Development and Performance Analysis of RESTful APIs in Core and Node.js using MongoDB Database

Endrit Shkodra, Edmond Jajaga[a] and Mehmet Shala
Department of Computer Science and Engineering, Lagjja Kalabria, UBT Higher Education Institution, Prishtina, Kosovo

Keywords: Core, Node.js, MongoDB, Non-relational Database, Performance.

Abstract: The purpose of this paper is to present a comparative study of Core and Node.js for the development of Representational state transfer (RESTful) Application Programming Interface (API) using MongoDB non-relational database. The study includes Create, Read, Update and Delete (CRUD) functionality, authentication and authorization using the JavaScript Object Notation (JSON) Web Token token as well as the easiness and development time of the two competing technologies. Tests show that in general the performance between the two technologies does not differ much. Different tests indicate that the performance of one technology is better than the other and vice versa. However, Core outperforms Node.js in a test case with large loads.

1 INTRODUCTION

There are a number of technologies used to develop ReST APIs and backend applications, content management systems (CMS), real-time service, different tools and applications etc. Core and Node.js are two popular technologies that can be used for such applications. Both are open source and cross-platform. The first version of Core was released in 2016 which resulted in an increase of the ASP.NET's popularity. Node.js by contract is more mature with its first released appearing in 2009. Node.js Express or more specifically Express is a bookstore which enables the development of RESTful services in Node.js. Both Core and Express are recognized for their speed and power and are highly scalable. They posses corresponding advantages and disadvantages and the choice of one over the other is driven by different needs and requirements. A direct comparison study often offers crucial insights that may help to choose one technology over the other. For example, there are a number of studies that assess the performance of Java EE with ASP.NET (Kronis and Uhanova, 2018; Hamed and Kafri, 2009; Hamed, 2009). However, studies that compare the performance of Core with Express are scarce and the authors are not aware of any of them published in literature to date. Both technologies can operate on relational and non-relational databases to store and retrieve large numbers of records. Relational databases have been around for a long time and are used for structured data when ACID (Atomicity, Consistency, Isolation, Durability) principles need to be enforced. Non-relational databases or NoSQL are particularly suitable in rapid development and when storing large number of unstructured data. Data are easily scalable which makes it the preferred choice for cloud storage and computing. To date, the most popular NoSQL database is the MongoDB with the first version appearing in 2009. There are a number of studies that compare relational and non-relational databases. A general comparison outline of relational databases and MongoDB is given in (Krishnan et al., 2016). A comparative study of MongoDB vs MySQL is undertaken in (Gyorodi et al., 2015) and MongoDB vs SQL in (Parker et al., 2013). The work presented in this paper compares Core with Express using the MongoDB database. The study is split into two parts: implementation and performance comparison. Implementation focuses on speed of development and library utilisation as well as authentication and authorization using JWT token. Development of Representational state transfer (RESTful) Application Programming Interface (API) and the definition of the MongoDB database are described. Performance is judged by execution time using exactly the same test cases with same parameters.

[a] https://orcid.org/0000-0003-1833-5856

Shkodra, E., Jajaga, E. and Shala, M.
Development and Performance Analysis of RESTful APIs in Core and Node.js using MongoDB Database.
DOI: 10.5220/0010621200003058
In *Proceedings of the 17th International Conference on Web Information Systems and Technologies (WEBIST 2021)*, pages 227-234
ISBN: 978-989-758-536-4; ISSN: 2184-3252

2 PROBLEM DEFINITION

Most of high-level programming languages such as C#, Java, PHP, Node.js, Python, etc., contain the implementation of RESTful web services. Given this variety we cannot be sure which implementation is better. Hence, each of them has its advantages and disadvantages. The selection of certain technology for the implementation of RESTful web services depends on various factors based on our needs. In a particular case, a certain technology will perform better, but in another case the performance will suffer. In general, the selection of the choice of RESTful web services is decided taking into account different criteria, such as: (i) the deployment environment, (ii) manner of implementation, (iii) security, which is a very important factor, (iv) availability of developers and their knowledge of working in certain technology, and (v) performance. In this paper we will compare the performance between .NET Core and Node.js Express's implementation of RESTful web services using the MongoGB non-relational database. Thus, the main hypothesis is: .NET Core, as a multi-threaded implementation, outperforms single-threaded Node.js for larger inputs. Hence, the study addresses the following research questions:

