Projektni rad iz kolegija Infrastrukture za podatake velikog obujma

Autori: Deni Kernjus i Dino Ladavac

Mentori: dr. sc. Rok Piltaver i mag. inf. Tomislav Slaviček Car

Rijeka, 29.1.2024.

Uvod

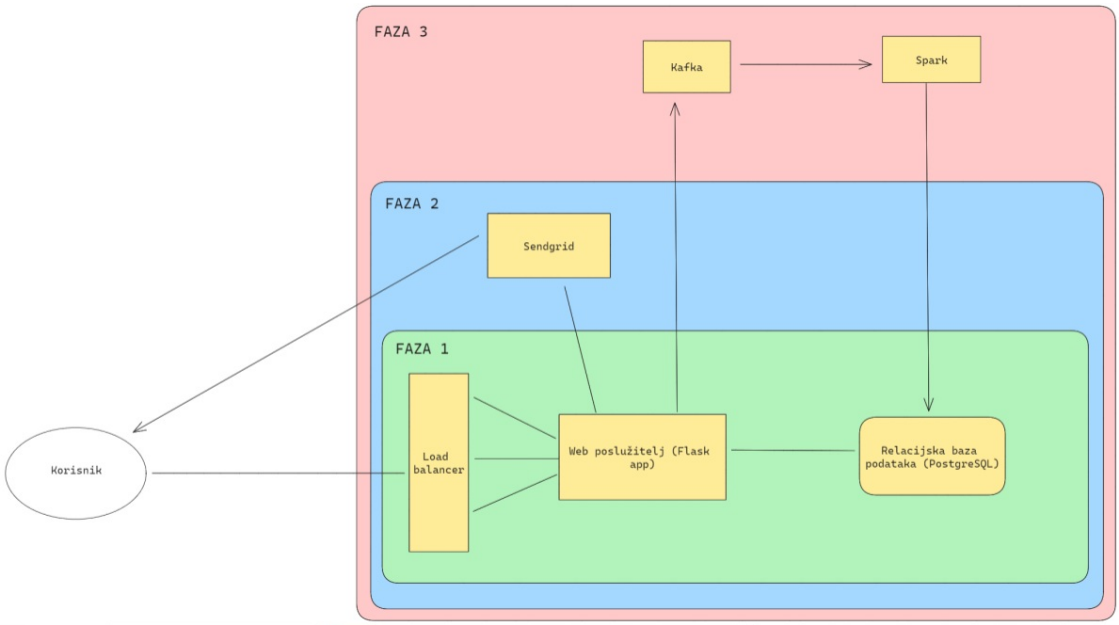
U sklopu projektnog rada iz kolegija Infrastrukture za podatke velikog obujma potrebno je timski izraditi web aplikaciju za podatke velikog obujma u obliku start-up firme. U tu svrhu izrađivat će se web aplikacija za osobno praćenje zdravlja (Health Tracker) u kojem se korisniku omogućuje postavljanje ciljeva te unošenje dnevnih vrijednosti vezanih uz unos kalorija, tekućina, sati spavanja i slično. Razvoj aplikacije podjeljen je u tri faze koje opisuju način izrade aplikacije i funkcionalnosti koje aplikacija ima prilikom izrade.

U prvoj fazi, korisniku se omogućuje unos dnevnih zdravstvenih parametara te mu se isti prikazuju na ekranu. Korisniku se također omogućuje prikaz prethodnih paprametara unesenih kroz određene vremenske periode. Ova faza sastoji se od uspostavljanja web poslužitelja za obradu podataka te relacijske baze u koju se podaci spremaju.

Drug Faza sastoji se od dodatne usluge obavijesti koja korisnika obaviještava, nakon unesenih ciljeva, ukoliko oni nisu ispunjeni kako bi ga daljnje potaknuli na ostvarenje istih.

Treća faza uvodi analitiku podataka tako da računa trendove ostalih korisnika te računa trendove korisnika istih dobnih razreda, težine ili visine te iste podatke prikazuje korisniku kako bi mogao suporediti svoj napredak u odnosu na ostale korisnike.

