WROCŁAW UNIVERSITY OF SCIENCE AND TECHNOLOGY
FACULTY OF ELECTRONICS

Field:	Informatyka Techniczna (INF)
Specialization:	Internet Engineering (INE)

MASTER OF SCIENCE THESIS

Comparison analysis and efficiency evaluation of back-end frameworks in database applications

Analiza porównawcza i ocena wydajności frameworków back-endowych w aplikacjach bazodanowych

Author:
Marcin Wojciechowski

Supervisor:
Dr inż. Paweł Głuchowski W4/K9

Grade:

WROCŁAW 2021

Contents

Acronyms **2**

1 Introduction **4**

2 Technology description **5**
- 2.1 PostgreSQL . . . 5
- 2.2 Django . . . 5
  - 2.2.1 Overview . . . 5
  - 2.2.2 Requirements and dependencies . . . 6
- 2.3 ExpressJS . . . 6
  - 2.3.1 Overview . . . 6
  - 2.3.2 Requirements . . . 6
- 2.4 ASP.NET Core . . . 7
  - 2.4.1 Overview . . . 7
  - 2.4.2 Requirements . . . 7
- 2.5 K6 and related tools . . . 7
- 2.6 Docker and Docker Compose . . . 7

3 Criteria description **8**
- 3.1 Performance benchmark . . . 8
  - 3.1.1 Scenarios . . . 8
  - 3.1.2 Data . . . 9
  - 3.1.3 Application isolation . . . 9
  - 3.1.4 Test progress . . . 9
  - 3.1.5 Software versions and hardware . . . 10
- 3.2 Security . . . 10

4 System design **12**
- 4.1 Database . . . 12
  - 4.1.1 Initial population creation . . . 12
- 4.2 Applications endpoints . . . 13
- 4.3 Environment . . . 15

5 Implementation **16**
- 5.1 Docker and Docker Compose . . . 16
- 5.2 Django . . . 16
  - 5.2.1 Model . . . 16
  - 5.2.2 Database connection and initialization . . . 16
  - 5.2.3 Routing and serialization . . . 17
  - 5.2.4 Endpoints . . . 18

5.3 Express 18
5.3.1 Model 18
5.3.2 Database connection and initialization 19
5.3.3 Routing and endpoints 20
5.4 ASP.NET 21
5.5 Performance tests 21

6 Results **22**
6.1 Performance 22
6.2 Security 22

7 Conclusion **23**

Bibliography **23**

List of Figures **24**

List of Code Listings **25**

List of Algorithms **26**

Acronyms

API Application Programming Interface. 6, 7

ASGI Asynchronous Server Gateway Interface. 6

CRUD Create Retrieve Update Delete. 8, 13, 18

DRF Django Rest Framework. 5, 6, 18

HTTP Hypertext Transfer Protocol. 6

JSON JavaScript Object Notation. 18

MVC Model-View-Controller. 5, 7

NPM Node Package Manager. 6

PG PostgreSQL. 1, 5, 6, 7, 12, 16

SQL Structured Query Language. 18

VU Virtual User. 8

WSGI Web Server Gateway Interface. 6

Chapter 1

Introduction

Complex web applications keep becoming more popular in recent years. They are very easily accessible from any place in the world and can run on almost any modern device, as it requires only web browser and the Internet connection to run.

Web frameworks simplify the development process of web applications significantly improving developers productivity. There is a huge variety of choices between the mentioned software, so choosing one may be a difficult task.

The choice of server-side framework is crucial, as it is responsible for handling sensitive data and plays a key role in the overall performance of the application.

The goal of this document is to focus on three of the most popular server-side frameworks and compare their performance under high load as well as basic security measurements. The results of this thesis will be important for web developers and architects, that need to decide on which framework should they choose for their application.