1. *Which technology offers the best performance for CRUD functionalities?*
2. *Which technology is more easily implemented?*
3. *How is security implemented using JWT token in these technologies and how much does authentication and authorization affect the performance?*

3 RELATED WORKS

The wide variety of available frameworks often makes it difficult to decide which is more appropriate to be used. This has led to extensive research on their implementation, performance, safety, usability and many other aspects. However, few research works have addressed the comparison between Core and Express. A work presented in (Söderlund, 2017) evaluates the performance of four different frameworks including Core and Express. Various controlled tests have been done indicating that Express has shown a greater sensitivity for large loads. For example, the update test of 1024 users took 189 milliseconds, which is slightly higher. Core performed best from the four tested frameworks. The delay increased by just about 1-2 milliseconds, in cases with 256, 512 and 1024 users. The tests were performed against MySQL database backend, while in the present study a non-relational database is used.

Table 1: Users entity metadata.

Column	Type	Description
_id	string	Unique identifier
name	string	Name
surname	string	Surname
email	string	Email
username	string	Username
password	string	Password
role	string	Role
salt	string	Encryption key

Regarding the advantages of using MongoDB versus MySQL, studies in (Krishnan et al., 2016) and (Truică et al., 2015) show that MongoDB has a shorter execution time compared to MySQL in all major operations such as reading, writing, editing, and deleting. A key feature of MongoDB is the update operation because it is atomic; it is possible to update a particular field as it is done in relational databases (Truică et al., 2015). MongoDB also has more flexible structure that can be tailored to the user's needs. Thus, MongoDB achieves a good performance under heavy read-load and when the application is read intensive.

4 IMPLEMENTATION

In order to target the main hypothesis and research questions, simple APIs for CRUD operations over sample entities as well as authentication and authorization operations using JWT tokens are implemented. The building of the APIs architecture was chosen to be as similar as possible to one another. Then, different load-testing scenarios were executed to analyze the cases in order to establish which technology performs better. The implementation details are described in the following subsections.

4.1 Sample MongoDB Database

Both APIs use the same MongoDB database. A database and a list or a collection similar to a table of relational databases was created to store sample user data as described in Table 1.

4.2 ASP.NET Core API

Table 2 lists the technologies that were used for the development of API in Core. The architecture of the API in Core is a basic and not too layered one. It consists of the following layers:

Table 2: .NET MongoDB components' versions.

Version	**.NET Core 2.2(Core)**
Database	MongoDB v.4.0.6
Other libraries	MongoDB.Driver v2.7.3 AutoMapper v.8.0.0 BCrypt-Core v.2.0.0

- Entities: Defines MongoDB lists or collections as an entity in the Core application layer.
- Services: This layer defines the interface for CRUD operations over the database as well as some other helper functions.
- Models: Sets the data model for the API including data constraints mapping a class with a database entity.
- Controllers: HTTP controllers define methods such as GET, POST, PUT and DELETE to support users URL access to retrieve or send data through the API.

Every MongoDB list or collection, which should be processed from within the application layer, should be defined as entity i.e. C# class, with the necessary attributes (or annotations) and configurations. A sample person entity `User` was created with standard fields including: `Id`, `name`, `surname`, `email`, `username`, `password` and `role`. The attributes that are used for MongoDB lists or collections include: `BsonId`, `BsonRepresentation` with `BsonType` and `BsonElement`. In our case we used `BsonId` to specify `Id` field as unique identifier, `BsonRepresentation` with `BsonType` for the data types and `BsonElement` for indicating which class field is associated with which field in the MongoDB database.

