6008-4d0a-af65-3216038216f9
6  //'id' se získá z cesty, implementace 'taskService' se získá z DI (dependency injection) kontejneru
7  app.MapGet("api/test/{id}", (ITaskService taskService, Guid id) =>
8  {
9      var task = taskService.GetTask(id);
10     return task is null ? Results.NotFound() : Results.Ok(task);
11 });
12 //POST metoda namapována na URI adresu http://{host:port}/api/test
13 //'model' se získá z těla dotazu
14 app.MapPost("api/test", (ITaskService taskService, TaskModel model) =>
15 {
16     var task = taskService.CreateTask(model);
17     return Results.Ok(task);
18 });
19
20 app.Run();
21
22 //Definice modelu, API vrací serializovanou podobu
23 public record TaskModel(string Name, string Description);
```

Výpis 3.1: Ukázka implementace jednoduchého REST API serveru ve frameworku ASP.NET Core.

[6]Minimal API https://docs.microsoft.com/en-us/aspnet/core/fundamentals/minimal-apis?view=aspnetcore-6.0

Způsob Minimal API lze použít pouze ve verzi .NET 6 a výš. Tradičnějším a stále používanějším přístupem jsou kontrolery. To jsou třídy, kde se definují a implementují koncové body. Tyto třídy jsou pak zaregistrovány do aplikace. Cesta a HTTP metoda dotazu se konfiguruje pomocí atributů. Na další ukázce 3.2 je vidět kontroler s jedním koncovým bodem. Třída může mít libovolný počet koncových bodů. Třídy z DI kontejneru se získávají přes konstruktor. Parametrem metody je buď objekt, který je serializován z těla dotazu, nebo nějaký primitivní datový typ jako `int`, `string`, `guid` použitý v cestě nebo jako parametr dotazu v URI adrese.

```
1  //Všechny koncové body budou mít prefix 'api/query'
2  [Route("api/[controller]")]
3  [ApiController]
4  public class QueryController : ControllerBase
5  {
6      protected ITaskService taskService;
7
8      public QueryController(ITaskService taskService)
9      {
10         this.taskService = taskService;
11     }
12
13     //GET metoda namapována na http://{host:port}/api/query/path?name=petr&nextparam=123
14     [HttpGet("path")]
15     public List<TaskModel> GetFilteredTasks(string name, int? nextParam = null)
16     {
17         return taskService.GetFilteredTasks(name, nextParam).ToList();
18     }
19 }
```

Výpis 3.2: Implementace jednoduchého REST API koncového bodu pomocí kontroleru v ASP.NET Core frameworku.

Kapitola 4

Návrh architektury služby

V této kapitole se detailně popisuje návrh celé služby. Nejdříve se rozebírají požadavky služby. Jak se služba bude používat a jak by měla fungovat. Dále se uvede použitá architektura a její alternativy.

4.1 Požadavky

Služba by měla umožňovat vytváření a spouštění uživatelských úloh. Aplikace umožňuje registraci nových uživatelů a přihlášení pro existující uživatele. Uživatel po úspěšné registraci a úspěšném ověření přes e-mail získá právo na spouštění úloh. Zároveň získává určitou částku nějaké měny, kterou utrácí za vytvořené úlohy. Úlohu vytvoří tak, že nahraje na server soubor, který specifikuje, co se bude počítat. Uživatel může vytvořit libovolný počet úloh. Server úlohy řadí do fronty podle času vytvoření úlohy a priority. Prioritní úlohy jsou úlohy založené VIP uživatelem. VIP uživatel platí za úlohy více. Uživatel získává VIP status po kontaktování administrátora. Navýšení zůstatku na uživatelském účtu provádí pouze administrátor.

Fronta bude neustále narůstat, pokud se v systému nevyskytuje žádný aktivní worker. Worker je aplikace (proces), která postupně odebírá úlohy z fronty. Odebranou úlohu následně spouští jako kontejner a čeká na jeho ukončení. Worker získává úlohu tím, že se dotáže na server (podle typu komunikace může server přidělit úlohu konkrétní worker aplikaci). Může existovat několik instancí workeru, fronta se tak zpracuje rychleji. Worker aktualizuje stav úlohy. Uživatel dostává notifikace o změně stavu spuštěných úloh. Jakmile se úloha dokončí, uživatel může stáhnout soubor s výsledkem.

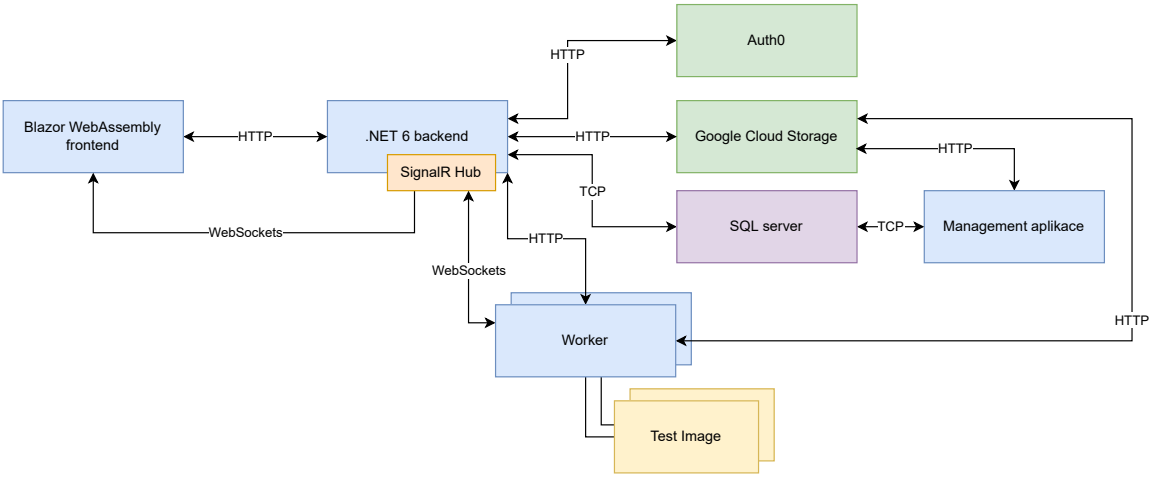
V systému existuje administrátorská role. Administrátor není standardní uživatel, nemůže zadávat úlohy. Bude moci sledovat všechny uživatele, úlohy a statistiky. Může blokovat uživatele, kterému bude kompletně znemožněn přístup ke službě. Blokace následně jde zrušit. Administrátor dále může měnit systémové proměnné jako částku, kterou dostane nově zaregistrovaný uživatel, nebo dobu, po jaké se smažou soubory a podobně. Administrátor může přidělit nebo odebrat VIP status uživatele a přidávat určitou částku na účet uživatele.

Celý systém by se tak měl skládat z klientské aplikace poskytující uživatelské prostředí, serveru, databáze a workerů. V systému figurují uživatelé (někteří mohou být VIP), administrátoři a workeři.

Shrnující diagram případů užití služby je v příloze A.1.

4.2 Návrh architektury

Návrh architektury je zobrazen na následujícím obrázku 4.1. Skládá se z celkem sedmi částí. Modře označené části představují aplikace nebo služby, které jsem naprogramoval. Zelenou barvou jsou zobrazeny existující služby, které jsem v práci využil a zintegroval s ostatními částmi systému. V této sekci se popíší jednotlivé části této architektury.



Obrázek 4.1: Architektura služby. Větší diagram se nachází v příloze A.4.

Model Blazor WebAssembly ASP.NET Core Hosted

V ASP.NET Core frameworku jsem zvolil právě tento hostovací model. Tento model je hostován na serveru Kestrel. Použití tohoto modelu má za následek vytvoření jedné aplikace (například v Dockeru vznikne po zabalení pouze jeden kontejner). Prakticky je aplikace rozdělena na frontend a backend část, jak je vidět na obrázku 4.1. Tyto části spolu nesdílí žádný kód. V backend části, kde se konfiguruje server Kestrel, je však nastaveno, aby Kestrel zároveň poskytoval statické soubory pro prohlížeč. Výhodami je jednoduchost nasazení takovéto aplikace. Jakmile se stáhnou statické soubory, aplikace může fungovat bez internetu, pokud se nepotřebuje provést HTTP dotaz na REST API. Může se použít cookie autentizace, protože frontend a backend aplikace existují ve stejné doméně (nelze nastavovat cookie z jiné domény). Pokud se přejde na url adresu `https://{doména}/..` z prohlížeče, otevřou se klasické statické webové stránky (akorát pomocí WebAssembly). Pokud se přistupuje na url adresu `https://{doména}/api/..`, přistupuje se k REST API. To je samozřejmě dostupné i mimo prohlížeč. Tento model je v podstatě návrhový vzor BFF[1].

Ve větších produkčních systémech se tento způsob používá tak, že backend zajišťuje pouze autentizaci typicky přes cookie. Ostatní dotazy pak směruje na jiné služby v systému, chová se jako reverzní proxy. V této práci by takové řešení bylo zbytečné, projekt by se stal zbytečně komplexnějším.

Pokud by existoval nějaký požadavek, který by vyžadoval rozdělení aplikace do dvou kontejnerů/aplikací, existuje několik jiných hostovacích modelů v ASP.NET Core frameworku.

1. První alternativou je hostovat REST API (ASP.NET Core 6 backend) na serveru Kestrel. Statické stránky (SPA) by poskytovala už jiná instance Kestrel serveru. Bla-

[1] BFF – Backend for frontend

zor stránky jdou nasadit i na kompletně jiný server, jako je třeba Nginx. Takto nasazené aplikace však znemožní cookie autentizaci.

2. Další řešení by mohlo být realizováno přes model Blazor Server. Tento model zajišťuje větší bezpečnost oproti předchozímu návrhu, jelikož Blazor Server je serverová technologie (UI je renderováno na serveru) a mohly by se opět použít cookie.

Výhodou tohoto frameworku je, že lze měnit typ modelu nasazení snadno a rychle (například na Blazor Server). Většinou se jedná pouze o konfigurační záležitosti, do kódu se v podstatě nemusí zasahovat.

ASP.NET Core 6 backend

Backend aplikace obsluhuje databázi a komunikuje se službou Google Cloud Storage pro ukládání souborů. Je zintegrována se službou Auth0, díky které se uživatelé přihlašují a autorizují. V této aplikaci se dále hostuje SignalR[2] Hub. Tak jak v aplikaci existují REST API koncové body, tak zde vznikají další koncové body pro SignalR. SignalR je knihovna implementující oboustrannou komunikaci v reálném čase bez aktivního čekání přes protokol WebSocket[3]. Na serveru se uchovává fronta s úlohami. Server rozděluje práci worker aplikacím právě pomocí SignalR. Pokud worker neimplementuje SignalR komunikaci, může úlohu získat přes HTTP dotaz. Pomocí algoritmu round robin se zajišťuje, aby neprioritní úlohy nehladověly.

Blazor WebAssembly frontend

Frontend jsem napsal v technologii Blazor, která funguje na WebAssembly. To mi umožnilo naprogramovat aplikaci v programovacím jazyce C# a kompletně se vyhnout tradičnímu Javascriptu. Jak bylo v předchozích odstavcích zmíněno, aplikace je nasazena na serveru Kestrel, který poskytuje prohlížeči statické soubory. Kromě komunikování s REST API se aplikace připojuje na SignalR Hub. Díky tomu dokáže aplikace zobrazovat uživatelům notifikace v reálném čase. Alternativně lze použít jakýkoli Javascript framework. ASP.NET Core podporuje stejný hostovací model jako tento i pro Javascript. Klientská SignalR knihovna existuje i pro Javascript.

Worker – Aplikace spouštějící výpočty

Aplikace má za úkol spouštět výpočty. Worker je navržen tak, že naslouchá příchozím zprávám ze SignalR Hub hostovaném na serveru. Zprávy obsahují informace o úloze, která se má spustit. Úloha se spustí jako kontejner. Události informující o změně stavu úlohy se posílají zpět na server (úloha se dokončila úspěšně, neúspěšně apod.). Tyto události se zobrazují danému přihlášenému uživateli v reálném čase (díky SignalR spojení). Worker spouští demonstrační Docker image, který pouze simuluje výpočet. Tento image se dá vyměnit za jiný, který už implementuje nějaký skutečný výpočet.

[2] SignalR https://en.wikipedia.org/wiki/SignalR
[3] WebSocket https://en.wikipedia.org/wiki/WebSocket

Management aplikace

Tato aplikace má za úkol vykonávat rutinní práce jako mazání starých souborů. Aplikace byla implementována pomocí balíčku Hangfire[4] pro .NET. Balíček umí plánovat práce a vykonávat je po určité době. Pokud se objeví nový požadavek, například mazat neaktivní uživatele, lze rutinu naprogramovat a naplánovat ji do této aplikace.

SQL server

Pro databázi byl zvolen SQL Server. Databáze byla vytvořena pomocí ORM Entity Framework technikou Code-First[5]. Nejprve se v kódu vytvoří třídy všech entit. Následně se nakonfiguruje databáze a vytvoří se migrace. Po spuštění migrace se vytvoří databáze. Pro přidání entity nebo nějaké položky se přidá nová migrace. Není problém použít jinou SQL databázi jako PostgreSQL, MySQL apod., jelikož se jedná o konfigurační záležitost. Jediným kritériem je podpora dané databáze Entity Frameworkem.

Google Cloud Storage

Systém potřebuje ukládat soubory. Zvolil jsem Google Cloud Storage kvůli jednoduchému nastavení a jednoduché správě. Alternativou je například Azure Storage. Ukládání souborů přes některého cloud poskytovatele jsem navrhl kvůli tomu, že se systém může nasadit distribuovaně. Každá část, nejen server, potřebuje pracovat s úložištěm. Pokud by byl celý systém nasazen na jednom stroji, soubory by se mohly sdílet přes souborový systém. Pokud by se jednalo pouze o prostředí Dockeru nebo Kubernetes, kontejnery by sdílely volume.

Auth0

Auth0 je poskytovatel identity a OIDC (OpenIdConnect). Služba implementuje nejnovější protokoly OAuth2 a OpenIdConnect. Vytváří jednoduchou platformu pro zajištění autentizace a autorizace k jakékoli webové i newebové aplikaci. Zároveň spravuje a ukládá všechny informace o uživatelích. Služba je komerční. Za cenu přijatelných limitů nabízí většinu funkcionalit zdarma. Alternativou Auth0 je služba Okta, která však poskytuje ještě více komplexnější řešení, zbytečná pro tento projekt. Firma Google poskytuje službu Firebase. Ta však nepodporuje přihlašování přes e-mail a heslo přes OpenIdConnect. Služba sice podporuje přihlášení přes některého OIDC poskytovatele, sama ale poskytovatelem OIDC není. Na rozdíl od Auth0 nemá oficiální SDK pro C#.

4.3 Stavový protokol úlohy

Vytvořil jsem jednoduchý protokol, který popisuje všechny stavy úlohy. Zaručuje, že se systém bude chovat deterministicky. Protokol jsem znázornil pomocí stavového automatu. V diagramu A.3 jsou zaznamenány všechny možné stavy. Pro každý stav je vysvětleno, jak se do daného stavu úloha dostane. Protokol umožňuje rušit úlohy ve frontě, zaručuje restartování zhavarované úlohy a poskytuje možnost implementování ochrany proti zacyklení.

[4] Hangfire https://www.hangfire.io/
[5] Code-First https://www.entityframeworktutorial.net/code-first/what-is-code-first.aspx

4.4 Návrh databáze

Na schématu v příloze A.2 je zobrazen návrh databáze. Navrhl jsem použít databázový server SQL Server. Díky Entity Frameworku však na volbě moc nezáleží. Framework podporuje většinu existujících databází. V reálné implementaci se tabulky mírně liší. Více v kapitole o implementaci 5.

Tabulka uživatelů je navržena obecně a v konečném systému ji bude implementovat autorizační služba Auth0, která potřebná data uchovává. V tabulkách se bude pouze vyskytovat Id uživatele od této služby.

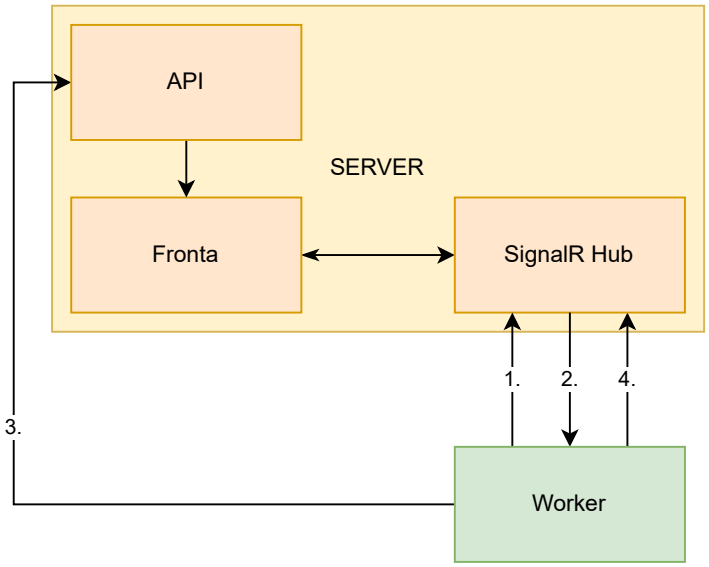
Databáze podporuje ukládat všechny změny stavu úlohy pomocí tabulky Události. K záznamu o úloze je vždy uvedena i provedená platba uložená v tabulce Platby. Systém následně potřebuje ukládat uživatelské statistiky, blokace a zůstatky na uživatelských účtech. Do databáze se pomocí tabulky Konfigurace ukládají veškeré systémové proměnné, které mohou být měněné administrátorem.

4.5 Detail komunikace mezi serverem a worker aplikacemi

V této sekci se detailněji popisuje architektura a komunikace mezi serverem a worker aplikacemi. Nejdříve se zmíní, jak je navržena architektura, která se i implementovala. Poté se rozeberou alternativy tohoto návrhu, jejich výhody a nevýhody. Uvedou se také problémy jako hladovění neprioritních úloh.

SignalR řešení

Tento návrh komunikace mezi serverem a workery je v této práci implementován. Obrázek 4.2 ukazuje potřebné části serveru a komunikaci mezi serverem a workerem. Server udržuje v paměti frontu úloh. Server hostuje SignalR Hub, na který se připojují klienti. Jakmile se podaří navázat spojení, může server posílat zprávy klientovi nebo naopak. Díky tomu lze snadno implementovat notifikace nebo obousměrnou komunikaci. Hub se zde využívá pro komunikaci s workery. Server přes Hub přidělí úlohu danému workerovi hned po jeho připojení na Hub. Worker následně začne zpracovávat úlohu. Nejdříve stáhne potřebný soubor. Následně se oznamuje serveru událost, že se podařilo získat všechny zdroje pro spuštění úlohy. Worker spouští image, kam přimontuje volume se staženým souborem. Poté se opět zasílá na server informace o této události. Jakmile se vygenerovaný kontejner dokončí, zasílá se zpráva o jeho dokončení. Server okamžitě přidělí další práci, pokud je fronta neprázdná.



Obrázek 4.2: Diagram zachycující komunikaci mezi workerem a serverem pomocí SignalR.

1. Worker se připojí na server.
2. Server odebere úlohu z fronty a přidělí úlohu workerovi.
3. Worker posílá události informující o změně stavu na API. (lze i přes SignalR)
4. Worker oznamuje dokončení úlohy. Poté se přejde na druhý krok.

Systém takto funguje, pokud je fronta neprázdná. Jakmile je prázdná, worker nedostane úlohu a přejde do klidového stavu. Jakmile se přes API vytvoří nové úlohy, které se vloží do fronty, server vzbudí workery. Následně se pokračuje druhým krokem.