Web frameworks for this comparison were chosen from the list of Stack Overflow Developer Survey 2020 Web Frameworks popularity [1]. The chosen frameworks (and their respective languages) are:

- Express.js (JavaScript)
- ASP.NET (C#)
- Django (Python)

Chapter 2 describes chosen frameworks, including their architecture and requirements. For the definition of the criteria against which the results will be measured, see Chapter 3. Chapter 4 presents design of the system - database models, application structure and environment preparation. Chapter 5 shows framework specific application implementation details. Results of the tests and comparison of the technologies can be found in chapter 6. For final conclusion of this document, see chapter 7.

Chapter 2

Technology description

2.1 PostgreSQL

For the Database Management System I chose PostgreSQL, which is the second most popular choice among database technologies

2.2 Django

2.2.1 Overview

Django is a high-level Python web framework, that allows programmers build dynamic, interesting sites in an extremely short time. Django is designed to let you focus on the fun, interesting parts of your job while easing the pain of the repetitive bits [2]. Additionally the framework is being supported by a wide variety of libraries and frameworks, like Django Celery, Crispy Forms, Django Rest Framework (DRF).

Django is based on MVC architecture:

- Model - a data structure, represented by a database
- View - interfaces visible in the browser
- Controller - connects Model and View together - describes how the data should be presented to the user

In Django application there are multiple files and at first it may not be obvious what their role is. Basic structure of an application looks like this:

- apps.py - common to all django apps configuration file
- models.py - custom models
- serializers.py - define how our model objects should be converted into response
- views.py - custom controllers, which is not intuitive for most of people; as the developers explain, in their interpretation of MVC the view describes which data gets presented to the user [3]
- urls.py - defines which endpoint responds to given controller

Templates, which are not mentioned above, are the Django's custom views - in case of application for my tests, it is going to be using build in json parsers.

2.2.2 Requirements and dependencies

Fortunately Django has a built in PostgreSQL engine, so no additional packages need to be added for this to work. However, for the sake of this research, a popular package Django Rest Framework package was added (over 21k stars on github), as it provides well thought set of base components for building APIs, often reducing the amount of code that needs to be written. Deploying django can be currently done by two interfaces:

- WSGI, which is the main Python standard for communicating between servers and applications
- and ASGI, which is new, asynchronous-friendly standard, allowing to use asynchronous Python features (and Django features as they are developed) **??**

Since the applications will be tested under a heavy load, the logical choice was the ASGI interface. As recommended in the documentation **??**, deploying an application with ASGI can be done using Uvicorn, which is a fast ASGI server implementation **??**, which results in two additional packages being added to the list of requirements.

2.3 ExpressJS

2.3.1 Overview

To describe what Express.js is, it would be appropriate to introduce Node.js. Node is an open-source runtime environment that runs on the same engine as the JavaScript in the browsers, but it allows developers to build server-side tools and applications. Node package manager (NPM) provides hundreds of thousands of packages, and that is where we can find Express.

Express is the most popular web framework with almost 17 million weekly downloads from NPM. It provides mechanisms to write handlers for HTTP requests, integrate with rendering engines and handle middlewares. The minimalism of the framework is compensated by huge supply of third party libraries, which are a giant expansion to it's functionality.

Express.js is an unopinionated framework, which means theres more than one "right" way to achieve a given result - there is no strict architecture that the developer has to follow.

2.3.2 Requirements

Application for the research in this thesis is very minimalistic itself, hence there are not many additional packages that need to be added. There is only one bundle downloaded from the NPM repository, which is pg-promise. Initially it was a package that expanded the base library - node-postgres by adding promises to the base driver, but since then the library's functionality was vastly expanded. The expanded version of the driver was chosen, because of it's popularity - at the day of accessing the libraries the number of weekly downloads of node-postgres (2k) was only a tiny fraction of pg-promise's (172k).

2.4 ASP.NET Core

2.4.1 Overview

ASP.NET Core is an open-source framework for building modern Internet-connected applications. With it you can build web applications and services, IoT apps, and mobile backends. "Core" is an a new, upgraded version of ASP.NET - it includes architectural changes, that result in more modular framework.

ASP.NET Core MVC is a framework optimized for use with ASP.NET Core, that allows to build APIs and web applications using Model-View-Controller pattern.