The model is used for enforcing data constraints and validation rules. For example, `name`, `surname` and `password` were set as required, `password` length was set with minimum length of six characters, etc.

In the controllers layer, two controllers were defined: `UsersController`, supporting CRUD features over `Users` entity, and `AuthController`, responsible for authentication features of the API. The controllers hold only HTTP methods such as GET, POST, PUT and DELETE. The business logic related to communication with the database and the generation of JWT token are located in the layer of services i.e. `UserService` and `AuthService`. `UserService` contains the services that have to deal with the `User`, while `AuthService` is responsible JWT token generation for authentication.

A sample API method `Create` of `UserService` that stores `User` data into the database and encrypts the password is described as follows:

Table 3: Node.js MongoDB components' versions.

Version	**Node.js v.10.14.2**
Database	MongoDB v.4.0.6
Other libraries	express v.4.16.4 mongoose v.5.4.19 joi v.14.3.1 config v.3.1.0 bcrypt v.3.0.4 jsonwebtoken v.8.5.1

```
public UserModel Create(UserModel userModel)
{
User user =
Mapper.Map<UserModel,User>(userModel);
string salt =
Bcrypt.BcryptHelper.GenerateSalt(10);
string passwordHashed =
Bcrypt.BcryptHelper.
HashPassword(userModel.Password, salt);
user.Salt = salt;
user.Password = passwordHashed;
_users.InsertOne(user);
return Mapper.Map<User, UserModel>(user);
}
```

Incoming user data are firstly mapped with the `UserModel` model. Then, a key (`salt` string) is generated, which is used to encrypt the password using the Bcrypt library. The user data together with the encrypted password is stored into the database and the user model object is returned to the user. This method is called from the `UsersController`, which firstly checks whether the user already exists in the database.

Core does not need any additional library for supporting JWT authentication and authorization. It requires setting up some configuration parameters. Hence, each route containing the `Authorize` annotation will enforce authenticated access. JWT token is generated using a unique key. It also contains other attributes through which the user's identity is verified.

4.3 Node.js Express API

The Express platform versions used in this paper are depicted in Table 3. The API architecture in Express is also as simple as that of the Core. Unlikely, it is a bit more separate because the difference lies in some configurations, but the basic logic is similar. The architecture consists of the following layers:

- Models: Defines a scheme similar to the one of the MongoDB database, including a function validating data sent by the API user.
- Controllers: Similar to Core Services, this layer defines the interface for CRUD operations over

the database as well as some other helper functions.

- Config: Set global API variables.
- Middleware: Used for setting functions that are intended to authenticate and authorize user requests, but also other configurations for requests such as Cross-Origin Resource Sharing (CORS).
- Router: Similar to Core Controllers, this layer defines the HTTP methods to support users to access through the defined URLs.

Like in Core, the definition of entity in NodeJS Express requires setting up attributes and configurations. The entity is built using the `model` and `Schema` functions of the mongoose library. These functions are used for specifying data fields and data types. In Express we do not define any extra model for data restriction and validation. Instead, we used a function to validate textual or numerical fields, restricting minimum and maximum length and specifying required/optional fields.

Unlike Core, in Express HTTP methods are located in the routing part, while the logic of database communication and authentication is located on the controller. In general, there are different architectural forms of how API can be organized, but it was decided to use a standard and very usable form by the developers.

The following code illustrates the analogous counterpart of the Core's `Create` method:

```
addUser = async (request, response) => {
const { error } = validate(request.body);
if (error) return response.status(400)
.send(error.details[0].message);
let user = await User
.findOne({username:
request.body.username});
if (user) return response.status(400)
.send('User already registered.');
user = new User(request.body);
const salt = await bcrypt.genSalt(10);
user.salt = salt;
user.password = await
bcrypt.hash(user.password, salt);
await user.save();
response.send(user);
```

The logic of the `addUser` function is similar to the `Create` method. Initially, there is a check whether a user with the choosen `username` does already exist in the database. If so, a notification notification is returned to the user, otherwise using the bcrypt library a key is generated, doing hash / encryption of the password. Furthermore, this function is used by the routing's POST method as follows:

```
router.post('/', controller.addUser);
```

Implementation of authentication and authorization in Express is done similarly to Core using only some configurations. The only difference is that in Express we need an additional library *jsonwebtoken*, which enables authentication and authorization using JWT token.

