Performance comparison between Entity Framework Core 6 and Dapper

Mohammed Syamand Yaba

Bachelor's thesis
2023

Tomas Bata University in Zlín
Faculty of Applied Informatics

Tomas Bata University in Zlín
Faculty of Applied Informatics
Department of Informatics and Artificial Intelligence

Academic year: 2022/2023

ASSIGNMENT OF BACHELOR THESIS
(project, art work, art performance)

Name and surname:	**Mohammed Syamand Yaba**
Personal number:	**A19911**
Study programme:	**B0613A140021 Software Engineering**
Type of Study:	**Full-time**
Work topic:	**Výkonnostní porovnání mezi Entity Framework Core 6 a Dapper**
Work topic in English:	**Performance Comparison Between Entity Framework Core 6 and Dapper**

Theses guidelines

1. Prepare the literature review of the thesis topic.
2. Compare Entity Framework Core 6 and Dapper.
3. Design and implement performance tests for Entity Framework Core 6 and Dapper.
4. Compare performance test results.
5. Evaluate and discuss the results.

Form processing of bachelor thesis:	**printed/electronic**

Recommended resources:

1. GORMAN, Brian. Practical Entity Framework Core 6: Database Access for Enterprise Applications. Berkeley, CA: Apress, 2022, 797 pages. ISBN 978-1-4842-7300-5. Available from: https://link.springer.com/book/10.1007/978-1-4842--7301-2
2. SCHWICHTENBERG, Holger. Modern Data Access with Entity Framework Core: Database Programming Techniques for .NET, .NET Core, UWP, and Xamarin with C#. Berkeley, CA: Apress, 2018, 665 pages. ISBN: 978-1-4842-3552-2, Available from: https://vufind.katalog.k.utb.cz/Record/102603
3. RIPPON, Carl. ASP.NET Core 5 and React. 2. Birmingham, UK: Packt, 2021. ISBN 9781800206168. Available from: https://www.packtpub.com/product/asp-net-core-5-and-react/9781800206168
4. DapperLib. Dapper – a simple object mapper for .Net [online]. GitHub, 2022 [cited 2022-06-14]. Available from: https://github.com/DapperLib/Dapper
5. Dapper.NET [online]. RIP Tutorial, 2022 [cited 2022-09-14]. Available from: https://riptutorial.com/dapper

Supervisors of bachelor thesis:	**Ing. Tomáš Vogeltanz, Ph.D.** Department of Computer and Communication Systems
Consultant of bachelor thesis:	**MSc. Luis Antonio Beltran Prieto** Department of Informatics and Artificial Intelligence
Date of assignment of bachelor thesis:	**October 5, 2022**
Submission deadline of bachelor thesis:	**May 12, 2023**

UNIVERZITA TOMÁŠE BATI VE ZLÍNĚ
Fakulta aplikované informatiky
L.S.
①

doc. Ing. Jiří Vojtěšek, Ph.D. m.p.
Dean

prof. Mgr. Roman Jašek, Ph.D., DBA m.p.
Head of Department

In Zlín October 10, 2022

ABSTRAKT

Hlavním zaměřením této práce jsou veřejně dostupné nástroje pro mapování objektů na relační databáze (ORM), konkrétně na Dapper a Entity Framework Core 6. Tyto nástroje jsou často první volbou vývojářů, kteří potřebují spravovat a manipulovat s daty v relačních databázích v .NET. S rozvojem technologií jsou vyvíjeny nové nástroje pro ORM a stávající jsou neustále aktualizovány a zdokonalovány. Cílem této práce je vybrat nejpopulárnější a moderní nástroje pro ORM, Dapper a Entity Framework Core 6, sestavit standardní sadu kritérií pro porovnání a provést podrobnou analýzu na základě těchto kritérií. Výsledkem této studie je informační zdroj pro vývojáře, kteří se chtějí seznámit s výhodami a nevýhodami těchto dvou předních ORM a na základě svých specifických požadavků si vybrat vhodný nástroj.

Klíčová slova:
Objektově-relační mapování (ORM), Entity Framework Core 6, Dapper, .NET

ABSTRACT

The focus of this thesis is on publicly accessible Object-Relational Mapping (ORM) tools, specifically Dapper and Entity Framework Core 6. These tools are often the first choice for developers who need to manage and manipulate data in relational databases in the .NET ecosystem. As technology evolves, new ORM tools are developed, and existing ones are constantly updated and improved. The goal of this thesis is to select the most popular and contemporary ORM tools, Dapper and Entity Framework Core 6, formulate a standard set of criteria for comparison, and perform a detailed analysis based on these criteria. The result of this research aims to be an insightful resource for developers seeking to understand the pros and cons of these two leading ORMs and make an informed choice based on their specific requirements.

Keywords:

Object-Relational Mapping (ORM), Entity Framework Core 6, Dapper, .NET,

ACKNOWLEDGEMENTS

I would like to express my deepest and most heartfelt gratitude to everyone who has been an unwavering pillar of support throughout my academic journey. Your constant love, encouragement, and unwavering belief in my abilities have been instrumental in my achievements and success.

