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

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.

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 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 ExpressJS 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.

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

2.4.1 Overview

2.4.2 Architecture

2.4.3 Requirements

2.5 K6 and related tools

For testing the performance of applications, I chose a tool named k6. 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 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 scenarios 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)

Scenarios 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 numbers 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 valuable information and only brought CPU overheating problems, thus they were shortened, which also made work with the results much easier.

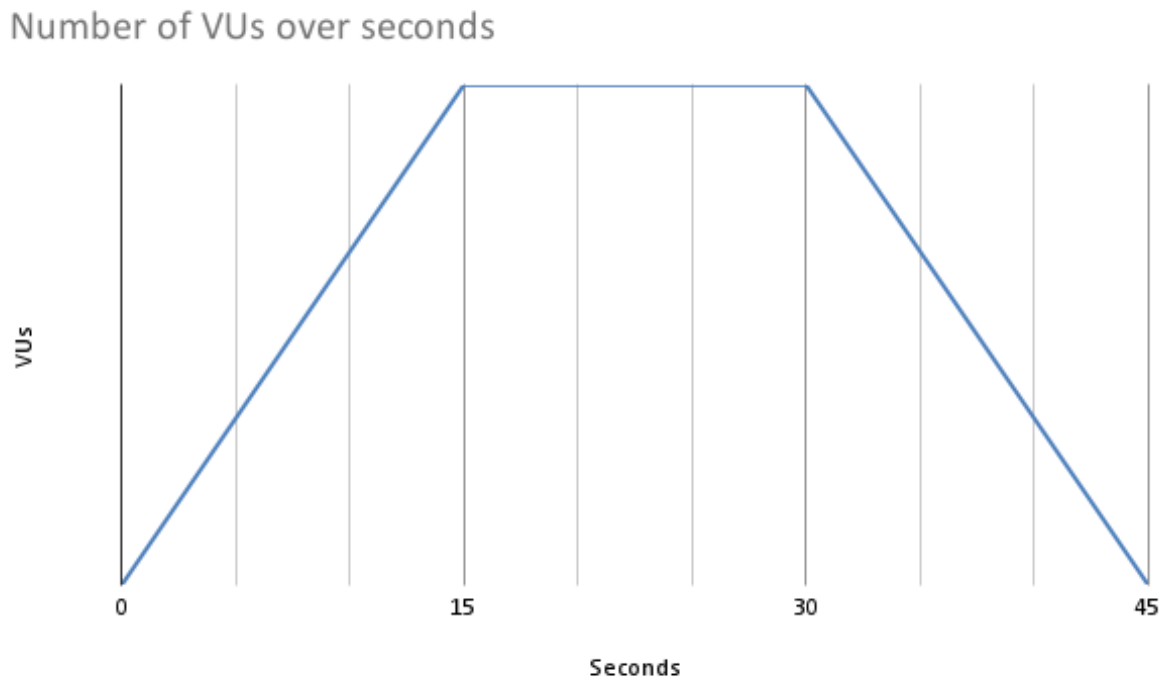

Figure 3.1: Amount of VUs per second during the tests

3.1.2 Database snapshots

To avoid any differences in the database between the tests, at the beginning of the tests the database is populated and the snapshot is stored locally. Before each test, the snapshot is 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 measures are gathered from each application in the following manner:

```
populate database;
store snapshot;
foreach test case 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 5 seconds do
        warmup requests;
    for 45 seconds do
        measured test;
    store result;
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

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

3.2 Security

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	integer
username*	string
email*	string
first_name*	string
last_name*	string
password*	string

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.

4.1.1 Database connection and initiation

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. With that done, the connection is automatically done on the system startup.

4.1.2 Initial population creation

4.1.3 Operations on users

4.2 Application

4.3 Environment

Chapter 5

Application implementation

5.1 Django

5.2 Express

5.3 ASP.NET

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

List of Code Listings

List of Pseudocodes