5 RESULTS

In order for the results to be as correct as possible, both APIs are deployed on the same server. The platform used for testing APIs has the following specifications: Intel CPU i7-8565U 4 Core 1.8 – 2.6 GHz, 16 GB DDR4 2400 MHz of RAM and 500 GB SSD NVMe. The simulated server for testing purposes uses Windows 10 Pro, while the application or web server used to set up APIs is Internet Information Services (IIS) version 10.

In the API testing, we used the Apache JMeter application. Apache JMeter is an open source application developed entirely in Java and designed to test different APIs as needed. In our case, different load tests were analyzed for both APIs. The tests were the same or identical for both APIs. Four test scenarios were designated, three of which test specific API features, while the last one tests all the features offered by the API. All test scenarios were executed by simulating different numbers of users within a given time interval. Each test was executed ten times in a row, which means that all the results presented in the tables represent the average of ten executions. The execution time of the requests is presented in milliseconds [ms].

5.1 First Test Scenario

The first scenario tests the HTTP POST and DELETE methods, where the main focus is the POST method, which is used to record data in the database. The scenario consists of a two step process; firstly inserting sample data in the database and secondly deleting the inserted data by its ID. The scenario was evaluated with 10 and 100 users.

In the first case, simulated with 10 users where each user made two calls to the API: one to register a person in the database and another one to delete the person previously registered. As depicted in Figure 1, results show that Core was 10 [ms] faster than Node.js. On the other hand, Node.js' average time for POST request was 156 [ms]. Regarding the deletion of the registered person, we had almost identical time for both technologies with an average time of 3 [ms].

In the second case we simulated 100 users where again each of them made two requests to the API. In other words, 100 users made requests within 5 sec-

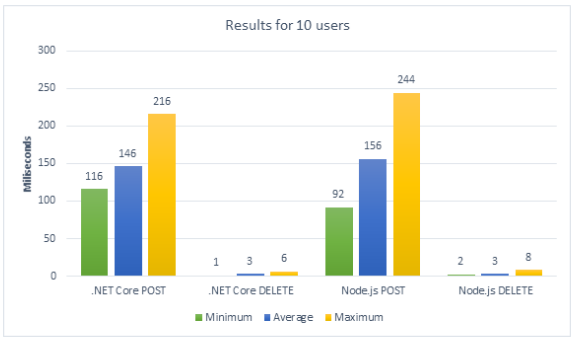

Figure 1: First test scenario – First case.

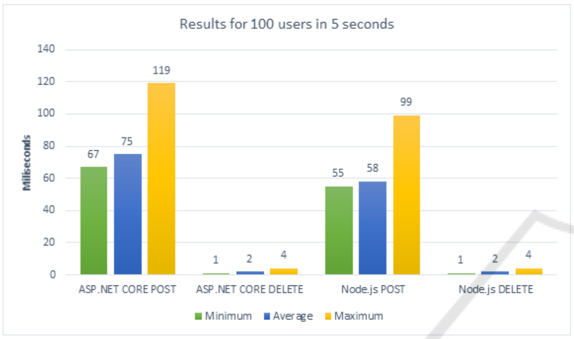

Figure 2: First test scenario – Second case.

onds or 20 users per second. As described in Figure 2, in this case we see a change in results, where Node.js performed better in terms of POST requests with an average time of 58 [ms].