My beloved family and friends, I am sincerely grateful for your unending support, understanding, and encouragement. Your belief in my abilities, even during the most challenging times, has pushed me to overcome obstacles and reach new heights. I am forever blessed to have such an incredible support system in my life.

I am deeply grateful to my esteemed supervisor, Ing. Tomáš Vogeltanz, Ph.D. , for his invaluable guidance, unwavering commitment, and mentorship. His trust, wisdom, and passion have shaped my academic journey profoundly. Additionally, I deeply appreciate Luis Antonio Beltran Prieto, MSc, for his invaluable contributions as my consultant. His insights, expertise, and unwavering support greatly enriched my research.

I hereby declare that the print version of my Bachelor's/Master's thesis and the electronic version of my thesis deposited in the IS/STAG system are identical.

CONTENTS

INTRODUCTION

This thesis explores deeply into the realm of ORM frameworks. In the first part, we will compare the relative merits of two popular solutions in terms of speed: EF Core 6 and Dapper. It is impossible to overstate the significance of ORM frameworks in modern software engineering. These frameworks simplify the process of interacting with databases while simultaneously boosting output. The purpose of this study is to shed light on the extent to which these frameworks may be used in a diverse assortment of use cases, with a particular concentration on how well they perform. Investigating important variables such as query execution time, memory utilization, and scalability allows developers to choose an ORM framework that not only satisfies the requirements of their application but also achieves the highest possible level of performance. The effective management of data is of the utmost importance in our increasingly computerized society. Object-Relational Mapping (ORM) plays a significant role in the process of integrating object-oriented programming languages with relational databases. There are two prominent ORM tools available for use in the context of the.NET framework, Dapper and Entity Framework Core 6. The current investigation intends to fill the gap in knowledge by carrying out a focused analysis that places an emphasis on performance, contrasting Dapper with Entity Framework Core 6. The primary objective of this study is to carry out exhaustive benchmarking tests, with an emphasis on the performance characteristics of several Object-Relational Mapping tools while performing a variety of database operations under a range of different load conditions. The major purpose of this inquiry is to supply the.NET community with a comprehensive grasp of the performance of each Object-Relational Mapping (ORM) tool, thereby facilitating the.NET community's capacity to make well-informed judgements that are tailored to their individual requirements.

I. THEORY

1 LITERATURE REVIEW

Data is a pervasive and essential force that is driving the progression of the digital environment in the current age of technology. To make the most of the opportunities presented by their data, businesses are increasingly relying on data-centric technologies such as big data, data science, and data visualization. Relational database management systems RDBMS are typically the tools of choice for businesses when it comes to effectively storing and managing data. SQL queries are employed to extract data and make use of it so that a variety of business applications may be improved. Developers that work with object-oriented programming languages (OOP) can successfully align with the structure of relational database systems RDB. ORM is a technique that links two disparate methods of organizing and storing data by providing a foundation for doing so. ORM is a valuable resource for tasks that are not too difficult. However, it can be difficult to use it for quick queries due to the complexity of the tool and the difficulty of expressing such queries via code. In my own experience, I've learned that the most effective way to convey complicated questions is through the use of programming languages. This is due to the inherent complexity of these tools and the difficulty of successfully expressing such queries via code [1].

1.1 Object Relational Mapping (ORM)

Since it may make the development process more efficient, object-relational mapping is widely used in the field of web development. This is especially true in circumstances that entail the use of a database. This article provides a definition of ORM, as can be seen in Figure 1 as well as a description of the procedures necessary to implement it in a software application developed with. NET. Within the realm of.NET web development, the idea of ORM is very common knowledge and is likely a requirement for professional roles. Because the use of ORM makes the creation and maintenance of a database easier, it is necessary for developers to acquire a full grasp of this concept [2].

When working with databases and object-oriented programming (OOP) languages, one of the issues that might arise is aligning the programming code with the architecture of the database. ORM is an approach that builds an intermediate layer between the database and the computer language. Because of this, programmers are given the opportunity to modify data in a way that is not constrained by the OOP paradigm. OOP developers need to have a full grasp of the structured query language SQL and be able to write code in SQL fluently to be able to link their application to a SQL database. Developers that are well-versed in SQL are more than capable of developing programs that can access data. It may be a very time-consuming operation for developers to extract data items from code strings in raw SQL. SQL query generators give an extra layer of abstraction to the SQL code, which allows for a more complete representation of the data. Nevertheless, developers are expected to have an in-depth knowledge of SQL and the ability to apply it in an effective manner. Despite the widespread usage of ORMs, which are also known as Object-Relational Mappers, they continue to be the subject of discussion in academic circles. Those who are in favor of object-relational mapping ORM claim that its implementation results in a gain in productivity, an enhancement of application architecture, a better potential for code reuse, and an increased ease of application maintenance over an extended period. According to the findings of study carried out by specialists in the industry, the efficacy of ORMs is seen as a disadvantage by a lot of people. This article's goals are to provide a full explanation of Object-Relational Mapping ORMs, compare ORMs and SQL tools, and discuss the relative benefits and drawbacks of each of these two categories of software. This article's goal is to provide the reader with sufficient information to enable them to establish a knowledgeable assessment of the suitability of employing ORMs for the purpose of constructing database applications in their separate projects. This assessment will be made possible by providing the reader with[3].

