.NET Core 3.1 & .NET 5

Performance benchmarking in Web API use

Tero Hyttinen

Bachelor's thesis
May 2021
Information and Communication Technologies
Bachelor's Degree Programme in Information and Communication Technology

jamk | Jyväskylän ammattikorkeakoulu University of Applied Sciences

Hyttinen Tero

.NET Core 3.1 & .NET 5, Performance benchmarking in Web API use

Jyväskylä: JAMK University of Applied Sciences, May 2021, 34 pages.

Information and Communication Technologies. Bachelor's Degree Programme in Information and Communication Technology.

Permission for web publication: Yes

Language of publication: English

Abstract

The aim of this study was to compare the performance of two Microsoft .NET (Core) product versions. Previous version upgrades to .NET (Core) had seen performance improvements over their preceding version. The need for performance assessment arose from software company RockOn's software project, during which a new version of the used .NET (Core) product was released. It was argued should the project switch to use the newer software version.

The main task was to gather performance data of the company's used software platform with the then current version and with the upgraded and version. To accomplish the task, quantitative research method was used to gather performance data of the software using two different software testing tools. Software tests were divided in to two separate sections. Practical web API performance was tested with load testing tool on the company's produced software application. Non-practical code level tests were done on a separate software application.

The load test result for the application web API performance saw 160 % speed reduction for the new software version due to software application irregularities. The code level performance saw increase for the new version from 0,76% to 94,63%.

By analyzing the results it was concluded that the new .NET version had performance benefits over the older .NET version. The application anomalies and inconsistent load test data lead to deem the load test results as unreliable while the code level test results proved to be in line with findings by other data and as such were regarded as reliable.

Keywords/tags (subjects)

Microsoft .NET, performance testing, performance, software testing, benchmarking

Miscellaneous (Confidential information)

Kuvailulehti

Hyttinen Tero

.NET Core 3.1 & .NET 5, Suorituskykyvertailu Web API käytössä

Jyväskylä: Jyväskylän Ammattikorkeakoulu, toukokuu 2021, 34 sivua.

Tieto- ja viestintätekniikka. Insinööri (AMK), tieto- ja viestintätekniikka.

Verkkojulkaisulupa myönnetty: Kyllä

Julkaisun kieli: Englanti

Tiivistelmä

Opinnäytetyön tavoitteena oli verrata suorituskykyä kahden Microsoftin .NET (Core) version välillä. Aikaisemmat .NET (Core) versiopäivitykset olivat tehneet ohjelmistoalustaan suorituskykyparannuksia. Tarve selvitystyölle lähti ohjelmistoyritys RockOn Oy:n ohjelmistoprojektista. Projektin aikana Microsoft julkaisi uuden version projektissa käytetystä ohjelmistosta. Tarvittiin selvitys olisiko uuden version tuoma oletettu suorituskykyhyöty riittävä projektin ohjelmiston siirtämiseksi uudelle versiolle.

Tehtävänä oli tuottaa suorituskykydataa käytetystä ohjelmistoalustan silloisesta versiosta sekä ohjelmistoalustan uudesta versiosta. Suorituskykydataa tuotettiin käyttämällä kvantitatiivisia menetelmiä hyödyntämällä kahta eri ohjelmistotestausohjelmaa. Ohjelmistotestit jaettiin kahteen itsenäiseen osioon. Käytännön verkkosovelluksen käyttöliittymän suorituskykyä testattiin RockOn:n tuottamaan applikaatioon käyttämällä kuormitustestausohjelmistoa. Yleistä kooditason suorituskykyä testattiin erillisellä applikaatiolla.

Verkkosovelluksen käyttöliittymän kuormitustestauksessa havaitut epäsäännöllisyydet johtivat uuden version 160 % hitaampaan suorituskykyyn. Yleinen kooditason suorituskyky oli mittauksissa 0,76% - 94,63% nopeampaa uudessa versiossa.

Saaduista tuloksista voitiin päätellä suorituskyvyn nousseen .NET:n uudessa versiossa. Kuormitustestaustulosten epäsäännöllisyydet johtivat tulosten luotettavuuden kyseenalaistamiseen ja tulosten hylkäämisen. Kooditason suorituskykytestien tulokset olivat linjassa ulkopuolisen testitulosten kanssa ja siten niitä voi pitää luotettavana.

Avainsanat (asiasanat)

Microsoft .NET, suorituskykytestaus, suorituskyky, ohjelmistotestaus, testaus

Muut tiedot (Salassa pidettävät liitteet)

Contents

Figures

Tables

1 Introduction

The aim of this work was to provide comparative performance data of two versions of Microsoft 's open-source software platform .NET. Versions tested in this work were .NET Core 3.1 and .NET 5. Thesis contractor required data about the performance of the specified versions of the software. Collected data would be analyzed and used in evaluating their software architectural choices.

The open-source .NET is a free, cross-platform software application building platform. As a software platform .NET is widely liked and popular (Ramel, 2019) with over "18%" (Datanyze, n.d.) of market share in server software space powering servers worldwide. Systems running .NET Core 3.1 versions would potentially have beneficial performance increases by upgrading their system version to .NET 5. By using .NET 5 version developers and service providers could produce more user-friendly applications and services.

System performance is one of the key attributes used to evaluate user-experience of a software application (Mifsud, n.d.), therefore software applications that are more performant than their counterparts, have a greater success in achieving better overall usability.

.NET 5 is Microsoft's strategic move in unifying .NET platforms and is the successor to .NET Core 3.1 and the proprietary .NET Frameworks. The shift to move all .NET products under one product name clarifies ".NET Ecosystem confusion" (Luijbregts, 2018).

The number of people using .NET Core 3.1 applications is unknown but is estimated to be significant, justified by the market share data reported by Datanyze (n.d.), this coupled with .NET Core 3.1 being a Long Term Support (later LTS) version which end of support date stated by Microsoft (n.d.) is December 3, 2022. Due to the longer software support times, LTS versions are generally regarded as preferrable versions to base software projects on, while non-LTS version updates can make significant improvements over LTS versions, many are hesitant to upgrade their system versions.

Using quantitative research method, it was show in the conducted performance measurements, that in selected code level test workloads .NET 5 is from 0,7% up to 94% faster than .NET Core 3.1.

2 Method

2.1 Performance & testing

Defining the meaning of performance is varied depending on the source. Definition by Cambridge dictionary (n.d.) describe performance as “how well a person, machine, etc. does a piece of work or an activity”. This definition leaves a lot to be desired for in clarity. At minimum two sub-definitions are needed to understand what performance is.

Definition for “well” is needed. “Well” or simply “good” is something positive depending on the viewpoint and is usually the desired outcome. To have an understanding how to conclude something as “well” the negative of “well” must be somewhat known. The opposite of “well” defining word could be “bad” or the phrasing “not well” and is usually the undesirable outcome dependent on the viewpoint.

Good and bad are highly relative terms which imposes a problem to performance measurement process. Every observer has their own relative viewpoint, resulting in multiple different performance results of any judged subject. Relativity of observers is ruled out by selecting a single agreed viewpoint. This viewpoint sets a framing in which are defined attributes in how performance is judged. Defining attributes removes subjectivity of the observation and sets an objective viewpoint to the measurements.

To get meaningful performance data, the object, metrics, measured data, methods and environment has to be defined. Clearly defined object answers the question what the object being measured is. Measured data is a selected property or properties from the object’s properties available for observation. Observation methods selected, can have an effect in how the measurements are conducted. Assessing observing methods impact on the measured system is useful in selecting the most useful methods of measurement. Environment has a direct impact on the resulting data and therefore consideration is to be upheld analyzing results. This process identifies aspects from the measured object, while leaving other possible factors outside.

Judging any system performance, outside affecting factors has to be assessed. Outside factors can be direct or indirect. Assessing outside factors, factors can be further identified by dividing them into factors that are alterable and to those are not. Out of those further assessment can be made by estimating how much of an impact the factors have on the measured system. Impactful factors are to be considered in the identification process.

Once the viewpoint is set and quantifiable meaningful data can be resulted with stable measurement methods, one can start assessing the performance of a system. A result observed in an isolation does not produce interesting analysis of any system. An arbitrary result can not be said to be neither “good” or “bad” if there is no scale to place results. Justification for performance testing results can be made to either serve as a baseline data or if pre-existing is available for comparison. Though comparing results with different environments may provide an indication of the performance of a measured system, conclusions might not lead to expected outcomes.

This underlines the relativity of performance testing as a system assessment tool. Results in a certain environment applies to tested environment only.

2.2 Testing tools

2.2.1 Selection process

Tools selected for this work were Apache JMeter and BenchmarkDotNet. Some time was used to research online sources for different testing tools available. Load testing tools suitable for performance testing purposes are varied and somewhat plentiful. In researching the toolkit, tools were found with both free open-source and commercial licenses.