5.2 Second Test Scenario

The main purpose of this scenario is testing the functionality of the authentication and identifying which of the APIs supports faster authentication. Firstly, a new person is registered in the database. Secondly, with the credentials of the newly inserted person a request is made for authentication. After successful authentication the API returns the JWT token as response and the person is deleted from the database. Thus, we have two POST requests in the API and one DELETE request. Similarly to the first scenario, this scenario was evaluated within two cases: with 10 and 100 users.

As depicted in Figure 3, Node.js again performs better for the case when 10 users made person's registration simultaneously, while we have a change in the results of the authentication feature where in this case Core has performed better with an average speed of 19 [ms] faster than Node.js. As for deleting the person from the database, the results are almost the same with an average time difference of only 1 [ms]. In the second case 100 users made requests within 5 seconds

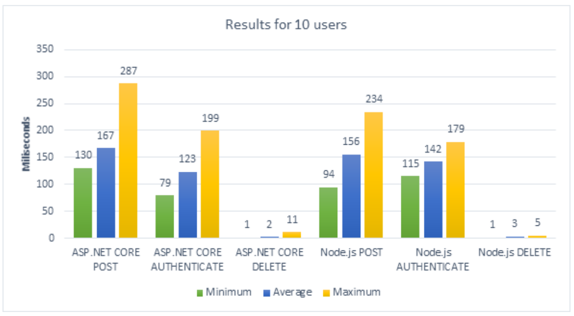

Figure 3: Second test scenario – First case.

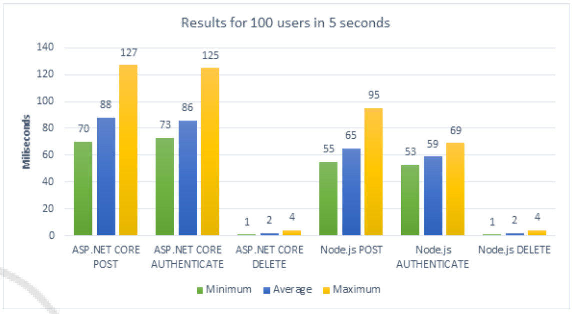

Figure 4: Second test scenario – Second case.

or 20 users per second. Results described in Figure 4 indicate that Node.js performed better in both person insertion and authentication features, while the deletion feature was the same for both technologies.

5.3 Third Test Scenario

The third scenario is similar to the first test scenario only that in this test the main focus is on the HTTP PUT method i.e. updating the data in the database. Again, in this case each user makes three requests to the API: the user inserts a new person, updates some of its attributes i.e. name and surname, and then the updated person is deleted from the database. HTTP methods that are used and tested include POST, PUT and DELETE.

In the first case, when 10 users made requests simultaneously in the API, Node.js performed faster in inserting i.e. running a POST method. Namely, as described in Figure 5, it resulted with 7 [ms] faster in average time. Regarding the PUT request, i.e. executing the update query, which is the main focus of this scenario, the results show that we do not have a big difference where Core in average time was only 1 [ms] better.

In the second case we simulated 100 users, where again each made three requests to the API resulting in 100 users making simultanous requests within 5 seconds or 20 users per second. As shown in Figure 6, results continue to be in favor of Node.js with POST

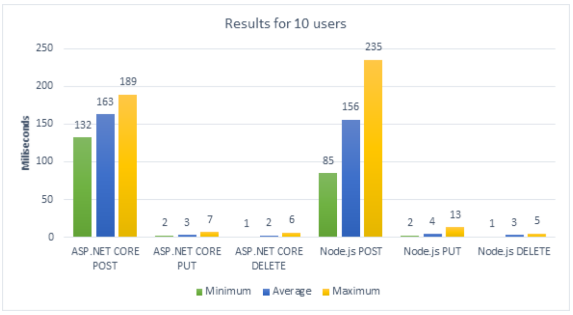

Figure 5: Third test scenario – First case.