Navržen je i jednoduchý systém, který zabraňuje zacyklení úloh. Administrátor nastaví v systému, jak dlouho se může úloha počítat. Worker tuto hodnotu získá přes REST API. Pokud se úloha nestihne vykonat za nastavený čas, worker úlohu zruší.

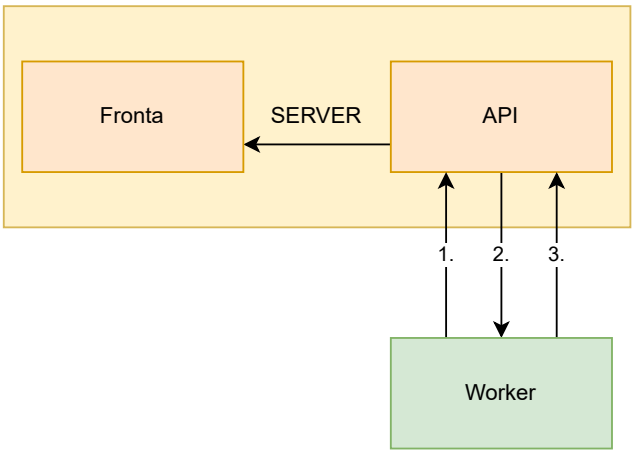
Výhodami tohoto řešení je jednoduché monitorování a úplná kontrola serveru. Server zná připojené workery, zná, jakou úlohu daný worker počítá. Pokud se worker nečekaně ukončí, server událost vždy zachytí díky SignalR implementaci. Díky obousměrné komunikaci lze realizovat i zrušení běžící úlohy.

Řešení přes HTTP dotazování

Worker jsem napsal také v programovacím jazyce C#, který obsahuje klientskou knihovnu SignalR. Existují jazyky nebo frameworky, které neimplementují tento protokol. Proto jsem navrhl další architekturu zobrazenou na obrázku 4.3, kde jedinou změnou oproti aktuálnímu návrhu (SignalR řešení) je způsob, jakým worker získává úlohu. Řešení komunikace je tedy přes tzv. HTTP polling (periodické dotazování na server). Worker se bude neustále dotazovat na REST API, zda ve frontě není nějaká úloha. Pokud je, server vrátí potřebné informace v odpovědi. Aktuálně implementovaný server poskytuje i takovéto REST API, umožňující napsat worker aplikaci v jakémkoli jiném jazyce podle tohoto návrhu.

Jedná se o nejjednodušší implementaci problému komunikace mezi serverem a workerem, avšak existují jisté problémy. Pokud worker zhavaruje, server tuto událost nedokáže zachytit. V aktuálním návrhu je možnost implementovat rušení běžících úloh. V této archi-

tektuře by se taková funkcionalita implementovala složitě (například přes webhook). Přes SignalR lze jednodušeji implementovat monitorování.

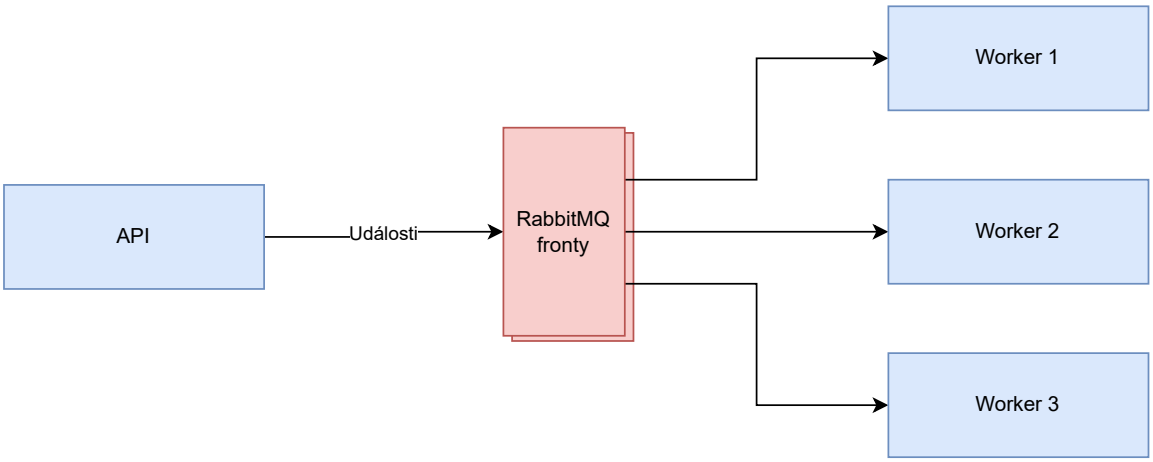


Obrázek 4.3: Komunikace mezi workerem a serverem pomocí HTTP. V prvním kroku se worker periodicky dotazuje na server. Krok se opakuje, pokud je fronta prázdná. Jakmile fronta není prázdná, worker získává úlohu a spustí ji. Poté worker posílá události o změně stavu úlohy.

Řešení přes message brokera

Navržený systém se může považovat za distribuovaný, ničemu nebrání nasadit řešení na více počítačů. Proč nevyužít message brokera jako je RabbitMQ pro posílání zpráv mezi částmi systému a uchovávání fronty úloh?

Návrh s brokerem by mohl vypadat následovně, obrázek 4.4. Fronta se na serveru vůbec nemusí implementovat. Stačí, když se vytvoří zpráva o vytvořené úloze a pošle se na exchange brokera. Zprávy se uchovávají do front brokera, na kterých naslouchají workeři. Broker zajistí distribuování zpráv mezi workery pomocí load balanceru. Worker tak dostane úlohu, kterou vykoná stejně jako v předchozích návrzích. Informace o spuštění a dokončení úlohy se mohou opět poslat přes brokera nebo přímo na REST API.



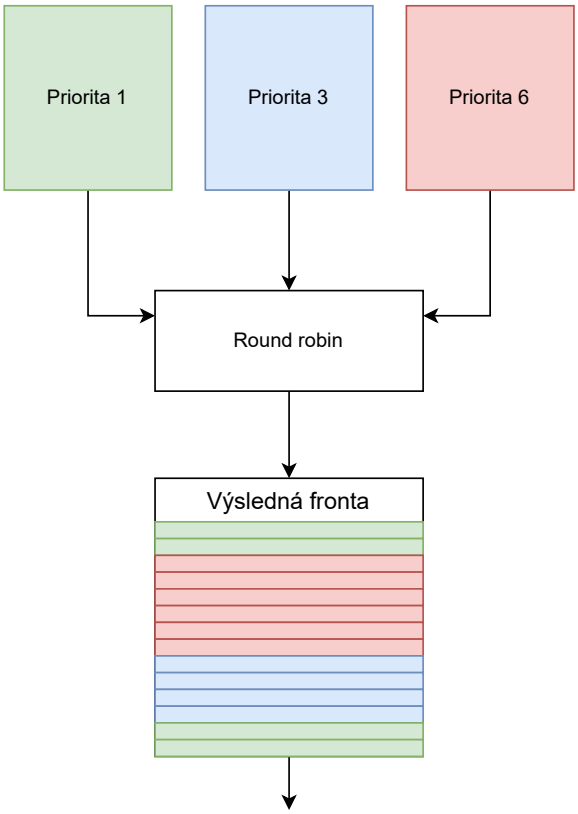
Obrázek 4.4: Řešení fronty a rozdělování úloh přes message brokera RabbitMQ.

Ačkoli toto řešení může znít elegantně, narazil jsem na několik problémů. První problém je v tom, že server už nemá pod kontrolou frontu. Jak se budou rušit úlohy, které jsou ve frontě? Co když zhavaruje RabbitMQ? Jak se bude řešit priorita úloh, popřípadě hladovění? Na všechny zmíněné otázky existují odpovědi a řešení. Celý systém se však velice komplikuje. Ať už se vybere řešení přes SignalR nebo HTTP polling, obě řešení jsou velice jednoduchá na implementaci. Implementování systému přes RabbitMQ přináší nové problémy, které se zkrátka v této práci nevyplatí řešit. O zvolení architektury s RabbitMQ jsem uvažoval už na začátku. V době implementace takové architektury jsem si však uvědomil, že se práce velmi komplikuje, a proto jsem na konec zůstal u prvního návrhu.

Fronta

Fronta se získává dotázáním do databáze a udržuje se v paměti po celou dobu běhu aplikace. To snižuje počet přístupů do databáze a zefektivňuje tak běh služby.

Fronta by měla prioritizovat úlohy spuštěné od VIP uživatelů. Může vznikat situace, kdy se neprioritní úlohy nikdy nespustí, protože je stále budou předbíhat prioritní úlohy. Tento problém jsem navrhl řešit pomocí algoritmu round robin a vytvořením několika front. Každá fronta má nějakou prioritu. Tato služba bude mít pouze frontu pro standardní úlohy a frontu pro prioritní úlohy. Podle vybrané architektury lze přidat třetí nejvíce prioritní frontu pro úlohy, které zhavarovaly z důvodu chyby workera. V tomto projektu je tato fronta také implementována. Algoritmus round robin pak bude spravedlivě odebírat úlohy podle vah front, jak je znázorněno na obrázku 4.5.



Obrázek 4.5: Jednoduché plánování pomocí round robin algoritmu řešící hladovění neprioritních úloh.

4.6 Návrh systému administrace uživatelů

Služba musí uživatelům poskytovat přihlašování a registraci. Mezi další požadavky správy uživatelů a administrace patří: role uživatelů, blokování uživatelů a možnost měnit uživatelské údaje. V této sekci se nejdříve rozebírají možnosti, jak řešit tyto problémy pomocí zvoleného frameworku. Následně se představí použitý návrh systému administrace pomocí služby Auth0.

Řešení ve frameworku ASP.NET Core

Ve frameworku ASP.NET Core existuje Identity systém[6]. Nenabízí kompletní řešení správy uživatelů, pouze usnadňuje její implementaci. Systém neimplementuje autentizaci ani autorizaci. Například zabezpečení REST API musí zajistit vývojář. Systém pouze poskytuje sadu tříd, které vývojář může použít pro správu uživatelů, správu rolí a podobně.

Další možné řešení je použití nástroje IdentityServer4[7]. IdentityServer4 implementuje autentizaci i autorizaci pomocí protokolů OAuth2 i OpenIdConnect. Pro správu uživatelů může využívat zmíněný ASP.NET Core Identity systém. Vývojář ale stále musí značnou část naprogramovat a nakonfigurovat, což je v tomto případě časově náročné.

Nezvolil jsem ani jedno zmíněné řešení. IdentityServer4 nabízí řešení pro komplexnější a složitější webové služby, než je tato. Pro tento projekt je zbytečný a komplikovaný. Rozhodl jsem se nevyužít ani Identity systém, protože stále vyžaduje vlastní implementaci zabezpečení REST API a dalších funkcionalit.

Řešení pomocí existující služby

Nakonec jsem tedy zvolil řešení přes využití nějaké existující služby nabízející kompletní administraci uživatelů a implementující nejnovější protokoly OAuth2 a OpenIdConnect. Dospěl jsem k závěru, že je lepší zvolit takovéto řešení než implementovat vlastní. Správné řešení správy uživatelů, jejich autentizace a autorizace spadá mezi těžší části vývoje webové aplikace, protože se musí dbát na zabezpečení údajů a celého procesu autentizace a autorizace. Vlastní řešení je časově náročnější a může do těchto procesů zavést chyby.

Mezi službami poskytujícími identitu jsem zvolil službu Auth0, kterou jsem již představil v sekci 4.2. Přihlašování je tedy navrženo následovně. Uživatel se při přihlašování přesměruje na stránku Auth0. Zde uživatel zadá své přihlašovací údaje a pokud byly údaje zadány správně, je zpět přesměrován do aplikace. Registrace probíhá podobně. V prvním kroku se přesměruje na registrační stránku Auth0. Tím se založí u této služby nový účet. Uživatel je přesměrován zpátky do implementované služby, kde však musí dokončit registraci (nastavit obrázek, přezdívku) a ověřit účet přes e-mail, který zaslala služba Auth0. Jakmile jsou oba požadavky splněny, uživatel získává právo na vytváření úloh.

Služba Auth0 implementuje kompletní správu uživatelů. Nabízí vytváření rolí v systému, blokování uživatelů a další. Lze tak jednoduše splnit výše zmíněné požadavky na správu a administraci uživatelů.

[6] https://docs.microsoft.com/en-us/aspnet/core/security/authentication/identity?view=aspnetcore-6.0&tabs=visual-studio

[7] IdentityServer4 https://github.com/IdentityServer/IdentityServer4

4.7 Návrh způsobu zpoplatnění za výpočetní služby

V této sekci se uvede návrh způsobu zpoplatnění za spuštěné úlohy. Dále jsou zmíněny vylepšení návrhu a alternativa.

V systému je zavedena určitá virtuální měna (token). Uživatel může spustit úlohu, pouze pokud má dostatek tokenů. Kolik tokenů stojí jedna úloha, konfiguruje administrátor. VIP uživatelé za úlohy platí více tokenů. Pokud uživatel zruší úlohu, tokeny jsou převedeny zpátky na účet. Tokeny se získávají pouze přes administrátora. Pouze on dokáže přidat tokeny na účet. Tento způsob tak vyžaduje komunikaci mezi administrátorem a uživatelem. Administrátor nicméně získává větší kontrolu.

Aby uživatel nemusel kontaktovat administrátora a žádat o tokeny za smluvenou částku, návrh lze vylepšit o přidání platební brány. Administrátor by pouze nastavil cenu tokenu. Brána by se mohla realizovat přes GoPay nebo Stripe.

Tento způsob může být stále nevýhodný, pokud úlohy trvají různě dlouhou dobu. Například nějaká úloha může trvat hodinu a stojí stejně jako úloha trvající minutu. Vylepšením je zavedení tzv. pay as you go způsobu. Tento způsob hojně využívají například cloud poskytovatelé jako Google, Microsoft nebo Amazon v různých formách. Uživatel platí pouze za to, jaké služby použije nebo jak dlouho službu používá. Způsob je náročnější na realizování a v této práci se neimplementuje. Nicméně dokáže řešit zmíněný problém, kdy se za úlohu, která je spuštěna kratší dobu, zaplatí méně.

4.8 Návrh REST API

Detailní dokumentace REST API se nachází v příloze E. V této sekci se objeví pouze vybrané důležité koncové body a použité konvence.

Všechny koncové body jsou označeny předponou `api`. Existuje však skupina koncových bodů nezačínající s touto předponou. Slouží především pro autentizaci, autorizaci, resetování hesla a podobně. Koncové body, které slouží pro administrátora nebo workera, jsou jasně označeny prefixem `api/admin` nebo `api/worker`. Označeny jsou proto, aby se jasně vědělo, že klient musí být autorizován patřičnou rolí případně nějakým pravidlem. Cesty, které takto nejsou označené, patří standardním uživatelům. Následují vybrané koncové body.

- **GET /auth/login** – Na tuto cestu se přihlašují uživatelé z prohlížeče. Uživatel bude přesměrován na Auth0. Využívá cookie autentizaci.
- **GET /auth/logout** – Odhlášení uživatele.
- **GET /passwordflow/login** – Přihlášení do služby z desktopové nebo mobilní aplikace. Místo cookie se získává JWT.
- **GET /api/tasks** – Dotazování pomocí protokolu OData nad kolekcí úloh přihlášeného uživatele.
- **POST /api/tasks** – Vytvoření nové úlohy.
- **GET /api/tasks/{id}** – Vrací detail úlohy společně s platbou a všemi událostmi.
- **PATCH /api/tasks/{id}** – Aktualizuje informace úlohy.
- **DELETE /api/tasks/{id}** – Smazání úlohy.

- **GET /api/tasks/{id}/file** – Vrací výsledný soubor úlohy.
- **GET /api/worker** – Worker získává novou úlohu k zpracování.
- **POST /api/admin/bans** – Administrátor vytváří novou blokaci uživatele.
- **GET /api/admin/tasks** – Dotazování pomocí protokolu OData nad celou kolekcí úloh v systému.
- **PUT /api/admin/config** – Aktualizace nějaké systémové proměnné.

Existují další koncové body pro statistiky, platby a podobně. Kompletní dokumentace REST API se nachází v příloze E.

Protokol OData

Při implementaci REST API, které vrací kolekce dat, existují některé výzvy. Jak filtrovat kolekci dat? Jak implementovat stránkování? Stránkování je důležité pro koncový bod, který vrací nějakou kolekci položek. Pokud existuje několik stovek nebo tisíc položek, sníží se efektivita, rychlost a data se nemusí vejít do HTTP odpovědi. Filtrování je stejně důležité, protože se často potřebují získat jen nějaká data, ne celá kolekce. UI aplikace potřebuje zobrazit například pouze prvních 20 položek, které začínají na uvedený řetězec. Nemusí se tak vracet celá kolekce o několika tisících položkách, stačí pouze vyfiltrovaná kolekce.

Filtrování a stránkování se realizuje pomocí parametrů v URL adrese dotazu. Implementace těchto problémů není složitá, nicméně není standardizovaná. REST architektura nespecifikuje, co přesně se má vracet v odpovědích a co přesně má obsahovat dotaz. Je tedy výhodné si vytvořit nějaký standard (protokol) definující výsledné API. Díky tomu vzniknou jednotná API, která půjdou snadněji konzumovat. Pokud například máme tři REST API servery a každý implementuje filtrování a stránkování jinak, musí pro každé API vzniknout kompletně nový HTTP klient, což není výhodné. Je lepší vytvořit jednoho HTTP klienta, který půjde využít na všechny API implementující stejný protokol určující, jak vypadá filtrování, stránkování a podobně.

Proto jsem se rozhodl navrhnout použití protokolu OData[8] vytvořený společností Microsoft. Jeho účel je právě standardizovat webová API pomocí jednoduchých pravidel a syntaxe HTTP dotazu. Protokol určuje, jak se filtrují kolekce, jak vypadají těla dotazů a odpovědí a jak vypadá URL adresa (především dotazovací parametry). Existující klientské knihovny, které implementují tento protokol, velmi usnadňují vývoj UI aplikací.

V implementované aplikaci se tak některé koncové body registrují dvakrát. Jsou to především koncové body nakonfigurované na GET metodu vracející kolekci entit. Ať už se požadavek přesměruje z jedné nebo druhé cesty, vykoná se stejný kód koncového bodu. Pro obě cesty na koncový bod se nakonfigurují dotazové parametry protokolu OData. Jedna cesta na koncový bod prochází OData middlewarem a do těla odpovědi přidává pomocné položky (metadata) tohoto protokolu. Druhá cesta se přesměruje na bod standardně. Zůstanou dotazovací a filtrovací parametry v URL adrese, ale odpověď neobsahuje žádná OData metadata. Klientské knihovny implementující tento protokol se dotazují na cesty s prefixem `odata`. Potřebují metadata pro navigování mezi stránkami nebo pro zefektivnění práce. Koncové body s prefixem `api` slouží pouze pro jednodušší serializaci dat, pokud klient neimplementuje celý protokol a pro dokumentaci API.

[8] OData https://www.odata.org/

Následuje ukázka konkrétního koncového bodu, který je konfigurován na cestu `GET api/admin/tasks`. Díky protokolu OData lze přidávat dotazovací a filtrovací parametry. Následující dotaz se poslal na cestu `/api/admin/tasks?$count=true&$top=3&$skip=1 &$orderby=CreationDate+desc&$select=Name,isPriority,ActualStatus`. Odpověď lze vidět na ukázce 4.1. V druhém dotazu se v cestě použil prefix `odata` místo prefixu `api`. Odpověď tak v ukázce 4.2 obsahuje metadata protokolu OData. V obou případech se vrací seřazená kolekce podle data `CreationDate` s položkami obsahujícími vlastnosti `Name`, `isPriority` a `ActualStatus`. V kolekci se přeskočí jedna položka a vrátí se maximálně tři.