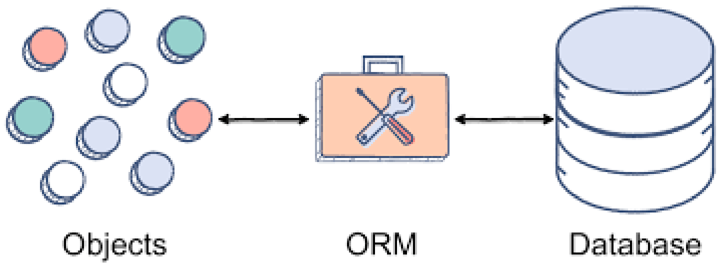
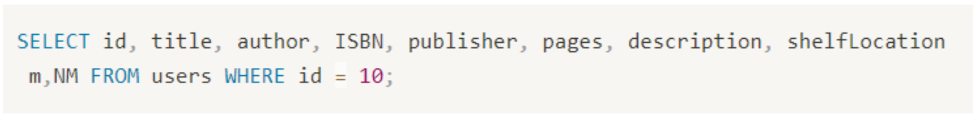


Figure 1: Architecture of ORM [29]

The use of object-relational mapping technology offers a considerable advantage in terms of shortening the amount of time required for development and removing SQL code that is not needed or repetitive. An extra perk is gained by having the capability to monitor changes made to databases. There are ORM-specific libraries that log every alteration that is made to a database and maintain track of the history of those changes[4].

1.1.1 ORM Tools for .NET

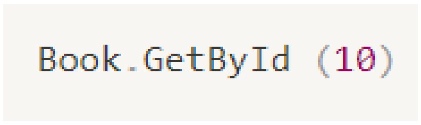
An ORM utility, is a piece of software that was designed to assist developers of Object-Oriented Programming in their interactions with relational databases. The creation of a one-of a kind ORM software solution from the ground up could be avoided if one chooses to make use of the technologies instead. The example SQL code that follows is one way that data may be retrieved from a database that is specific to a book table by using the SQL as shown in Figure 2.



```
SELECT id, title, author, ISBN, publisher, pages, description, shelfLocation
 m,NM FROM users WHERE id = 10;
```

Figure 2 :The SQL script provides an example of SQL code for retrieving data from a specific book table in a database.

The code in Figure 2 retrieves a specific field from the Book's table. Using the WHERE clause, it was specified that the data must come from a subscriber with id 10. On the other hand, an ORM tool may conduct a similar query by making use of simplified approaches to do the task. To put it another way:



```
Book.GetById (10)
```

Figure 3 : Retrieving specific data using ORM.

Therefore, the piece of code in Figure 3 is equivalent to the SQL query. It is essential to recognize that each Object-Relational Mapping tool is developed in a distinctive way, which results in differences in the approaches that they take. However, the primary purpose that each of these technologies was designed to accomplish has not changed[5].

1. **Entity Framework:** The EF Core is a data access technology that is open source, cross-platform, flexible, and lightweight. It serves as a version of the Entity Framework, which is extensively used[6].
2. **nHibernate:** is an established ORM that is open source and de-signed for utilization within the .NET framework. The software is currently undergoing active development[7].

3. **Dapper:** is an open-source ORM framework that can be used in.NET and.NET Core programmers to map objects to databases[8].
4. **Base One Foundation Component:** The toolkit is a solution that is very efficient for accelerating the process of developing database applications on the Windows and ASP.NET platforms, while also guaranteeing that these applications are secure and fault tolerant. In conjunction with the integrated development environment IDE that is provided by Microsoft's Visual Studio[9].
5. **iBatis:** is a data mapping system that simplifies the integration of relational databases with OOP[10].
6. **LINQ to SQL:** is a runtime architecture that was included in version 3.5 of the.NET Framework by Microsoft. Its objective is to facilitate the management of relational data in the form of objects[11].
7. **nHydrate:** is a solution available on the Microsoft.NET platform. This solution provides a framework for mapping.NET objects to relational database tables. When it comes to assembling persistence domains, one of the most monotonous tasks for software developers is often doing so with the help of this design[12].

1.1.2 Why an ORM is Needed

Even while ORMs may be of aid, it is essential to see them to an end rather than the goal themselves. It is essential to factor in the possibility of making concessions, since the usefulness of these things could not be applicable in every situation. If a significant number of object-oriented functions of a programming language are being used to manage many states, then it is possible that Object-Relational Mapping, also known as an ORM, is the most suitable solution to use.

When dealing with objects that have complex inheritance links, the manual handling of state management might provide difficulties in terms of accounting for its ramifications. The management of adjustments to the data structure may be made easier thanks to the schema migration functionality's ability to aid. This, in turn, will make it simpler for the developer to start a project.

ORM has the potential to be beneficial yet, it is not free of drawbacks and does have certain restrictions. When compared to the original state, the employment of an ORM could result in a higher degree of abstraction being achieved. which will make the process of debugging more difficult. It is possible that the representation provided by the ORM, which is used to facilitate communication between the database and the application, may not be totally exact or may mistakenly divulge information about the application's internal implementation. This scenario might occur in several different scenarios.