2.4.2 Requirements

To have the PostgreSQL support in the application, it is necessary to add PG provider for the framework - Npgsql.EntityFrameworkCore.PostgreSQL.

2.5 K6 and related tools

For testing the performance of applications, a tool named k6 was chosen. It is a modern load testing tool written in Golang, which provides clean and well documented APIs for writing and running tests, while still being easily configurable to the developers needs. Test logic and configuration options are both to be written in JavaScript, which allows developers for using JavaScript modules, which aids in code reusability. The creators of k6 prepared two types of execution:

- local, through command line interface
- and cloud, which is a commercial SaaS product, made to make performance testing in bigger applications easier.

For the sake of this experiment, local testing has fulfilled all expectations.

Installation on Ubuntu operating system is fairly simple and all necessary commands were described in the documentation. However, to make the testing simpler, a k6 Docker image was used, that together with Docker Compose allowed to create a single script that would handle all test cases as described in the following section.

K6 allows to create visualizations, using built-in InfluxDB and Grafana integration, where InfluxDB is used as storage backend and Grafana to visualize the data. In this research, only InfluxDB was added to store the data and after each test the data was exported to file, which later allowed to compare the results between applications on a single chart.

2.6 Docker and Docker Compose

Docker is a software that allows virtualization on an operating system level (containerization). Containers wrap the applications code in a small virtual environment based on the provided configuration. Docker and Docker Compose were used to simplify the development. This made starting all services that had be run together possible with only one command, eg. for django performance tests - django, postgres, k6 and influx. For every application a production ready Dockerfile was created. Additionally, Docker provides applications a layer of isolation from each other and the host.

Chapter 3

Criteria description

3.1 Performance benchmark

3.1.1 Scenarios

The task for each application is to complete simple CRUD operations as fast as possible. For comparison of the performance of the applications, a few test cases were developed:

- retrieving multiple objects (getMany)
- retrieving single object (get)
- updating a single object (put)
- creating a single object (post)
- deleting a single object (delete)

They were tested with a few different application loads, which are represented by a number of Virtual Users (VUs) - as mentioned in the k6 repository description they are glorified, parallel while(true) loops. The scenarios chosen for tests are:

- 1 VU
- 8 VUs
- 32 VUs
- 128 VUs
- 512 VUs

For a single virtual user case the number of concurrences does not change throughout the duration of the test, however, as suggested in , for bigger numbers of concurrent users, the tests should include warmup and cooldown period. All tests are 45 seconds long, and tests with more than 1 virtual user include 15 seconds of ramp up time and 15 seconds of ramp down time as shown on figure 3.1.

Longer test times did not bring any satisfactory results and only caused CPU throttling problems, thus they were shortened to the period of 45 total seconds, which also made comparison of the results much easier.

For one scenario the results could be slightly different, that is why every test was repeated 10 times. For creation of the graphs presented in chapter ?? for each test an average response times were calculated.