```
[
  {
    "Name": "TaskA",
    "IsPriority": false,
    "ActualStatus": 6
  },
  {
    "Name": "TaskB",
    "IsPriority": false,
    "ActualStatus": 6
  }
]
```

Výpis 4.1: Ukázka těla odpovědi na dotaz směrovaný na koncový bod `api/admin/tasks?...` Neobsahuje žádná metadata.

```
{
  "@odata.context": "https://adresa:port/adata/admin/$metadata#Tasks(Name
    ,IsPriority,ActualStatus)",
  "@odata.count": 105,
  "value": [
    {
      "Name": "TaskA",
      "ActualStatus": "FinishedSuccess",
      "IsPriority": false
    },
    {
      "Name": "TaskB",
      "ActualStatus": "Downloaded",
      "IsPriority": false
    },
    {
      "Name": "TaskC",
      "ActualStatus": "FinishedSuccess",
      "IsPriority": false
    },
  ]
}
```

Výpis 4.2: Ukázka těla odpovědi na dotaz směrovaný na OData cestu. Obsahuje metadata protokolu. Protokol OData umí serializovat výčtový typ.

OpenApi a generování HTTP klienta

Pro dokumentaci REST API jsem navrhl použít standard OpenApi[9]. Standard umožňuje vytvořit jednoduché a jednotné rozhraní, které dokumentuje API. Jednoduše se dá například vygenerovat interaktivní dokumentace pro jiné vývojáře. Jsou vytvořeny nástroje, které pomocí této specifikace dokáží vygenerovat HTTP klienta v jakémkoli programovacím jazyce. Pokud se API správně zdokumentuje, frontend vývojář si může vygenerovat celého HTTP klienta, kterého nemusí dále modifikovat, pouze pohodlně používat.

[9]OpenApi https://swagger.io/specification/

Kapitola 5

Implementace

V této kapitole jsou rozebrány některé implementační detaily systému. Autentizace byla inspirována podle [16] (třídy `AuthorizedHandler, HostAuthenticationStateProvider`). Z oficiální dokumentace ASP.NET Core jsem využil [3, 15]. Pro vygenerování OpenApi dokumentace (nemá na funkčnost služby žádný vliv) jsem použil [20, 24].

5.1 Zabezpečení

V sekci se řeší zabezpečení přístupu do služby a databáze.

Zabezpečení uživatelského přístupu do databáze

Tímto se myslí případ, kdy uživatel přistupuje k REST API a dotazuje se na svá data. Uživatel ale nesmí přes toto API přistoupit k datům jiného uživatele.

Problém se dá naimplementovat tak, že každý uživatelský koncový bod bude kontrolovat, zda se uživatel skutečně dotazuje pouze na svá data. Pokud ve službě existuje několik desítek takových bodů, je pravděpodobné, že jako vývojář zapomenu tuto kontrolu implementovat. Jednodušší cestou je implementování middlewaru, který bude nakonfigurován tak, že se vyvolá při dotazech na uživatelské koncové body. Implementace tak bude na jednom místě a závisí jen na konfiguraci specifikující, u jakých dotazů se middleware spustí. Dalším způsobem jsou globální filtry dotazů.

Vybral jsem poslední způsob, který se pro problém hodí nejvíce. Ve třídě `AppDbContext`, která přistupuje do databáze, se v metodě `OnModelCreating` nakonfiguruje globální filtr. Při přístupu do databáze se nejdříve aplikují tyto filtry, které vyfiltrují nežádoucí data. Implementovaný filtr zaručuje, že se uživatel přistupující na API nikdy nedostane k datům jiného uživatele. Bude moci přistoupit pouze ke svým datům. Aby filtr fungoval spolehlivě, každá databázová tabulka s uživatelskými daty musí obsahovat sloupec s ID uživatele a ke každé takové tabulce se musí nakonfigurovat filtr. To lze vidět na ukázce 5.1. Filtr funguje jednoduše tak, že při každém příchozím požadavku se z HTTP kontextu získá ID uživatele. To se použije pro vyfiltrování dat. Pro administrátora jsou tyto filtry vypnuty.

```
modelBuilder.Entity<TaskEntity>().HasQueryFilter(i => i.UserId == idService.GetUserId());
modelBuilder.Entity<EventEntity>().HasQueryFilter(i => i.UserId == idService.GetUserId());
```

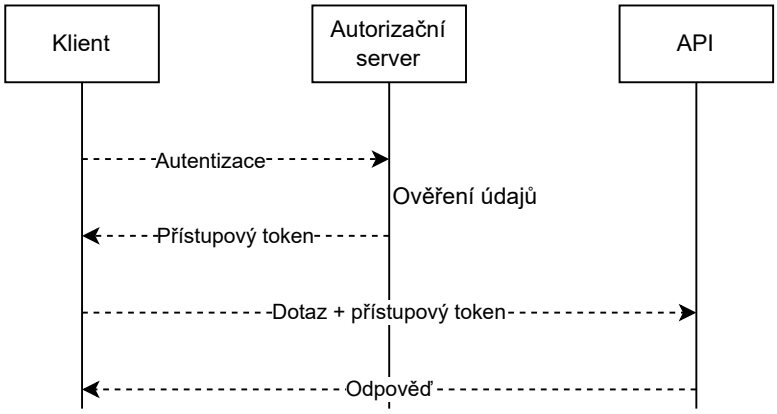
Výpis 5.1: Globální filtr nastavený pouze pro tabulky Úlohy a Události.

Zabezpečení REST API

Zabezpečení souvisí se službou Auth0 a zvolenou architekturou. Pokud jsou požadavky posílány z prohlížeče, používá se cookie autentizace a autorizace. Z tradiční aplikace se autorizuje pomocí bearer tokenu.

Obě schémata obsahují tyto autorizační údaje: role, je registrován, je vip, je ověřen e-mailem. Více na následující ukázce 5.2. REST API obsahuje koncové body, které je potřeba mít zabezpečené. Nemají být veřejné pro nepřihlášené uživatele. Některé koncové body vyžadují konkrétní roli nebo nějaké pravidlo. Implementoval jsem následující způsoby autorizace.

- **Autorizace podle role** - V aplikaci vznikají role uživatel a administrátor. Framework má vestavěný pomocný atribut `Authorize`, který dekoruje kontroler nebo metodu reprezentující koncový bod. V atributu se specifikuje role, která musí být obsažena v cookie nebo bearer schématu (například `Authorize(Roles = "admin")`).
- **Autorizace podle definovaných pravidel** - Pro autorizaci kolikrát nestačí pouze role, jsou třeba i jiná kritéria. ASP.NET Core povoluje definovat vlastní pravidla (policies). Lze specifikovat pravidlo určující, co přesně má uživatel splňovat. Uživatel musí být například registrován, musí mít nějaký nárok nastavený na `true` a zároveň musí mít danou roli (používané nároky a role lze vidět v ukázce 5.2). Dá se nastavit i omezit použité autorizační schéma. Na některé koncové body se tak lze dostat pouze pomocí cookie autentizace. Aby se koncový bod řídil takovým pravidlem, musí se dekorovat pomocí atributu `Authorize`, například `Authorize(Policies = "Pravidlo")`.
- **Autorizace pomocí přístupového tokenu** - Tento způsob využívá worker aplikace. Worker je technicky další rolí v systému. Z praktického úhlu pohledu je však použití role nevýhodné, protože pro každého workera by se u služby Auth0 musel vytvořit speciální uživatel pod speciální rolí. Lepším způsobem je, aby si workeři zažádali o přístupový token na API. Přístupový token je klasický bearer token s určitou expirací a je přímo určen pro koncové body workera. Tyto koncové body pak kontrolují, zda token byl vydán pro toto API službou Auth0. Tento token nemůže získat nikdo jiný kromě workera. Ten má v konfiguraci uložen tajný klíč (ClientSecret), díky kterému získává od Auth0 požadovaný token. Postup získání tokenu je znázorněn na obrázku 5.1.



Obrázek 5.1: Autorizace pomocí přístupového tokenu.

```
{
  "https://bp-claims.com/vip": false,
  "https://bp-claims.com/registered": true,
  "http://schemas.microsoft.com/ws/2008/06/identity/claims/role": ["user"],
  "nickname": "fila",
  "name": "Filip",
  "picture": "https://storage.com/picture.png",
  "updated_at": "2022-04-22T00:33:30.396Z",
  "email": "filip268@email.com",
  "email_verified": true,
  "iss": "https://bptasklauncher-test.eu.auth0.com/",
  "sub": "auth0|625944233841e3006a184020",
  "aud": "u15oOJyR9PhEdhn05hFfQBsYItfydSWL",
  "iat": 1650587610,
  "exp": 1650623610
}
```

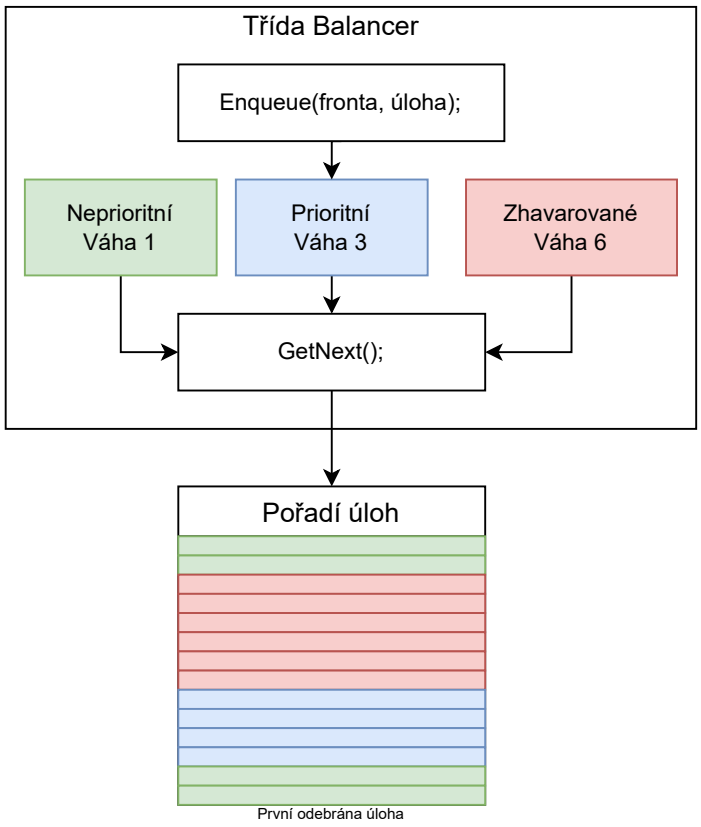
Výpis 5.2: Ukázka všech nároků a informací z bearer tokenu nebo cookie.

5.2 Fronta úloh

V této sekci se popisuje implementace fronty úloh.

Vytvořil jsem tři fronty. Fronta je implementována vestavěnou třídou `ConcurrentQueue`. Do jedné se řadí podle času vytvoření neprioritní úlohy, do druhé prioritní úlohy a do třetí zhavarované úlohy. Fronty jsou uchovány v paměti. Po vytvoření úlohy se data uloží do databáze a ihned se úloha vloží do příslušné fronty. Dále jsem zvolil nejjednodušší implementaci navrženého algoritmu round robin. Pro implementace jsem využil existující implementaci algoritmu v balíčku RoundRobin[1]. Vkládání do front a odebírání úloh podle algoritmu round robin implementuje třída `Balancer`, která je zachycena na obrázku 5.2. Fronty jsou ve třídě uloženy ve slovníku. Vložení tak vyžaduje specifikování klíče slovníku (název fronty např.: prioritní) a model reprezentující úlohu. Balíček RounRobin poskytuje třídu `RoundRobinList`, ve které se nakonfigurují fronty s váhami. Jakmile worker požádá o další práci, musí se rozhodnout, z jaké fronty se má úloha odebrat. Volá se tak metoda `Next` třídy `RoundRobinList`, která vrací frontu, ze které se odebere úloha. Implementace si nevytváří žádnou pomocnou frontu. Vnitřně si pouze pamatuje stav algoritmu (váhy front, aktuálně vrácená fronta), podle kterého se dalším voláním `Next` vrátí další položka ze stejné nebo už jiné fronty. Pokud je vybraná fronta prázdná, přeskočí se na další neprázdnou frontu.

[1] Balíček RoundRobin https://github.com/alicommit-malp/roundrobin



Obrázek 5.2: Diagram implementace fronty a rozdělování úloh ve třídě `Balancer`. Spravedlivé rozdělování je dosaženo algoritmem round robin.

Úlohy se odebírají z front tak, že se z fronty s největší váhou odebere více úloh za sebou než z fronty s menší váhou. Prioritní úlohy tak sice nebudou vždy předbíhat neprioritní, nicméně se vyřeší problém hladovění.

5.3 Worker

Tato část popisuje implementaci worker aplikace, která komunikuje se serverem pomocí SignalR a spouští výpočet. Detailně se zmíní, jak se spouští docker kontejner pomocí Docker Engine API. Popíší se zprávy, které si Hub vyměňuje s workerem.

Aplikace nejdříve získá přístupový token podepsaný službou Auth0. Ten se použije pro autorizaci na SignalR Hub a REST API. SignalR klient se připojí na Hub a začne naslouchat příchozím zprávám. Diagram zachycující tuto komunikaci je zobrazen na obrázku 4.2 v předešlé kapitole. Implementuje se návrh popsaný v sekci 4.5. Worker získá úlohu po připojení na Hub nebo po dokončení úlohy nebo pokud byla v předchozích případech prázdná fronta, po vzbuzení serverem. Aplikace zpět zasílá různé zprávy informující o stavu úlohy. Worker funguje v jednoduché smyčce. Dostane úlohu, vykoná ji a zažádá si o další.

Získávání úlohy

Je to proces, kdy worker komunikuje se serverem za účelem odebrání úlohy z fronty a následné spuštění této úlohy. Tento proces jsem implementoval následovně.

Jakmile se autorizovaný worker připojí na SignalR Hub, daný worker obdrží zprávu typu `StartTask` s přidělenou úlohou. Pokud přijde od workera na Hub zpráva `TaskStatusUpdate`, kde v těle zprávy je stav úlohy dokončen (je jedno jestli úspěšné nebo neúspěšně), worker opět získá zprávu `StartTask`. Worker na této zprávě naslouchá a jakmile ji přijme, začne ji vykonávat. Poté posílá zmíněné zprávy typu `TaskStatusUpdate` informující o změně stavu úlohy. Worker dále může posílat dvě speciální zprávy. Jedna slouží pro informaci, že se úloha nestihla dokončit – `TaskTimeouted`. Druhá oznamuje, že úloha zhavarovala – `TaskCrashed`, v aplikaci došlo k nějaké nečekané výjimce.

Pokud však fronta na serveru byla prázdná, server zpět neodešle žádnou zprávu. Worker přejde do klidového stavu. Server si poznačí do pomocné struktury ty workery, které jsou připojeni, ale nic nedělají. Toto je volitelné a jiná implementace nemusí uchovávat tyto informace. Jakmile se fronta bude plnit (uživatelé spustí úlohy), server zašle pomocí SignalR zprávu `WakeUpWorkers`. V mé implementaci se tak zprávy pošlou jen těm workerům, kteří nic nedělají. Pokud by server neuchovával informace o workerech, zpráva by se zaslala všem připojeným. V obou případech worker odpoví zprávou `RequestNewWork`, pouze pokud skutečně nic nevykonává. Server na příchozí zprávu `RequestNewWork` již odebere úlohu z fronty a zašle ji na daného workera. Toto jsem implementoval proto, aby nikdy nenastala situace, kdy server bez kontroly přímo zašle úlohu workerovi, který teoreticky může nějakou úlohu počítat. Worker tímto potvrzuje serveru, že skutečně nic nedělá a může tak přijmout úlohu. Díky tomuto řešení se tak eliminují nečekané chyby, které by bez této implementace mohly nastat.

Pokud bych implementoval druhý návrh ze sekce 4.5, nemusel bych tento problém řešit. Tato použitá implementace nicméně není složitá. Navíc jak je zmíněno v návrhu, implementované řešení je celkově spolehlivější a bezpečnější. Server má nad úlohami kontrolu a dokáže jednoduše monitorovat workery.

Nečekané odpojení workera

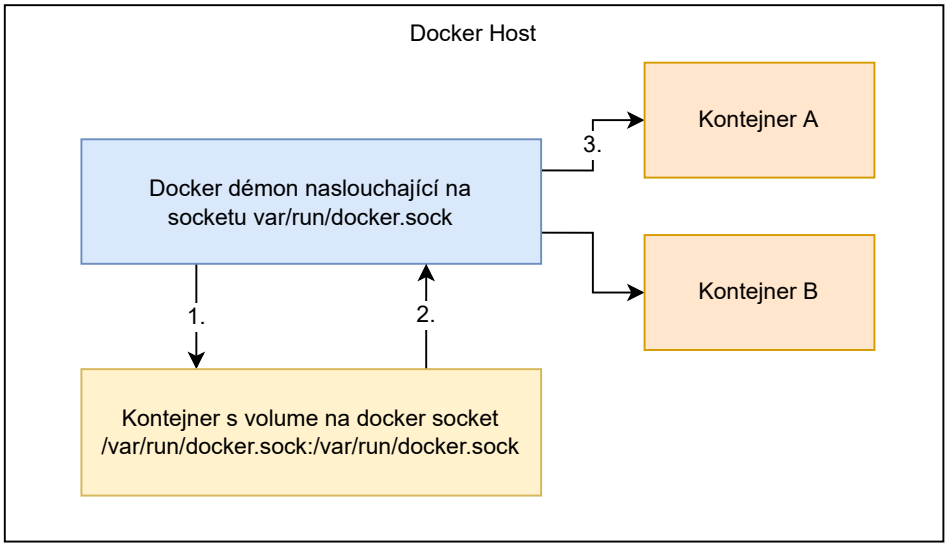
Díky SignalR protokolu je zajištěno, že server vždy obdrží událost, pokud se worker nečekaně odpojí. Jestliže worker počítal nějakou úlohu, server ji označí za zrušenou (stav `Zhavarováno`) a zařadí ji do fronty s největší váhou.

Předání souboru a zpracování úlohy

Jakmile worker získá úlohu, stáhne se z API konfigurace pomocí `GET api/worker/config`. Následně se stáhne soubor z Google Storage. Pokud aplikace běží na počítači, soubor se uloží do souborového systému na daném počítači. Tento soubor lze předat do spuštěného kontejneru pomocí bind mount. V této práci se worker spouští v Dockeru. Je tak jednodušší vytvořit volume a přimontovat ho k worker kontejneru. Do tohoto volume se uloží stažený soubor s úlohou. Stejný volume je pak přimontován do spouštěného image představující výpočet. Worker s vygenerovaným kontejnerem sdílí volume a předávají si přes něj soubor. Worker informuje server o změnách stavu úlohy ve zprávě `TaskStatusUpdate`. Nejdříve se stav změní na `Připraven`, kdy je vše připraveno ke spuštění (stažen soubor, nakonfigurován časový limit). Jakmile se podaří spustit image, změní se stav na `Spuštěno`. Po dokončení kontejneru se stav změní na jeden z dokončených stavů (`Úspěšně`/`Neúspěšně`). Ostatní stavy jsou uvedeny ve stavovém protokolu úlohy v příloze A.3. Spuštěný kontejner, který simuluje výpočet, zapíše výsledek do souboru ve sdíleném volume. Worker pak tento soubor zpátky nahraje na Google Storage po skončení kontejneru.