Zbog povećanja skalabilnosti i brzine izvođenja, dodat će se balanser opterećenja sa tri kopije poslužitelja. Više o konfiguraciji balansera opterećenja biti će prikazano kasnije.



Slika: Arhitektura aplikacije

Prva Faza

Prva faza sastoji se od razvoja web poslužitelja, neralacijske baze te veze između njih. Za izradu logike web poslužitelja korišten je okvir Flask jer je jednostavan za korištenje i brz što nam omogućuju laku daljnju skalabilnost, nadograđivanje i ispravljanje grešaka. Flask koristi programski jezik python stoga je pisanje određenih djelova poput ruta ili modela poprilično jednostavno. Ova aplikacija sadržavat će 3 modela.

Prvi model su korisnički podaci poput njegovog emaila koji će nam kasnije služiti za slanje obavijesti, korisničkog imena i lozinke za prijavljivanje (iako konkretan sustav autorizacije nije implementiran).

Sljedeći model su korisnikovi podaci koje unosi na dnevnoj bazi. Tu se ubrajaju unos kalorija, litara tekućine the sati spavanja i vježbanja (uz datum kada se podaci unose). Ovaj model koristit će se kod slanja obavijesti u drugoj fazi.

Zadnji model čine korisnikovi ciljevi (za unos kalorija, sati spavanja i dr.), težina i visina. Ovaj model koristit će se kod slanja obavijesti u drugoj fazi kao i kod analitike u trećoj.

```
class User(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(80), unique=True, nullable=False)
    email = db.Column(db.String(120), unique=True, nullable=False)
    password = db.Column(db.String(255), nullable=False)
    health_metrics = db.relationship('HealthMetrics', backref='user',
lazy=True)
    health_goals = db.relationship('UserHealthGoals', backref='user',
uselist=False, lazy=True)


class HealthMetrics(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    user_id = db.Column(db.Integer, db.ForeignKey('user.id'), nullable=True)
    date = db.Column(db.Date, default=datetime.utcnow)
    calorie_intake = db.Column(db.Float)
    exercise_duration = db.Column(db.Float)
    sleep_hours = db.Column(db.Float)
    water_consumed = db.Column(db.Float)

class UserHealthGoals(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    user_id = db.Column(db.Integer, db.ForeignKey('user.id'), nullable=True)
    goal_type_calorie = db.Column(db.Float)
    goal_type_exercise = db.Column(db.Float)
    goal_type_sleep = db.Column(db.Float)
    goal_type_water = db.Column(db.Float)
    age = db.Column(db.Float)
    height = db.Column(db.Float)
    weight = db.Column(db.Float)
```

Svi podaci i modeli zapisuju se u PostgreSQL relacijsku bazu podataka koja je kompatibilna sa okvirom Flask te omogućuje skalabilnost i pouzdanost ukoliko se koriste pouzdane veze i indeksi.

Da bi se aplikacija postavila, koristi se Docker za kontenjerizaciju koji svaki servis postavlja u zaseban kontenjer te, uz navedene zavisnosti, postavlja web poslužitelj i ostale servise istovremeno prilikom pokretanja.

Osim modela, u kodu aplikacije navode se i rute na određene web stranice. Navođenjem ruta i funkcija nakon njih, apliakciji naređujemo što da čini kod pokretanja navedene web stranice.

Kako aplikacija nema predviđen sustav autorizacije u ovim fazama razvoja, ovdje će biti implementirana demo vezija autorizacije koja radi na princip čitanja korisničkog imena iz baze podataka i provjeravanja ukoliko lozinka odgovara korisničkom imenu. Ovi podaci nisu haširani,

već su zapisani kao običan tekst.