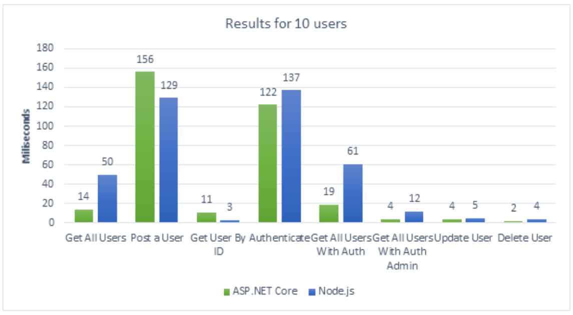

Figure 7: Fourth test scenario – First case.

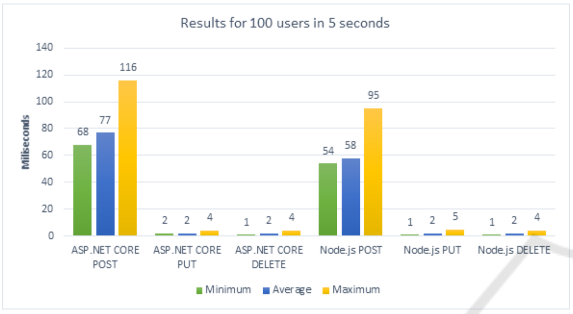

Figure 6: Third test scenario – Second case.

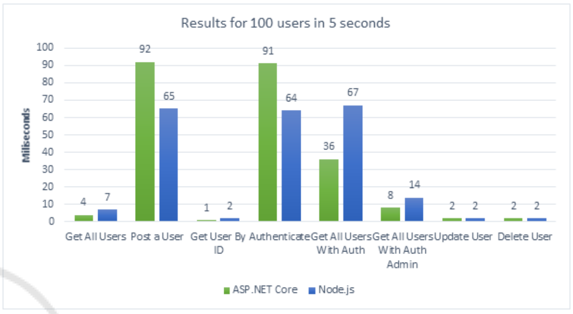

Figure 8: Fourth test scenario – Second case.

requests executed faster with a difference of 19 [ms] on average time. The results of updating and deleting a user are similar with an average time of 2 [ms] per request.

5.4 Fourth Test Scenario

The fourth test scenario includes all the key functionalities that APIs offer. This test was done for four different cases where the only difference is the number of simulated users and the time interval within which the simulated users make requests to the API.

In the first case, the first test consist of 10 users making 8 different requests each, while in the second test 100 records were retrieved from the database. As can be seen from Figure 7 for some methods one technology performs better and for others the other one. However, from the clients point of view the differences are insignificant.In the third test where a specific person data were retreived based on his ID from the database, Express performed better with a difference of only 8 [ms]. However, the generation of the JWT web token has been faster on Core with a difference of 15 [ms] or 122 [ms] total average time. In the other two tests where the records were taken from the database, which required authentication and authorization, Core has shown better performance in the first case for 42 [ms] faster while in the second for 8 [ms].

In the second case, described in Figure 8, 100 users have made eight requests to the API in a time interval of 5 seconds. The results are almost similar with Core slightly faster (only 3 [ms]). As for the POST method Express has come out better with a similar difference as to the first case i.e. 27 [ms]. The authentication process in the second case was performed faster by Express with a difference of 27 [ms] than the authentication in Core. In other two tests, obtaining data from the database requiring authentication and authorization Core performed faster with a difference of 31 [ms] and 6 [ms], resepectively.

In the third case, described in Figure 9, we dealt with 500 users or clients who accessed the APIs in a time interval of 20 seconds and made 8 requests each. Previously, in the test of taking 100 people from the database we had a change, now Express again has performed faster in average for all users. As a bottom line, we can say that in the third case Express has outperformed Core in all tests.