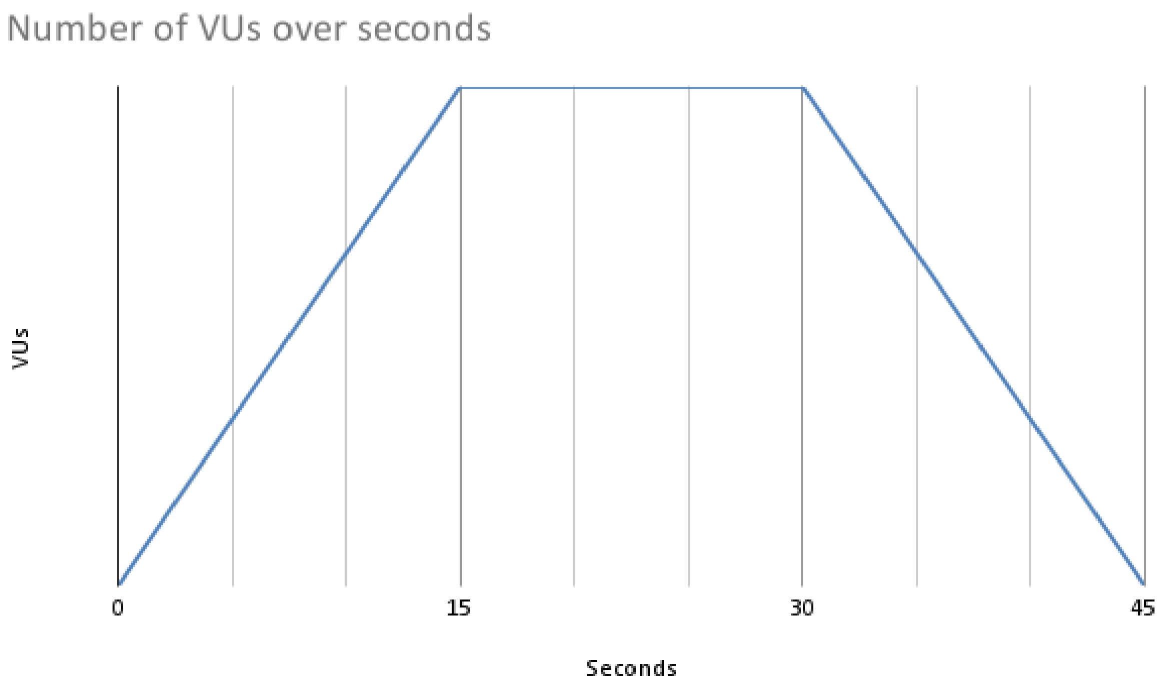

Figure 3.1: Amount of VUs per second during the tests

3.1.2 Data

For the performance tests some frameworks required initial data to exist in the database (for example PUT endpoint for editing database models), thus at the beginning of the tests for each framework the database is populated. To be sure that the data is consistent, after seeding the database the docker volume that keeps the data is being stored locally, and before each test it is being restored.

3.1.3 Application isolation

To be sure that the applications are running in an isolated environment, docker containers were used. The configuration prepared for the applications included environment preparation (installing necessary packages, providing environment variables), To simplify the research, a Docker Compose configuration was prepared, that builds and starts all the necessary containers at once.

3.1.4 Test progress

The main script prepared for this thesis, that gathers all the measurements is described in the algorithm 1.

Algorithm 1: Pseudocode describing load testing process

```
frameworks = [aspnet django express];
scenarios = [1 8 32 128 512];
iterations = [1 2 3 4 5 6 7 8 9 10];
test cases = [get post put delete getMany];
forall frameworks do
    populate database;
    store snapshot;
    forall scenarios do
        forall iterations do
            forall test cases do
                if application is running then
                    kill application;
                remove volumes;
                if test case is not post then
                    restore snapshot;
                start application;
                while application is not responding do
                    wait for application;
                for 45 seconds do
                    measured test;
                store k6 result ;
                store influxDB result ;
                for 20 seconds do
                    cooldown ;
merge results;
```

3.1.5 Software versions and hardware

The tests were run on a laptop with the specification presented in table 3.1. Frameworks used to build the application were in the versions presented in table 3.2.

Table 3.1: Hardware

Hardware	
Processor	Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz
RAM memory	16 GB @ 2400MT/s
Operating system	Ubuntu 20.04.2 LTS

3.2 Security

Table 3.2: Frameworks and libraries versions

Framework versions	
Django	3.1.4
Django REST Framework	3.12.2
gunicorn	20.0.4
uvicorn	0.13.1
ASP.NET	2.1.1
Npgsql.EntityFrameworkCore.PostgreSQL	2.1.1.1
Express	4.17.1
pg-promise	10.9.5

Chapter 4

System design

4.1 Database

For the tests, a database consisting of a single table was created - a model is presented in a table 4.1.

Table 4.1: Database model

User	
id	serial not null primary key
password	varchar(128) not null
username	varchar(30) not null
first_name	varchar(30) not null
last_name	varchar(30) not null
email	varchar(75) not null

For every application the operations on the PostgreSQL database may be slightly different, since in Django and ASP.NET they are handled by the Object-Relational Mapping libraries, while Express works on SQL statements.