Microsoft (2019) lists 10 load testing tools in their ASP.NET documentation pages. Instead of doing our own tool comparison, pre-existing tool reviews can be found online. Site like Baeldung.com has a load testing tool comparison article by Doyle (2021) where some load testing tools were compared and rated for points. JMeter was one of the rated tools and received a good score.

Code level benchmarking tools on the other hand are scarce, although some alternatives to BenchmarkDotNet were found. Tools like Perfx or NBench could have been used, but strong positive

sentiment towards BenchmarkDotNet swayed the selection for it.
Main aspects behind choosing these tools were non-commercial licensing, ease of use, online tool reviews and availability of tool specific additional material.

2.2.2 Apache JMeter

JMeter, figure 1, is a multi-purpose cross-platform open-source software testing tool. The software has a GUI for test building purposes and supports As a Java application JMeter can be run on any platform that supports Java. Current requirement for Java support is Java 8 or higher.

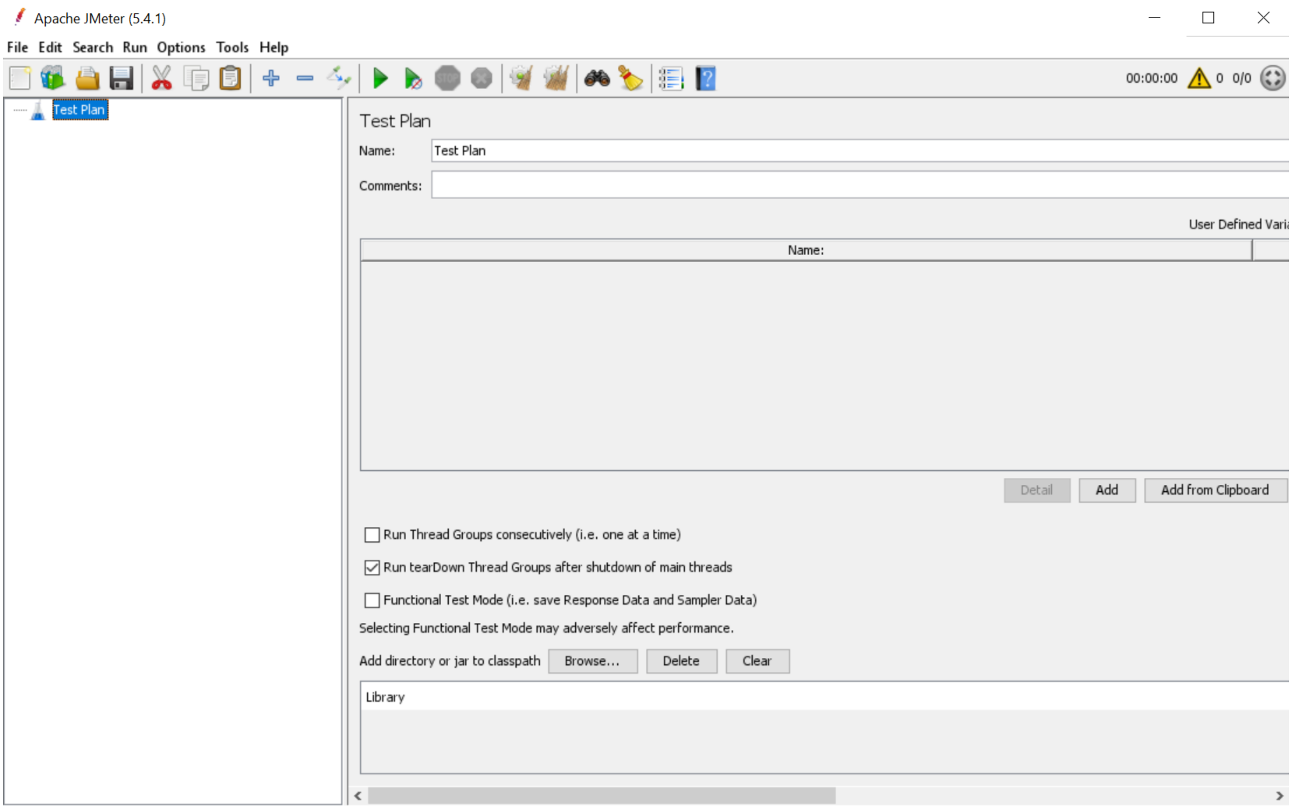


Figure 1. Test tool JMeter.

2.2.3 BenchmarkDotNet

For code level performance testing BenchmarkDotNet version 0.12.1 was used. Toub (2020) describes BenchmarkDotNet as a “canonical” tool in .NET testing. BenchmarkDotNet current 0.12.1 version listed in Microsoft’s package manager Nuget (n.d.) has nearly 1.8 million downloads. Figure 2 displays the BenchmarkDotNet logo.

Figure 2. Test tool BenchmarkDotNet logo.

2.3 Environment

2.3.1 Measured system

The system tested serves Application Programming Interface (later API) endpoints for automated data integrations and uses Model-View-Controller (later MVC) for the user interface (later UI). For data accessing Entity Framework Core (later EF Core) is used. As the database provider for the system, in-memory provider was used. In-memory database is created every time the application is started. Using in-memory database also reduces the outer systems effects on the testing results.

2.3.2 Software and .NET

Figure 3 shows the test platform's .NET information. Runtime for .NET Core 3.1 3.1.13. Runtime for .NET 5 5.0.4. .NET SDK version 5.0.201. Operating system Windows 10 Pro 20H2 version 10.0.1904. Visual Studio 2019 Community version 16.9.3 was used as the development platform. Git BASH a GNU bash, version 4.4.23(1) was used to execute application builds.

```
C:\Users\teroh>dotnet --info
.NET SDK (reflecting any global.json):
 Version:   5.0.201
 Commit:    a09bd5c86c

Runtime Environment:
 OS Name:     Windows
 OS Version:  10.0.19042
 OS Platform: Windows
 RID:         win10-x64
 Base Path:   C:\Program Files\dotnet\sdk\5.0.201\

Host (useful for support):
  Version: 5.0.4
  Commit:  f27d337295

.NET SDKs installed:
  5.0.201 [C:\Program Files\dotnet\sdk]

.NET runtimes installed:
  Microsoft.AspNetCore.All 2.1.26 [C:\Program Files\dotnet\shared\Microsoft.AspNetCore.All]
  Microsoft.AspNetCore.App 2.1.26 [C:\Program Files\dotnet\shared\Microsoft.AspNetCore.App]
  Microsoft.AspNetCore.App 3.1.13 [C:\Program Files\dotnet\shared\Microsoft.AspNetCore.App]
  Microsoft.AspNetCore.App 5.0.4 [C:\Program Files\dotnet\shared\Microsoft.AspNetCore.App]
  Microsoft.NETCore.App 2.1.26 [C:\Program Files\dotnet\shared\Microsoft.NETCore.App]
  Microsoft.NETCore.App 3.1.13 [C:\Program Files\dotnet\shared\Microsoft.NETCore.App]
  Microsoft.NETCore.App 5.0.4 [C:\Program Files\dotnet\shared\Microsoft.NETCore.App]
  Microsoft.WindowsDesktop.App 3.1.13 [C:\Program Files\dotnet\shared\Microsoft.WindowsDeskto
  Microsoft.WindowsDesktop.App 5.0.4 [C:\Program Files\dotnet\shared\Microsoft.WindowsDesktop

To install additional .NET runtimes or SDKs:
  https://aka.ms/dotnet-download

C:\Users\teroh>_
```

Figure 3. Dotnet info.

2.3.3 Hardware

The hardware for was provided by the thesis contractor. Dell 5450 laptop with 32Gb of random accessed memory (later RAM), Intel Core i7-9850H central processing unit (later CPU) running at 2.60GHz and 512GB Non-Volatile Memory Express (later NVME) hard drive. The system was at all times plugged in in the power outlet and ‘Power mode’-was selected in the battery options.

2.4 Test plan and tests

2.4.1 Test plan

The scope of this work set a clear framing for the test plan. The plan consisted testing the two versions, gather data and analyze the results. The metrics selected for the measurements was the processing speed of the selected tasks. Tasks would result in millisecond (ms) and nanosecond (ns) times being recorded. Test were divided into two categories: practical and non-practical. Practical in this context meaning testing the application’s main purpose, web API performance testing. Non-practical tests test more general workload, but which are relevant processes in the application.

Two parts of the system were identified as points of interest. Practical testing targeted the main section, the applications API-layer. Non-practical tests targeted the secondary subjects, the mocked ApiService.cs class and the more general workloads: string pattern matching and serialization/deserialization using two different serializers.

2.4.2 Practical tests

Load and stress tests can be used for example in evaluating software applications operating capacity, sufficiency of infrastructure, peak user load sustainability, maximum concurrent user, and scalability. Load testing in this work context was used to find out a baseline for request-response times of the application.

Practical load tests targeted the applications API-layer with the load testing tool JMeter. The API-layer is of great interest in performance testing in an API centered application. The system API-layer serves clients with API-endpoints to data accessing. Targeting testing to the API-layer provides meaningful data of the system applications main purpose performance. Load testing the API-endpoint with the tool gives data of the overall performance of the system regarding the main function of the application. JMeter measures request-response times of HTTP requests send to the application.