The fourth case is a test that simulates 1000 users or clients who simultaneously in the interval of 30 seconds make requests to the API. In this case the results are shown in seconds in order to be more readable. In this case the difference in the results obtained during the test is much greater. Based on Figure 10, it is clear that Core has outperformed Express in all tests. In the case of receiving 100 people (rows) from the database, in Core took approximately 0.7 seconds faster. The POST request of a new person in the database in this case had better results in Core by a

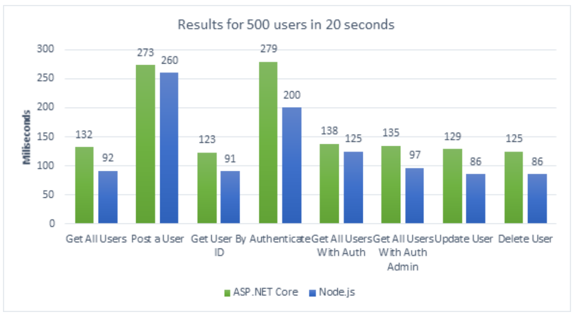

Figure 9: Fourth test scenario – Third case.

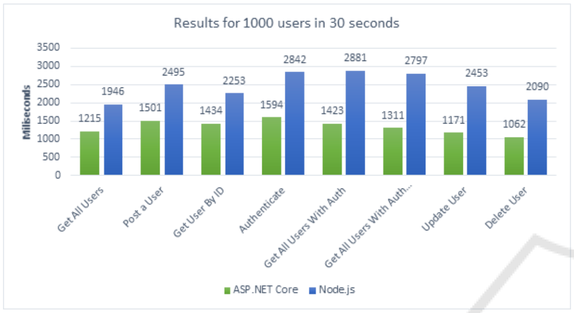

Figure 10: Fourth test scenario – Fourth case.

margin of approximately one second faster. Obtaining a specific person data from the database based on his ID on Core was 0.8 seconds faster than Express. The authentication process which in the first three cases was faster in Node.js, in this case appeared faster in Core with a difference of approximately 1.2 seconds. In the tests for obtaining data which require authentication and authorization Core has performed faster with a very noticeable difference of approximately 1.4 seconds and 1.5 seconds, respectively. Data update and delete in previous cases did not differ too much and in one case were even the same. In this case the difference was quite noticeable for the update resulting in 1.1 seconds versus 2.4 seconds for Core and Node.js, respectively. Similarly, the time of deleting records has pretty noticeable difference of about 1 second in favor of Core.

6 DISCUSSION

In terms of development and implementation of APIs in Core and Express, the differences are not noticeable. Node.js uses JavaScript as a programming language, which when combined with Node.js enables faster development. Frontend developers can also do backend development without any major problems. On the other hand, Core is preferred for large projects because it offers almost ready-made architecture. Although multi-threaded Core is a more advanced programming language, single-threaded Node.js is not far behind.

6.1 Core

Although Core is a new technology that came into use in 2016 by Microsoft it has shown good results during testing. Depending on the increase in the level of testing the changes have been constant with an increase in the number of users and at certain time intervals. In the case with 1000 users the Core shows much better and stable performance compared to Express. From the results presented earlier we note that Core shows better performance in the case when we had smaller time intervals even though the number of users was the same as in Express.

6.2 Node.js Express

Express or more specifically Express, a bookstore which enables the development of RESTful services in Node.js, is older technology than Core. If we look at the results obtained we notice that in the case of the large loads Express was more sensitive giving several times unsatisfactory results compared to Core. Depending on the tests in some cases Express turned out to be better especially in the case of 500 users in a time interval of 20 seconds where in all tests it was faster, but this changed drastically in test case with 1000 users in the interval within 30 seconds. The big difference is attributed to ”single-threaded” Node.js. We also see that the HTTP POST method in many cases shows better performance in the case of registration and authentication.

6.3 Answers to Research Questions

1. *Which of the technologies offers better performance for CRUD functionalities?* The whole logic of an API is almost built on the CRUD or Create (POST), Read (GET), Update (PUT) functionality. Looking at the test results in terms of each method separately depending on the type of test we see that performance changes in favor of one technology to the other. The most obvious is the POST method which in most cases performs better in Express but performance shifts in favor of Core only in the case with large loads. The methods for update or PUT and deletion or DELETE in most of the time have had almost the same speed, a change which is negligible from the user’s point of view. The GET method had better performance in Core overall but there were times when it performed worse compared to Express.

