DE GRUYTER OPEN
RIGA TECHNICAL UNIVERSITY
Applied Computer Systems
ISSN 2255-8691 (online)
ISSN 2255-8683 (print)
May 2018, vol. 23, no. 1, pp. 37–44
doi: 10.2478/acss-2018-0005
https://www.degruyter.com/view/j/acss

Performance Comparison of Java EE and ASP.NET Core Technologies for Web API Development

Kristiāns Kronis[1*], Marina Uhanova[2]
[1, 2] *Riga Technical University, Riga, Latvia*

***Abstract* – The paper describes the implementation of organic benchmarks for Java EE and ASP.NET Core, which are used to compare the performance characteristics of the language runtimes. The benchmarks are created as REST services, which process data in the JSON format. The ASP.NET Core implementation utilises the Kestrel web server, while the Java EE implementation uses Apache TomEE, which is based on Apache Tomcat. A separate service is created for invoking the benchmarks and collecting their results. It uses Express with ES6 (for its async features), Redis and MySQL. A web-based interface for utilising this service and displaying the results is also created, using Angular 5.**

***Keywords* – Benchmark testing, computer languages, programming, software performance.**

I. Introduction

Both Java EE (Java Platform, Enterprise Edition), developed by Oracle, and ASP.NET (Active Server Pages .NET), developed by Microsoft, offer features fit for the creation of web-based applications. However, in recent history, Java EE has had better cross-platform support – it is possible to install the HotSpot implementation of the JVM (Java Virtual Machine), which is supported by Oracle, on both Windows and GNU/Linux operating systems. However, when dealing with .NET, the full .NET framework does not run on GNU/Linux and Mono must be used, which was originally an open source project and was only acquired by Microsoft in 2016 [1]. It does not offer support for WPF (Windows Presentation Foundation), WWF (Windows Workflow Foundation), while offering limited support for WCF (Windows Communication Foundation) and ASP.NET [2].

However, with the release of .NET Core in 2016 and, subsequently, the ASP.NET Core [3], Microsoft is supporting more operating systems. Now, as there is a first-party CLR (Common Language Runtime) implementation available on GNU/Linux, in addition to a modern rewrite of ASP.NET and a new web server – Kestrel [4], it would be beneficial to re-evaluate which technology stack is better for new projects.

The evaluation can be performed by examining their differences, i.e., how the Kestrel web server is different from IIS, which it is supposed to replace, and the most popular Java web servers, such as Apache Tomcat [5], how the runtime performance differs in typical use cases, running under similar, commonly utilised configurations.

The present paper describes an implementation of a system, which is to be used for running organic benchmarks (real-world tests) and collecting their results, offering immediate visual feedback to the user. The main goal of the benchmarking is to gain an approximation of how performant both technology stacks are on the GNU/Linux operating system and to highlight any obvious differences. General guidelines are also laid out for the software architecture and implementation practices to ensure the capability of generating hundreds of concurrent requests and efficiently processing them, as well as handling any errors.

A common REST (Representational State Transfer) API (Application Programming Interface), which uses JSON (JavaScript Object Notation) for data transfer is described and implemented in both technologies and deployed on identical servers. A separate application, consisting of a front-end for test configuration written in Angular 5, and a back-end service for test execution and result processing, written in Express and Node.js, which uses Redis for temporary storage and MySQL for result logging, are also created. This system is designed modularly – the servers implementing the testing APIs can be configured in the front-end interface, in addition to configuring Redis and MySQL logging.

While no claims are made that the results will be objective, the system should serve as a starting point, allowing for extensibility – adding more servers, which can run different languages and software or hardware configurations, with no code changes, or extending the list of benchmarks to be run, should there be necessity for more specific tests in the future.

II. Java EE

Java EE (Java Platform, Enterprise Edition) is a superset of the Java SE (Java Platform, Standard Edition), which extends the general-purpose Java APIs to provide features, which are useful in an enterprise setting, such as dependency injection (CDI, EJB), transaction management (JTA) and dynamic webpage functionality (JSP, JSF), as well as features for creating web services (JAX-RS, JAX-WS), at the same time also shortening the development time and the software complexity (Fig. 1) [6]. The development is organised with the Java Community Process (JCP) and based on Java Specification Requests (JSR).

The present paper describes a setup that uses Java EE 7, which was released in 2013 [7]. The Java EE 7 platform can be further divided into the Full Platform and Web Profile, the purpose of which is to provide a more limited set of features, which is easier to support [8]. The developed API implementation takes advantage of Servlets, JSON, CDI and

* Corresponding author's e-mail: kristians.kronis@edu.rtu.lv

JAX-RS, thus using elements of the Full Platform specification. Java EE 7 was chosen, because at the time of writing, Java EE 8 adoption was still limited, only Glassfish 5.0 and Payara 5 supported the specification, neither of which was as popular as Apache Tomcat, upon which Apache TomEE was based.