2.4.3 Non-practical tests

To test more generalized workloads, a few workloads were identified in the application data processes which were selected to testing. To test the selected workloads, the methods were ported and tested in a separate .NET project. This would give more isolated data about the workloads processed by the system that could then be compared against the two versions.

Workloads tested in this manner were: serializing and deserializing data objects, string pattern matching and mocking the applications ApiService class. Serializing and deserializing is a process of making a string representation of data model object and vice versa. String pattern matching is task of comparing a defined string pattern from a string data. Mock ApiService test tests fetching the data from the main applications API-endpoint /api/addresses using a HTTP client, and deserializing the data.

2.5 Test environment

2.5.1 Tested applications

To get meaningful data out of the tests, debugging code used by the developing software is not wanted or needed in performance tests. To achieve debug codeless code for testing, release builds of the application and the separate benchmarking project were built using Visual Studios default release build configurations. Aside from removing the debugging code, the compiler can make significant alterations to the produced machine code by evaluating the code resulting in more optimized software applications.

2.5.2 Operating system

Normal operation by the testing platform is referred to as noise. This system noise can have an affect to the testing process. Noise generated by the system is unavoidable and steady baseline noise is usually not a problem. Unsteady noise spikes on the other hand can drastically alter test result data. When recording milli- or nanosecond resolution timeframes, small test platform noise anomalies can cause big variations to the resulting data. To minimize this test platform noise, non-test critical applications were shutdown. Normal operating system idling background applications were left running.

2.5.3 JMeter test build

Figures 4 to 12 show the JMeter test built to test the applications endpoint1 /api/companies. Figure 4 shows the opened test plan named benchmark5.jmx. Test build for endpoint2 /api/addresses is the same in structure but differs in HTTP requests count, target url and sent HTTP POST json object.

In the test plan are two thread groups. The “setup Thread Group” is a setup type thread group ensuring it will be executed before the regular thread group. HTTP POST request were sent first to the application to insert data to the application database which would then be ready to be requested by HTTP GET request.

As per JMeter best practices listed in the tool’s user manual section 16.7 (JMeter, n.d.), reducing tool resource usage the “View Results Tree” listeners display name is greyed meaning it is set in disabled state and therefore is not run during actual test run. While useful in test building phase “view results” listeners should not be used during the load test.

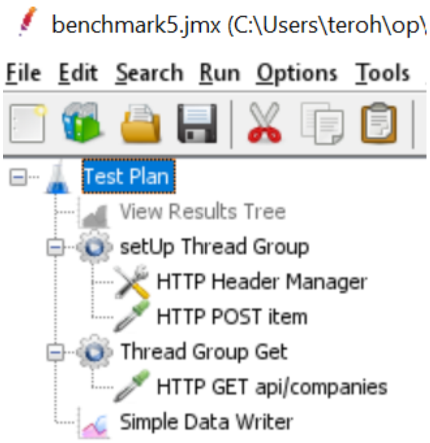


Figure 4. JMeter test plan.

Thread group settings defines the requests JMeter will execute in the thread groups HTTP request. Figure 5 shows the thread groups configuration with thread count set to 40 and the loop count to 10, in total of 400 requests will be executed by the thread group. Inside the thread group are added a Config Element HTTP Header Manager “HTTP Header Manager” and a Sampler type HTTP Request element named “HTTP POST item”.

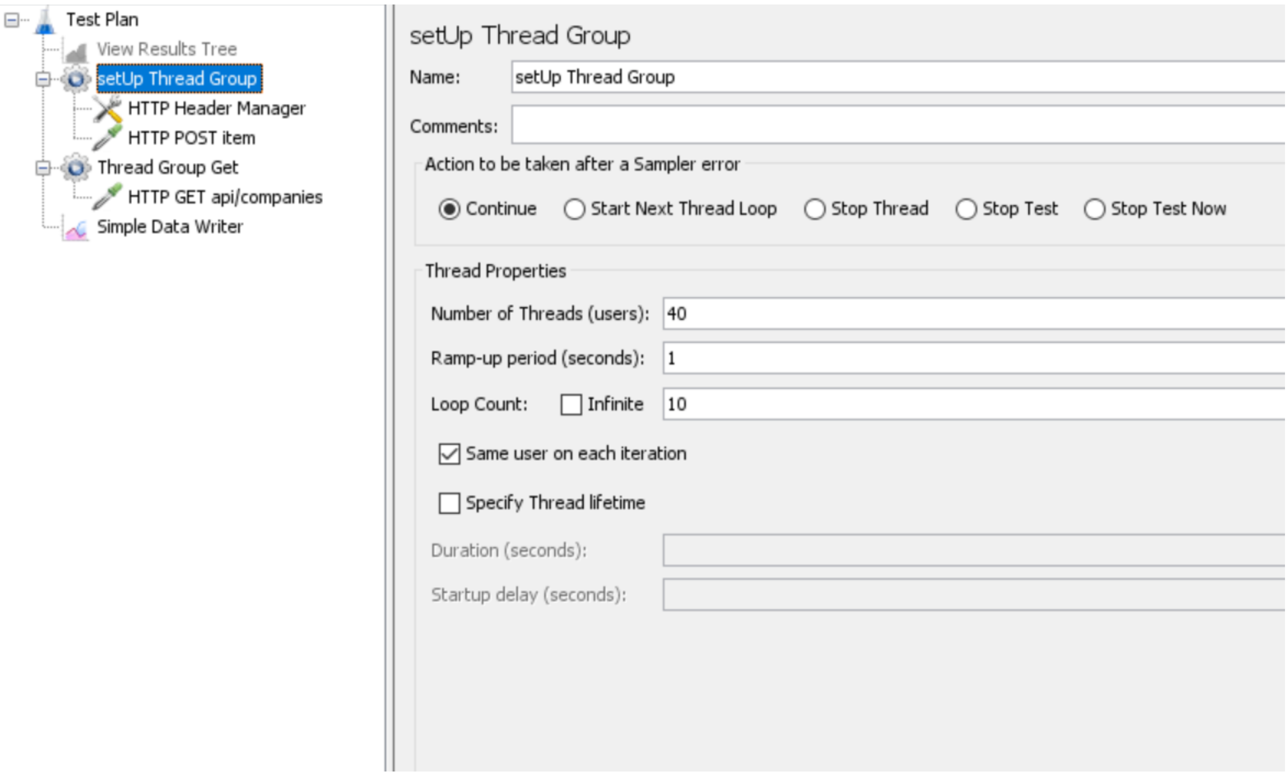


Figure 5. JMeter setUp Thread Group.

Figure 6 shows example header managers settings. One header named "Content-Type" with value "application/json-simple" is added. Headers configured here are used by the thread groups HTTP request elements.

Figure 6. JMeter Header Manager.

HTTP request sampler in the setup thread group has the required settings for the HTTP request shown in figure 7. Protocol, server name, port number, HTTP Request and path configures the request as a POST type request targeting local environment in port 5001. Below in the Body Data tab is the Javascript Object Notation (later JSON) data structure. JSON object depicts a complex type object sent in the HTTP POST request body.