Prilikom učitavanja web aplikacije, očekuje se od korisnika da se registrira. To radi unosom email-a, korisničkog imena i lozinke te se u slučaju da korisničko ime već postoji, korisniku ispisuje povratna poruka. U suprotnom u bazu se zapisuju podaci za novog korisnika.

```
@app.route('/', methods=['GET', 'POST'])
def register():
    if request.method == 'POST':
        username = request.form.get('username')
        email = request.form.get('email')
        password = request.form.get('password')

        existing_user = User.query.filter((User.username == username) |
(User.email == email)).first()

        if existing_user:
            error_message = 'Error: Username already exists'
            flash(error_message)
            return redirect(url_for("register"))
        else:
            new_user = User(
                username=username,
                email=email,
                password=password
            )
            db.session.add(new_user)
            db.session.commit()

            return redirect(url_for('user_dashboard', username=username))

    return render_template('register.html')
```

Stranica za registraciju pisana je u html datoteci *register.html*.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Register | Health Tracker</title>
    <link rel="stylesheet" href="{{ url_for('static', filename='css/style.css')
}}">
</head>
<body class="auth-body">
    <section class="auth-container">
        <h1>Welcome!</h1>
        <p>Register to track your Health</p>
        <body>
            {% set messages = get_flashed_messages() %}
                {% if messages %}
                     <h1>{{ messages[-1] }}</h1>
                {% endif %}
          </body>
            <form id="register-form" method="post" action="{{
url_for('register') }}">
            <div class="input-group">
                <label for="email">Email</label>
                <input type="email" id="email" name="email" required>
            </div>

            <div class="input-group">
                <label for="username">Username</label>
                <input type="text" id="username" name="username" required>
            </div>

            <div class="input-group">
                <label for="password">Password</label>
                <input type="password" id="password" name="password" required>
            </div>

            <button type="submit">Register</button>
            <p>Already have an account? <a href="{{ url_for('login')
}}">Login</a></p>
        </form>
    </section>
</body>
</html>
```

izvor: ChatGPT

Na sličan način funkcionira prijava korisnika. Ukoliko kod prijave korisnik postoji i njegova se lozinka podudara sa lozinkom zapisanom u bazi, tada se korisnika preusmjerava na početnu stranicu (ruta */username* gdje je *username* korisničko ime korisnika). U suprotnom, korisnika se preusmjerava na registraciju te se od njega očekuje da stvori svoj račun.

```
@app.route('/login', methods=['GET', 'POST'])
def login():
    if request.method == 'POST':
        username = request.form.get('username')
        password = request.form.get('password')

        user = User.query.filter_by(username=username,
password=password).first()

        if user:
            session['username'] = username  # Store username in session
            return redirect(url_for('user_dashboard', username=username))
        else:
            error_message = 'Error: Username or password is incorrect.'
            flash(error_message)
            return redirect(url_for("login"))

    # If it's a GET request or login fails, render the login template
    return render_template('login.html')
```

Izgled login stranice sličan je izgledu *register.html-a*.

Nakon što se korisnik uspješno prijavi, prikazuje se nekoliko mogućnosti:

- Logout koji samo odjavljuje korisnika i vraća ga na login stranicu.
- Alatna traka na kojoj može odabrati tab za *Account, Health Metrics i Add Metrics* te sadržaj otvorenog taba ispod.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Health Tracker</title>
    <link rel="stylesheet" href="{{ url_for('static', filename='css/style.css')
}}">
    <script src="https://unpkg.com/htmx.org" defer></script>
</head>
<body>
    <header>
        <nav>
            <ul>
                <li><a href="/account" hx-get="/account" hx-trigger="click" hx-
target="#content" hx-swap="innerHTML">ACCOUNT</a></li>
                <li><a href="/" hx-get="/health-metrics" hx-trigger="click" hx-
target="#content" hx-swap="innerHTML">HEALTH METRICS</a></li>
                <li><a href="/" hx-get="/add-metric" hx-trigger="click" hx-
target="#content" hx-swap="innerHTML">ADD METRIC</a></li>
            </ul>
            <a href="/logout" class="logout">Logout</a>
        </nav>
    </header>

    <main id="content">
        {% block content %}
           {{ content|safe }}
        <!-- The content from child templates will be injected here -->
        {% endblock %}
    </main>
```