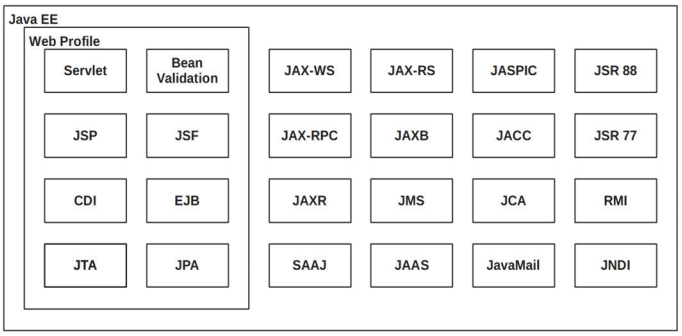

Fig 1. A diagram of the Java EE Full Platform and Web Profile specifications.

III. ASP.NET

ASP.NET (Active Server Pages .NET) is a web application framework that is developed by Microsoft and utilises the CLR [9]. The features provided are similar to Java EE, for example, Web Forms allows creating dynamic content, in a similar fashion to JSP and JSF (Fig. 2) [10]. A number of components, notably ASP.NET Web API, ASP.NET AJAX, ASP.NET MVC provide various functions [11]. ASP.NET is typically run on IIS (Internet Information Services), which is not supported on GNU/Linux by Microsoft; thus, Kestrel must be used instead.

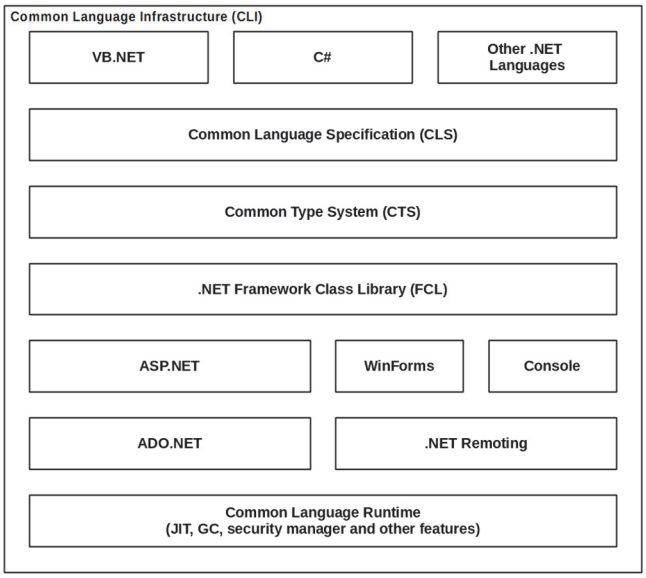

Fig. 2. A diagram of .NET Framework Architecture with ASP.NET displayed.

IV. ASP.NET Core

ASP.NET Core is the next generation of ASP.NET developed by Microsoft and community contributors [12] and can be run on both the full .NET framework and the .NET Core platform. It is a rewrite, which is meant to provide a slimmer, but more up-to-date set of features, as well as a new lightweight web server, Kestrel (although using IIS is still possible on Windows). It supports multiple components – Entity Framework Core, MVC Core and Razor Core [13], amongst others, which act as alternatives to those found in ASP.NET [14].

V. Requirements for Testing

To ensure optimal results, without interference caused by the configuration of the used operating system, some guidelines are defined for the design of the system:

1. The implementations for the APIs are to be placed on separate servers. In this case, VPS (virtual private servers) are used.
2. Both servers should have the same type and version of operating system. In this case, Ubuntu 16.04.4 LTS was chosen.
3. To ensure that the servers perform equally, they are to be tested by synthetic benchmarks. Both are to be subjected to the same types of benchmarks. Here, sysbench 0.4.12 was used.
4. The servers should run the same software with the same updates, differing only in the packages that are required to run the implementations – in this case, OpenJDK and .NET Core.
5. To test how well the web servers of the specific technology stacks perform (Apache TomEE and Kestrel), no reverse proxies (such as Apache httpd or Nginx) are to be used.

Requirements were also laid out for how the testing should be conducted to prevent memory leaks and errors caused by the implementation of the testing system itself:

1. The service that is invoking the benchmarks should not have its load capacity exceeded. This is ensured by each of the APIs being tested sequentially to prevent the service from being affected by the load, while it is still possible to run multiple iterations of a single test in parallel.
2. It should be possible to easily add and remove tests to be run, using a scheduler – this is implemented in the front-end, thus not limiting the testing service to sequential execution, but only utilising it in such a manner. This allows for future-proofing, should there ever be a requirement for the concurrent testing of multiple APIs.
3. The exchange of data that are not relevant among the layers of the system (APIs, testing service and the front-end) should be minimised. Here, the contents of the testing requests are generated on the testing service itself and are only exchanged with the APIs, which are to be tested. The front-end only receives the results.
4. The test results should be chunked and provided to the front-end upon request, before the completion of all of the scheduled test iterations, however, should also expire after a set amount of time if not retrieved.
5. The web interface should be lightweight. This is achieved by grouping the results and showing the averages in the groups when over 100 iterations are run, to prevent the chart framework from negatively affecting the performance. The MySQL database can be used for a finer analysis of the results.