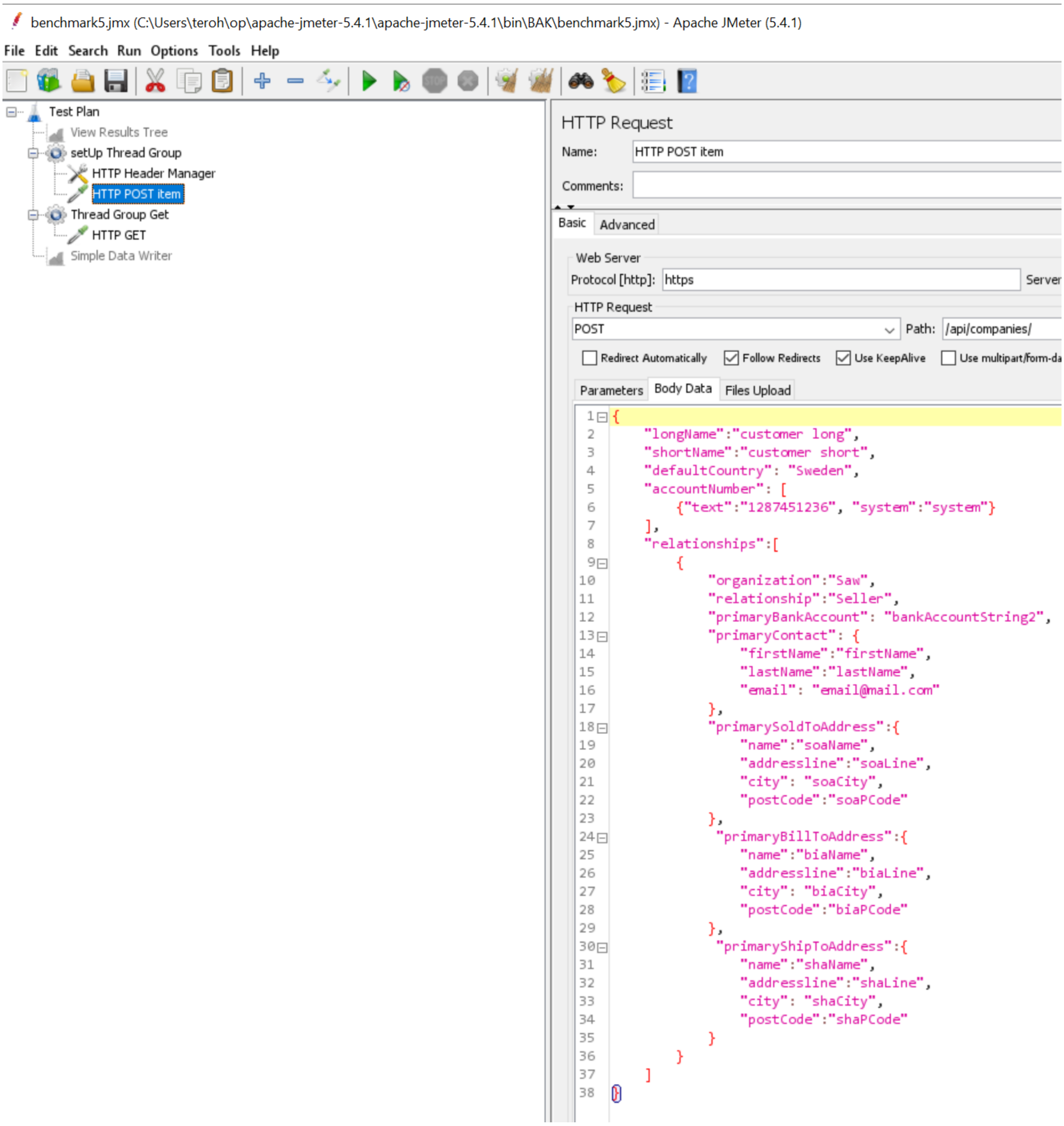


Figure 7. JMeter HTTP POST request.

HTTP POST request alters the tested systems database state by making requests to add data into the database. To get reliable data out of the load tests, the database state has to be the same before each test run. To achieve steady database base starting point state the tested application instance was restarted after every test run. Restarting the application reinitializes the non-persistent in-memory database.

Thread Group Get in figure 8 is configured with 100 threads with 100 loop count totaling 10000 requests. Single GET request is responded by a list of ten objects from the application API.

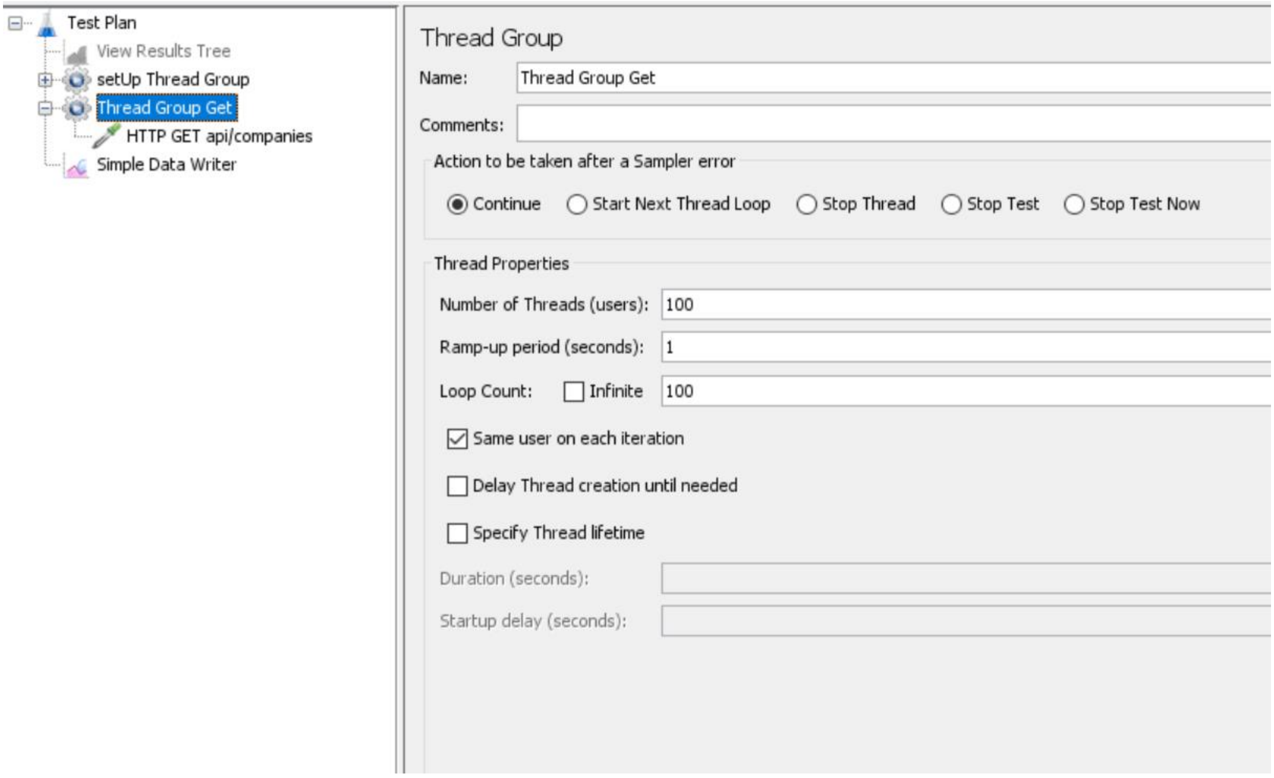


Figure 8. JMeter Thead Group Get.

Configuration for HTTP GET request in figure 9 is set up as a GET type HTTP request to local environment targeting api/companies endpoint in port 5001.

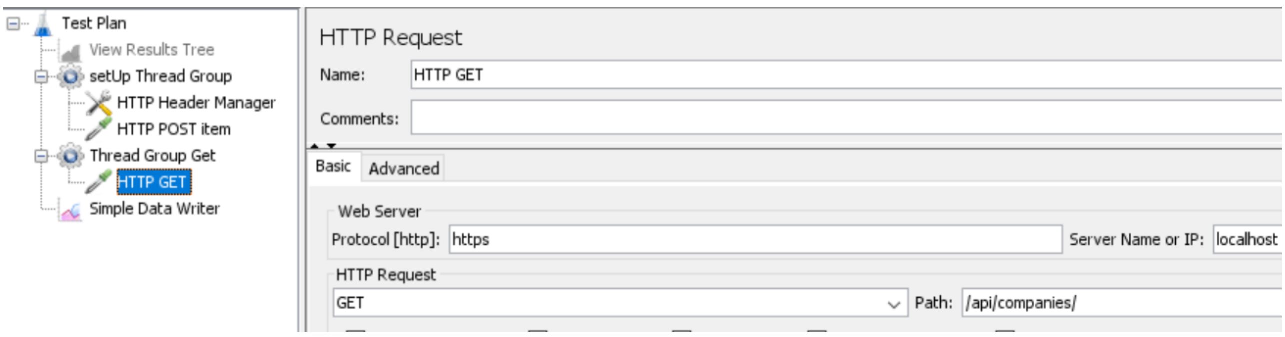


Figure 9. JMeter HTTP GET request.

Simple Data Writer in figure 10 is a listener type element which can be used to save test result data.

Figure 10. JMeter Simple Data Writer.

While JMeter saves test result data in plain CSV, Simple Data Writer can be configured to save data in XML format as depicted in figure 11.