Pod tabom *Account* korisnik može unositi vrijednosti za model Korisnika (svoje ciljeve, težinu i visinu kao i email).
Pod tabom *Health Metrics* prikazano je nekoliko gumbova pod kojima korisnik odabire prikaz svojih unesenih podataka za neka razdoblja (danas, jučer, prošlih 7 dana…) te analitika za određene kategorije (ja, moje godište, moja težina…).
Pod tabom *Add metric* korisnik unosi dnevne aktivnosti koje čine model HealthMetrics.

```
<section class="add-metric">
    <h2>New Entry</h2>
    <form id="add-metric-form" hx-post="/submit-metric" hx-target="#content"
hx-swap="innerHTML">
        <div class="input-group">
            <label for="date">Date</label>
            <input type="date" id="date" name="date" required
class="datepicker" required>
        </div>

        <div class="input-group">
            <label for="calorie-intake">Calorie intake in kcal</label>
            <input type="number" id="calorie-intake" name="calorie_intake"
required>
        </div>

        <div class="input-group">
            <label for="exercise">Exercise in minutes</label>
            <input type="number" id="exercise" name="exercise" required>
        </div>

        <div class="input-group">
            <label for="sleep">Sleep in hours</label>
            <input type="number" id="sleep" name="sleep" required>
        </div>

        <div class="input-group">
            <label for="water-intake">Water intake in liters</label>
            <input type="number" id="water-intake" name="water_intake"
required>
        </div>

        <button type="submit">Save</button>
    </form>
</section>s
```

Unosom ovih podataka, oni se u bazu spremaju te se pod tabom *Health metrics* prikazuju ovisno o datumu koji je unesen (primjerice ako je unesen današnji datum, ovi podaci će se prikazat pod odjeljkom “danas” kao i “prethodna 7 dana” gdje će biti prikazani uz još podataka koji su uneseni u tom vremenskom periodu).

```
@app.route('/submit-metric', methods=['POST'])
def submit_metric():
    if 'username' not in session:
        return redirect(url_for('login'))

    username = session['username']
    user = User.query.filter_by(username=username).first()

    if user:
        date = request.form.get('date')
        calorie_intake = request.form.get('calorie_intake')
        exercise_duration = request.form.get('exercise')
        sleep_hours = request.form.get('sleep')
        water_consumed = request.form.get('water_intake')

        health_metrics = HealthMetrics(
            user=user,
            date=date,
            calorie_intake=calorie_intake,
            exercise_duration=exercise_duration,
            sleep_hours=sleep_hours,
            water_consumed=water_consumed
        )

        db.session.add(health_metrics)
        db.session.commit()
    return render_template('partials/health_metrics.html')
```

Učitavanje podataka ovisno o vremenskom periodu:

```
 if period == 'today':
            metrics_query = metrics_query.filter(HealthMetrics.date ==
datetime.today().date())
        elif period == 'yesterday':
            metrics_query = metrics_query.filter(HealthMetrics.date ==
(datetime.today() - timedelta(days=1)).date())
        elif period == 'last7days':
            metrics_query = metrics_query.filter(HealthMetrics.date >=
(datetime.today() - timedelta(days=7)).date())
        elif period == 'lastmonth':
            metrics_query = metrics_query.filter(HealthMetrics.date >=
(datetime.today() - timedelta(days=30)).date())
```

Druga faza

U drugoj fazi razvoj web aplikacije fokus je na uvođenju usluge obavijesti. Kako bi obaviještavanje korisnika bilo pouzdano, potrebno je odabrati dobar način obaviještavanja, a kod web aplikacija to je najčešće putem e-mail adrese. Ovaj način je, uz to, i preaktičan jer se korisnik registrira u web aplikaciju korištenjem e-mail adrese, stoga je ovaj podatak već spremljen u bazu podataka.

Za slanje obavijesti koristiti će se servis *SendGrid*. To je servis temeljen u oblaku koji omogućuje slanje elektroničkih pošta putem programa, a obično se koristi kod slanja transakcijskih i marketinških mailova u takvim tipovima aplikacija.