VI. Details of the Testing System Components

The system is composed of several components, each using common technologies to re-create configurations, which could be found in real-world usage.

A. Java API Implementation

The Java API is implemented using Java EE features and avoiding third party frameworks, such as Spring, where possible. It is running on Apache TomEE Web Profile 7.0.2, which provides the functionality of Java EE 7 Web Profile using open-source components (OpenEJB, Apache CXF, etc.) [14]. It is based on Apache Tomcat – a popular application container [15]. It is run through OpenJDK, version 1.8.0_151.

B. ASP.NET Core API Implementation

The ASP.NET Core API is implemented using ASP.NET Core features and avoiding third party frameworks, where possible. It is running on Kestrel web server 2.0.1. The distribution is run through .NET Core, version 2.1.3.

C. Node.js + Express Testing Service

The testing service is running on Node.js v9.2.1 and based on Express 4.15.5. It uses cors, redis, express-redis and request-promise-native packages (installed through npm), amongst others, to provide the necessary functionality.

D. Angular 5 Web Interface

The front-end is created with Angular 5.0.0, TypeScript 2.5.3 and Bootstrap 4, which is used for providing styling and behaviour of the user interface, in combination with jQuery 3.2.1. In addition, ng2-charts is used to serve as a bridge between Angular and Chart.js, a framework that provides HTML5-based graph display capabilities [16].

VII. The Design of the Front-end

The front-end serves as a façade to the rest of the system and allows configuring the Redis and MySQL instances to use, as well as the servers to be subjected to testing. It does not handle generating the test request contents, but makes schedule invoking the testing service in the back-end.

The application structure is based on a single Angular module, which has services for storing data about settings and tests, the latter of which contain both the test entries themselves – with data about the test type, iterations and other parameters – and the servers that are to be used, each test having a reference to one of the server objects.

The rest of the app is composed of utility classes (such as enumerables, or notification components) and components for displaying the data and organising input and output – tab, menu and chart components.

The interface is tabbed to display only the information that is relevant to the user at any given moment – there are tabs for running tests and displaying their results (Fig. 3), as well as configuring the servers themselves (Fig. 4), and changing the system settings.

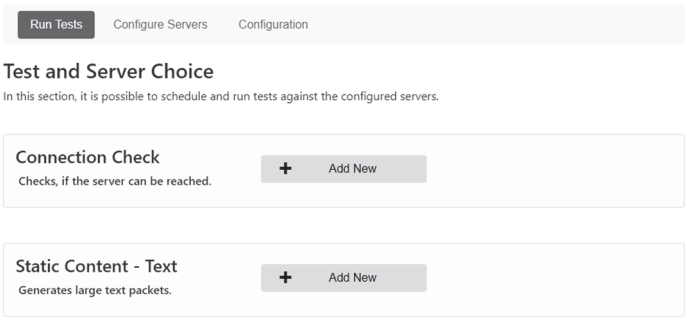

Fig. 3. The web interface, with the test tab open, which lists the available test types.

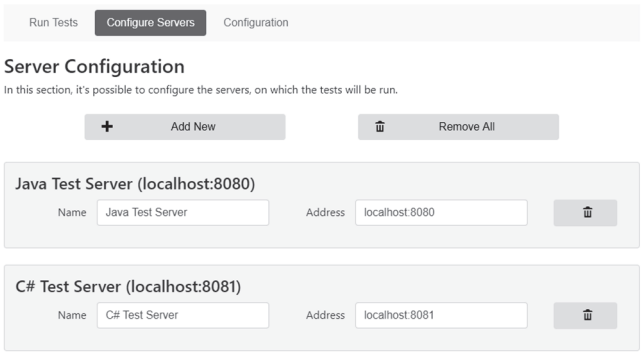

Fig. 4. The web interface, with the server configuration tab open, which lists the current servers.

Lastly, the benchmarking process also attempts to divide the request run time into its components, which are also displayed differently in the graphs (Fig. 5) – the time that a request spends on the network and the time that it spends being processed. The horizontal axis displays the iteration or a span of iterations (if more than 100 iterations are run), whereas the vertical axis displays the execution time, in milliseconds.