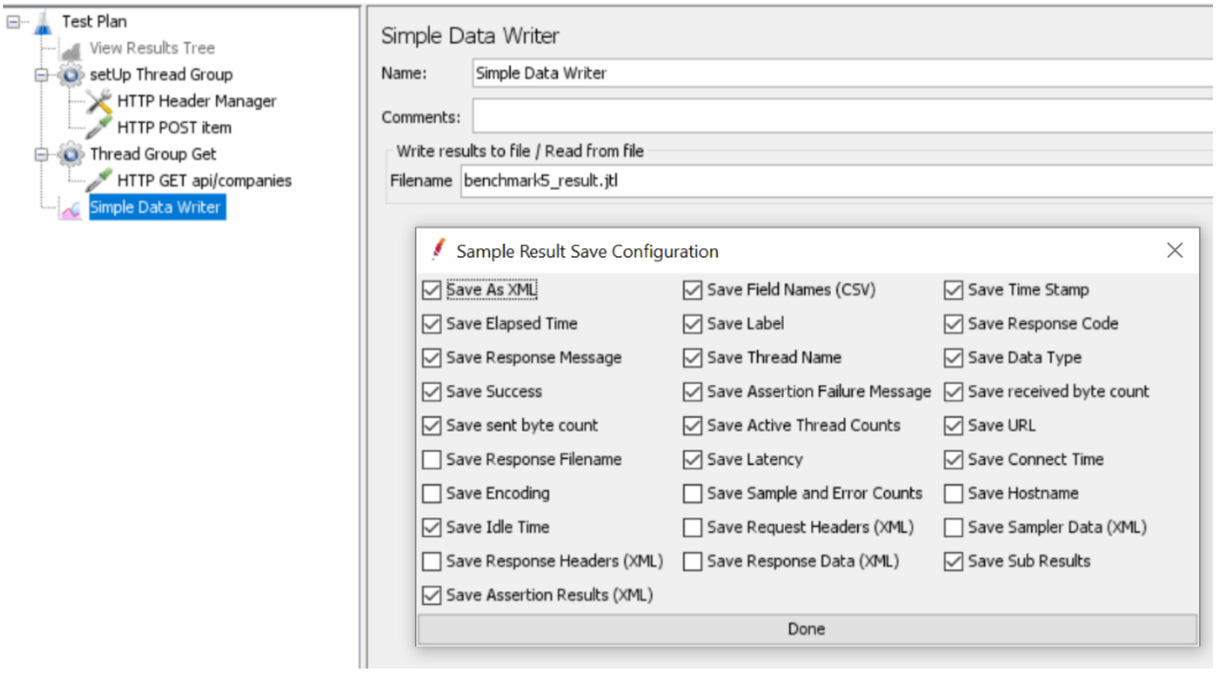


Figure 11. Simple Data Writer options.

2.5.4 BenchmarkDotNet

.NET Benchmark project tree for non-practical test depicted in figure 12 shows the needed files to conduct the code level benchmarking tests. The project is setup as a basic .NET console application. An empty console application starts with the Program.cs file in the project tree where the rest of the files are added.

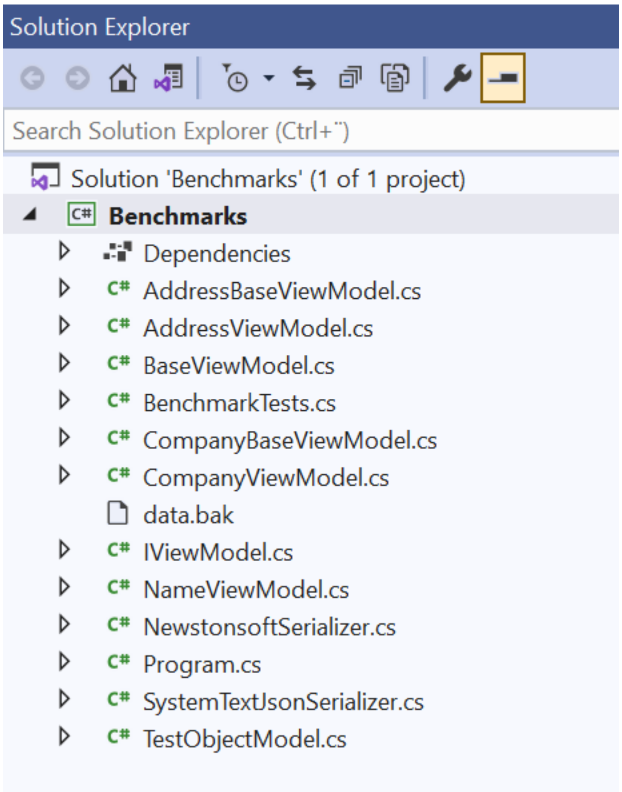


Figure 12. .NET Benchmarks project tree.

Figure 13 shows the projects csproj file. This file holds the relevant project information. In <Target-Frameworks> tags are the targeted frameworks. Defining both target frameworks here builds both application target version with single build command.

Benchmarks.csproj

```xml
<Project Sdk="Microsoft.NET.Sdk">

    <PropertyGroup>
        <OutputType>Exe</OutputType>
        <AllowUnsafeBlocks>true</AllowUnsafeBlocks>
        <ServerGarbageCollection>true</ServerGarbageCollection>
        <TargetFrameworks>net5.0;netcoreapp3.1;</TargetFrameworks>
    </PropertyGroup>

    <ItemGroup>
        <PackageReference Include="benchmarkdotnet" Version="0.12.1" />
        <PackageReference Include="Newtonsoft.Json" Version="13.0.1" />
    </ItemGroup>
</Project>
```

Figure 13. Benchmarks.csproj.

The main program entry point in .NET project is the Program.cs. Main() method in figure 14 is set to run the BenchmarkDotnet's BenchmarkRunner. When the application is launched the BenchmarkRunner executes the BenchmarkTests defined in BenchmarkTests.cs class.

```csharp
using BenchmarkDotNet.Running;
using Benchmarks;

0 references
public class Program
{
    0 references
    static void Main(string[] args)
    {
        BenchmarkRunner.Run<BenchmarkTests>();
    }
}
```

Figure 14. Program.cs

Newtonsoft serializer in figure 15 with serialize/deserialize methods used to serialize/deserialize data model objects.

```
using Newtonsoft.Json;

namespace Benchmarks
{
    2 references
    class NewstonsoftSerializer
    {
        1 reference
        public TestObjectModel Deserialize(string modelString)
        {
            var testModel = JsonConvert.DeserializeObject<TestObjectModel>(modelString);
            return testModel;
        }
        1 reference
        public string Serialize(TestObjectModel model)
        {
            var testSerializedModel = JsonConvert.SerializeObject(model);
            return testSerializedModel;
        }
    }
}
```

Figure 15. Newtosoft serializer.

.NET serializer in figure 16 with serialize/deserialize methods used to serialize/deserialize data model objects.

```
using System.Text.Json;

namespace Benchmarks
{
    2 references
    class SystemTextJsonSerializer
    {
        1 reference
        public TestObjectModel Deserialize(string modelString)
        {
            var testModel = JsonSerializer.Deserialize<TestObjectModel>(modelString);
            return testModel;
        }
        1 reference
        public string Serialize(TestObjectModel model)
        {
            var testSerializedModel = JsonSerializer.Serialize(model);
            return testSerializedModel;
        }
    }
}
```

Figure 16. System.Text.Json serializer.

Figure 17 depicts the test object defined which was used by the serializer tests.

```
namespace Benchmarks
{
    7 references
    class TestObjectModel
    {
        0 references
        public TestObjectModel() { }
        0 references
        public string FieldOne { get; set; }
        0 references
        public string FieldTwo { get; set; }
        0 references
        public string FieldThree { get; set; }
        0 references
        public string FieldFour { get; set; }
        0 references
        public string FieldFive { get; set; }
        0 references
        public string FieldSix { get; set; }

    }
}
```

Figure 17. TestObjectModel.

BenchMarkTests.cs class in figure 18 is where all the benchmarking tests are defined. [MemoryDiagnoser] tells the benchmark to gather memory data, [Orderer(SummaryOrderPolicy.FastestToSlowest)] orders results from fastest to slowest.

```
using BenchmarkDotNet.Attributes;
using BenchmarkDotNet.Order;
using Newtonsoft.Json;
using System;
using System.Collections.Generic;
using System.Net.Http;
using System.Text.RegularExpressions;
using System.Threading.Tasks;

namespace Benchmarks
{
    [MemoryDiagnoser]
    [Orderer(SummaryOrderPolicy.FastestToSlowest)]
    1 reference
    public class BenchmarkTests
    {
```

Figure 18. BenchMarkTests.cs.

Serialize/Deserialize tests in figure 19 were done using two different JSON serializing providers. Newtonsoft serializer and .NET provided System.Text.Json serializer. TestObjectSerialized string is used in deserializeing tests and the instantiated testObject is used in serializing tests.

```
private string _data = new HttpClient().GetStringAsync(dataUrl).Result;

private static readonly NewstonsoftSerializer NewstonsoftSerializer =
    new NewstonsoftSerializer();
private static readonly SystemTextJsonSerializer SystemTextJsonSerializer =
    new SystemTextJsonSerializer();

private const string testObjectSerialized = "{\"FieldOne\":" +
    "\"fieldValueString\",\"FieldTwo\":\"fieldValueString\"," +
    "\"FieldThree\":\"fieldValueString\",\"FieldFour\":\"fieldValueString\"," +
    "\"FieldFive\":\"fieldValueString\",\"FieldSix\":\"fieldValueString\"}";

private TestObjectModel testObject = new TestObjectModel() {
    FieldOne = "fieldValueString",
    FieldTwo = "fieldValueString",
    FieldThree = "fieldValueString",
    FieldFour = "fieldValueString",
    FieldFive = "fieldValueString",
    FieldSix = "fieldValueString"
};

[Benchmark]
0 references
public int EmailPatternSearch() => _emailPattern.Matches(_data).Count;

[Benchmark]
0 references
public void DeserializeNewtonsoft() => NewstonsoftSerializer.Deserialize(testObjectSerialized);

[Benchmark]
0 references
public void DeserializeSystemTextJson() => SystemTextJsonSerializer.Deserialize(testObjectSerialized);

[Benchmark]
0 references
public void SerializeNewtonsoft() => NewstonsoftSerializer.Serialize(testObject);

[Benchmark]
0 references
public void SerializeSystemTextJsont() => SystemTextJsonSerializer.Serialize(testObject);
```

Figure 19. Serialize/deserialize tests.