These concerns could become problematic in certain contexts and situations. When designing software, it is necessary to have a thorough understanding of the project's requirements as well as its preferences on the distribution of resources. An approach to software engineering known as ORM is one that makes it simpler to develop applications that are driven by a database. The apparent advantages that an Object-Relational Mapping (ORM) framework may give to an application that is currently under construction are ultimately what determines whether or not a software development project will include the use of an ORM framework[13].

1.1.3 Advantages of ORM

1. Enhance the productiveness of the development process while reducing costs. Making development more object-oriented
2. It offers portability by supporting multiple databases and programming languages.
3. It is possible to implement new features and capabilities without much difficulty, such as the capacity to cache data[14].

1.1.4 Disadvantages of ORM

1. It is typical for developers to have lower levels of productivity when they are going through the process of learning how to program using Object-Relational Mapping (ORM).
2. Developers tend to have a lessening understanding of the real operation of the code; the usage of SQL supplies developers with more authority is typically a sluggish process.
3. When compared to SQL queries, the Object-Relational Mapping (ORM) technique is insufficient for processing sophisticated queries[15].

1.2 DAPPER

Sam Saffron, a veteran software developer at Stack Overflow, is the person responsible for the creation of Dapper. It was released in 2011. It was first built for production use at Stack Overflow with the intention of improving performance with the understanding that LINQ to

SQL was not considered to be sufficiently efficient during that time[16]. Dapper is open source and publicly accessible. It was developed to work with.NET and.NET Core software applications. The library eliminates the need for developers to engage in laborious coding by providing a quick and simple interface for accessing data stored in databases. Dapper offers a wide variety of functions, such as the capability to carry out stored procedure executions, map results to objects, and carry out raw SQL query operations. The NuGet package may be obtained without much difficulty and is easily accessible. Because it is both quick and lightweight. The object mapping tool is a professional solution for any.NET language, such as C#, that allows developers to effectively map query results from ADO.NET data readers to business object instances in an easy way. This is made possible by the object mapping tool's support for any.NET languages. The platform provides extensive support for combining several database searches into a single invocation, and it also makes it possible to conduct synchronous and asynchronous database queries simultaneously. In addition to this, Dapper makes it easier to perform parameterized queries, which is another action that can be taken to protect against the possibility of SQL injection issues[17].

1.2.1 Advantages of Dapper

1. The ORM framework known as Dapper is widely regarded as being among the most time-efficient of its kind. It’s functional for both elementary and advanced computations.
2. Establishing a connection to the database and gaining access to its features is made easier with the help of the IDBConnection object, which is responsible for facilitating the execution of these activities. The database system is SQL Query compatible.
3. Fewer lines of code are needed to establish a connection to the database.
4. It allows users to input data in bulk[18].

1.2.2 Disadvantages of Dapper

1. Dapper’s major shortcoming is its inability to construct a class model automatically.
2. It is unable to monitor objects or their associated modifications.
3. Manual SQL query creation and maintenance are required.
4. While the raw dapper library doesn't support CRUD operations, it need to utilize the contribute library as a separate package to complete the job [19].

1.2.3 Importance of Dapper

1. One of the most notable qualities of this specific micro-ORM is its lightning-fast speed.
2. Reduces by a significant margin the amount of code that must be written in order to access the database.
3. The programme is equipped with the capacity to participate in cooperative endeavours with a wide number of databases, including SQL Server, Oracle, SQLite, MySQL, and PostgreSQL, amongst other potential choices.
4. The handling of SQL queries and stored procedures has been made easier to understand and use thanks to this achievement.
5. The technology is able to help with a single as well as several requests at the same time.
6. This feature makes it easier to get several datasets all at once depending on a variety of inputs[20].

1.2.4 When to Use Dapper

Utilizing Dapper unquestionably results in an increase in performance, which is the fundamental advantage of doing so. When circumstances call for exceptional performance, Dapper could be thought of as the best option to choose. It is essential to keep in mind that the use of Dapper may call for a longer period of development since it requires the construction of SQL queries that will be carried out by the framework. Taking this into consideration is essential. In addition, the platform does not come equipped with a native capacity for automatic migrations, unlike EF Core, which offers this feature. This implies that all essential construction or alteration of databases and tables must be accomplished using SQL scripts, regardless of whether they are being created or modified[21].

1.2.5 Versions of Dapper

The below table show the only 2 versions of dapper that are available as can be observed in Table 1. This gives an idea of which version the user prefers. There are many sub versions for users to download, but most are the same after the new version 2.0.0. It is very clear that the version which is used by the application (v2.0.x) is the most used version by other programmers around the world which is also the newest one.

Table 1: Versions of Dapper [22]

Version	Downloads
2.0.X	140,030,281
V1.0.x	83,882,346

The above tables show the only 2 versions of dapper that are available. This gives an idea of which version the user prefers. There are many sub versions for users to download, but most are the same after the new version 2.0.0. It is very clear that the version which is used by the application (v2.0.x) is the most used version by other programmers around the world which is also the newest one. This is also shown in the chart form in Figure 4: The most popular and recent version used worldwide is v2.0.x.