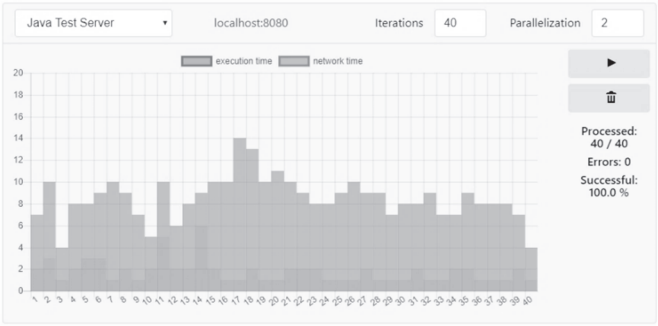

Fig 5. A part of the web interface, displaying the results of a test and controls for changing its parameters.

This is achieved through self-reporting by the systems and is displayed in the form of a stacked bar chart, in which the bottom bar displays the reported test execution time on the API, while the top bar displays the remainder of the time, which it took for the request to reach the testing service (which is calculated by subtracting the execution time from the total time).

VIII. Communication Between the Front-end and the Back-end

Layers of the system communicate (Fig. 6) using HTTP requests, which are provided by the HttpClient class in combination with the JavaScript JSON class on the Angular side, and Express routing with body-parser on the back-end. This format was chosen because it allowed for easy information exchange between the layers, as they use the contents as any other JavaScript object.

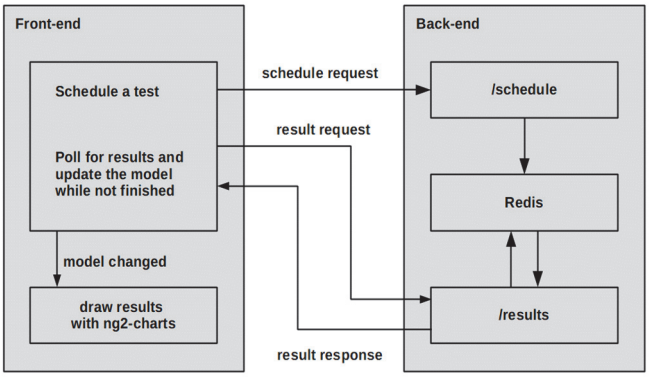

Fig. 6. A diagram of network requests, with Redis shown as an internal part of the back-end as a possible configuration.

Two paths are exposed by the testing service: /schedule and /results, the former of which allows scheduling a new test for execution, while the latter can be polled by the front-end to periodically receive and clear the result list stored in Redis, as well as check whether the execution has finished. Both utilise Redis for temporarily storing data, chosen because of its performance, as it uses RAM (random access memory) for temporary key-value storage. It is accessed through the express-redis library, which mirrors the official API and allows for atomic access [17]. The only exception is setting the TTL value for lists, which is done in a separate call, to make the values expire should they not be requested in a certain amount of time. For debugging purposes, all the communication between the layers of the system can be logged, either in the browser console or the Node.js output, which is redirected to a file.

When starting a test on the testing service, all of the relevant data to its execution must be passed in the first request: its kind, the iterations to be run, its unique identifier, information about the testing API to be used, as well as the configuration data for Redis and MySQL.

The response to a request for test results contains information about whether it is finished, as well as an array of the results. The results include the contents as well as the response of the request, the specific iteration, which the entry describes, as well as information about errors, should any have occurred (in the form of the contents of a stack trace of the language used). Timestamps are also included for measuring execution times.

The received timestamps come in pairs, the source ones are generated by the Node.js server, whereas the target ones are generated by the test API. Only their subtraction is important, so they do not have to match, allowing for server time configurations to differ without impacting the functionality.

Example of a logged request, which is used to launch the execution of a test through the testing service:

```
{
  "testKind": "connectionCheck",
  "iterations": 5,
  "testId": 1396483829,
  "results": [],
  "errorCount": 0,
  "successfulPercent": 100,
  "controlsBusy": true,
  "isRunning": "true",
  "parentServer": {
    "serverName": "Java Server",
    "serverAddress": "java.kronis.gdn:8080",
  },
  "chartData": null,
  "settingsRedisURL": "kronis.tk:6379",
  "settingsRedisDBNumber": "15",
  "settingsRedisPassword": "39fsdk2491",
  "settingsParallelTestIterations": 1,
  "settingsUseAudit": false,
  "settingsAuditURL": "kronis.tk:3306",
  "settingsAuditDBName": "audit_prod",
  "settingsAuditUsername": "audit_prod",
  "settingsAuditPassword": "4jsl3679kj"
}
```

Example of a logged response, which returns the information about the status of a test execution, as well as the results:

```
{
  "testingFinished": "finished",
  "testResponses": [
    "responseContents": {
      "text": "pong"
    },
    "targetReceivedTimeStamp": 1520196324474,
    "targetSentTimeStamp": 1520196324476,
    "requestContents": {
      "text": "ping"
    },
    "error": false,
    "responseIteration": 0,
    "sourceSentTimeStamp": 1520196324404,
    "sourceReceivedTimeStamp": 1520196324495
  ],
  "resultsTruncated": true,
  "resultsLength": 5
}
```