2. *Which of the technologies is most easily implemented?* If we look at the implementation or development of RESTful services in both technologies from the developer's point of view the logic of implementation is similar but it is understood that there are differences on both sides depending on the support libraries which makes further development easier. If the developer is more front-end oriented then obviously the Express would be the best choice because it is developed using JavaScript which is widely used in front-end development. However, if the developer is more back-end oriented with experience in .NET or even Java technologies then Core would be an easier solution due to similarities in the development logic. Additional libraries were needed to enable communication with non-relational databases. Libraries used in Express enabled faster development with less code while the library used in Core in some cases needed a little more complicated logic so that the functions give the same result compared to the libraries in Express.
3. *How is security implemented using JWT token in these technologies and how much does authentication and authorization affect performance?* Implementation of authentication and authorization using JWT token in both technologies is done using the necessary libraries which enable this functionality. The logic is almost the same in both technologies where the JWT token must be configured, where in the configurations it is also determined which secret key will be used for security reasons and then as needed configure the routes which need to have implemented authentication and authorization. If we look at getting 100 users from the database without authentication and with authentication, we notice that we have a slight increase in delay in some cases, in most cases Core has had better performance in calls which have required authentication and authorization.

7 CONCLUSION

Choosing a technology to develop an application or APIs requires consideration of many parameters which affect both performance, security, extensibility, usability, ease of development and implementation, etc. The main purpose of this paper was to provide developers with information on the implemnatation and performance of technologies such as Core and Node.js in conjunction with MongoDB non-relational database. The implementation of a RESTful API in these technologies included basic functions such as Reading, Creating, Changing and Deleting as well as implementing authentication and authorization using JWT token.

If we dwell on the performance of the two technologies by looking at the results obtained we notice that in general the difference in performance when each test is analysed separately. A differing factor is the performance in the case with large loads in a shorter time interval where Core outperforms Node.js. In some other cases and especially in the POST method or recording of database data Node.js performs better than Core. There is no significant difference between the two technologies when changing and deleting data or during Authentication and authorization stage.

REFERENCES

Asp.net core. https://dotnet.microsoft.com/apps/aspnet. Accessed: 2021-05-03.

Experience on working with asp.net (core) and nodejs. https://guillaumejacquart.medium.com/experience-on-working-with-asp-net-core-and-nodejs-5e6c6351fc1f. Accessed: 2021-05-03.

Mongodb. https://www.mongodb.com/. Accessed: 2021-05-03.

Node.js. https://nodejs.org/en/. Accessed: 2021-05-03.

Gyorodi, C., Gyorodi, R., Pecherle, G., and Olah, A. (2015). A comparative study: Mongodb vs. mysql. *2015 13th International Conference on Engineering of Modern Electric Systems (EMES)*, pages 1–6.

Hamed, O. (2009). Performance prediction of web based application architectures case study: .net vs. java ee. *IJWA*, 1:146–156.

Hamed, O. and Kafri, N. (2009). Performance testing for web based application architectures (.net vs. java ee).

Krishnan, H., Elayidom, M., and Santhanakrishnan, T. (2016). Mongodb – a comparison with nosql databases. *International Journal of Scientific and Engineering Research*, 7:1035–1037.

Kronis, K. and Uhanova, M. (2018). Performance comparison of java ee and asp.net core technologies for web api development. *Applied Computer Systems*, 23:37–44.

Parker, Z., Poe, S., and Vrbsky, S. (2013). Comparing nosql mongodb to an sql db.

Söderlund, S. (2017). Performance of rest applications: Performance of rest applications in four different frameworks (dissertation). Retrieved from http://urn.kb.se/resolve?urn=urn:nbn:se:lnu:diva-64841.

Truică, C.-O., Rădulescu, F., Boicea, A., and Bucur, I. (2015). Performance evaluation for crud operations in asynchronously replicated document oriented database.