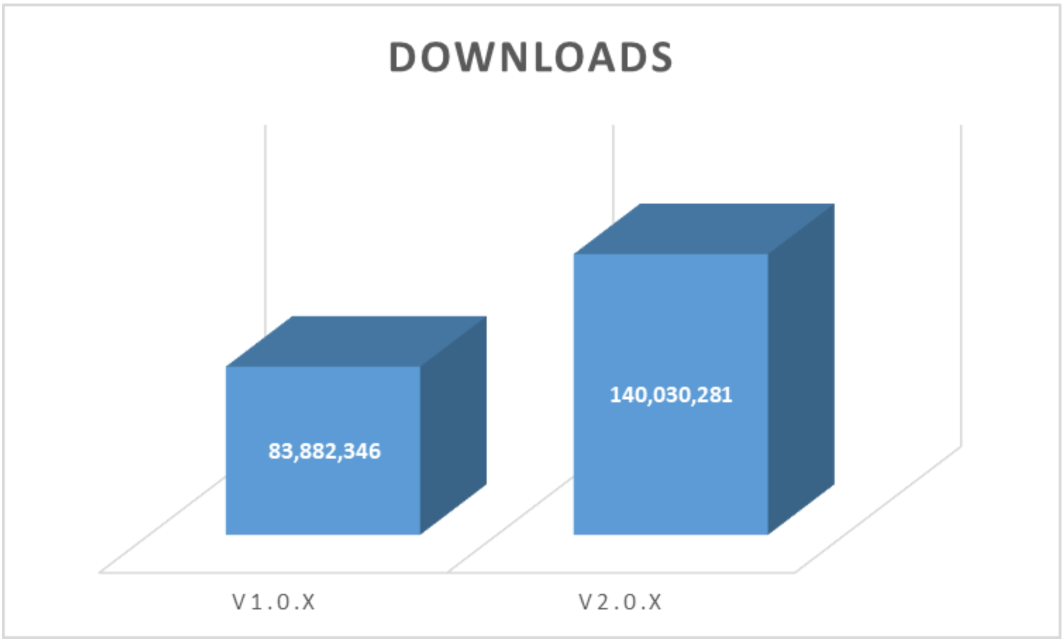


Figure 4: The most popular and recent version used worldwide is v2.0.x.

1.3 ENTITY FRAMEWORK CORE

For.NET applications, developers may access databases using the EF Core Object Relational Mapping ORM framework. The framework is a development tool that improves ADO.NET and gives programmers an automated approach to retrieve and save data in a database. The key difference between EF Core and its predecessor is that this upgraded version is interoperable across Linux, OS, and Windows. This framework offers a DbContext feature that sits in between the domain and database. It functions as a bridge between the domain model and the database, enabling developers to easily define how mappings operate. This feature provides a connection to a database and can be used to query and store instances of the entities. It allows software developer teams to work more effectively [23].

1.3.1 Advantages of Entity Framework Core

1. It has been shown that making use of EF has a favorable influence on the amount of time spent on development as well as the cost.
2. The platform offers automatically produced code and gives developers the ability to graphically construct models and map databases. In addition, the platform offers a visual mapping feature.
3. This functionality makes it possible to map business items in an easy and straightforward manner.
4. The use of .NET Applications makes the speedy execution of CRUD tasks simpler and more convenient [24].

1.3.2 Disadvantages of Entity Framework Core

1. The use of lazy loading is one of the most significant drawbacks associated with EF.
2. One notable aspect of its syntax is its intricacy.
3. The logical schema demonstrates a limitation in terms of its ability to grasp business entities and the interrelationships between them.
4. A model with a big domain may not be the best option [25].

1.3.3 Importance of EF

The EF gives developers the ability to deal with data in the form of domain-specific objects and attributes, such as customers and customer addresses, without having to worry about the underlying database tables and columns where this data is kept. In contrast to conventional programs, data-oriented applications may be built and maintained with fewer code thanks to the EF, which allows developers to work at a higher degree of abstraction when working with data. Entity Framework applications may be used on any machine that has the.NET Framework installed since the EF is a part of the.NET Framework, beginning with version 3.5 SP1 [26].

1.3.4 Determining optimal use case for Entity Framework in software development

The added support for LINQ that is provided by the EF Core framework helps to make the process of building queries to run against the information that is stored in the database more straightforward. Instead of using SQL or any other kind of query language, it is recommended to use LINQ since it enables the construction of queries in the C# programming language [27]. This is an advantage over using any other query language. Because it makes it easier to save data and retrieve it from memory, the EF Core In-Memory Database Provider is well suited for use in programs that need to store data temporarily or for testing reasons. This is because it offers support for in-memory storage. An in-memory database is a very handy alternative for use in unit testing due to its expeditious setup and characteristics for speedy processing. This makes it an excellent choice to consider [27].

1.3.5 Versions of Entity Framework Core

The table below represent the 7 versions of Entity Framework Core as shown in Table 2. The most used versions are around version 6 and below inclusively. This can be due to many reasons but mainly it takes time and testing to adapt the software to newer versions of EF Core. Which means that not many users/programmers have adapted to the newer versions. At the time of creating this application, version 6.0.0 was the one available at hand and the most used version. This is also shown in the chart form in Figure 5.