Example of a logged request, which requests an updated list of results from the testing service:

```
{
  "testId": 139648329,
  "settingsRedisURL": "kronis.tk:6379",
  "settingsRedisPassword": "39fsdk2491",
  "settingsRedisDBNumber": 15
}
```

Example of a logged response, which responds to a request, to create a new test:

```
{
  "createdID": 1396483829,
  "creationStatus": "successful",
  "creationError": null
}
```

IX. Communication between the Back-end and Testing APIs

The communication between the back-end and the APIs to be tested varies more as there is not a unified data format, differences existing from one benchmark to the next; however, they share a common lifecycle (Fig. 11). The test execution is implemented with the ES6 async and Promise functionality, which allow for parallelisation of iterations, depending on the parameters set in the front-end. As soon as a response from the API is received, it is logged to Redis and that iteration is forgotten apart from it.

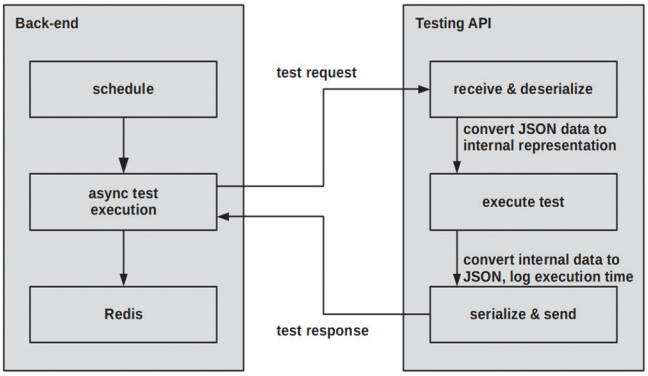

Fig. 7. A diagram of network requests and internal processes of the back-end and testable APIs, when processing test iterations.

X. Test API Benchmark Lifecycle

The testing APIs follow a common format of request processing lifecycle – first, a request is received by the REST controller, where the time it has been received is logged.

Then, the request is de-serialised from the JSON format into objects that the language can then work with natively. Since the individual benchmarks differ, this process is done dynamically instead of utilising predefined resource formats.

Then, the benchmark is executed with the acquired data and the results are written into memory, and are later serialised back to JSON, so that a response can be made. After the result serialisation is complete, the current time is logged once again and written to JSON separately in order not to cause an overhead. The response is then sent back to the testing service.

A. Java Implementation

In Java, `System.currentTimeMillis()` is used for keeping track of time, and `javax.json.*` packages are utilised to handle the JSON data format. The `JsonObjectBuilder` is used to allow writing the response dynamically, for example:

```
this.responseContentsBuilder
  = Json.createObjectBuilder();
this.testVO.getResponseContentsBuilder()
  .add("text", responseText);
```

In a similar fashion, data can be de-serialised from JSON in a `JsonObject` and typecast to Java's types.

```
int size = this.testVO
  .getRequestContents().getInt("size");
```

B. ASP.NET Core Implementation

Keeping track of the time was achieved with the following:

```
this.receivedTime = DateTime.UtcNow.Ticks /
TimeSpan.TicksPerMillisecond;
```

In regard to processing JSON, C# provides automatic type casting; in this case a `Dictionary` was used (later the response contents were passed to the `Json()` method):

```
Dictionary<string, string> responseContents
  = new Dictionary<string, string>();
responseContents.Add("text", responseText);
this.testVO.responseContents
  = responseContents;
```

As for reading JSON, the dynamic data type was used:

```
int size = this.testVO.requestContents.size;
```

XI. Testing Benchmarks

The web-based interface is only aware of the benchmarks in the form of an enumeration, which is passed to the testing service. Therefore, implementing new tests mainly takes place in the testing service, which generates the requests and the testing APIs themselves.

The tests implemented here were intended as a proof of concept, and as such avoided the use of external systems or databases (such as MySQL with JDBC or ADO), as I/O (input/output) would most likely increase their execution times noticeably.