U ovoj web-aplikaciji, *SendGird* je integriran u falskovu biblioteku *flask_mail* te je za korištenje potreban poziv funkcije i postavnjanje konfiguracijih postavka.

Koristiti će se SendGrid-ov SMTP server *smtp.sendgrid.net (http://smtp.sendgrid.net)* te je potrebno postaviti port na 587 za slanje mailova. Pošto ovo tip edukacijski tip aplikacije, mailove ćemo slati putem naše email adrese koju je potrebno navesti te se pravilno logirati koristeći api ključ.

Više o tome moguće je pročitati u uputama za postavljanje servisa SendGrid putem flask_mail-a (https://sendgrid.com/en-us/blog/sending-emails-from-python-flask-applications-with-twilio-sendgrid).

Većina web aplikacija koristi obavijesti nakon (ili tijekom) registracije korisnika gdje se korisniku na elektroničku adresu šalje aktivacijski kod ili potvrda o uspiješnoj registraciji.

Da bi se ova funkcionalnost implementirala, potrebno je pozvati funkciju *Message()* u kojoj navedodimo naziv poruke te adrese na koje se poruka šalje. U ovom slučaju, ta se adresa nalazi u varijabli *email* koja se popunjava nakon unosa korisnika. Sadržaj poruke piše se u *body*, a raspored se određuje u *html-u*.

```
msg = Message('Welcome to Health tracker app', recipients=[email])
            msg.body = ('Congratulations! You have successufully reegistered in
Health tracker app')
            msg.html = ('<h1>Welcome to Health tracker app</h1>'
                        '<p>Congratulations! '
                        '<b>You have successufully reegistered in Health
tracker app</b>!</p>')
            mail.send(msg)
```

Slanje obavijesti korisno je nakon što korisnik unese svoje ciljeve kako bi ga aplikacija na tjednoj bazi potakla na ostvarivanje ciljeva (primjerice: “U ovom tjednu spavali ste 50h, vaš cilj je postavljen na 70h, Večeras legnite prije spavati!”).

Korisnik unosi ciljeve pod tabom *Account* gdje može unijeti svoje godine, visinu, težinu i ciljeve vezane uz unos kalorija i tekućine te vremena spavanja i vježbanja.

Nakon unosa, korisniku se ispisuju unesene vrijednosti te se spremaju u bazu podataka. Ukoliko ih on želi izmjeniti, one ostaju zapamćene te mu se omogućuje izmjena.

```
if user:
        age = request.form.get('age')
        height = request.form.get('height')
        weight = request.form.get('weight')
        calorie_intake_goal = request.form.get('calorie_intake_goal')
        exercise_goal = request.form.get('exercise_goal')
        sleep_goal = request.form.get('sleep_goal')
        water_intake_goal = request.form.get('water_intake_goal')

        if user.health_goals:
            user_health_goals = user.health_goals
        else:
            user_health_goals = UserHealthGoals(user=user)

        user_health_goals.age = age
        user_health_goals.height = height
        user_health_goals.weight = weight
        user_health_goals.goal_type_calorie = calorie_intake_goal
        user_health_goals.goal_type_exercise = exercise_goal
        user_health_goals.goal_type_sleep = sleep_goal
        user_health_goals.goal_type_water = water_intake_goal

        db.session.add(user_health_goals)
```

Treća faza

Treća faza odnosi se na uvođenje analitike parametara zdravlja. Korisniku se u prethodnoj fazi omogućio unos ciljeva te pregledavanje osobnih postigunuća, u ovoj fazi omogućena mu je i analitika, odnosno suporedba svojih postignuća s prosijekom postigunća ostalih korisnika istih karakteristika kao što su Age, Height i Weight.

Za analitiku koristi se Apache Spark te se obrađeni podaci spremaju u nove tablice u bazu podataka tako da su dostupne u svim djelovima aplikacije. Kako bi pokrenuli Spark skriptu koristimo Apache Kafka. Ovaj alat omogućava definiranje producera (Flask app) i consumera (Spark skripta) na način koji omogućava pokretanje spark skripte samo kada je to potrebno te također na ovaj način osiguravamo da sve instance Flask aplikacije koriste samo jednu Spark sesiju.