Table 2 : Versions of Entity Framework Core [28]

Version	Downloads
V8.0.x	110,020
V7.0.x	20,190,582
V6.0.x	152,534,525
V5.0.x	126,200,671
V3.0.x	191,753,032
V2.0.x	143,462,726
V1.0.x	9,425,062

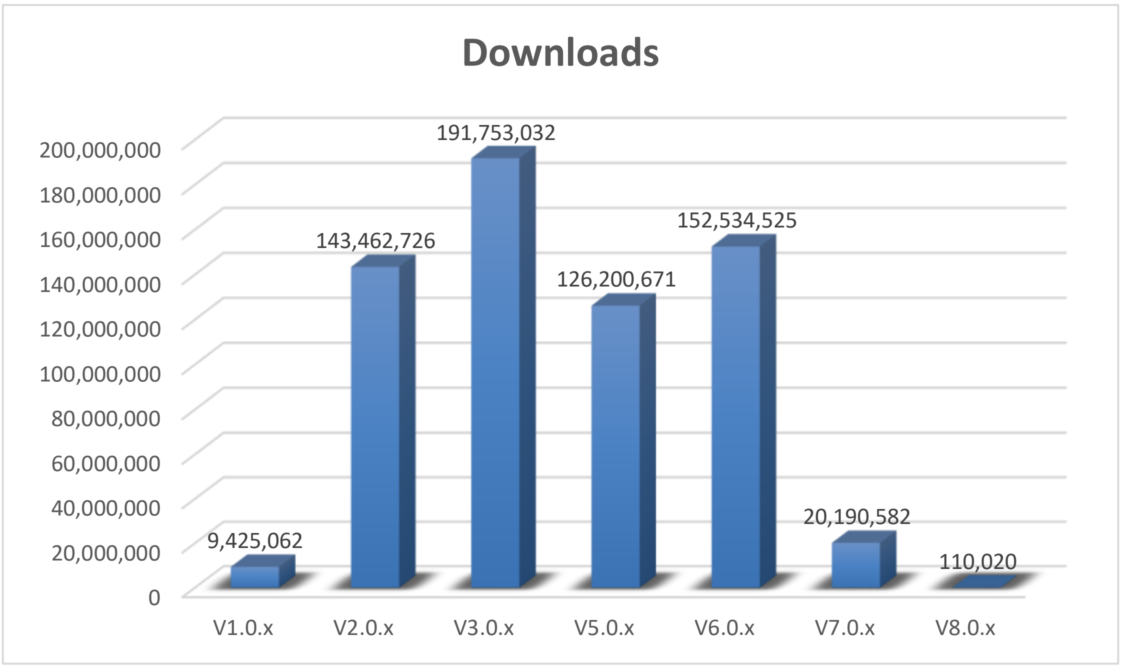


Figure 5 : Number of versions downloaded by users.

II. ANALYSIS

2 PERFORMANCE COMPARISON

There are 2 versions of rest API's Dapper and EF core 6. Their job is to connect the backend to the frontend of the application. The main jobs of the REST APIs are to use the functionalities of CRUD operation to send information to the database and vice versa.

There are 3 benchmarks used in the application: one is the open-source benchmark that was imported through an application, the other was created manually to do the job at hand and *`benchmarkDotNet`*. All benchmarks are used to calculate the times of the CRUD functionality timing. There is a performance counter that is created in the program to do the timing of the transactions and calculate the memory usage of the endpoints. There are 1000 transactions that are sent to the database and back, so that the performance counter can calculate the data.

Both implementations, the applications and the JMeter, have a performance counter and are compared to see which framework can execute the endpoints faster. In addition to the two implementations that calculate everything for us, there is also a third that is imported to be compared for better results.

2.1 Methodology

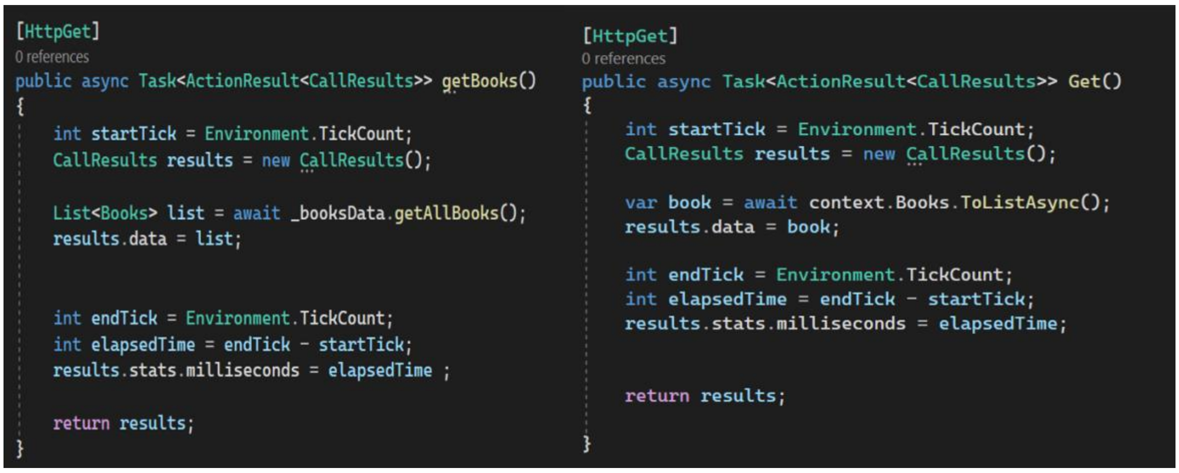
The methods that have been used to obtain our results are Read, Create, Update and Delete. Many may already know the function of each method, but we will give a brief detail about each.