The implemented tests are as follows:

- Connection Check – sends a short text string and receives a modified response;
- Static Content, Text – requests a response of a specific size from the server, consisting of randomly generated alphanumeric characters;
- Maths, Addition – performs addition with 1000 randomly generated real numbers;
- Maths, Multiplication – performs multiplication with 1000 randomly generated real numbers;
- Maths, Division – performs division with 1000 randomly generated real numbers;
- Maths, Powers – raises e to different powers with 1000 randomly generated real numbers;
- Maths, Logarithms – calculates natural logarithms with 1000 randomly generated real numbers, checks whether the results are finite numbers;
- Dynamic Content, Prime Numbers – generates 100 sequential prime numbers;
- Dynamic Content, JSON Structure, Reading – reads a JSON structure of nested objects, which contain arrays of numbers, and transforms it into a flat array;
- Dynamic Content, JSON Structure, Writing – writes a JSON structure of nested objects, which contain arrays of numbers, based on input parameters;
- Sorting, Whole Numbers – sorts an array of whole numbers, which has 1000 entries, returns the results;
- Sorting, Real Numbers – sorts an array of real numbers, which has 1000 entries, returns the results;

- LINQ / Streams – sorts the objects present in a JSON structure, which contains arrays of numbers, based on the average of all the elements present in each one;
- Cryptography, SHA256 – generates SHA256 hashes for the input strings;
- Cryptography, MD5 – generates MD5 hashes for the input strings.

XII. Server Benchmarking Results

To ensure that there were no notable discrepancies in the performance of the servers, synthetic benchmarks were run on them, provided by the sysbench package. There were minute differences in the results of the benchmarks, but overall the execution time was similar:

TABLE I

Synthetic Benchmark Execution Time

Benchmark used	Java server time	.NET server time
--test=cpu	15.4 s	13.0 s
--test=threads	11.8 s	13.2 s
--test=memory	62.0 s	54.0 s

XIII. Testing Results

After ensuring that the performance of the servers was comparable, the tests were run on each of the servers, with 10, 25, 50 and 100 iterations in each test run, each of these runs being tested with 5, 10, 15 and 25 parallel iterations.

A. Test Success Percentage

First, it is important to explore, how well the servers performed under load. This was achieved, by checking how many iterations were scheduled per test type and how many were actually run and had results returned by the APIs. These were considered to be successfully executed.

TABLE II

Test Success Percentage, by Test Type

Test Type	Success Percentage	
	Java	ASP.NET
Connection Check	100.0	100.0
Static Content, Text	100.0	87.67
Maths, Addition	100.0	100.0
Maths, Multiplication	100.0	100.0
Maths, Division	100.0	100.0
Maths, Powers	100.0	100.0
Maths, Logarithms	100.0	100.0
Dynamic Content, Prime Numbers	100.0	100.0
Dynamic Content, JSON Structure, Reading	100.0	100.0
Dynamic Content, JSON Structure, Writing	100.0	100.0
Sorting, Whole Numbers	100.0	100.0
Sorting, Real Numbers	100.0	100.0
LINQ / Streams	100.0	100.0
Cryptography, SHA256	100.0	96.0
Cryptography, MD5	100.0	100.0

B. Test Network Time

Next, it was checked, how well the web server components of the technology stacks performed, to see whether there were any clear disadvantages to using Kestrel or Apache TomEE on GNU/LINUX (average values were checked for all successful iterations, for simplicity).

TABLE III

Test Network Time, by Test Type

Test Type	Time, milliseconds	
	Java	ASP.NET
Connection Check	132	107
Static Content, Text	679	2096
Maths, Addition	212	395
Maths, Multiplication	208	403
Maths, Division	206	377
Maths, Powers	200	403
Maths, Logarithms	201	380
Dynamic Content, Prime Numbers	111	106
Dynamic Content, JSON Structure, Reading	274	356
Dynamic Content, JSON Structure, Writing	148	241
Sorting, Whole Numbers	166	177
Sorting, Real Numbers	251	458
LINQ / Streams	114	124
Cryptography, SHA256	467	844
Cryptography, MD5	405	454

C. Test Processing Time

Lastly, it was checked, how well the actual language runtime performed in processing the request contents, to see, how their performance differed and whether there were any clear advantages to either technology in some use case (average values were checked for all successful iterations, as before).

TABLE IV

Test Processing Time, by Test Type

Test Type	Time, milliseconds	
	Java	ASP.NET
Connection Check	5	2
Static Content, Text	23	11
Maths, Addition	11	1
Maths, Multiplication	8	2
Maths, Division	8	1
Maths, Powers	10	3
Maths, Logarithms	10	2
Dynamic Content, Prime Numbers	5	1
Dynamic Content, JSON Structure, Reading	12	2
Dynamic Content, JSON Structure, Writing	4727	64
Sorting, Whole Numbers	14	2
Sorting, Real Numbers	17	2
LINQ / Streams	11	1
Cryptography, SHA256	28	23
Cryptography, MD5	22	11

XIV. Interpretation of the Test Results

After reviewing the data, a few points of interest appear, which require further explanation.

A. ASP.NET Core Test Success Percentage

While Java was able to serve 100 % of the requests that were made to it, across all levels of parallelisation and test types, this was not the case with ASP.NET Core. Although in most test types its success rate was the same, it was unable to serve all of the requests while generating the large static text responses (over 1 megabyte in size) and while generating the SHA256 hashes.