Database creation is based on environment variables, as it makes it easier for connecting the applications to the database later. For the connection, the following variables need to be present:

- POSTGRES_DB, which is the name of the database, set to postgres,
- POSTGRES_USER, the default database user, set to postgres,
- POSTGRES_PASSWORD as the defaults' user password, set to postgres,
- POSTGRES_HOST required only for applications - it is the hostname where the database can be found - because of how the docker networks work, it needs to be set to the container name - postgres,
- POSTGRES_PORT which is the port on which the database is exposed - set to 5432

4.1.1 Initial population creation

For every application the script responsible for seeding the database looks basically the same, but for the different frameworks the piece of code had to be adjusted to three different languages. The amount of users is parameterized and can be changed on all applications by one variable in the main script, to keep the consistency between applications and avoid potential mistakes. Users are created in bulk in a single transaction to speed up the process.

Each framework (or library providing database support) has it's own built in methods for CRUD operations, and they may be slightly different from each other. For example, Express.js and ASP.NET Core framework automatically wrap statements in a transaction. ASP.NET in each transaction sets the transaction isolation level and uses prepared statements, which then are executed with required parameters. In listings 4.1 4.2 and 4.3 there are presented example log queries for initial user population creation.

```
LOG:  statement: BEGIN
LOG:  statement: SET TRANSACTION ISOLATION LEVEL READ COMMITTED
LOG: execute: INSERT INTO users (id, email, first_name, last_name,
    password, username) VALUES ($1, $2, $3, $4, $5, $6)
DETAIL: parameters: $1 = '1', $2 = 'First0@Last0.com', $3 = 'First0'
    , $4 = 'Last0', $5 = 'Pass0!', $6 = 'First0Last0'
LOG: execute: INSERT INTO users (id, email, first_name, last_name,
    password, username) VALUES ($1, $2, $3, $4, $5, $6)
DETAIL: parameters: $1 = '2', $2 = 'First1@Last1.com', $3 = 'First1'
    , $4 = 'Last1', $5 = 'Pass1!', $6 = 'First1Last1'
LOG:  statement: COMMIT
```

Listing 4.1: Log of ASP.NET initial user population creation

```
LOG:  statement: BEGIN
LOG:  statement: INSERT INTO "app_myuser" ("id", "password", "
    username", "first_name", "last_name", "email") VALUES (1, 'Pass0!
    ', 'First0Last0', 'First0', 'Last0', 'First0@Last0.com'), (2, '
    Pass1!', 'First1Last1', 'First1', 'Last1', 'First1@Last1.com')
    RETURNING "app_myuser"."id"
LOG:  statement: COMMIT
```

Listing 4.2: Log of Express.js initial user population creation

```
LOG:  statement: insert into "public"."users"("id","password","
    username","first_name","last_name","email") values('1','Pass0!','
    First0Last0','First0','Last0','First0@Last0.com'),('2','Pass1!','
    First1Last1','First1','Last1','First1@Last1.com')
```

Listing 4.3: Log of Django initial user population creation

4.2 Applications endpoints

For the load tests, applications were prepared with 6 endpoints:

- GET /status/
  - Description: required for the script, allows to check if the server has properly started and can properly respond to requests
  - Response code: 200
  - Response body: empty
- GET /users/{id}/

  - Description: retrieves user with given id from the database
  - Database operation: retrieve
  - Parameters:
    - id - path parameter, id of the user to be retrieved
  - Response code: 200
  - Response body: User model

- GET /users/?limit={limit}&offset={offset}
  - Description: retrieves multiple users - allows to check the frameworks serialization speed
  - Database operation: retrieve
  - Parameters:
    - limit - query parameter, amount of users returned
    - offset - query parameter, amount of rows to skip from the beginning
  - Response code: 200
  - Response body: User model array

- DELETE /users/{id}/
  - Description: Removes user with given id
  - Database operation: delete
  - Parameters:
    - id - path parameter, id of the user to be removed
  - Response code: 204
  - Response body: empty

- POST /users/
  - Description: Creates user from details provided in body
  - Database operation: create
  - Request body: User model to be created
  - Response code: 201
  - Response body: User model

- PUT /users/{id}/
  - Description: Updates user with given id from details provided in body
  - Database operation: update
  - Request body: User model to be created
  - Parameters:
    - id - path parameter, id of the user to be removed
  - Response code: 201
  - Response body: User model

4.3 Environment

As mentioned in the previous chapters, a script that gathers all the measurements was prepared, which follows the schema presented in algorithm 1.