The functionality of the applications ApiService class was mocked in the tests. The test in figure 20 creates an HTTP client and sends a GET request to local endpoint /api/companies. Api responses with a serialized json list of 10 items. Then the serialized list is deserialized to list of view model items. Endpoint /api/addresses was also targeted.

```
private readonly HttpClient client = new HttpClient();

[Benchmark]
[Arguments("https://localhost:44314/api/companies")]
0 references
public async Task<List<CompanyViewModel>> MockApiServiceGetList(string url)
{
    using var request = new HttpRequestMessage()
    {
        RequestUri = new Uri(url),
        Method = HttpMethod.Get
    };

    using HttpResponseMessage response = await client.GetAsync(url).ConfigureAwait(false);

    response.EnsureSuccessStatusCode();

    var responseBodyString = await response.Content.ReadAsStringAsync();

    var itemList = JsonConvert.DeserializeObject<List<CompanyViewModel>>(responseBodyString);

    return itemList;
}
```

Figure 20. Mock ApiService test.

EmailPatternSearch test in figure 21 tests the speed of string pattern matching. The variable _emailPattern defines the pattern to be matched. The string variable _data was sourced from Juárez (2021) public regex-benchmark GitHub project, it contains the string data to where the email pattern is matched. The function returns the count of matches found in the _data.

```
private Regex _emailPattern =
    new Regex(@"[\w\.+-]+@[\w\.-]+\.[\w\.-]+", RegexOptions.Compiled);

private static string dataUrl ="https://"+
    "raw.githubusercontent.com/mariomka"+
    "/regex-benchmark/652d55810691ad88e1c2292a2646d301d3928903/input-text.txt";
private string _data = new HttpClient().GetStringAsync(dataUrl).Result;

[Benchmark]
0 references
public int EmailPatternSearch() => _emailPattern.Matches(_data).Count;
```

Figure 21. String pattern test.

2.6 Tests

Test were run so many times that at least three stable results could be recorded. Of the three recorded runs deemed stable the mean values of each data were taken into account.

2.6.1 JMeter

Two API endpoints were tested with the load testing tool with following setups: endpoint 1, /api/companies, 400 POST and 10 000 GET requests, endpoint 2, /api/addresses, 4 000 POST and 100 000 GET requests. JMeter tests were run in non-GUI mode. Non-GUI mode test were run with command: jmeter -n -t <testfilename> -l <logfilename>. Figure 22 show an example test output from console view.

```
c:\Users\teroh\op\apache-jmeter-5.4.1\apache-jmeter-5.4.1\bin>jmeter -n -t benchmark31.jmx -l jmeter31.jtl
Creating summariser <summary>
Created the tree successfully using benchmark31.jmx
Starting standalone test @ Fri Apr 02 20:54:12 EEST 2021 (1617386052372)
Waiting for possible Shutdown/StopTestNow/HeapDump/ThreadDump message on port 4445
summary +    121 in 00:00:18 =    6.8/s Avg:   320 Min:     2 Max:  2235 Err:     0 (0.00%) Active: 1 Started: 8 Finished: 7
summary +    300 in 00:00:30 =   10.0/s Avg:     3 Min:     1 Max:    71 Err:     0 (0.00%) Active: 1 Started: 38 Finished: 37
summary =    421 in 00:00:48 =    8.8/s Avg:    94 Min:     1 Max:  2235 Err:     0 (0.00%)
summary +    300 in 00:00:30 =   10.0/s Avg:     3 Min:     1 Max:   134 Err:     0 (0.00%) Active: 1 Started: 68 Finished: 67
summary =    721 in 00:01:18 =    9.3/s Avg:    56 Min:     1 Max:  2235 Err:     0 (0.00%)
summary +    300 in 00:00:30 =   10.0/s Avg:     3 Min:     1 Max:   177 Err:     0 (0.00%) Active: 1 Started: 98 Finished: 97
summary =   1021 in 00:01:48 =    9.5/s Avg:    41 Min:     1 Max:  2235 Err:     0 (0.00%)
summary +     29 in 00:00:02 =   14.3/s Avg:     2 Min:     1 Max:    12 Err:     0 (0.00%) Active: 0 Started: 100 Finished: 100
summary =   1050 in 00:01:50 =    9.6/s Avg:    40 Min:     1 Max:  2235 Err:     0 (0.00%)
Tidying up ...    @ Fri Apr 02 20:56:02 EEST 2021 (1617386162495)
... end of run
```

Figure 22. JMeter example test run.

After every test run using command: jmeter -g <logfilename> -o <directoryname>, was used to generate a directory and a html report from each runs log file. The generated directory with the html report file depicted in figure 23 contains files needed to display the data in browser.

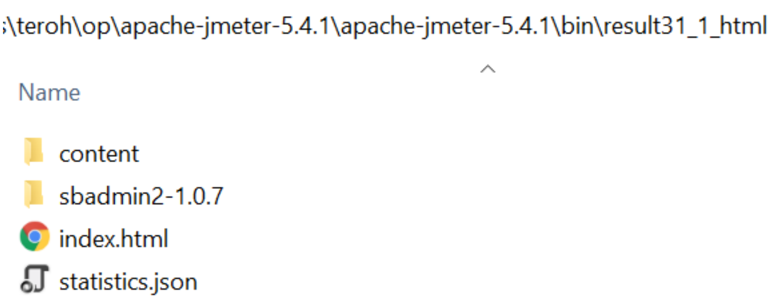
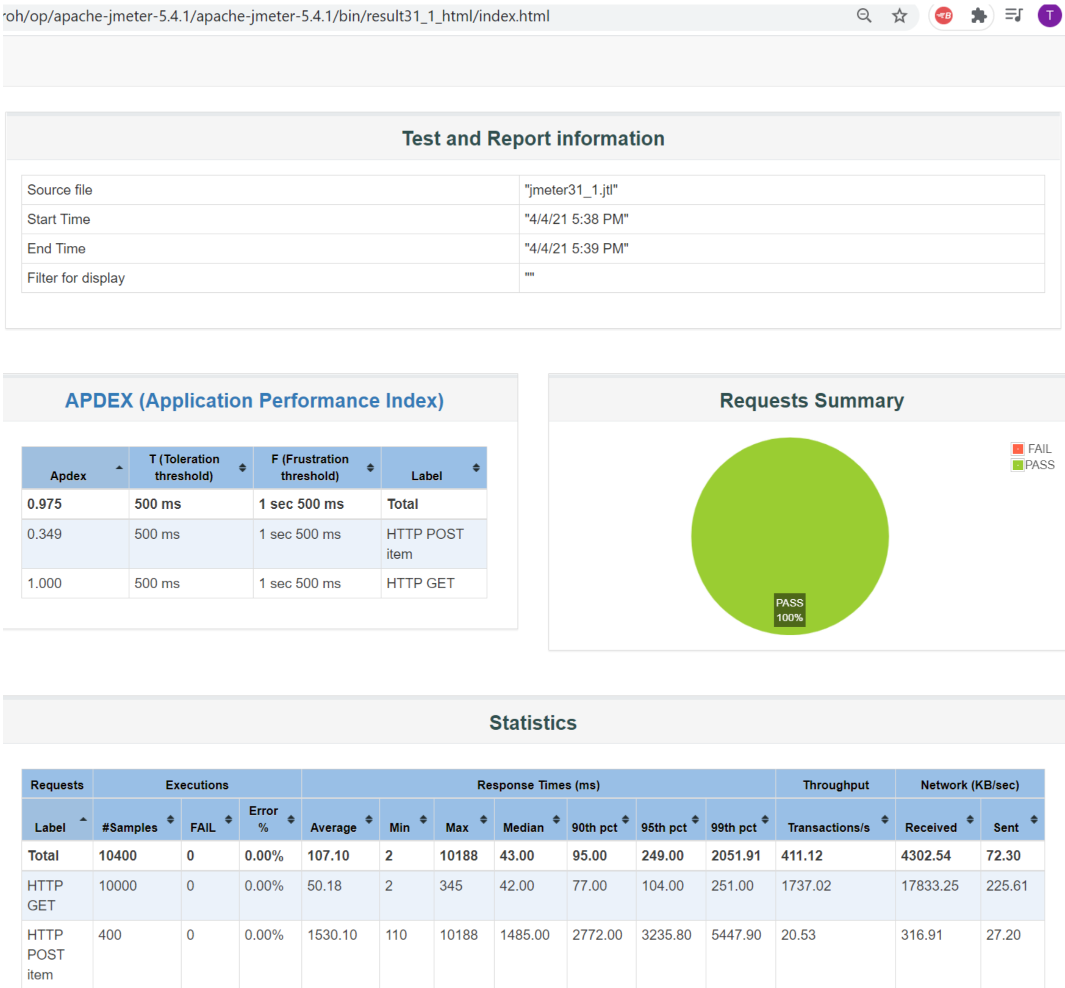


Figure 23. JMeter report director.