The first method is Read, which calls and gets the information from the database. For example, getting to read a product's detail or reading/getting a user's details.

Second Creating, which adds data to a database which has not been there or has not existed. An example of creating would be adding a product to an inventory or creating some user in the database.

Third which is the update. This deletes the already existing information and replaces it which something else such as updating a product's price in the inventory.

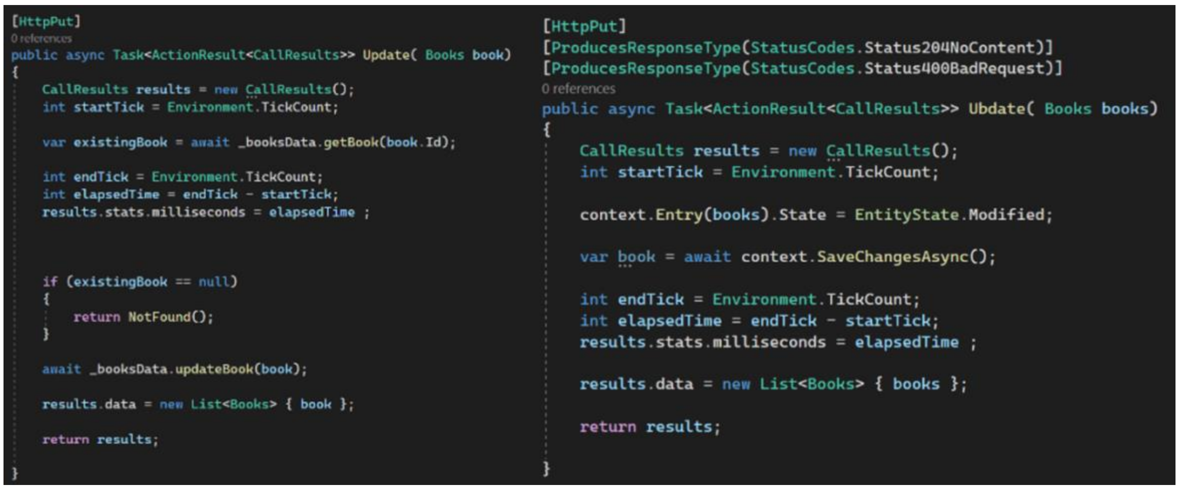
Finally deleting, the operation is for removing something that already exists in the database. For example, permanently removing a product that does not exist in the inventory anymore. In Figure 6, Figure 7, Figure 8, Figure 9, Figure 10 both APIs we can see that the endpoint retrieves books from the database. It starts to count between the starting time laps and the ending time, then it gets the difference of both. Then it shows the result stats in milliseconds. In the following figures, we can see these endpoints post, put, get by ID, and delete respectively. In all these endpoints there is one operating that is common between all five which is calculating the time. As the results are already shown below, these five endpoints are distinct from one another as compared to one another., each giving a different result. This clarifies the case of dapper vs EF6 by showing that dapper has the higher hand in almost every aspect of the procedure.



```
[HttpGet]
0 references
public async Task<ActionResult<CallResults>> getBooks()
{
    int startTick = Environment.TickCount;
    CallResults results = new CallResults();

    List<Books> list = await _booksData.getAllBooks();
    results.data = list;


    int endTick = Environment.TickCount;
    int elapsedTime = endTick - startTick;
    results.stats.milliseconds = elapsedTime ;

    return results;
}
```

```
[HttpGet]
0 references
public async Task<ActionResult<CallResults>> Get()
{
    int startTick = Environment.TickCount;
    CallResults results = new CallResults();

    var book = await context.Books.ToListAsync();
    results.data = book;

    int endTick = Environment.TickCount;
    int elapsedTime = endTick - startTick;
    results.stats.milliseconds = elapsedTime;


    return results;
}
```