This can be explained with the hardware constraints (the amount of RAM available to the servers) causing the API runtime to be terminated by the operating system, after exhausting all of the available resources (Fig. 12).

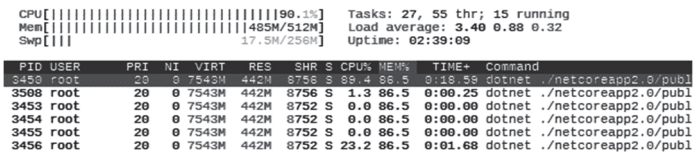

Fig. 8. Output of htop on the server running the ASP.NET Core implementation, shortly before a crash.

While the implementations of the StringBuilder concept, provided by the standard libraries, were used in both implementations, for both Java and ASP.NET Core, it appeared that ASP.NET Core's usage of memory was slightly less optimised; therefore, it could not successfully run the tests with the resources provided. This behaviour was also reproducible for other test types, with increased request for data size.

B. ASP.NET Core Static Text Benchmark Results

It appeared that the ASP.NET Core text generation benchmark not only had worse success rates, but also much greater test execution times, which can be attributed to the network time component of the results – ASP.NET Core took 2096 milliseconds, on average, to receive and send responses, compared to Java's average of 679 milliseconds, a difference by a factor of approximately 3.

One possible explanation is that the Kestrel web server, which is used by ASP.NET Core, currently is worse optimised when compared to Apache Tomcat and Apache TomEE, as it is relatively new. This is evidenced not only by the fact that this particular test involved sending and receiving the largest packets, but also by the fact that the network times for the ASP.NET API implementation were higher in 13 of the 15 benchmarks.

C. Java JSON Write Benchmark Results

Another data point that could attract attention is the average request processing time for the JSON write benchmark, which was run on Java. This average of 4727 milliseconds is not only much longer than that of the ASP.NET Core implementation – 64 milliseconds, but also the longest one in any of the benchmarks run across all of the test types and technologies used.

This could be explained by the fact that for both languages, the JSON for the test requests and responses were both generated and parsed dynamically, in contrast to the more traditional approach of generating static entities. While this sacrifices type safety to a degree, it allows for much greater flexibility and faster development time, which is why this approach was chosen.

This is where a difference between the languages manifests itself – C#, which is the language in which the ASP.NET Core implementation was written, supports the `dynamic` data type, while Java offers no such a feature.

In Java, the Json class was used, which provided support for creating object and array builders, in addition to reading JSON data. This does, however, come with the disadvantage of an object that needs to be created for every item, which must be serialised. This appears to be noticeable when working with a deep, nested structure, because of which the performance degraded, even if it caused no failures.

For example, code in C# can look like

```
List<dynamic> generatedObjectsArray
  = new List<dynamic>();
return generatedObjectsArray.ToArray();
```

While in Java, the following must be done

```
JsonArrayBuilder generatedObjectsArray
  = Json.createArrayBuilder();
return generatedObjectsArray.build();
```

XV. Conclusion

After examining the results, some conclusions can be made, as well as suggestions for improvement of the testing process and methodologies used.

A. Test API Implementations

While using the dynamic data type features to circumvent having to write resource objects for each of the different benchmarks saves time, it can lead to sub-optimal results, as it was in the case of the encountered instability.

Testing should also be done, while following the approach of defining entity classes in the future to see whether the Java performance improves.

Additionally, the performance of the technologies used could be compared on servers running Windows as well, where IIS is also available and could provide different results in relation to Kestrel. For a deeper insight, it would be useful to also log the server resource usage and attempt testing on more powerful hardware configurations, to allow for greater parallelisation and to see how multi-core processor utilisation differs.

B. Runtime

While problems with ASP.NET Core memory management appeared, the overall performance of both technologies was subjectively similar and comparable – neither was noticeably slower or faster than the other in all of the test types.

It would stand to reason, that because of this result, the choice of algorithms utilised to solve problems and, in turn, the code contained in the libraries, which might be used in the development process, could have a greater impact, as opposed

to choosing either of the technologies because of otherwise insignificant differences in performance.

Scientific studies performed in the past [18]–[21], as well as more contemporary attempts at benchmarking [22] seem to indicate that the performance of Java (and Java EE), as well as C# (and thus ASP.NET and ASP.NET Core) depends on particular tasks they are applied to.

While ASP.NET Core was faster at processing the requests, it appeared that the Kestrel web server took longer to deliver the responses in almost all of the cases. It should also be noted that in scenarios where no blocking processes were present, such as waiting for a database to return results, or reading data from a disk, it appeared that a request would spend the majority of the time travelling through the network, as the averages of the processing times were noticeably smaller than those of the network times.

It should be noted that ASP.NET Core is a new technology and is in active development. As such, it is not as mature as Java EE yet, and it is subject to change.