Resulting html has data in different charts, depicted in the figure 24 is the report opened in browser with the total statistic view of a test run.



Test and Report information

Source file	"jmeter31_1.jtl"
Start Time	"4/4/21 5:38 PM"
End Time	"4/4/21 5:39 PM"
Filter for display	""

APDEX (Application Performance Index)

Apdex	T (Toleration threshold)	F (Frustration threshold)	Label
0.975	**500 ms**	**1 sec 500 ms**	**Total**
0.349	500 ms	1 sec 500 ms	HTTP POST item
1.000	500 ms	1 sec 500 ms	HTTP GET

Statistics

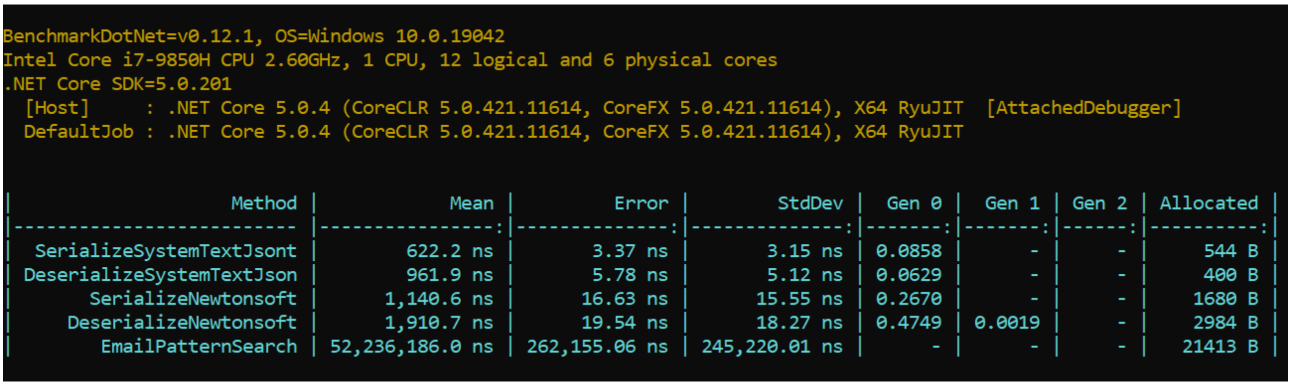
Requests	Executions			Response Times (ms)							Throughput	Network (KB/sec)	
Label	#Samples	FAIL	Error %	Average	Min	Max	Median	90th pct	95th pct	99th pct	Transactions/s	Received	Sent
Total	**10400**	**0**	**0.00%**	**107.10**	**2**	**10188**	**43.00**	**95.00**	**249.00**	**2051.91**	**411.12**	**4302.54**	**72.30**
HTTP GET	10000	0	0.00%	50.18	2	345	42.00	77.00	104.00	251.00	1737.02	17833.25	225.61
HTTP POST item	400	0	0.00%	1530.10	110	10188	1485.00	2772.00	3235.80	5447.90	20.53	316.91	27.20

Figure 24. JMeter html report.

2.6.2 BenchmarkDotNet

Running BenchmarkDotNet tests is straightforward and simple to execute. Building a release build of the benchmark application and running the resulted dll with the command: dotnet <application>.dll.

BenchmarkDotNet identifies the BenchmarkRunner in Program.cs class and will start executing the tests. After tests are run the results are printed in the console output. Following figure 25 depicts an example BenchmarkDotNet test run result report.



```
BenchmarkDotNet=v0.12.1, OS=Windows 10.0.19042
Intel Core i7-9850H CPU 2.60GHz, 1 CPU, 12 logical and 6 physical cores
.NET Core SDK=5.0.201
  [Host]     : .NET Core 5.0.4 (CoreCLR 5.0.421.11614, CoreFX 5.0.421.11614), X64 RyuJIT  [AttachedDebugger]
  DefaultJob : .NET Core 5.0.4 (CoreCLR 5.0.421.11614, CoreFX 5.0.421.11614), X64 RyuJIT

|                   Method |             Mean |          Error |         StdDev |  Gen 0 |  Gen 1 | Gen 2 | Allocated |
|------------------------- |-----------------:|---------------:|---------------:|-------:|-------:|------:|----------:|
|  SerializeSystemTextJsont |          622.2 ns |        3.37 ns |        3.15 ns | 0.0858 |      - |     - |     544 B |
| DeserializeSystemTextJson |          961.9 ns |        5.78 ns |        5.12 ns | 0.0629 |      - |     - |     400 B |
|      SerializeNewtonsoft |        1,140.6 ns |       16.63 ns |       15.55 ns | 0.2670 |      - |     - |    1680 B |
|    DeserializeNewtonsoft |        1,910.7 ns |       19.54 ns |       18.27 ns | 0.4749 | 0.0019 |     - |    2984 B |
|       EmailPatternSearch | 52,236,186.0 ns | 262,155.06 ns | 245,220.01 ns |      - |      - |     - |   21413 B |
```

Figure 25. BenchmarkDotNet test report.

3 Results

3.1 Reported values

Of the gathered data, each test runs mean values are reported here, full measurement tables of the test runs in Appendix 1. To get mean values, the equation $Mean = \frac{sum\ of\ datapoints}{datapoint\ count}$ was used.

3.2 Practical load test results (JMeter)

Practical test results for endpoint1 show calculated averages in Table 1. Longer response times for .NET 5 was observed for both HTTP GET and POST request types.

Table 1. Endpoint1 load test results.

endpoint1 /api/companies/		
Request type	Time (ms) .NET Core 3.1	Time (ms) .NET 5
HTTP GET	56,47	78,07
HTTP POST	1577,81	1958,53
HTTP total	114,98	150,40

Practical test results for endpoint2 in Table 2 presents the calculated averages. Endpoint2 results show interesting data about the application. Almost 3 times longer HTTP GET response time recorded for .NET 5 is a peculiar finding. Response time for HTTP POST request was similar in both versions of the application.

Table 2. Endpoint2 load test results.

endpoint2 /api/addresses/		
Request type	Time (ms) .NET Core 3.1	Time (ms) .NET 5
HTTP GET	26,37	74,45
HTTP POST	20,48	19,14
HTTP total	26,48	72,33

3.3 Non-practical test results (BenchmarkDotNet)

BenchmarkDotNet reports results as averages as the number of samples can vary a lot between the tested methods and test runs. .NET Core 3.1 non-practical test calculated averages in Table 3.

Table 3. Code level performance results.

Test	Time (ns) .NET Core 3.1	Time (ns) .NET 5
SerializeNewtonsoft	1100,3	1030,7
DeserializeNewtonsoft	1956,3	1748,3
SerializeSystemTextJson	744,9	593,7
DeserializeSystemTextJson	1047,73	922,47
EmailPatternSearch	938166,7	50420,0
MockApiServiceGetList	1962,0	1947,0

4 Analysis

4.1 Comparison of results

Some of results gathered were expected and some a bit surprising. Results was analyzed by comparing the .NET Core 3.1 result mean values with .NET 5 result mean values by applying the formula:

$$Relative\ difference = (5\ mean\ value - \ Core\ 3.1\ mean\ value)\ /\ Core\ 3.1\ mean\ value$$

The resulting table 4 shows the .NET 5 values relative to .NET Core 3.1 values, where negative value denotes .NET 5 is faster to .NET Core 3.1.

Table 4. Code level .NET 5 speed relative to .NET Core 3.1.

.NET 5 speed relative to .NET Core 3.1	
Test	Percentage difference
SerializeNewtonsoft	-6,33 %
DeserializeNewtonsoft	-10,63 %
SerializeSystemTextJson	-20,30 %
DeserializeSystemTextJson	-11,96 %
EmailPatternSearch	-94,63 %
MockApiServiceGetList	-0,76 %

The practical API load tests resulted in more unexpected results. While improved performance was expected from.NET 5 over .NET 3.1 Core, the opposite was recorded. To get the total weighted relative difference following equations were used:

$$Http\ request\ mean\ difference = \frac{(5\ http\ req.mean - \ Core\ 3.1\ http\ req.average)}{Core\ 3.1\ mean}$$

$$
\begin{aligned}
&\textit{Weighted total http request difference} \\
&\qquad = (endpoint1\ http\ req.mean\ diff.*\ weight1 \\
&\qquad + endpoint2\ http\ req.mean\ diff.*\ weight2)
\end{aligned}
$$

Table 5 shows the weighted difference percentages, where positive value means .NET 5 slower performant than .NET Core 3.1. The resulting performance difference recorded was surprising. The seeming speed regression of the regarding the HTTP GET request .NET 5 was not investigated in this work. The anlyze revealed a problem in the application that needs to be solved.