Chapter 5

Implementation

5.1 Docker and Docker Compose

The Dockerfile that I prepared was inspired by Anuj Sharma, who created one for his series of Django development guide articles. It uses the python:3.9.1-slim image, installs all necessary packages, places the code in suitable folders, changes the user to a non-root user and sets the entrypoint to a script that starts the application. Docker Compose configuration contains the build folder, imports the environment variables for connection with PG and variable with amount of users to create, ensures that the application starts after the database has started (makes the app wait for PostgreSQL healthcheck) and reserves memory for the application.

5.2 Django

5.2.1 Model

Django offers a built in User model, however it was decided not to use it in this case. The reason for that is that the built in model handles extra operations for the User, like creating groups, permissions, authentication and a few additional fields. Instead of using it, the implementation of model presented in listing 5.1 was created. It does not contain the primary key definition, as django.db.models.Model class handles it automatically.

```python
class MyUser(models.Model):
    password = models.CharField(max_length=128)
    username = models.CharField(max_length=30)
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=30)
    email = models.CharField(max_length=75)
```

Listing 5.1: Django user model

5.2.2 Database connection and initialization

Django handles a lot of things for the user - connection is very simple - in initial project generation a file settings.py is created, which consists of all the necessary variables for the system to work. In the file we can find a variable named Databases, initially with SQLite backend. All that needs to be done to connect to our PostgreSQL is to change the engine to built-in PG backend and change the remaining fields - name, user, password, host and port, as presented in listing 5.2. With that done, the connection is automatically done on the system startup.

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': os.environ.get('POSTGRES_DB', 'postgres'),
        'USER': os.environ.get('POSTGRES_USER', 'postgres'),
        'PASSWORD': os.environ.get('POSTGRES_PASSWORD', 'postgres'),
        'HOST': os.environ.get('POSTGRES_HOST', 'postgres'),
        'PORT': os.environ.get('POSTGRES_PORT', 5432),
    }
}
```

Listing 5.2: Django database connection object, fragment of settings.py file

Creating the table is handled by migration system - to create the migrations the command from listing 5.3 had to be run.

```
python3 manage.py makemigrations
```

Listing 5.3: Django migration creation command

This creates the tables based on the model presented in models.py files through the whole project. In this case, it only created one table. For creating the initial population a management function was prepared, that is being executed from the main script. Seeding the database was presented in listing 5.4.

```
amount = int(os.environ.get("USER_AMOUNT"))
users = []
for id in range(amount):
    first_name = f"First{id}"
    last_name = f"Last{id}"
    user = MyUser(
        id=id+1,
        username=first_name+last_name,
        first_name=first_name,
        last_name=last_name,
        email=f"{first_name}@{last_name}.com",
        password=f"Pass{id}!"
    )
    users.append(user)
MyUser.objects.bulk_create(users)
```

Listing 5.4: Populating django DB

5.2.3 Routing and serialization

Django routing is to be placed in urls.py files. There is one main file in the project configuration folder, and one in each module. Since this application consists of only one modules, two urls.py files exist - presented in listings 5.5 and 5.6.

```
urlpatterns = [
    path('', include('app.urls'))
]
```

Listing 5.5: Fragment of Django route configuration file - config/urls.py

```
from .views import UserViewSet, status
```

```
router = routers.DefaultRouter()
router.register(r'users', UserViewSet)

urlpatterns = [
    path('', include(router.urls)),
    path('status', status),
]
```

Listing 5.6: Fragment of Django route configuration file - app/urls.py

Serialization is another great thing about Django and Django Rest Framework - DRF has built in abstract ModelSerializer class - to create a serializer for our model, all that needs to be done is to specify which model and which fields we want to serialize, as presented in listing 5.7.