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, avg, monotonically_increasing_id,
get_json_object, from_json
from pyspark.sql.types import StringType, StructField, StructType,
TimestampType

# Define the schema of the Kafka messages
schema = StructType([
    StructField("trigger", StringType()),
    StructField("groupBy", StringType())
])

# Initialize Spark Session
spark = SparkSession.builder \
    .appName("Health-Data-Analysis") \
    .config("spark.jars.packages", "org.apache.spark:spark-sql-kafka-0-
10_2.12:3.0.1") \
    .getOrCreate()

# Read data from Kafka
df_kafka = spark.readStream \
    .format("kafka") \
    .option("kafka.bootstrap.servers", "kafka:9092") \
    .option("subscribe", "spark_trigger") \
    .load()

# Deserialize the value
df_values = df_kafka.selectExpr("CAST(value AS STRING)")


schema_ddl = schema.simpleString()

# Apply the schema to the data
df_structured = df_values.selectExpr(f"from_json(value, '{schema_ddl}') as
data").select("data.*")

def process_trigger(df, epoch_id):
    # JDBC URL
    jdbc_url = "jdbc:postgresql://postgres_db:5432/postgres"
    properties = {
        "user": "postgres",
        "password": "postgres",
        "driver": "org.postgresql.Driver"
    }

    group_by_values = df.select("groupBy").collect()

    if group_by_values:
        group_by_column = group_by_values[0]["groupBy"]
    else:
        # Handle the case when there are no rows
        group_by_column = None
```

```
    # Read data from PostgreSQL
    df_health_metrics = spark.read.jdbc(url=jdbc_url, table="health_metrics",
properties=properties)
    df_user_goals = spark.read.jdbc(url=jdbc_url, table="user_health_goals",
properties=properties)

    # Join the tables on user_id
    df_joined = df_health_metrics.join(df_user_goals, "user_id")

    if group_by_column == "age" or group_by_column == None:
        # Group by Age, Weight, and Height and calculate averages
        df_age_analysis = df_joined.groupBy("age").agg(
            avg("calorie_intake").alias("calorie_intake"),
            avg("exercise_duration").alias("exercise_duration"),
            avg("sleep_hours").alias("sleep_hours"),
            avg("water_consumed").alias("water_consumed")
        ).withColumn("id", monotonically_increasing_id())
        # Write the results back to PostgreSQL
        df_age_analysis.write.jdbc(url=jdbc_url, table="age_analysis",
mode="overwrite", properties=properties)

    elif group_by_column == "weight" or group_by_column == None:
        df_weight_analysis = df_joined.groupBy("weight").agg(
            avg("calorie_intake").alias("calorie_intake"),
            avg("exercise_duration").alias("exercise_duration"),
            avg("sleep_hours").alias("sleep_hours"),
            avg("water_consumed").alias("water_consumed")
        ).withColumn("id", monotonically_increasing_id())
        # Write the results back to PostgreSQL
        df_weight_analysis.write.jdbc(url=jdbc_url, table="weight_analysis",
mode="overwrite", properties=properties)

    elif group_by_column == "height" or group_by_column == None:
        df_height_analysis = df_joined.groupBy("height").agg(
            avg("calorie_intake").alias("calorie_intake"),
            avg("exercise_duration").alias("exercise_duration"),
            avg("sleep_hours").alias("sleep_hours"),
            avg("water_consumed").alias("water_consumed")
        ).withColumn("id", monotonically_increasing_id())
        # Write the results back to PostgreSQL
        df_height_analysis.write.jdbc(url=jdbc_url, table="height_analysis",
mode="overwrite", properties=properties)