Table 5. Combined endpoints weighted total relative difference.

.NET 5 weighted total difference relative to .NET Core 3.1	
HTTP request	Percentage difference
HTTP GET	169,22 %
HTTP POST	-3,78 %
HTTP total	160,17 %

4.2 Reliability

As a good practice the meaningfulness and the reliability of the testing and the process was questioned throughout the work. In the context of this work, recorded results are somewhat reliable. The data gathered can be compared between the tested versions and results in indication of the applications current states performance in the tested environment.

During the practical load tests an application bug was recorded when testing the endpoint1 of the application. This resulted in unreliable data being recorded from endpoint1. Since the bug presented in both versions tested, the data is somewhat comparable in this work context. The performance discrepancy of the practical and the non-practical performance where .NET 5 was more performant than .NET 3.1 Core and practical performance is too significant to regard the load tests reliable. The load test discrepancy in performance paired with the non-practical MockApiService test results, in which the API-layer was called by the test methods HTTP client, .NET 5 had similar results as .NET Core 3.1, also raises suspicion of anomalies in the load tests caused by unknown factor.

The non-practical tests done with BenchmarkDotNet were inline and confirmed the findings made by Toub (2020), that .NET 5 has performance advantages over .NET Core 3.1. Non-practical tests resulted in more expected outcome and as a whole are regarded as reliable.

5 Discussion

Performance testing is an interesting part of the software testing field. It provides almost endless set of problems and test variations. Starting this work with zero experience in performance testing, at first seemed challenging but the problem presented itself as an intriguing one to solve.

Upon taking the work, the project seemed quite vast with many different subsections requiring expertise in their respective parts. While researching the subject the depth of performance testing as a testing field became clear. The level of detail, scale, and resolution in which things can be measured by can be overwhelming. Dissecting the problem into smaller sections, more clearly defined set of tasks started to appear.

Interestingly the well-defined scope by the thesis contractor greatly aided in the planning phase of the work. The scope targeted performance differences of two versions of the same software in Web API use. The required steps to achieve the scope target then consisted of researching what to measure and how get the required data, what environment or environments can be used, what tools are suitable for the purpose, how to conduct the tests and analyze data.

The setup part of the test plan was to select the tools and build the tests. Some errors in the test building phase were identified. The load test plan using the complex object sent in the HTTP POST request targeted at /api/companies had resulted in LINQ errors in the application, this resulted in longer processing times in both HTTP GET and HTTP POST requests in this endpoint. Data gathered in this endpoint is somewhat skewed but still comparable between versions. The other endpoint /api/addresses tests were included after finding the error. Second endpoint performed drastically better with simpler object. Building the non-practical tests was fairly straightforward task of identifying operations in the application process flow and writing those methods in to the testing project.

Implementation phase of the work was to run the built tests and record the results. Release builds of the application and benchmark projects were built for the tests. More planning could have been used in recording the gathered data to make the data processing in excel sheets easier. The data copy pasted from the tools resulted in wrong formatting requiring data reinputting in properly formatted cells. Manual input is never a good option if it can be avoided.

Analyzing the data was comparing the calculated average results. This would give the contractor meaningful information of the data gathered about the measured speed differences between the software versions of the application and code level performance.

Finishing the work gave some insight into software performance testing to the worker. The issue of software performance testing can be greatly taken more in-depth. In this work scope the more in-depth approach was not practical to be applied nor in the scope context.

References

BenchmarkDotNet. (n.d.). *BenchmarkDotNet*. Accessed on 18 February 2021. Retrieved https://benchmarkdotnet.org

Cambridge dictionary. (n.d.). *PERFORMANCE | meaning in the Cambridge English Dictionary*. Accessed on 19 April 2020. Retrieved https://dictionary.cambridge.org/dictionary/english/performance

Datanyze. (n.d.). *ASP.Net market share*. Datanyze. Accessed on 18 February 2021. Retrieved https://www.datanyze.com/market-share/programming-languages--67/asp.net-market-share

Doyle, K. (2021, February 21). *Gatling vs JMeter vs The Grinder | Baeldung*. Accessed on 18 February 2021. Retrieved https://www.baeldung.com/gatling-jmeter-grinder-comparison

Guru99. (n.d.). *Performance testing process*. Accessed on 18 February 2021. Retrieved https://www.guru99.com/performance-testing.html#5

Guru99. (n.d.). *What is performance testing*. Accessed on 18 February 2021. Retrieved https://www.guru99.com/performance-testing.html#1

JMeter. (n.d.). *Apache JMeter - User's Manual: Best Practices*. Accessed on 11 May 2021. Retrieved https://jmeter.apache.org/usermanual/best-practices.html#lean_mean

Juárez, M. (2017, September 10). Accessed on 6 April 2021. Retrieved https://github.com/mariomka/regex-benchmark/blob/master/input-text.txt

Luijbregts, B. (2018, January 8). *The .NET Ecosystem Demystified*. Accessed on 26 April 2020. Retrieved https://stackify.com/net-ecosystem-demystified/

Microsoft. (2019, April 5). *ASP.NET Core load/stress testing*. Accessed on 25 April 2021. Retrieved https://docs.microsoft.com/en-us/aspnet/core/test/load-tests?view=aspnetcore-5.0

Microsoft. (n.d.). *.NET Core and .NET 5 Support Policy*. Accessed on 25 April 2021. Retrieved https://dotnet.microsoft.com/platform/support/policy/dotnet-core

Mifsud, J. (n.d.). *The Difference (And Relationship) Between Usability And User Experience*. Usabilitygeek. Accessed on 22 April 2021. Retrieved https://usabilitygeek.com/the-difference-between-usability-and-user-experience/

Nuget. (n.d.). *NuGet Gallery | BenchmarkDotNet 0.12.1.* Accessed on 12 April 2021. Retrieved https://www.nuget.org/packages/BenchmarkDotNet/

Ramel, D. (2019, April 9). *.NET Core Is 'Most Loved' Framework in Stack Overflow Survey*. Visual Studio Magazine. Accessed on 18 February 2021. Retrieved https://visualstudiomagazine.com/articles/2019/04/09/so-survey.aspx

Toub, S. (2020). *Performance Improvements in .NET 5*. .NET Blog. Accessed on 18 February 2021. Retrieved https://devblogs.microsoft.com/dotnet/performance-improvements-in-net-5/

Trianz. (2020). *Why Master Data Management is a Business-Critical Topic*. Accessed on 18 February 2021. Retrieved https://www.trianz.com/insights/why-master-data-management-is-a-business-critical-topic

Appendices

Appendix 1. Measurement tables

JMeter Api-endpoint load test measurement tables.

.NET Core 3.1 endpoint1 /api/companies			
	HTTP total (samples 10400) (ms)	HTTP GET (samples 10000) (ms)	HTTP POST (samples 400) (ms)
Run1	107,1	50,18	1530,1
Run2	119,35	59,62	1612,61
Run3	118,5	59,61	1590,72

.NET Core 3.1 endpoint2 /api/addresses			
	HTTP total (samples 104000) (ms)	HTTP GET (samples 100000) (ms)	HTTP POST (samples 4000) (ms)
Run1	27,17	26,03	29,53
Run2	25,96	26,37	15,77
Run3	26,32	26,72	16,16

.NET 5 endpoint1 /api/companies			
	HTTP total (samples 10400) (ms)	HTTP GET (samples 10000) (ms)	HTTP POST (samples 400) (ms)
Run1	153,87	80,6	1985,45
Run2	150,39	76,81	1989,79
Run3	146,95	76,82	1900,37

.NET 5 endpoint2 /api/addresses (ms)			
	HTTP total (samples 104000) (ms)	HTTP GET (samples 100000) (ms)	HTTP POST (samples 4000) (ms)

Run1	72,47	74,6	19,21
Run2	72,15	74,27	19,09
Run3	72,37	74,5	19,12

Relative weight table.

	Sample size	Weight
weight1	10400	0,090909
weight2	104000	0,909091

BenchmarkDotNet benchmark measurement tables.

.NET Core 3.1 benchmarks (ns)						
	SerializeSystemTextJson	DeserializeSystemTextJson	SerializeNewtonsoft	DeserializeNewtonsoft	EmailPatternSearch	MockApiServiceGetList
Run1	745,1	1022,3	1103	1981,7	925400,0	1950,0
Run2	744,6	1053,9	1130,9	1926,7	940800,0	1969,0
Run3	745	1042,7	1067	1960,5	948300,0	1967,0

.NET 5 benchmarks (ns)						
	SerializeSystemTextJson	DeserializeSystemTextJson	SerializeNewtonsoft	DeserializeNewtonsoft	EmailPatternSearch	MockApiServiceGetList
Run1	594	911,1	1067,8	1745,3	50540,0	1936,0
Run2	601,7	931,3	1014,6	1741,8	50330,0	1958,0
Run3	585,4	925	1009,7	1757,8	50390,0	1947,0