```
class UserSerializer(serializers.ModelSerializer):
    class Meta:
        model = MyUser
        fields = ['id', 'username', 'email', 'first_name', '
           last_name', 'password']
```

Listing 5.7: Django User serialization class

5.2.4 Endpoints

It is pretty certain at this point that django offers a lot of functionality. It should come with no surprise that the endpoints can also be implemented with a few lines of code, thanks to the built in methods. As shown in the listing 5.8, creating views does not require much, only the queryset containing all user models and a serializer class. With this ViewSet and one standalone function we get all the endpoints described in section 4.2 of this document. UserViewSet class was used in the routing in file presented in listing 5.6.

```
class UserViewSet(viewsets.ModelViewSet):
    queryset = MyUser.objects.all()
    serializer_class = UserSerializer

@api_view(['GET'])
def status(r):
    return Response()
```

Listing 5.8: Controllers for django endpoints

5.3 Express

5.3.1 Model

User model in express.js needs to contain all the necessary logic for handling CRUD operations. Implementation of the model is shown in listing 5.9 - it shows all SQL statements except include, which is placed in separate file. Database queries return JSON object that is ready to be

```
class UsersModel {
  constructor(db, pgp) {
    this.db = db;
    this.pgp = pgp;
    createColumnsets(pgp);
  }

  async insert(user) {
    return this.db.one(sql.insert, user);
  }

  async update(fields, id) {
    const updateFields = ‘(${Object.keys(fields)}) = ROW(${
      Object.values(fields).map((value) => ‘’${value}’‘)})‘;

    return this.db.one(
      "UPDATE users SET $1^ WHERE id = $2 RETURNING *",
      [updateFields, +id]
    );
  }

  async delete(id) {
    return this.db.any("DELETE FROM users WHERE id = $1", [+id]);
  }

  async retrieve(id) {
    return this.db.oneOrNone(
      "SELECT * FROM users WHERE id = $1",
      [+id]
    );
  }

  async get(limit, offset) {
    return this.db.multi(
      "SELECT * FROM users LIMIT $1 OFFSET $2",
      [+limit, +offset]
    );
  }
}
```

Listing 5.9: Express.js user model

5.3.2 Database connection and initialization

Work with express is a bit more difficult, as most of the configuration needs to be done by the user. To connect with the database, a postgres promise instance needs to be created. Because there is no migration system, creating the table also needs to be done manually. To do so, a database configuration file was created (listing 5.10) and imported into the entrypoint file. It contains all the necessary information about the connection and creation of the table.

```
const DBCONFIG = {
    host: process.env.POSTGRES_HOST,
    password: process.env.POSTGRES_PASSWORD,
    database: process.env.POSTGRES_DB,
    user: process.env.POSTGRES_USER,
    port: process.env.POSTGRES_PORT,
};

const initOptions = {
    extend(obj, dc) {
            obj.users = new Users(obj, pgp);
        }
    };
    const pgp = pgPromise(initOptions);
    const db = pgp(DBCONFIG);

    (async () => { await db.users.createTable(); })();

    module.exports = { db, pgp };
```

Listing 5.10: Express.js database connection

Initialization of the database is handled by function presented in listing 5.11.

```
const users = [];
for (var id = 0; id < USER_AMOUNT; id++) {
  const first_name = `First${id}`;
  const last_name = `Last${id}`;
  const user = {
    id: `${id + 1}`,
    first_name,
    last_name,
    username: first_name + last_name,
    email: `${first_name}@${last_name}.com`,
    password: `Pass${id}!`,
  };
  users.push(user);
}
const query = this.pgp.helpers.insert(users, cs.insert);
await this.db.none(query);
```

Listing 5.11: Populating Express.js DB

5.3.3 Routing and endpoints

Unlike in Django, routes and endpoints are not separated in two files. It is very common to keep them in a single file, as shown in listing 5.12. All they do is registering a route on the provided path in the first argument and executing the functions provided in the second argument. The mentioned functions call the respective method from User model sends appropriate response based on the return data of the query.