Spouštění image s volume

Spouštění image probíhá zasláním HTTP dotazu na Docker Engine API. Toto usnadňuje balíček Docker.DotNet[2], který implementuje komunikaci s tímto API. Knihovna nabízí třídu `DockerClient`, kterou lze pohodlně používat. Aby bylo možné komunikovat s démonem přes REST API, musí se nakonfigurovat socket, na kterém démon naslouchá. Na systémech Unix je to socket `unix:///var/run/docker.sock`, na systému Windows je to pojmenovaná roura `npipe://./pipe/docker_engine`. Jelikož se worker spouští v Dockeru a běží jako kontejner, musí být do kontejneru přimontován tento socket (používají se linux kontejnery). Kontejner jinak nemá jak komunikovat s démonem, protože v kontejneru o něm není žádná informace. Kontejner je kompletně izolován. Teorie je popsána v kapitole 2. Pokud je socket do kontejneru přimontován, může pomocí něho přes REST API ovládat démona. Tento kontejner je vyznačen na obrázku 5.3.



Obrázek 5.3: Diagram zobrazující Docker kontejner spouštějící další docker kontejnery.

1. Démon vytvoří kontejner s volume na jeho socket, na kterém naslouchá.
2. Tento kontejner přes REST API na socket zasílá dotaz o vytvoření jiného kontejneru.
3. Démon na socketu zachytí dotaz, vykoná ho a vygeneruje další kontejner.

Worker tedy přes REST API může spustit image, přimontovat volume, sledovat kontejner, čekat na skončení kontejneru, smazat ho a další. Worker tak má pod kontrolou spuštěnou úlohu, která běží ve spuštěném kontejneru. Pokud ji potřebuje zrušit, jednoduše pošle dotaz na REST API Dockeru.

Implementace zabezpečení proti zacyklení úlohy

Před spuštěním úlohy se pomocí REST API získá aktuální hodnota času v minutách, do které se musí stihnout vykonat úloha. Pokud se nestihne, bude se považovat za zacyklenou. Worker toto oznamuje zprávou `TaskTimeouted`. Úloha se poté bude nacházet ve stavu `Nedokončeno`. Uživatel stále může restartovat úlohu, protože administrátor může na jeho podnět zvýšit časový limit, protože úloha je jednoduše náročnější na výpočet.

[2]Docker.DotNet https://github.com/dotnet/Docker.DotNet

Rušení úlohy se implementuje přes vestavěnou třídu `CancellationTokenSource`, ve které se nastaví časovač na získaný časový limit. Třída poskytuje tzv. `CancellationToken`, který se předává do asynchronních metod. Jakmile uplyne časový limit, vyvolá se zrušení tokenu. Pokud asynchronní metody implementují reakci na zrušení tokenu, metoda se zruší a uklidí po sobě. Použitý balíček Docker.Dotnet toto chování implementuje. Po vyvolání zrušení tokenu, knihovna dokáže zastavit a smazat vytvořený kontejner.

Kapitola 6

Nasazení služby na veřejný cloud

V kapitole se popisuje nasazení služby na veřejný cloud server. Volí se konkrétní cloudové služby a řešení. Porovnávají se alternativy. Kapitola obsahuje ukázky nasazení aplikace, SQL serveru a kontejneru na cloudová řešení od firem Google a Microsoft.

6.1 Architektury nasazení

Služba se může nasadit buď na jeden stroj, nebo jako distribuovaný systém na více počítačů.

Virtuální stroj

Nasazení na virtuální stroj je nejjednodušší řešení. Na vytvořeném virtuálním počítači se nainstaluje a nastaví Docker. Služba se poté spustí například pomocí Docker Compose. Za virtuální stroj se obecně platí více. Musí se platit disk a další služby potřebné pro provoz virtuálního stroje. Pokud virtuální stroj spadne, spadnou všechny služby nasazené na stroji. Nasazený systém se také hůře škáluje.

Distribuovaná architektura

Distribuovaná architektura může využít více služeb. Celý systém je tak nasazen na více počítačů. Pokud například vypadne počítač nebo cloud služba s management aplikací nebo worker aplikací, nic se nestane. Webový server bude totiž nasazen na jiném stroji a uživatelé budou moci stále využívat službu. Server této služby je navržen tak, aby se obešel bez management aplikace i všech workerů. Nasazení může využívat cenově výhodné služby, díky kterým může být nasazení levnější než virtuální stroj.

6.2 Výběr cloud služeb

Mezi nejpopulárnější cloud poskytovatele patří AWS (Amazon Web Services), GCP (Google Cloud Platform) a Azure od společnosti Microsoft. Pro nasazení této služby jsem vybral řešení od Googlu a Microsoftu. Platformu AWS jsem nepoužil, protože s ní nemám žádné zkušenosti. Řešení od AWS je navíc asi nejvíce komplikované a naučit se základy používání jakéhokoli cloud poskytovatele může být časově náročné. Jak Azure, tak GCP poskytují služby pro nasazení kontejnerizovaných aplikací. Nabízí spolehlivá řešení pro nasazení SQL databází. U každého poskytovatele si lze zaplatit virtuální stroj.

Výběr nasazení SQL databáze

Azure i Google poskytují svá řešení pro vytváření SQL databází. V kapitole je ukázáno nasazení databáze na obě platformy. Celkem je jedno, jaký poskytovatel se využije. Záleželo by opravdu na specifických požadavcích. SQL služby fungují koncepčně podobně u obou poskytovatelů. Z mé zkušenosti vychází cenově platforma Azure o něco lépe.

Proč databázi nenasadit jako všechny jiné části služby do kontejneru? Nasazovat databázi v kontejneru se zkrátka nedoporučuje. SQL servery jsou daleko škálovatelnější, bezpečnější a nabízí spoustu dalších funkcionalit. Databáze v kontejneru se může bez problému použít při vývoji nebo při méně kritickém projektu. V produkčních systémech jsou databáze téměř vždy nasazeny na SQL serverech [29].

Výběr služby pro vytvoření virtuálního stroje

Google nabízí Compute Engine. Alternativa od Microsoftu se nazývá Azure Virtual Machines. Opět nelze rozhodnout, jaké řešení je lepší. Proces vytvoření stroje a připojení se na něj je jednoduchý na obou platformách. Cenově záleží na spoustě ohledů a nelze určit, jaká služba je cenově výhodnější[1]. Dostupnost serverů v Evropě je skvělá u obou poskytovatelů. Záleží tedy zase na různých požadavcích. V mém řešení jsem pouze potřeboval virtuální stroj schopný nainstalovat Docker, spustit kontejnery a přesměrovat provoz na nějaký veřejný port virtuálního stroje. Tyto požadavky splňují obě řešení.

Nakonec používám Azure, který mi opět vycházel levněji.

Výběr služby pro spouštění kontejnerizovaných aplikací

GCP nabízí především tři způsoby. Prvním je GKE (Google Kubernetes Engine). Nabízí nejrobustnější řešení pro správu a spouštění kontejnerů. Druhým je zmíněný GCE (Google Compute Engine), kde se vytváří virtuální stroje. Toto řešení nabízí i operační systém, který je optimalizován přímo pro spouštění kontejnerů. Posledním způsobem je služba Google Cloud Run. Ta nabízí jednoduché nasazení jednoho kontejneru. Nepodporuje však spouštění více kontejnerů. Na to se musí využít virtuální stroj nebo Kubernetes. Cloud Run je oproti těmto službám nejjednodušším a nejlevnějším řešením pro spuštění jednoho kontejneru. Nabízí funkcionality jako škálování kontejnerů. Za cílem snížení nákladů lze kontejner nastavit tak, aby se spustil jen při příchozím požadavku.

GKE alternativou je v Azure tzv. AKS (Azure Kubernetes Services). Platforma dále nabízí ACI (Azure Container Instances) a ACA (Azure Container Apps), které jsou možnou alternativou služby Cloud Run. ACA, na rozdíl od Cloud Run, nabízí spouštění několika kontejnerů a jejich orchestraci pomocí Kubernetes. Na rozdíl od AKS neposkytuje přímý přístup do Kubernetes. Služba ACI zase nepodporuje škálování a vyrovnání zátěže. I přes to, že pomocí ACI nebo ACA lze nasadit kontejner, Cloud Run nabízí jednodušší a levnější řešení nasazení jednoho kontejneru. Azure nicméně poskytuje ještě službu Azure App Service. Ta je zintegrována například se službou ACA a dalšími. Aplikaci lze nasadit na službu v kontejneru i bez kontejneru. Služba velmi zjednodušuje nasazení jakékoli aplikace. Navíc nabízí úroveň použití, která je kompletně zdarma (omezená některými limity jako CPU čas a podobně). Azure App Service dále poskytuje různé další nástroje pro správu REST API a .NET aplikací.[17]

[1] Srovnání služeb https://acloudguru.com/blog/engineering/cloud-comparison-aws-ec2-vs-azure-virtual-machines-vs-google-compute-engine

Pro nasazování kontejneru s management aplikací jsem tak z těchto možností zvolil Cloud Run, protože nabízí nejjednodušší způsob nasazení jednoho kontejneru. Dále jsem využil Azure App Service pro nasazení ASP.NET Core server aplikace (backend a frontend služby) kvůli zmíněným výhodám. Kubernetes jsem v projektu nepoužíval. Pokud by bylo třeba, vybral bych si GKE od Googlu.

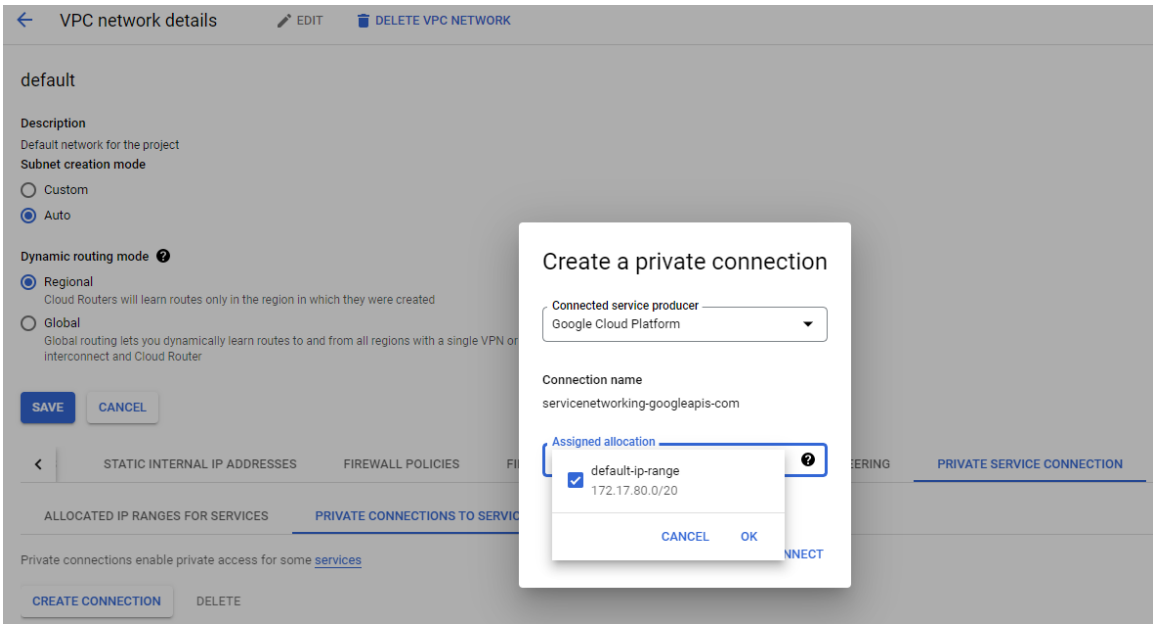
6.3 Nasazení databáze

Základem je vytvořit databázi, protože jinak nebude fungovat server ani management aplikace. Tyto aplikace se nebudou moci bez SQL databáze nasadit.

Google SQL server

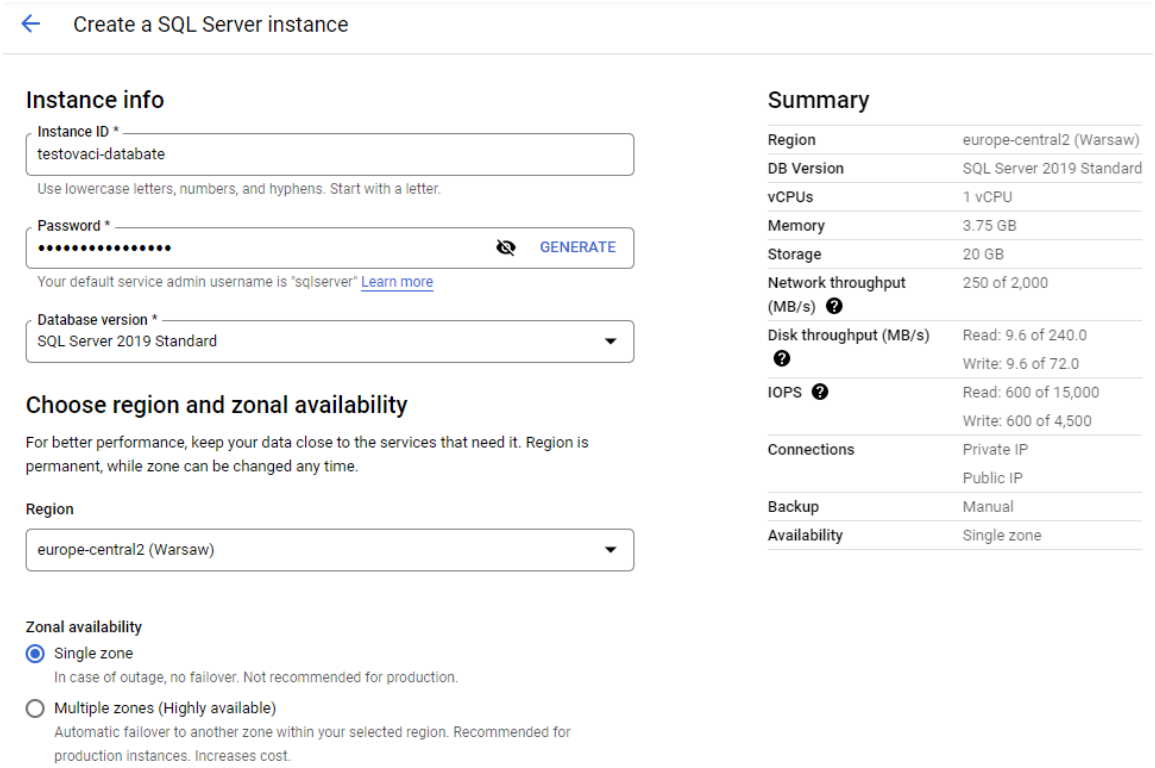
Pro vytvoření databáze na GCP se musí vytvořit účet u Googlu. Dále se musí založit nový projekt. Projekt je v podstatě skupina používaných a vybraných služeb. U projektu, který používá placené služby (například SQL), musí být nastavené a zapnuté účtování. Pro používání služby Cloud SQL musí být povoleno Compute Engine API. Vytvořením SQL Serveru na stránce služby Cloud SQL se otevře stránka, kde se volí CPU, paměť a další. Zobrazeno na obrázku 6.2. Stojí zmínit způsob připojení na server přes nástroj Cloud SQL Proxy. Poskytuje zabezpečenější přístup k databázi než jen přes jméno a heslo.

Z kontejneru nasazeném na Cloud Run se nelze připojit přes veřejnou IP adresu Cloud SQL serveru. Proto je vytvořena interní privátní adresa sloužící pro kontejnery nasazené na Cloud Run[2]. Vytváření privátní adresy lze vidět na obrázku 6.1.



Obrázek 6.1: Pro vytvoření privátní adresy na GCP se nejdříve musí vytvořit VPC Network. Toto je detail výchozí sítě, kde se vytváří soukromé připojení, které se použije pro komunikaci s SQL serverem.

[2] https://cloud.google.com/sql/docs/sqlserver/connect-run

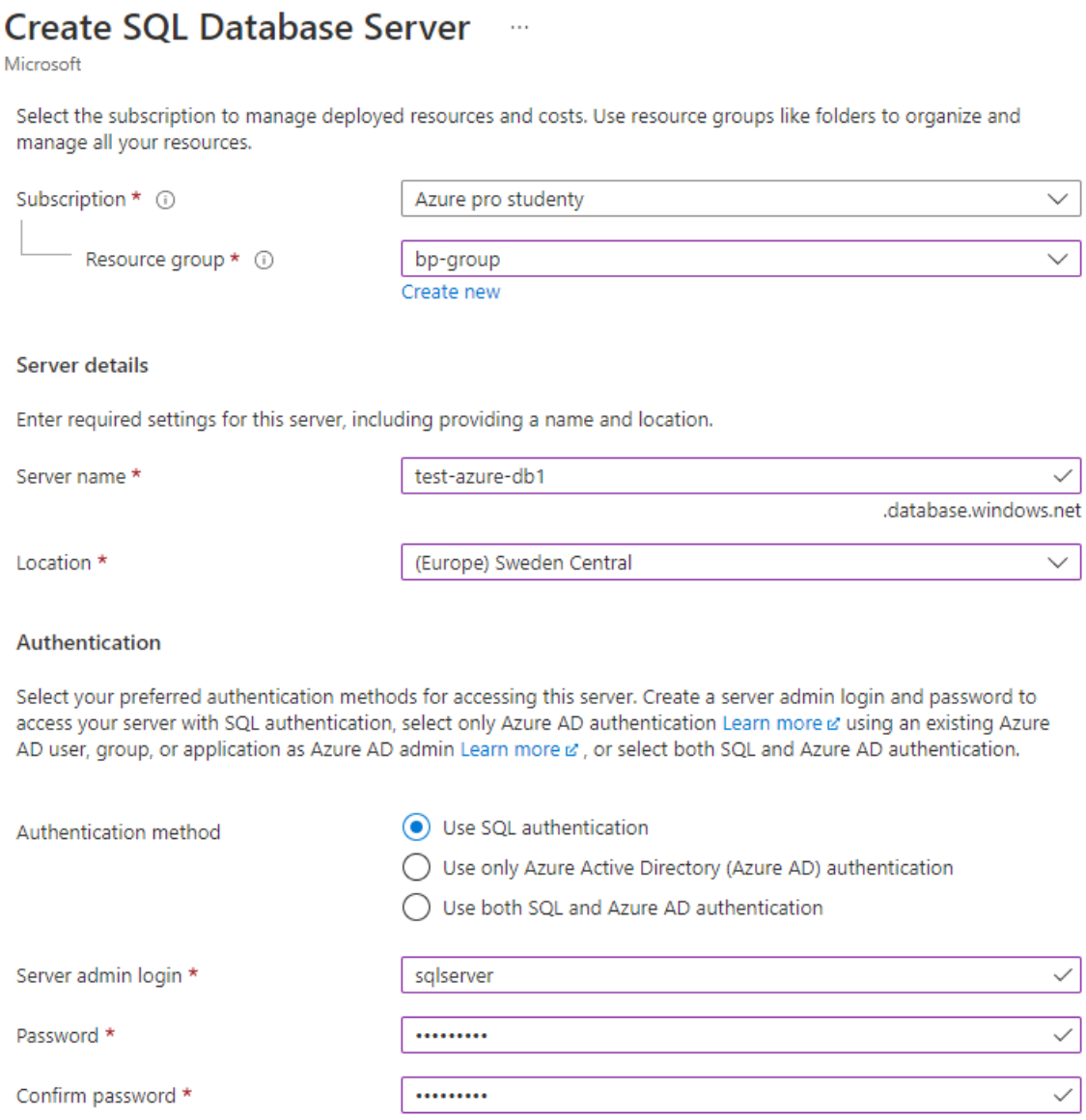


Obrázek 6.2: Stránka konfigurující SQL Server na službě Cloud SQL. Je zvolena nejlevnější konfigurace. Na stránce se nacházejí další konfigurace ohledně IP adresy a údržby (je třeba alokovat privátní adresu).