# Start the streaming query
query = df_structured.writeStream \
    .foreachBatch(process_trigger) \
    .start()

query.awaitTermination()
```

Plan za budućnost

Kako je ovo aplikacija izrađena u edukacijske svrhe, ona je daleko od prve uporabljive verzije. Da bi njen razvoj mogao teći u pravom smjeru, potrebno se fokusirati na trenutne probleme i predložiti moguća rješenja.

Problem	Rješenje
1. Optimizacija i Skalabilnost baze podataka	Implementacija indeksiranja i particioniranja, ACID, Backup podataka
2. Slaba sigurnosti	Sustav autentifikacije, haširanje poruka i osjetljivih korisničkih podataka
3. Nedostatak zapisnika grešaka	Uvođenje praćenja performansi, logova, Error handling
4. Monolitna arhitektura	Moguća tranzicija na mikroservisnu arhitekturu za ubrzavanje izvođenja aplikacije i povećanja skalabilnosti

Kako bi u budućnosti nadogradili sustav, moguće je dodati izvan mrežni sustav kojim bi se korisnicima omogućio unos parametara zdravlja bez povezivanja na mrežu. Ova nadogradnja zahtjevala bi stvaranje lokalne baze podataka u koju korisnik unosi podatke te se ti podaci ovde u web aplikaciju tek kad postoji veza sa internetom. Tada se baza podataka prazni kako se nebi bespotrebno popunjavala. Unesene podatke je potrebno dodatno sinkronizirati kako nebi došlo do konflikta te se za to uvodi dodatan servis. Pretpostavka je da se prije uvođenja novih nadogradnji, arihtektura sustava prebaci na mikroservisnu kako bi se ubrzao proces te povećala skalabilnost. Lokalna baza podataka nebi trebala biti prezahtjevna stoga bi se u tu svrhu mogao korisiti SQLite dok bi se za komunikaciju i sinkronizaciju sa poslužiteljom koristili RESTful API.

Konfiguracija Balansera opterećenja i kontenjerizacija

Za balanser opterećenja koristi se *nginx* servis čija je konfiguracija navedena u nastavku

nginx.conf:

```
events {
    worker_connections 1024;
}
http {
    upstream flask_app {
    server flask_app:4000;
    server flask_app2:4000;
    server flask_app3:4000;
}

    server {
        listen 80;
        server_name localhost;

        location / {
            proxy_pass http://flask_app;
    }
}
}
```

Kontenjerizacija je postignuta *dockerom* i *docker-compose.yml*

```
version: "3.9"

services:
  postgres_db:
    container_name: postgres_db
    image: postgres:12
    ports:
      - "5432:5432"
    environment:
      - POSTGRES_PASSWORD=postgres
      - POSTGRES_USER=postgres
      - POSTGRES_DB=postgres
    volumes:
      - pgdata:/var/lib/postgresql/data

  flask_app:
    container_name: flask_app
    build: ./main
    ports:
      - "4010:4000"
    environment:
      - DB_URL=postgresql://postgres:postgres@postgres_db:5432/postgres
    depends_on:
      - postgres_db

  flask_app2:
    container_name: flask_app2
    build: ./main
    ports:
      - "4011:4000"
    environment:
      - DB_URL=postgresql://postgres:postgres@postgres_db:5432/postgres
    depends_on:
      - postgres_db

  flask_app3:
    container_name: flask_app3
    build: ./main
    ports:
      - "4012:4000"
    environment:
      - DB_URL=postgresql://postgres:postgres@postgres_db:5432/postgres
    depends_on:
      - postgres_db

  nginx:
    build:
      context: .
      dockerfile: Dockerfile.nginx
    ports:
      - "4000:80"
    depends_on:
      - flask_app
      - flask_app2
      - flask_app3

volumes:
  pgdata:
```

Literatura

Flask službena dokumentacija (https://flask.palletsprojects.com/en/3.0.x/)
Dokumentacija SendGrid (https://sendgrid.com/en-us/blog/sending-emails-from-python-flask-applications-with-twilio-sendgrid)
Apache Spark Dokumentacija (https://spark.apache.org/docs/latest/)
ChatGPT (https://chat.openai.com/)
StackOverflow (https://stackoverflow.com/)