```
app.get("/status", (req, res) => res.sendStatus(200));

app.get("/users/:id", async (req, res) => {
    const data = await db.users.retrieve(req.params.id);
    return data ? res.json(data) : res.sendStatus(400);
});

app.get("/users", async (req, res) => {
    const limit = req.query.limit;
    const offset = req.query.offset;
    const data = await db.users.get(limit, offset);
    return data ? res.json(data) : res.sendStatus(400);
});

app.post("/users", async (req, res) => {
    const data = await db.users.insert(req.body);
    return data ? res.status(201).json(data) : res.sendStatus(400);
});
```

```
app.delete("/users/:id", async (req, res) => {
    await db.users.delete(req.params.id);
    return res.sendStatus(204);
});

app.put("/users/:id", async (req, res) => {
    const data = await db.users.update(req.body, req.params.id);
    return data ? res.json(data) : res.sendStatus(400);
});
```

Listing 5.12: Express.js routing and endpoint logic

5.4 ASP.NET

5.5 Performance tests

Chapter 6

Results

6.1 Performance

Table 6.1 shows an average of 95th percentile from the 10 tests mentioned in the previous chapters.

Table 6.1: Average p(95) response time in tests

AVERAGE z p(95)		filename		
test	concurrency	aspnet	django	express
delete	1	8.35	48.36	4.14
	8	9.75	143.84	10.94
	32	48.84	385.77	32.68
	128	192.80	1259.32	118.67
	512	675.12	4923.61	458.99
get	1	1.37	12.54	1.00
	8	6.42	54.46	5.98
	32	19.89	189.95	20.05
	128	84.64	643.71	79.42
	512	309.29	2605.04	321.64
getMany	1	48.16	56.93	24.14
	8	76.09	344.77	42.59
	32	177.36	1280.14	73.80
	128	325.66	4831.64	159.44
	512	683.23	19525.67	511.82
patch	1	9.13	45.58	5.20
	8	10.53	129.42	11.65
	32	50.86	358.92	35.65
	128	198.23	1237.40	130.69
	512	655.22	4806.19	501.05
post	1	6.73	40.73	5.13
	8	7.32	122.30	10.37
	32	42.36	321.98	29.24
	128	171.97	1069.47	120.47
	512	611.54	4185.22	407.89
put	1	9.12	45.20	5.26
	8	10.49	132.10	11.73
	32	51.82	359.56	34.78
	128	202.52	1236.24	129.99
	512	735.95	4817.58	507.11

6.2 Security

Chapter 7

Conclusion

Bibliography

[1] “Stack overflow 2020 developer survey.” https://insights.stackoverflow.com/survey/2020#technology-web-frameworks. Accessed: 2021-03-14.

[2] J. K.-M. Adrian Holovaty, “The django book,” 2007.

[3] “Django documentation - faq: General.” https://docs.djangoproject.com/en/1.11/faq/general/#django-appears-to-be-a-mvc-framework-but-you-call-the-controller-the-view-and-the-view-the-template-how-come-you-don-t-use-the-standard-names. Accessed: 2021-03-14.

List of Figures

3.1 Amount of VUs per second during the tests 9

List of Code Listings

4.1 Log of ASP.NET initial user population creation 13
4.2 Log of Express.js initial user population creation 13
4.3 Log of Django initial user population creation 13
5.1 Django user model 16
5.2 Django database connection object, fragment of settings.py file 17
5.3 Django migration creation command 17
5.4 Populating django DB 17
5.5 Fragment of Django route configuration file - config/urls.py 17
5.6 Fragment of Django route configuration file - app/urls.py 17
5.7 Django User serialization class 18
5.8 Controllers for django endpoints 18
5.9 Express.js user model 18
5.10 Express.js database connection 19
5.11 Populating Express.js DB 20
5.12 Express.js routing and endpoint logic 20

List of Algorithms

1 Pseudocode describing load testing process 10