Poté, co se server vytvoří, lze založit databázi připojením se na server. Připojení na server nebude fungovat, pokud IP adresa, ze které se přistupuje na server, nebude uvedena v autorizovaných sítích serveru. IP adresa počítače nebo sítě se přidává do autorizovaných sítí v záložce Connections. Pro vytvoření tabulek a vložení dat se může použít jakýkoli SQL nástroj, který je schopen se připojit na vzdálený SQL server. Pro SQL Server se může použít například SSMS (SQL Server Management Studio). K serveru se připojuje přes veřejnou IP adresu. Jméno je vždy sqlserver, heslo je zvolené heslo při vytváření serveru. Zabezpečenější komunikace a autentizace se zajišťuje přes Cloud SQL Proxy. Databázi lze vytvořit i přímo v GCP v daném detailu serveru na záložce Databases. Databázi vytváří i backend aplikace. Pokud je tato aplikace nasazena například na Azure, nesmí se zapomenout přidat danou IP adresu do autorizovaných sítí Cloud SQL serveru.

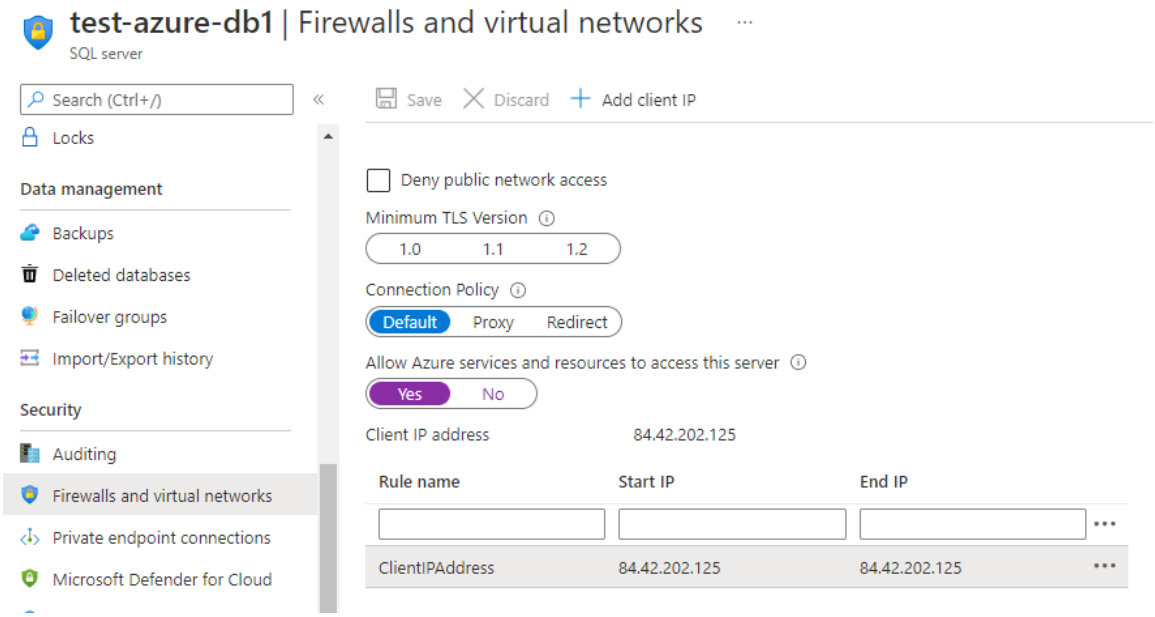
Azure SQL server

Proces nastavení platformy je podobný jako u GCP (vytvoření účtu, nastavení účtování). Na rozdíl od GCP se v Azure vytváří tzv. Resource Group. Podobný koncept jako projekt v GCP. V portálu Azure se server vytváří přejitím na stránku SQL servers. Na obrázku 6.3 je vidět průvodce vytvoření SQL serveru.



Obrázek 6.3: Vytvoření SQL serveru na platformě Azure.

Po úspěšném vytvoření serveru se přes webový portál přidá databáze. Do databáze se pak následně dá připojit pomocí nějakého nástroje, jako je třeba zmíněný SSMS. Podobně jako u Google SQL je zde potřeba nastavit autorizované IP adresy, ze kterých se bude se serverem komunikovat. Na Azure je nasazen i backend služby, je potřeba povolit přístup Azure službám k serveru. Povolení přístupu a nastavení sítě serveru je zobrazeno na dalším snímku 6.4.



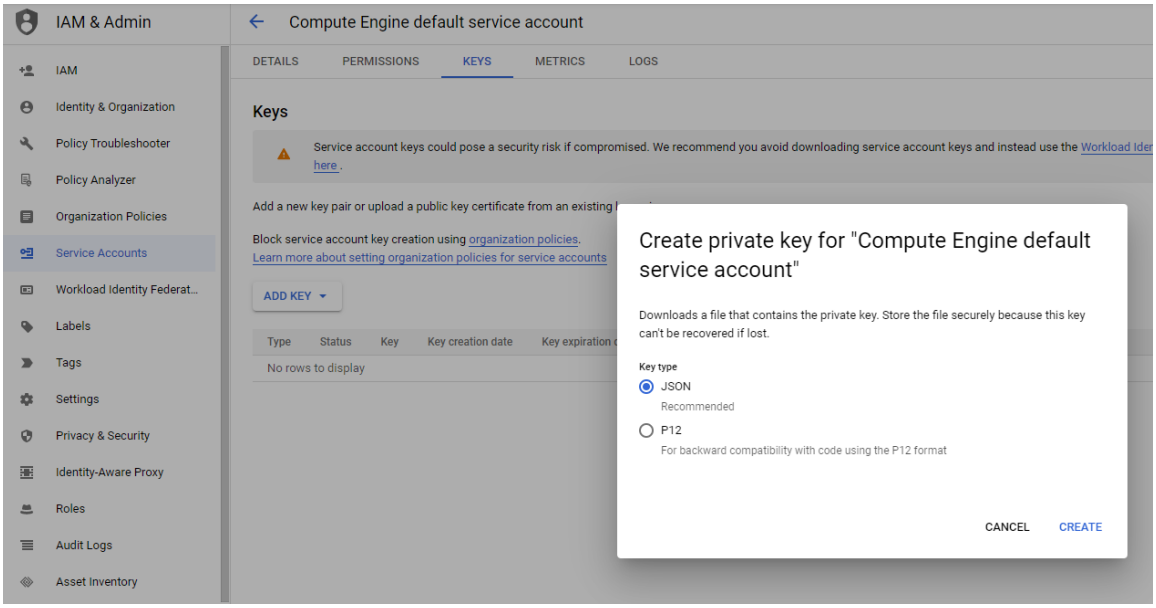
Obrázek 6.4: Takto vypadá nastavení firewallu a sítí SQL Serveru na platformě Azure.

6.4 Nasazení kontejneru v Google Cloud Run

Pro spuštění kontejneru v této službě je nutné mít git repositář, ze kterého se sestaví Docker image. Nebo se Docker image musí nahrát do Container Registry.

K nahrání image je potřeba vytvořit image lokálně s tagem `gcr.io/projekt/image:tag`. Aby fungoval příkaz `docker push`, který nahraje image do Google registru, musí být počítač autorizován. K tomu slouží nástroj gcloud[3]. V počítači musí být také přítomen klíč k servisnímu účtu. Ten se získá ze správy přístupu a rolí na stránce IAM & Admin. Na kartě Service Accounts doporučuji zvolit implicitní Compute Engine servisní účet. V detailu účtu (obrázek 6.5) se vytváří privátní klíč. V lokálním prostředí se provede autorizace pomocí příkazu `gcloud auth activate-service-account email --key-file=cesta-ke-klíči`. Následně je umožněno nahrát image do registru.

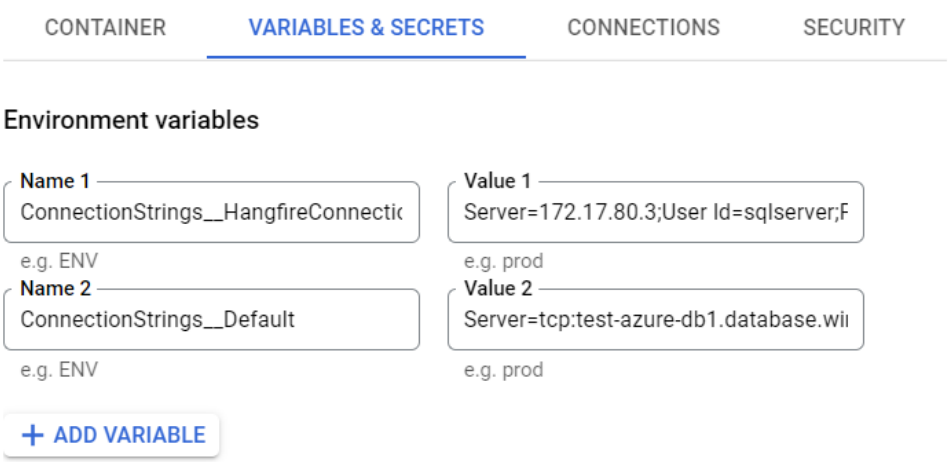
[3]Instalace nástroje gcloud https://cloud.google.com/sdk/docs/install



Obrázek 6.5: Vytvoření privátního klíče k servisnímu účtu na platformě GCP.

Jakmile se image nahraje do registru, v Cloud Run se může vytvořit služba. Konfiguruje se škálování, povolený provoz do kontejneru, nastavení CPU alokace. Toto zachycuje snímek A.5 v příloze. Kontejner může běžet celou dobu a CPU je neustále alokováno nebo jde CPU alokovat pouze s příchozím požadavkem, což je levnější varianta. V konfiguraci se může měnit port kontejneru a konfigurační proměnné.

Pro nasazení management aplikace vykonávající rutinní práce se potřebuje změnit port kontejneru na port 80 (pouze pro health check). Dále se musí přepsat připojovací řetězce do databází. Pokud se použije Google Cloud SQL, musí se v připojovacím řetězci zadat privátní adresa. Potom se musí vytvořit VPC connector na kartě Connections. Ten slouží pro připojení pomocí privátní IP adresy na Cloud SQL server. Pokud se jedná o Azure SQL, nemusí se VPC konektor vytvářet. Nicméně je nutné, aby se IP adresa kontejneru povolila ve firewallu Azure SQL serveru. Po spuštění kontejneru se mohou zobrazit logy a podobně. Kontejner se může znovu nasadit vytvořením nové revize, například z důvodu nové konfigurace. Konfigurování kontejneru je vidět na obrázku 6.6.

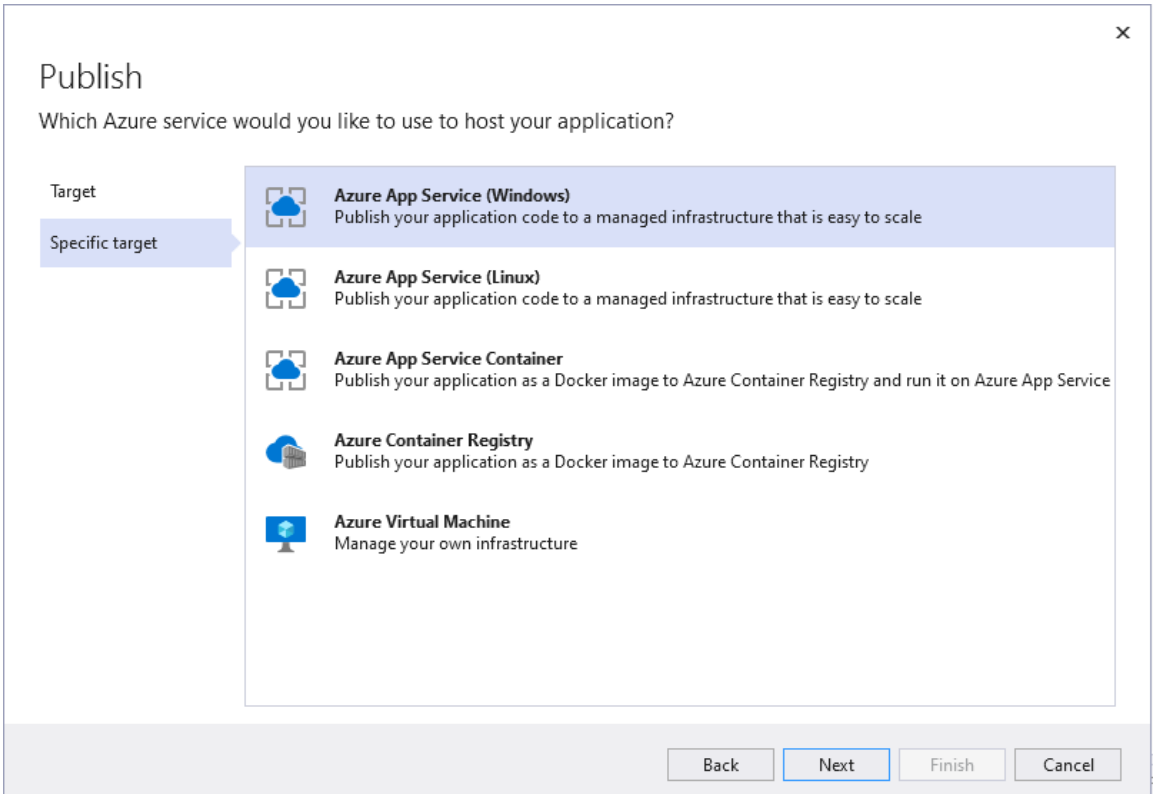


Obrázek 6.6: Specifikování připojovacích řetězců kontejneru v Cloud Run. První řetězec je pomocná databáze pro management aplikaci, druhý řetězec je do hlavní databáze služby. Řetězec vypadá například takto: `Server=34.116.215.3;User Id=sqlserver;Password=zIP2pFqc1nNy8BLp;Initial Catalog=Hangfire;`

6.5 Nasazení aplikace v Azure App Service

Další ukázkou je nasazení ASP.NET Core aplikace (backend a frontend služby) do Azure App Service. K tomuto doporučuji použít Visual Studio, které velmi usnadní celý proces. Aplikace se dá nasadit i z repositáře, tento proces je ale náročnější, jelikož se musí nastavit řada věcí včetně sestavovací pipeliny.

Ve Visual Studiu musí být uživatel přihlášen s účtem na Azure. Po zvolení projektu, který se bude publikovat na Azure, se otevře průvodce zobrazený na obrázku 6.7. V tomto průvodci pro publikování aplikace se vybírá způsob nasazení. Lze nasadit aplikaci na Azure App Service na infrastrukturu Windows nebo Linux. Windows nasazení umožní aplikaci hostovat v plánu F1, který je zdarma. Pro nasazení na Linux je třeba mít placený plán úrovně S1. Nebo se může vytvořit image, který se nahraje na Azure registr a spustí se v App Service (také placené). Po výběru způsobu nasazení se musí vytvořit App Service instance. Pokud se nasazuje backend této práce, která je zintegrována se službou Auth0, musí se instance App Service pojmenovat na `testauth0blazorwasmserverapp`. Auth0 totiž potřebuje znát adresu, na kterou má přesměrovat uživatele při přihlašování. V Auth0 jsem toto nakonfiguroval dopředu na adresu `testauth0blazorwasmserverapp`. Spolu s App Service se vytváří Resource group a plán hostování. Mohou se použít existující. Vše lze vytvořit i přes webový portál. API management není třeba konfigurovat. Pokud se nasazuje aplikace jako kontejner, musí se k tomu vytvořit i registr, kam se nahraje Docker image.



Obrázek 6.7: Publikace projektu na Azure v programu Visual Studio 2022.

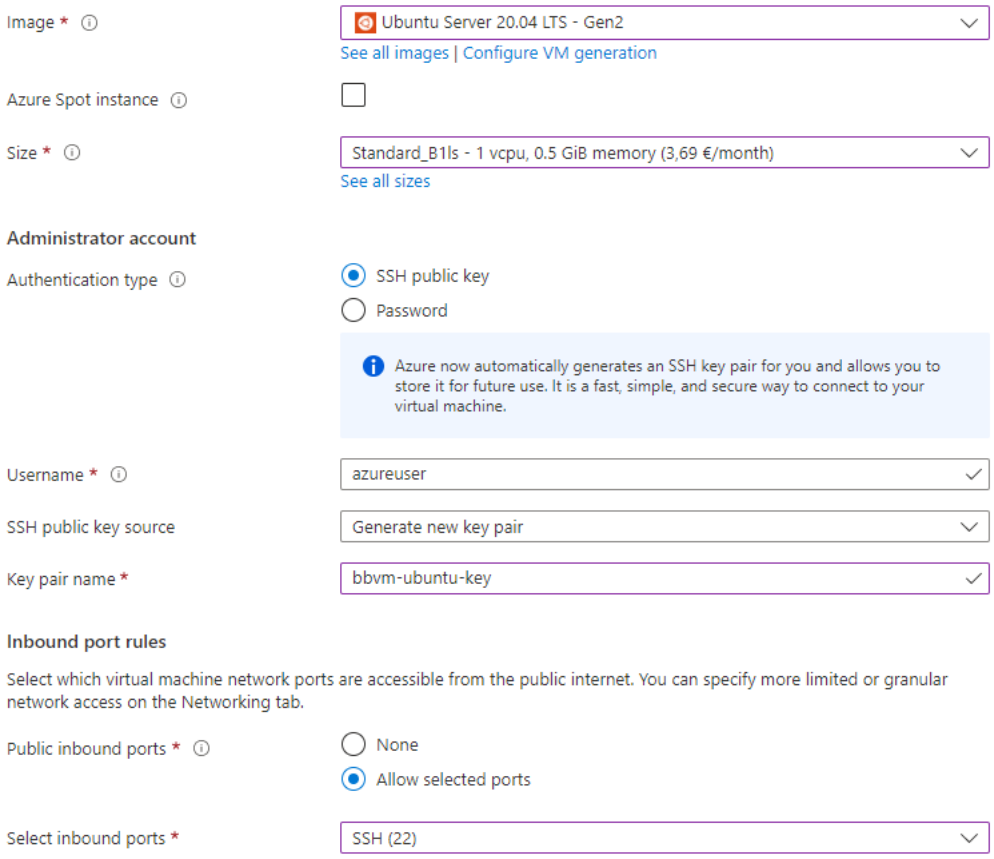
Po úspěšném publikování aplikace se musí stejně jako v Cloud Run změnit konfigurace připojovacího řetězce. Toto se dá nastavit již ve Visual Studiu nebo později přes portál v Azure. Stačí přejít na vytvořenou instanci App Service a na kartě Configuration v sekci Connection strings vytvořit nový řetězec pojmenovaný Default. Hodnota řetězce se získá buď z Google SQL, nebo z Azure SQL serveru. Pokud je databáze na Googlu, nesmí se opět zapomenout na přidání IP adresy sítě aplikace do autorizovaných sítí SQL serveru. Používané IP adresy služby App Service jsou uvedeny na kartě Networking v Outbound Traffic. Tento proces je nezávislý na způsobu nasazení (Linux, Windows nebo kontejner).

6.6 Vytvoření virtuálního stroje v Azure

Nasazení celé služby (i s databází) na jeden stroj je jednoduché. Stačí vytvořit virtuální stroj, stáhnout a nastavit Docker s Docker Compose (záleží na OS). Poté stačí přetáhnout zdrojové soubory s konfiguracemi a spustit celou službu přes Docker Compose. Nesmí se zapomenout na případné otevření portu do virtuálního stroje.