References

[1] Microsoft, "Microsoft to acquire Xamarin and empower more developers to build apps on any device," 2016. [Online]. Available: https://blogs.microsoft.com/blog/2016/02/24/microsoft-to-acquire-xamarin-and-empower-more-developers-to-build-apps-on-any-device/

[2] The Mono Project, "Compatibility". [Online]. Available: http://www.mono-project.com/docs/about-mono/compatibility/

[3] Microsoft, "Announcing ASP.NET Core 1.0," 2016. [Online]. Available: https://blogs.msdn.microsoft.com/webdev/2016/06/27/announcing-asp-net-core-1-0/

[4] Microsoft, "Kestrel web server implementation in ASP.NET Core," 2017. [Online]. Available: https://docs.microsoft.com/en-us/aspnet/core/fundamentals/servers/kestrel?tabs=aspnetcore2x

[5] The Apache Software Foundation, "Apache Tomcat". [Online]. Available: http://tomcat.apache.org/

[6] Oracle, "Java EE at a Glance" [Online]. Available: http://www.oracle.com/technetwork/java/javaee/overview/index.html

[7] Oracle, "JSR 342: Java Platform, Enterprise Edition 7 (Java EE 7) Specification". [Online]. Available: https://jcp.org/en/jsr/detail?id=342

[8] Oracle, "Java Platform, Enterprise Edition 7 (Java EE 7), Web Profile Specification". [Online]. Available: http://download.oracle.com/otn-pub/jcp/java_ee-7-fr-eval-spec/WebProfile.pdf

[9] Microsoft, "Common Language Runtime (CLR)", 2017. [Online]. Available: https://docs.microsoft.com/en-us/dotnet/standard/clr

[10] Microsoft, "Introduction to Razor Pages in ASP.NET Core", 2017. [Online]. Available: https://docs.microsoft.com/en-us/aspnet/core/mvc/razor-pages/?tabs=visual-studio

[11] GitHub, "ASP.NET". [Online]. Available: https://github.com/aspnet

[12] GitHub, "ASP.NET Core". [Online]. Available: https://github.com/aspnet/home

[13] Microsoft, "Getting started with Razor Pages and Entity Framework Core", 2017. [Online]. Available: https://docs.microsoft.com/en-us/aspnet/core/data/ef-rp/intro

[14] The Apache Software Foundation, "Apache TomEE". [Online]. Available: http://openejb.apache.org/apache-tomee.html

[15] Plumbr, "Most popular Java application servers: 2017 edition", 2017. [Online]. Available: https://plumbr.io/blog/java/most-popular-java-application-servers-2017-edition

[16] GitHub, "Chart.js – Simple HTML5 Charts using the <canvas> tag". [Online]. Available: https://github.com/chartjs/Chart.js

[17] npm, "redis". [Online]. Available: https://www.npmjs.com/package/redis

[18] G. A. Francia and R. R. Francia, "An Empirical Study on the Performance of Java/.Net Cryptographic APIs," *Information Systems Security*, vol. 16, no. 6, pp. 344–354, Dec. 2007. https://doi.org/10.1080/10658980701784602

[19] O. Hamed, "Performance Prediction of Web Based Application Architectures Case Study: .NET vs. Java EE.," *International Journal of Web Applications*, vol. 1, no. 3, pp. 146–156, Sept. 2009.

[20] A. Abu-Kamel, R. Zaghal, and O. Hamed, "A Comparison between EJB and COM+ Business Components, Case Study: Response Time and Scalability," *Communications in Computer and Information Science*, pp. 123–135, 2010. https://doi.org/10.1007/978-3-642-14306-9_13

[21] P. Sestoft, "Numeric performance in C, C# and Java," IT University of Copenhagen, Denmark, Feb. 2010

[22] The Computer Language Benchmarks Game, "C# .NET Core vs Java - Which are faster?," 2018. [Online]. Available: https://benchmarksgame-team.pages.debian.net/benchmarksgame/faster/csharp.html

Kristiāns Kronis is a third-year Bachelor's student majoring in Computer Science at Riga Technical University and a Java Developer at Ltd. "Autentica", where he professionally uses a variety of technologies.
He is currently studying the practical application of microservice architecture in web development and containerization platforms, such as Docker.
E-mail: kristians.kronis@edu.rtu.lv

Marina Uhanova graduated from Riga Aviation University, receiving the Bachelor's degree in 1995 and Master's degree in 1996. In 2007, she received the Doctoral degree in System Analysis, Modelling and Development from Riga Technical University. Since 2000, she has worked as an Assistant and Lecturer. Her current employment is an Assistant Professor at the Chair of Software Engineering, Riga Technical University. Her research interests include distributed application development and their applications in insurance.
E-mail: marina.uhanova@rtu.lv
ORCID iD: https://orcid.org/0000-0003-2994-3638