Při vytváření virtuálního stroje s operačním systémem Windows je potřeba dát pozor na volbu CPU. Systém Windows s některými CPU nepodporuje vnořenou virtualizaci a díky tomu nebude fungovat Docker[4]. Toto se netýká Linux systémů. Na následujícím obrázku 6.8 je zobrazen průvodce vytvoření virtuálního stroje. Na dalších kartách průvodce stačí vybrat standardní SSD disk. Pro připojení ke stroji lze vytvořit místo SSH klíče klasické heslo.

[4]Seznam podporovaných systémů https://docs.microsoft.com/en-us/azure/virtual-machines/acu



Obrázek 6.8: Konfigurace virtuálního stroje na Azure.

Jakmile se spustí stroj, lze se připojit pomocí SSH klíče nebo hesla přes program PuTTy. Pokud se vytvoří stroj s Windows systémem, lze se připojit pomocí RDP. Dále se může nainstalovat Docker. Na Windows stačí nainstalovat Docker Desktop. Já jsem vybral Linux distribuci Ubuntu 20.04, kde je instalace odlišná[5]. Dále stačí stáhnout zdrojové soubory a spustit celou službu přes Docker Compose. Takto se nasazuje i worker. U nasazení celé služby je třeba zveřejnit port, na kterém služba naslouchá (například 5001). Toho se docílí na kartě Networking v sekci Inbound port rules. Toto není třeba při nasazování worker aplikace.

Worker aplikaci lze nasadit na virtuální stroj pomocí Docker Compose. Detaily jsou v příloze C.

6.7 Datové úložiště

Pro datové úložiště jsem vybral již zmíněné řešení Google Cloud Storage. Je jednoduché a zdarma. Pro vytvoření úložiště stačí přejít na Cloud Storage, povolit potřebná API a vytvořit tzv. bucket. Následně je třeba vytvořit servisní účet s právy pro čtení a zápis do Cloud Storage. Aplikace, která bude komunikovat s úložištěm, bude potřebovat klíč tohoto

[5]Instalování Docker Compose na Ubuntu 20.04 https://www.digitalocean.com/community/tutorials/how-to-install-and-use-docker-compose-on-ubuntu-20-04

účtu. Alternativně lze využít služby od Azure. V takovém případě by se již musela měnit i implementace aplikace, protože se používají knihovny pro Google Cloud Storage.

6.8 Konečné nasazení

Službu jsem nakonec nasadil na Azure App Service, jelikož poskytuje nejjednodušší řešení s možností výběru platformy (Windows, Linux nebo kontejner). Pokud se využívá Windows, nasazení je kompletně zdarma. Navíc nabízí lepší diagnostické nástroje pro ASP.NET Core aplikace. Management aplikaci starající se o soubory jsem nasadil na službu Google Cloud Run. Poskytuje velmi jednoduché a levné nasazení samostatného kontejneru. Management aplikace navíc nemusí být stále spuštěna. Soubory se ukládají do Google Cloud Storage. SQL server běží také na Azure, ale nevedlo mě k tomu nic zásadního (možná jednodušší vytvoření a cena). Worker aplikaci lze nasadit lokálně v Dockeru nebo na virtuální stroj nebo do jiného prostředí umožňujícího spouštění kontejnerů. Pomocí sekcí v této kapitole lze nasazení služby jakkoli zreplikovat i s využitím jiných služeb, než které jsem zvolil já.

Kapitola 7

Simulování provozu služby

Kapitola pojednává o simulování provozu služby. Vytváří se simulační prostředí. Následně se simulace spouští s různými parametry a testuje se provoz. Z experimentů je na konci odvozen závěr.

Simulace je proces experimentování s modelem systému. Cílem simulace je získání nových znalostí ze zkoumaného systému. Model systému je systém, který napodobuje reálný systém. Nemusí tak obsahovat všechny jeho části. Model by se měl snadno vytvořit a měl by se co nejvíce podobat reálnému systému. Dále je důležité si určit cíle simulace, co je potřeba pozorovat, co se potřebuje zjistit. Experimenty je třeba spouštět několikrát a i s jinými parametry, aby se získalo co nejvíce nových informací a zkušeností. Je tedy dobré mít vhodné parametry modelu, které mohou měnit chování systému.

7.1 Vytvoření prostředí simulace

Simulace potřebuje model. V této práci lze rovnou použít serverovou aplikaci. Není třeba vytvářet nový model, stačí použít reálnou implementaci. Chování právě této aplikace chceme simulovat a testovat. Nemusí se využít management aplikace, která nemá zásadní vliv na systém. Nemusí se použít reálná databáze, postačí pouze in-memory databáze. Jelikož se bude simulace systému spouštět několikrát, je potřeba zajistit, aby se všechny závislosti, které vytvoří simulace, na konci smazaly. Na to jsem použil výhodně Docker Compose. Pomocí tohoto nástroje lze jednoduše spustit simulaci, uklidit po simulaci a měnit parametry systému. Parametry, které se mohou různě nastavovat a ovlivňují chování systému, jsou například: jak často uživatel vytváří úlohu, kolik jich vytvoří, kolik je v systému uživatelů, kolik z nich má VIP účet, jak dlouho běží úloha v kontejneru.

Cílem tohoto testování a simulování je napodobení reálného provozu, kdy uživatelé vytváří úlohy a čekají na výsledek. Především je třeba pozorovat frontu s úlohami. Jak dlouho úlohy čekají ve frontě? Kolik worker aplikací je třeba, aby se fronta zvládla plynule zpracovávat za daného provozu? Jak moc VIP úlohy předbíhají standardní?

Pro simulaci byl vytvořen pomocný program, který vytváří uživatele. Program pak zakládá úlohy pod těmito uživateli. Procesy uživatelů vytvářející úlohy běží paralelně. Další úloha se vytvoří po nějakém zpoždění. Simulaci lze sledovat přihlášením do aplikace pod administrátorem nebo pod programem vytvořeným uživatelem. Aplikace zobrazuje zpracování úloh v reálném čase, dále poskytuje pohledy zobrazující statistiky. V simulaci se nastavuje počet vytvořených uživatelů, kolik úloh uživatel celkově vytvoří, po jaké době se úlohy vytváří. V systému se může vytvořit několik workerů. Každý se dá různě konfiguro-

vat. Worker spouští demonstrační kontejner, kde lze specifikovat, jak dlouho se bude úloha počítat. To například umožní prodloužení čekání ve frontě, pokud existuje málo workerů.

Výsledné simulační prostředí se tedy vytváří pomocí Docker Compose a skládá se ze serveru a několika workerů.

7.2 Experimenty

V této sekci jsou ukázány některé experimenty. Detailní popis všech parametrů a konfigurací simulace se nachází v příloze D.

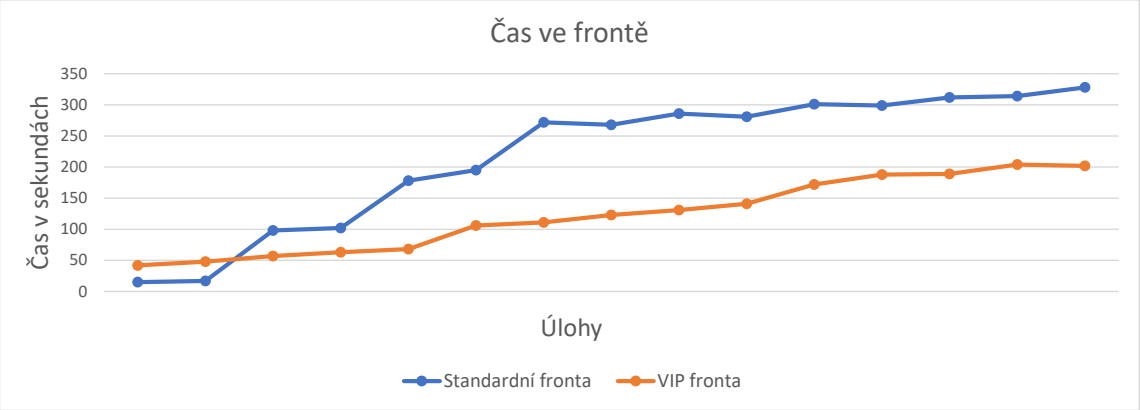
I přes to, že jde vytvářet několik uživatelů, stačí vytvořit jednoho VIP uživatele a druhého standardního. Pak stačí určit počet úloh, které oba vytvoří. Pokud chceme zvýšit provoz, stačí nastavit větší počet vytvořených úloh v kratším časovém intervalu. Tím se zajistí podobné chování, jako kdyby bylo v systému více uživatelů.

Je vhodné si zvolit výhodnou jednotku času. Simulace dovoluje nastavit, aby výpočet běžel hodinu, taková simulace by ale běžela velmi dlouho. Lépe se pracuje s jednotkami sekund. Systém se bude chovat podobně, když se všechny parametry změní na větší časovou jednotku. Akorát bude vše trvat déle.

1. experiment

Parametry experimentu:

- Každý uživatel vytvoří 15 úloh.
- V systému jsou dva workeři.
- Délka výpočtu se pohybuje v intervalu od 20 sekund po 32 sekund.
- Váha fronty s neprioritními úlohami je 1.
- Váha fronty s prioritními úlohami je 4.
- Doba, kdy opět uživatel vytvoří úlohu, se pohybuje na intervalu od 5 do 7 sekund.

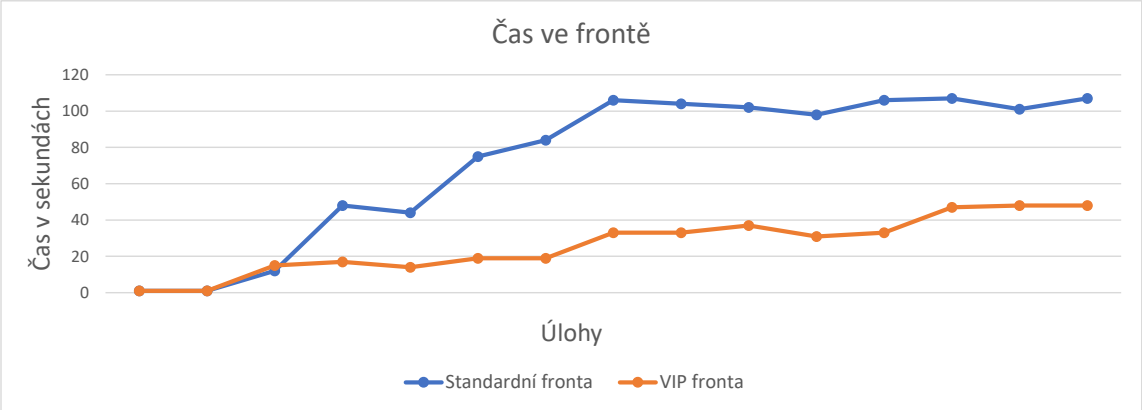


Obrázek 7.1: Výsledek prvního experimentu simulování. Graf zobrazuje čas úlohy strávený v dané frontě.

Z grafu 7.1 lze vidět, že ze začátku se fronta zvětšuje a díky tomu se prodlužuje čekání. Jakmile uživatelé přestanou spouštět úlohy, doba už výrazně nestoupá. Záleží už jen na tom, kolik je ještě ve frontě úloh a kolik existuje workerů. Je i vidět fungování implementovaného round robin algoritmu, kdy VIP úlohy musí na začátku čekat déle, protože algoritmus se nejdříve podívá do neprioritní fronty. Poté se ale vše srovná do normálu a prioritní úlohy tak celkově čekají kratší dobu. Závěr experimentu je takový, že pokud by nápor byl souvislejší, vytvoří se velká fronta s dlouhou čekací dobou.

2. experiment

Tento experiment má stejné parametry jako předchozí. Chci však zmírnit rychlý nárůst fronty zvýšením počtu workerů na čtyři.

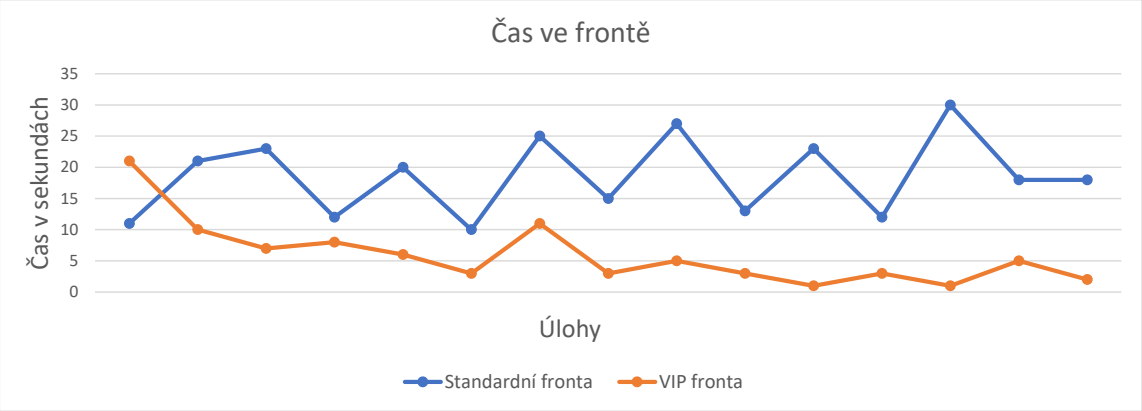


Obrázek 7.2: Výsledek druhého experimentu simulování. Graf zobrazuje čas úlohy strávený v dané frontě.

Stejný nápor služba zvládla samozřejmě lépe, jak je vidět na grafu 7.2. Čekání ve frontě se opět razantně zvýší kvůli velkému náporu. Fronta se ale zpracovává stabilněji, vidět je to hlavně na VIP frontě. Díky dalším dvěma workerům se fronty zpracovávají více jak dvakrát rychleji. Na rozdíl od prvního experimentu worker aplikace už běžely, takže začaly ihned pracovat. Toto ale nijak neovlivňuje celý výsledek.

3. experiment

V posledním experimentu jsem upravil hodnotu zpoždění mezi vytvářením úloh. Nový interval, ze kterého se doba určuje, se pohybuje od 13 do 16 sekund. Myšlenkou je odsimulovat klasický provoz bez velkého náporu. Ten sice opět vznikne, ale nebude tak velký jak v minulých experimentech. Všechny ostatní parametry jsou stejné a systém je spouštěn se čtyřmi worker kontejnery.



Obrázek 7.3: Výsledek třetího experimentu simulování. Graf zobrazuje čas úlohy strávený v dané frontě.

V tomto běhu je vidět (graf 7.3), že takový provoz dokáže systém spolehlivě zvládnout bez nějakého většího zvětšení fronty. Workeři byli spuštěni o něco později, proto se doba u některých úloh dokonce zmenšuje.

7.3 Vyhodnocení

Testováním provozu služby simulovanými přístupy lze odhadnout, kolik je potřeba workerů na daný nápor uživatelů. Ve službě se uchovávají statistiky o spuštěných úlohách. Pokud se například v nějakou dobu objevuje velká fronta, lze podobná data získat a odsimulovat. Podle toho se může odhadnout zvýšení workerů v systému. Simulace jsou tedy skvělým nástrojem pro testování provozu. Umožňují efektivněji upravovat parametry systému (počet worker aplikací, váhy front) podle odsimulovaného provozu.

Kapitola 8

Závěr

Cílem této práce bylo vytvoření internetové služby umožňující spouštět úlohy v kontejnerech. Uživatel za spuštěné úlohy platí ve formě tokenů, které získává od administrátora. Administrátor je speciální role v systému, která spravuje uživatele. Může sledovat frontu úloh, může zobrazit uživatelské úlohy, statistiky a podobně. Dále uděluje blokace za podezřelé chování, připisuje uživatelům tokeny a rozdává uživatelům VIP statusy. Úlohy založené VIP účtem jsou dražší, ale jsou prioritnější než standardní úlohy.

Navržená služba se skládá z frontend (klient) a backend (server, REST API) aplikace. Součástí služby je aplikace, která provádí rutinní práce jako mazání starých souborů. V systému existují tzv. workeři, kteří si stahují práci ze serveru a spouští ji v kontejneru. Celý systém tedy funguje tak, že uživatelé pomocí frontend aplikace zakládají nové úlohy, které se řadí do fronty na serveru. Workeři postupně odebírají z fronty úlohy a spouštějí je.

Úloha je pouze abstrakcí nad nějakým konkrétním výpočtem. Používá se demonstrační Docker image, který simuluje nějaký výpočet. Tento image se dá v budoucnu vyměnit za image, který už skutečně implementuje nějaký výpočet nebo simuluje něco reálného. To byla i hlavní motivace této práce. Služba nyní výpočty pouze simuluje, v budoucnu by však mohla spouštět nějaké konkrétní výpočty.

Celou službu se podařilo implementovat ve frameworku ASP.NET Core 6 v programovacím jazyce C#. Veškeré pomocné programy (například pro simulaci) byly napsány také v programovacím jazyce C#. Frontend aplikace se zhotovila pomocí technologie WebAssembly. Framework ASP.NET Core doporučuji pro vývoj REST API aplikací. Nabízí jednoduché vývojové prostředí pro implementaci REST architektury.

Službu se úspěšně podařilo nasadit pomocí platformy Azure a Google. Na Azure je nasazena webová aplikace[1] společně s SQL serverem. Na Google se ukládají soubory a je zde nasazena management aplikace. Worker aplikace běží lokálně nebo na virtuálním stroji pomocí Docker Compose.

V práci se také podařilo simulovat provoz služby. Lze sledovat frontu a chování serveru nastavováním různých parametrů (počet worker aplikací, délka výpočtu úlohy a podobně). Tím lze také jednoduše otestovat funkčnost systému. Simulaci by šlo obohatit o další statistiky a jednodušší způsob získání těchto dat (například UI aplikace).

Ve službě by se mohla integrovat nějaká platební brána, aby uživatel nemusel žádat o tokeny u administrátora. Dalším rozšířením je například implementace robustnějšího systému účtování za spuštěné výpočty. Služba by mohla být nasazena v systému Kubernetes.

[1] V období obhajoby práce je služba dostupná na https://tasklauncher.azurewebsites.net/

Literatura

[1] Anderson, R. a Smith, S. *ASP.NET Core Middleware* [online]. docs.microsoft.com, duben 2022 [cit. 2022-19-04]. Dostupné z: https://docs.microsoft.com/en-us/aspnet/core/fundamentals/middleware/?view=aspnetcore-6.0.

[2] Anderson, R. et al. *When to use Kestrel with a reverse proxy* [online]. docs.microsoft.com, duben 2022 [cit. 2022-19-04]. Dostupné z: https://docs.microsoft.com/en-us/aspnet/core/fundamentals/servers/kestrel/when-to-use-a-reverse-proxy?view=aspnetcore-6.0.

[3] Anderson, R. et al. *Add support for JSON Patch when using* [online]. microsoft.com, březen 2022. Dostupné z: https://docs.microsoft.com/en-us/aspnet/core/web-api/jsonpatch?view=aspnetcore-6.0.

[4] *Bearer Authentication* [online]. swagger.io [cit. 2022-17-04]. Dostupné z: https://swagger.io/docs/specification/authentication/bearer-authentication/.

[5] *Comparing Containers and Virtual Machines* [online]. docker.com [cit. 2022-07-04]. Dostupné z: https://www.docker.com/resources/what-container/.

[6] *Containerization* [online]. ibm.com, 23. června 2021 [cit. 2022-7-04]. Dostupné z: https://www.ibm.com/cloud/learn/containerization.

[7] *Cookies* [online]. auth0.com [cit. 2022-17-04]. Dostupné z: https://auth0.com/docs/manage-users/cookies.

[8] Docker. *Docker Documentation* [online]. [cit. 2022-07-04]. Dostupné z: https://docs.docker.com/.

[9] *Docker overview* [online]. docs.docker.com [cit. 2022-07-04]. Dostupné z: https://docs.docker.com/get-started/overview/.

[10] *Docker vs Virtual Machines (VMs) : A Practical Guide to Docker Containers and VMs* [online]. weave.works, 16. ledna 2020 [cit. 2022-10-04]. Dostupné z: https://www.weave.works/blog/a-practical-guide-to-choosing-between-docker-containers-and-vms.

[11] Fielding, R. T. *Architectural Styles and the Design of Network-based Software Architectures*. Irvine, USA, 2000. Disertační práce. University of California Irvine, Information and Computer Science. Dostupné z: https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm.

[12] Fielding, R., Gettys, J., Masinter, L. et al. *Header Field Definitions* [online]. w3.org, červen 1999 [cit. 2022-10-04]. Dostupné z: https://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html.

[13] Fielding, R., Reschke, J., Adobe et al. *Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing* [online]. datatracker.ietf.org, červen 2014 [cit. 2022-10-04]. Dostupné z: https://datatracker.ietf.org/doc/html/rfc7230.

[14] Franks, J., Hallam Baker, P., Hostetler, J. et al. *HTTP Authentication: Basic and Digest Access Authentication* [online]. datatracker.ietf.org, červen 1999 [cit. 2022-17-04]. Dostupné z: https://datatracker.ietf.org/doc/html/rfc2617.

[15] Gaster, B., FitzMacken, T. et al. *Mapping SignalR Users to Connections* [online]. microsoft.com, únor 2020. Dostupné z: https://docs.microsoft.com/en-us/aspnet/signalr/overview/guide-to-the-api/mapping-users-to-connections.

[16] Hirschmann, B. a Baier, D. *BlazorWebAssemblyCookieAuth* [online]. 2021. Dostupné z: https://github.com/berhir/BlazorWebAssemblyCookieAuth.

[17] Hollan, J. et al. *Comparing Container Apps with other Azure container options* [online]. docs.microsoft.com, březen 2022 [cit. 2022-24-04]. Dostupné z: https://docs.microsoft.com/en-us/azure/container-apps/compare-options.

[18] MacVittie, L. *The Three HTTP Routing Patterns You Should Know* [online]. dzone.com, 16. května 2018 [cit. 2022-17-04]. Dostupné z: https://dzone.com/articles/the-three-http-routing-patterns-you-should-know.

[19] Mathew, P. D. *Writing your first dockerfile to run test automation framework from docker* [online]. medium.com, 6. září 2021 [cit. 2022-10-04]. Dostupné z: https://praveendavidmathew.medium.com/running-webdriverio-in-wsl2-windows-91d3a0dc7746.

[20] McKenna, M. *JSON Patch Support with Swagger and ASP.NET Core 3.1* [online]. Říjen 2020. Dostupné z: https://michael-mckenna.com/swagger-with-asp-net-core-3-1-json-patch/.

[21] Nowak, R., Larkin, K. a Anderson, R. *Routing in ASP.NET Core* [online]. docs.microsoft.com, duben 2022 [cit. 2022-19-04]. Dostupné z: https://docs.microsoft.com/en-us/aspnet/core/fundamentals/routing?view=aspnetcore-6.0.

[22] *OIDC: What Is OpenID Connect and How Does It Work? What You Need to Know* [online]. pingidentity.com [cit. 2022-11-04]. Dostupné z: https://www.pingidentity.com/en/resources/content-library/articles/openid-connect.html.

[23] Perfect.org. *Routing* [online]. [cit. 2022-17-04]. Dostupné z: https://perfect.org/docs/routing.html.

[24] Rgudkov. *ODataOperationFilter* [online]. 2021. Dostupné z: https://github.com/rgudkov-uss/aspnetcore-net6-odata-swagger-versioned/blob/main/WebApiTest6/ODataOperationFilter.cs.

[25] Rosencrance, L., McKenzie, C. a Courtemanche, M. *Application containerization* [online]. techtarget.com, říjen 2021 [cit. 2022-7-04]. Dostupné z: https://www.techtarget.com/searchitoperations/definition/application-containerization-app-containerization.

[26] Roth, D., Anderson, R. a Luttin, S. *Overview to ASP.NET Core* [online]. docs.microsoft.com, březen 2022 [cit. 2022-11-04]. Dostupné z: https://docs.microsoft.com/en-us/aspnet/core/introduction-to-aspnet-core?view=aspnetcore-6.0.

[27] Slootweg, S. *Stop using JWT for sessions* [online]. cryto.net, 19. června 2016 [cit. 2022-17-04]. Dostupné z: http://cryto.net/~joepie91/blog/2016/06/13/stop-using-jwt-for-sessions/.

[28] Smith, S., Pine, D. et al. *Routing differences between ASP.NET MVC and ASP.NET Core* [online]. docs.microsoft.com, duben 2022 [cit. 2022-19-04]. Dostupné z: https://docs.microsoft.com/en-us/dotnet/architecture/porting-existing-aspnet-apps/routing-differences.

[29] Supalov, V. *Should You Run Your Database in Docker?* [online]. vsupalov.com [cit. 2022-24-04]. Dostupné z: https://vsupalov.com/database-in-docker/.

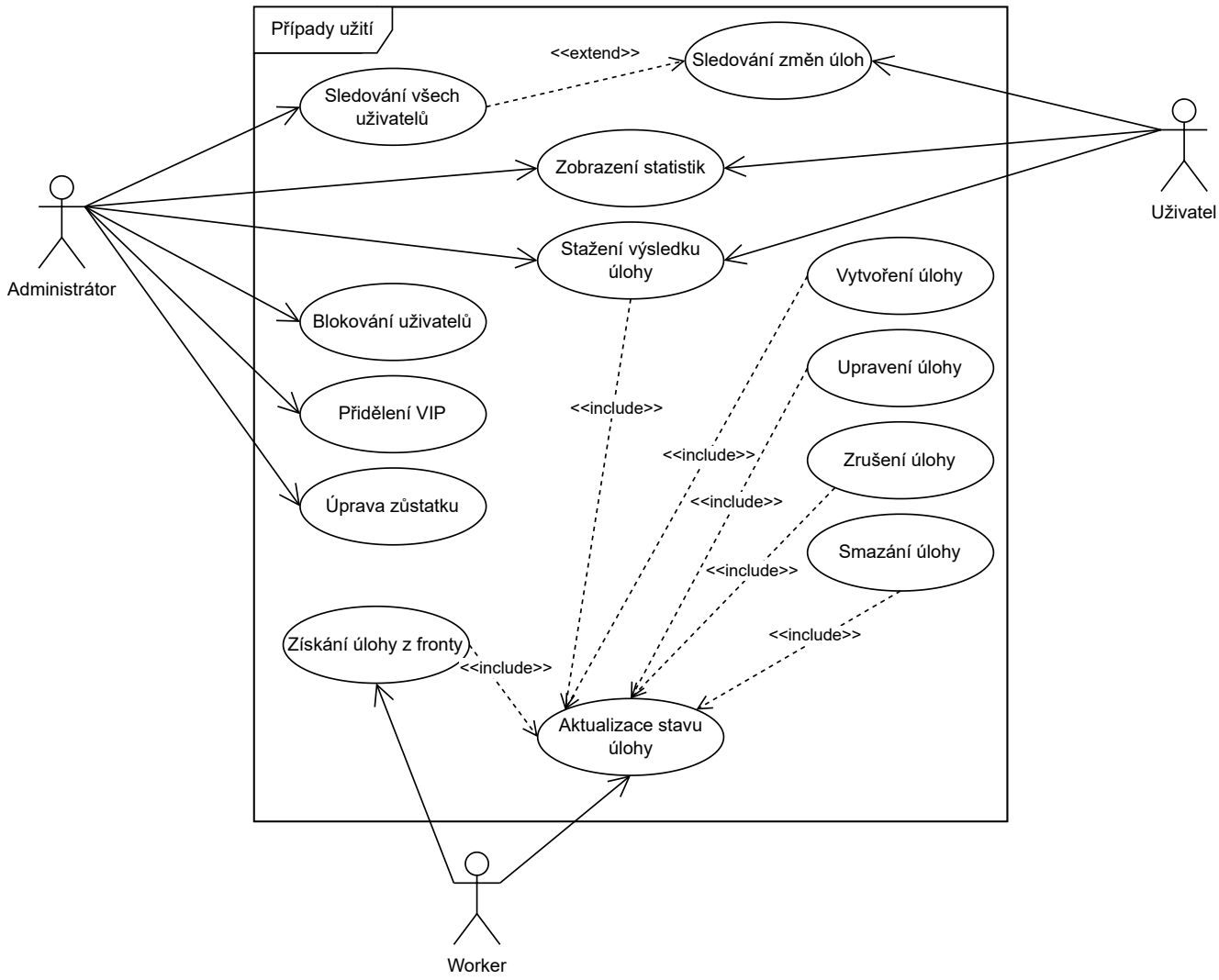
[30] *Using HTTP cookies* [online]. developer.mozilla.org [cit. 2022-17-04]. Dostupné z: https://developer.mozilla.org/en-US/docs/web/http/cookies.

[31] *What is a REST API?* [online]. redhat.com, 8. května 2020 [cit. 2022-10-04]. Dostupné z: https://www.redhat.com/en/topics/api/what-is-a-rest-api.

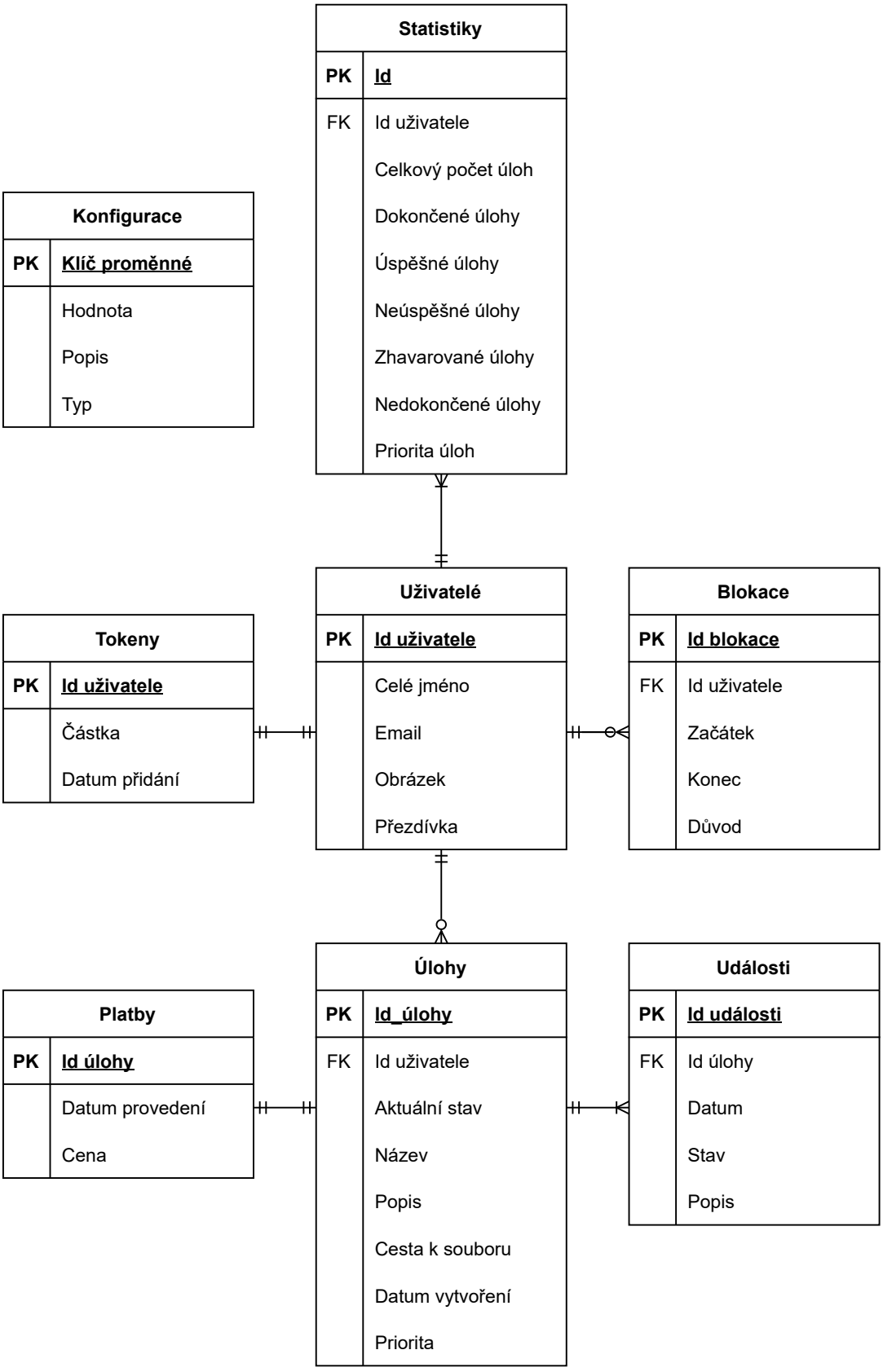
[32] *What is JSON Web Token?* [online]. jwt.io [cit. 2022-17-04]. Dostupné z: https://jwt.io/introduction.

Příloha A

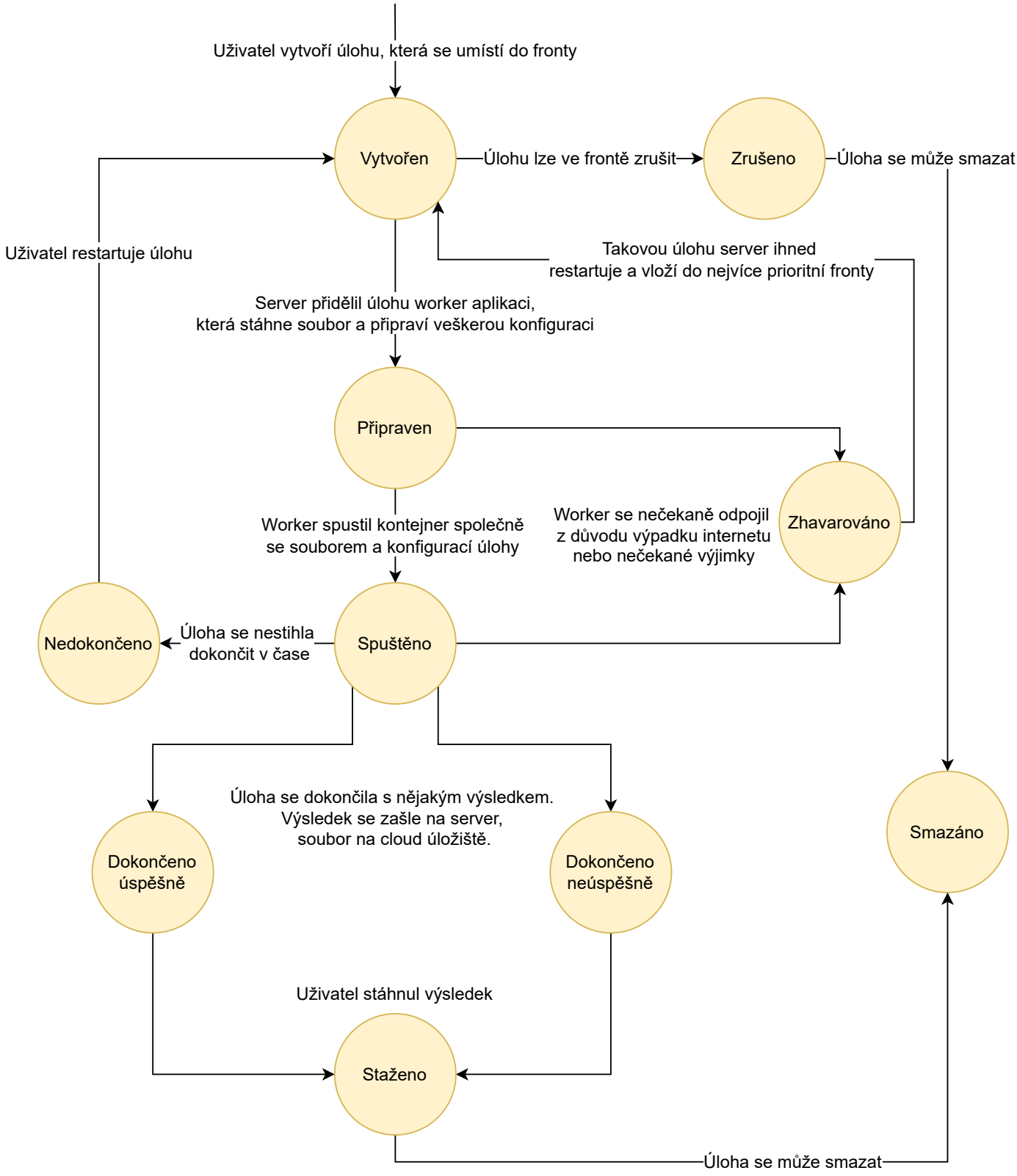
Diagramy



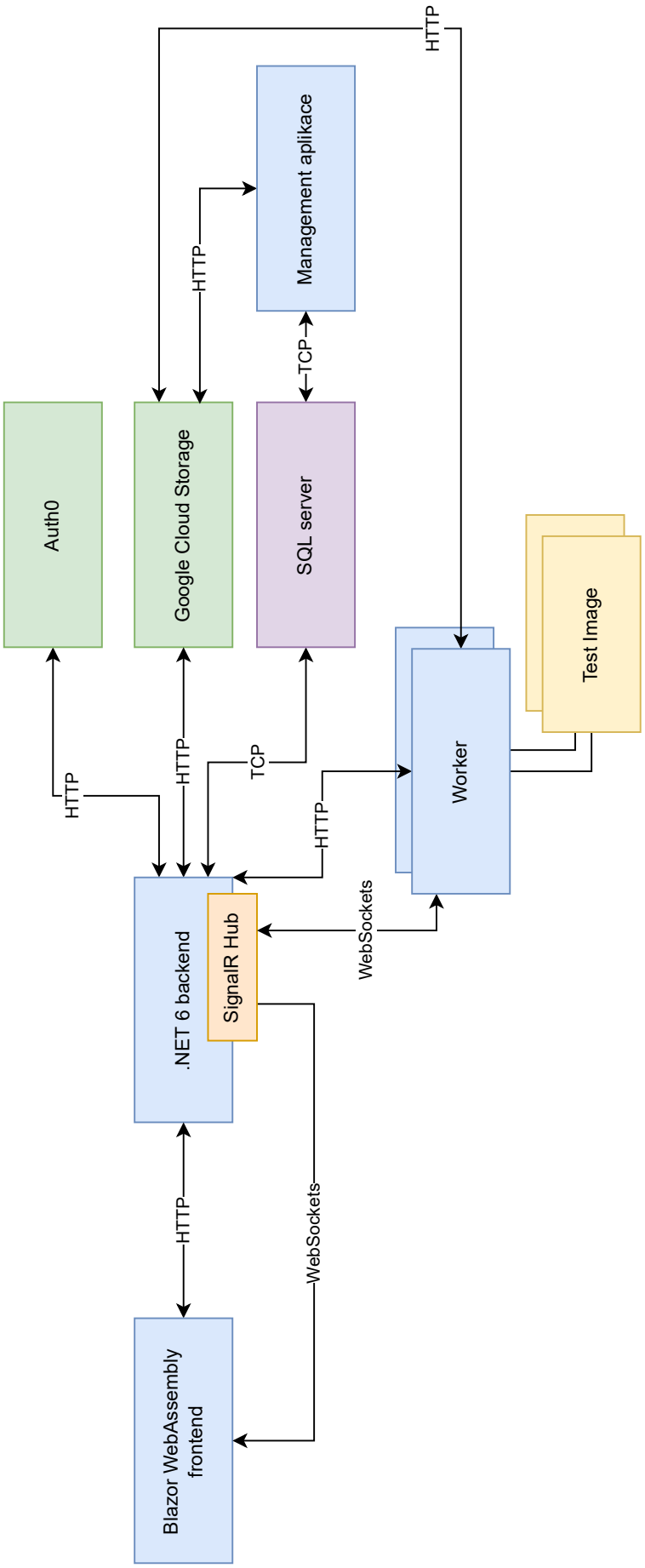
Obrázek A.1: Diagram případů užití služby.



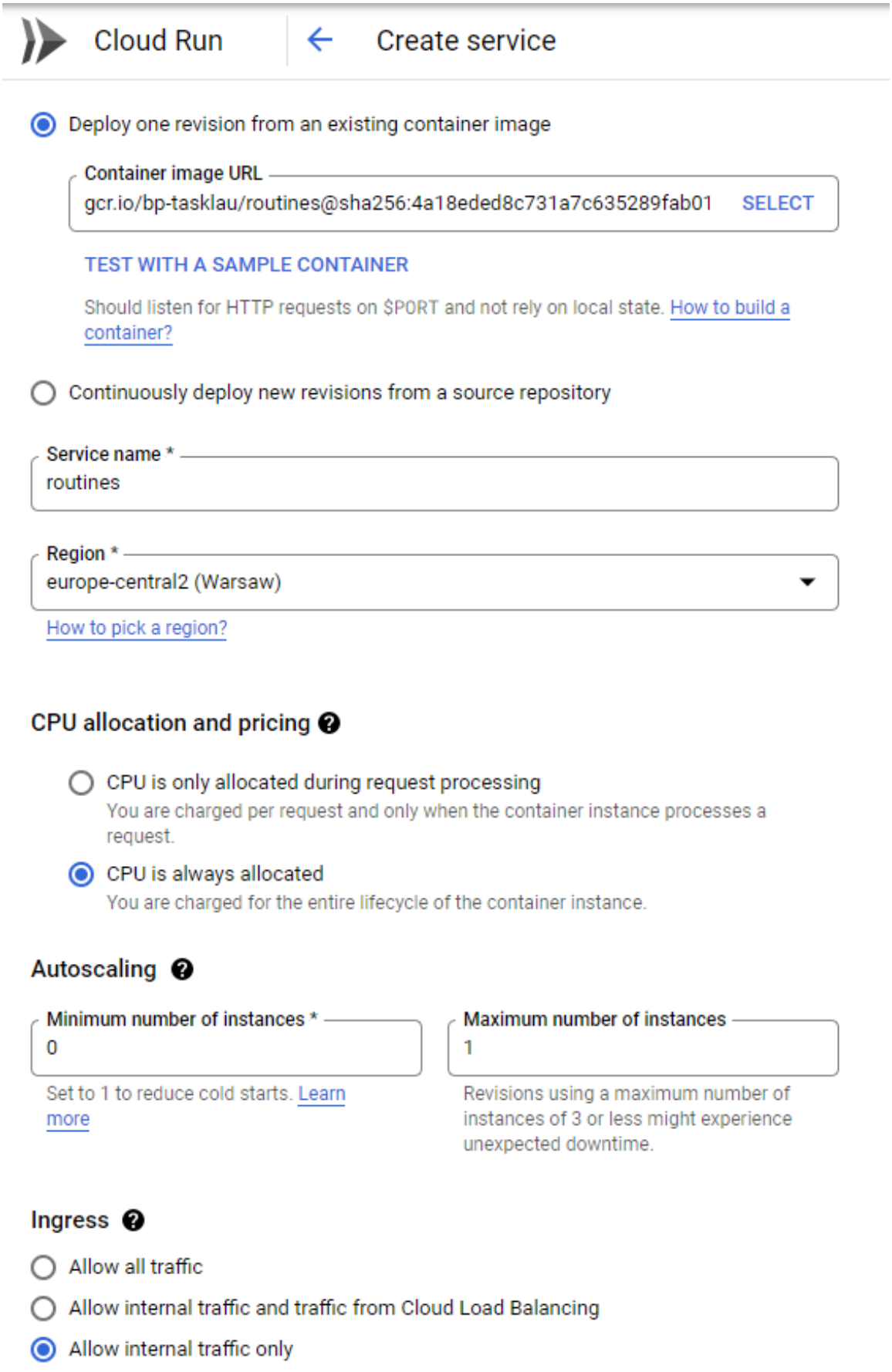
Obrázek A.2: Návrh databáze.



Obrázek A.3: Stavový protokol úlohy.



Obrázek A.4: Architektura služby.



Obrázek A.5: Vytvoření instance nahraného image na službě Google Cloud Run.

Příloha B

Spouštění projektu

Příloha obsahuje postup spouštění projektu.

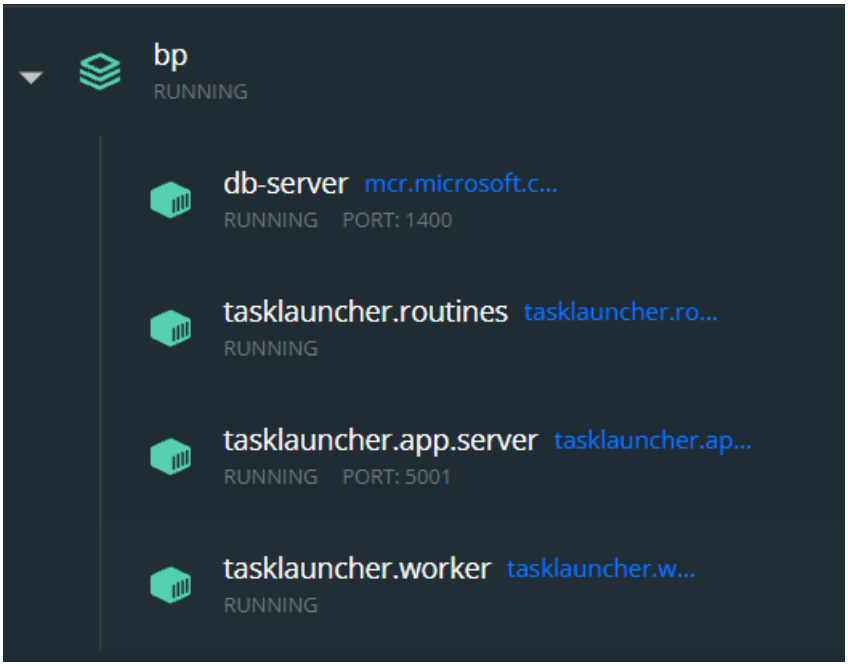
Zdrojové soubory a soubory pro Docker Compose se nachází v `src.zip` v příloze E. Obsah zip souboru se musí rozbalit do adresáře `BP`, ve kterém se budou provádět následující příkazy. Práce se spouští přes příkaz `docker-compose` (stažení a instalace pro Windows a Linux https://docs.docker.com/compose/install/). Nejdříve je ale potřeba vytvořit demonstrační image, který bude simulovat výpočet. Tento image se vytvoří přes příkaz `docker build -t testimage -f TestImage/Dockerfile .` (zadávat i s tečkou).

Před spuštěním služby je třeba vytvořit vývojářský certifikát vygenerovaný přes .NET CLI jinak nebude fungovat HTTPS provoz, a tudíž nepůjde komunikovat se službou Auth0. Lze použít i reálný ověřený certifikát. Certifikát musí být uložen v adresáři `BP` ve složce `https`, odkud se montuje do kontejneru.

1. Stažení .NET 6 SDK https://dotnet.microsoft.com/download/dotnet/6.0.
2. V adresáři `BP` se musí zadat příkaz
   `dotnet dev-certs https -ep https/aspnetapp.pfx -p mypass123`
3. A poté se zadává `dotnet dev-certs https --trust`

Celý projekt se spustí přes `docker-compose -f docker-compose.yml up --build -d`. Tento proces může trvat několik minut. Při dalším spuštění se v příkazu nemusí uvádět `--build`. Výsledkem by měly být čtyři spuštěné docker kontejnery (obrázek B.1). Tabulka B.1 obsahuje vytvořené uživatele po spuštění služby.
Na adrese https://localhost:5001 je dostupná webová aplikace.
Na adrese https://localhost:5001/swagger/index.html je dostupná dokumentace.
Vznikne kopie nasazené služby v Azure https://tasklauncher.azurewebsites.net/. Tato adresa je dostupná v období obhajob.

Služba se vypne přes `docker-compose -f docker-compose.yml down`.



Obrázek B.1: Výsledek spuštění služby přes Docker Compose.

Tabulka B.1: Tabulka uživatelů, kteří se ve službě vytvoří. Heslo ke všem účtům je `Password123*`. Ověřený uživatel je uživatel, který potvrdil e-mail. Registrovaný uživatel je uživatel, který dokončil registraci a může spouštět úlohy.

Uživatelský email	Popis
admin@email.com	Administrátor
tomashavel@test.com	VIP, ověřený a registrovaný uživatel
filipnovak@test.com	Ověřený a registrovaný uživatel
stepannemec@test.com	VIP (nějaký čas VIP neměl), ověřený a registrovaný uživatel
jakubstefacek@test.com	Neověřený a neregistrovaný uživatel

Příloha C

Konfigurace projektu

V této příloze se popisují konfigurační soubory a YAML soubory pro Docker Compose.

Projekt lze konfigurovat přes konfigurační proměnné v souboru `docker-compose.yml`. Konfigurační proměnné jsou definovány v souborech `appsettings.json` (například v projektu `TaskLauncher.App/Server`) a lze je přepsat pomocí YAML souboru z Docker Compose nebo z jiných nasazovacích nástrojů. Některé proměnné jsou proměnné prostředí frameworku ASP.NET Core.

Konfigurace serveru

Sekce a proměnné v konfiguračním souboru `TaskLauncher.App/Server/appsettings.json` slouží ke konfiguraci serveru. Lze měnit nastavení poskytovatele identity, připojovací řetězec k databázi, váhy front a podobně.

- `AllowedHosts, ReverseProxy, ReverseProxyExtensions` – Tyto sekce ignorovat.
- `Logging` – Nastavuje úroveň logování.
- `Auth0ApiConfiguration` – V sekci se konfiguruje Auth0. Je zde uvedeno ClientId a ClientSecret aplikace. Jsem si vědom, že tyto informace by se neměly objevovat v konfiguracích a neměly by být veřejné. Tyto údaje jsou ale pouze pro testování. Pro nasazenou službu na cloudu jsem v Auth0 vytvořil nového tenanta s jinými údaji. Vedle těchto dvou proměnných se nachází doména tenanta a API, ke kterému se získává autorizační token.
- `ProtectedApiAzp` – Toto je tzv. azp nárok (claim) obsahující ClientId Auth0 API.
- `Auth0Roles` – Zde se specifikují Id rolí z Auth0. Je to kvůli tomu, aby se mohlo vytvářet více tenantů. Každý tenant totiž vytváří jiné Id role.
- `StorageConfiguration` – Slouží pro připojení na Google Cloud Storage.
- `ConnectionStrings` – Důležitá sekce, kde se specifikují připojovací řetězce do databází. Proměnná `Default` je připojovací řetězec do hlavní databáze.
- `SeederConfig` – Nastavuje chování vkládání testovacích dat. Proměnná `seed` určuje, zda se budou testovací data vkládat. Pokud je nastaveno na true, vytvoří se testovací uživatelé s testovacími daty (pouze pokud je prázdná databáze).

- `PriorityQueues` – Sekce nastavuje priority daných front. Čím větší číslo, tím větší priorita.

Konfigurace management aplikace

Management aplikaci konfiguruje soubor `TaskLauncher.Routines/appsettings.json`. Zde se opakují některé proměnné a sekce z konfigurace serveru. V `ConnectionStrings` je akorát další připojovací řetězec na pomocnou databázi.

Konfigurace worker aplikace

Posledním konfiguračním souborem je `TaskLauncher.Worker/appsettings.json`. Ten konfiguruje worker aplikaci, která spouští demonstrační kontejner simulující výpočet. V tomto souboru se nastavují i parametry tohoto spouštěného kontejneru.

Kromě již zmíněných proměnných a sekcí z předchozích konfiguračních souborů je zde sekce `ServiceAddresses` určující, na jakou adresu se má připojit SignalR klient a na jaké adrese je hostován server služby.

Sekce `TaskLauncherConfig` konfigurující demonstrační kontejner obsahuje tyto proměnné:

- `Target` – Určuje cestu v kontejneru, kam bude namontován volume. Nedoporučuje se měnit tuto proměnnou, protože ji používá i spuštěný kontejner s výpočtem. Oba kontejnery přes volume sdílí soubor.
- `Source` – Jméno volume. Stejný volume by měl být přimontován jak k worker kontejneru, tak k workerem spuštěnému kontejneru. Jinak bude worker ukládat soubor jinam a kontejner s výpočtem nebude moct do tohoto souboru zapsat výsledek výpočtu.
- `ImageName` – Jméno testovacího/demonstračního image.
- `ContainerArguments__Mode` – Zde jsou možné pouze dvě možnosti: `seconds`, `minutes`. Určuje jednotku času v následujících proměnných.
- `ContainerArguments__Min` – Minimální možný čas, po který poběží spuštěný kontejner s výpočtem. Jednotku určuje proměnná `Mode`.
- `ContainerArguments__Max` – Maximální možný čas, po který poběží spuštěný kontejner s výpočtem. Jednotku určuje proměnná `Mode`.
- `ContainerArguments__Chance` – Určuje šanci na úspěch úlohy (zadává se jako celé číslo od 0 po 100).

Soubory pro Docker Compose

Zde se krátce popisují YAML soubory pro Docker Compose, které se využívají pro nasazení a spuštění služby. V těchto souborech lze přepisovat proměnné ze zmíněných konfiguračních souborů. V YAML se používá speciální notace pro přístup k proměnným v konfiguraci. Například pro změnu proměnné `seed` v sekci `SeederConfig` je třeba využít tento zápis: `SeederConfig__seed=false`.

- `docker-compose.yml` – Tento soubor se používá pro lokální nasazení služby. Vytvoří databázový server, serverovou aplikaci, management aplikaci a worker aplikaci. Je třeba vytvořit demonstrační image. Postup v příloze B.
- `docker-compose.worker.yml` – Tento soubor slouží pro nasazení worker aplikace například na nějaký virtuální server. Soubor je nakonfigurován tak, že se image stáhne z registru Docker Hub. Kromě tohoto souboru se tak nemusí nikam přesouvat zdrojové kódy. V souboru se pouze specifikuje Auth0 tenant a adresa serveru. Nesmí se zapomenout na vytvoření nebo stažení demonstračního image, který bude spouštěn worker aplikací. Image lze stáhnout přes `docker pull wutshot/testimage:latest`. Pokud se server služby nasadil na Azure podle 6.5 a zvolilo se jméno instance na `testauth0blazorwasmserverapp`, mělo by nasazení workera fungovat bez další konfigurace.
- `docker-compose.simulation.yml` – Slouží pro spouštění simulace. Více v následující příloze.

Příloha D

Spouštění a konfigurace simulace

V této příloze je uveden postup spuštění simulace. Dále jsou uvedeny parametry, které lze v simulaci měnit.

Spuštění simulace probíhá opět pomocí `docker-compose` v adresáři `BP`. V příkazu se použije soubor se simulací `docker-compose -f docker-compose.simulation.yml up --build -d`. Opět je nutné mít testovací image z přílohy B. Ukončení a uklizení simulace se provádí přes příkaz `docker-compose -f docker-compose.simulation.yml down`. Pokud je v YAML souboru definováno více workerů, je možné, že se na startu zaseknou (nebudou se generovat kontejnery s úlohami). Stačí workery restartovat, problém by se pak neměl znovu objevit.

Pro sledování simulace se dá přihlásit do služby na adrese `https://localhost:5001/` přes administrátorský účet. Lze se přihlásit i přes simulací vytvořené uživatele. Heslo je stejné jako z přílohy B.

Konec simulace lze poznat podle toho, že se negenerují další kontejnery. Statistiky lze získat přes REST API (`api/admin/stats`). Lze využít OpenApi dokumentace nebo nástroj Postman. Po ukončení simulace (`docker-compose down`) se vytvoří soubor s těmito daty, který bude dostupný v adresáři `BP` ve složce `stats`.

Konfigurace

Soubor `docker-compose.simulation.yml` vypadá obdobně jako `docker-compose.yml`, pouze zde není management aplikace. Je zde ale nová aplikace, která vytváří uživatele a vytváří jejich úlohy. V souboru jsou definovány dvě worker instance. Lze jich vytvořit více. Stačí zkopírovat jednoho workera a vytvořit nový volume. Worker i server aplikace jsou nastaveny na jiného Auth0 tenanta, aby se vytvoření uživatelé nemíchali do tenantu, který může být použitý v nasazené aplikaci. V části `environment` služby `tasklauncher.simulation` tohoto YAML souboru lze měnit parametry simulace. Tyto parametry definuje soubor `TaskLauncher.Simulation/appsettings.json`.

- `SimulationConfig__VipUsers` – Nastavuje počet vygenerovaných VIP uživatelů.
- `SimulationConfig__NormalUsers` – Nastavuje počet standardních uživatelů.
- `SimulationConfig__TaskCount` – Nastavuje počet úloh, které každý vytvořený uživatel vytvoří.

- `SimulationConfig__DelayMin` – Nastavuje spodní hranici časového intervalu v sekundách. Z intervalu se náhodně vygeneruje zpoždění, po kterém se opět spustí další úloha.
- `SimulationConfig__DelayMax` – Nastavuje horní hranici časového intervalu v sekundách. Z intervalu se náhodně vygeneruje zpoždění, po kterém se opět spustí další úloha.

U instancí workera lze změnit, jak dlouho bude demonstrační kontejner běžet. Tím se dá ovlivnit čekací doba ve frontě, jelikož výpočet bude třeba trvat delší dobu. Další informace ohledně konfigurace workera jsou v příloze C.

Příloha E

Obsah přiloženého paměťového média

- `xhajek50-text.pdf` – PDF soubor celé práce
- `xhajek50-text.zip` – Zdrojové soubory textu práce
- `restapidoc.pdf` – Generovaná REST API dokumentace
- `src.zip` – Zdrojové a konfigurační soubory
  - `README.md` - Jenoduchý návod na spuštění projektu
  - `TaskLauncher.Api.Contracts` – Těla odpovědí a dotazů pro REST API
  - `TaskLauncher.App` – Webová aplikace, obsahuje další členění na server a klient část.
  - `TaskLauncher.App.DAL` – Databázová vrstva webové aplikace
  - `TaskLauncher.Authorization` – Třídy integrující službu Auth0 zajišťující autorizaci
  - `TaskLauncher.Common` – Společné a pomocné třídy
  - `TaskLauncher.Routines` – Management aplikace vykonávající rutinní údržbové práce
  - `TaskLauncher.Simulation` – Simulační program
  - `TaskLauncher.Worker` – Implementace worker aplikace
  - `TestImage` - Demonstrační image
  - `docker-compose.simulation.yml` – Spuštění simulace přes Docker Compose
  - `docker-compose.worker.yml` – YAML soubor pro nasazení worker aplikace přes Docker Compose
  - `docker-compose.yml` – Nasazení celé služby přes Docker Compose
  - `TaskLauncherService.sln` – Soubor vytvářející řešení nad všemi uvedenými projekty