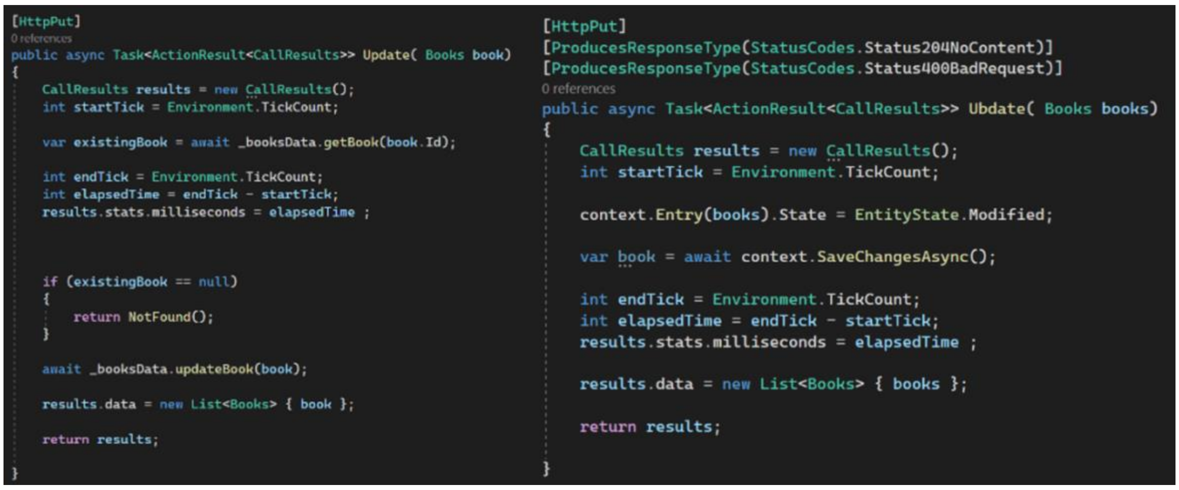
Figure 6 : Endpoint for GET method for retrieving books for Dapper and Entity Framework.



```
[HttpPut]
0 references
public async Task<ActionResult<CallResults>> Update( Books book)
{
    CallResults results = new CallResults();
    int startTick = Environment.TickCount;

    var existingBook = await _booksData.getBook(book.Id);

    int endTick = Environment.TickCount;
    int elapsedTime = endTick - startTick;
    results.stats.milliseconds = elapsedTime ;



    if (existingBook == null)
    {
        return NotFound();
    }

    await _booksData.updateBook(book);

    results.data = new List<Books> { book };

    return results;

}
```

```
[HttpPut]
[ProducesResponseType(StatusCodes.Status204NoContent)]
[ProducesResponseType(StatusCodes.Status400BadRequest)]
0 references
public async Task<ActionResult<CallResults>> Ubdate( Books books)
{
    CallResults results = new CallResults();
    int startTick = Environment.TickCount;

    context.Entry(books).State = EntityState.Modified;

    var book = await context.SaveChangesAsync();

    int endTick = Environment.TickCount;
    int elapsedTime = endTick - startTick;
    results.stats.milliseconds = elapsedTime ;

    results.data = new List<Books> { books };

    return results;

}
```

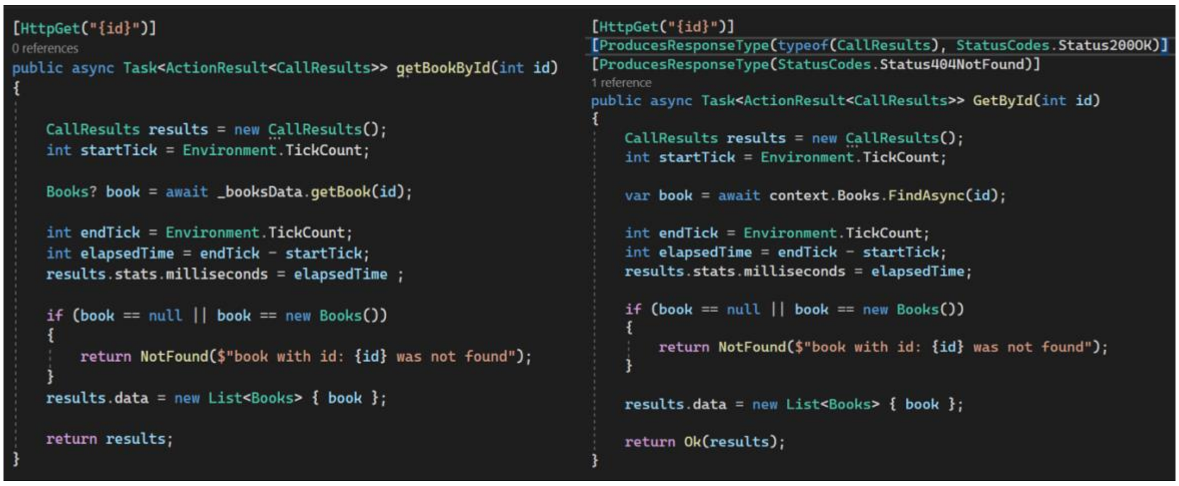
Figure 7 : Endpoint for POST method for inserting book into the database for Dapper and Entity Framework.



```
[HttpPut]
0 references
public async Task<ActionResult<CallResults>> Update( Books book)
{
    CallResults results = new CallResults();
    int startTick = Environment.TickCount;

    var existingBook = await _booksData.getBook(book.Id);

    int endTick = Environment.TickCount;
    int elapsedTime = endTick - startTick;
    results.stats.milliseconds = elapsedTime ;



    if (existingBook == null)
    {
        return NotFound();
    }

    await _booksData.updateBook(book);

    results.data = new List<Books> { book };

    return results;

}
```

```
[HttpPut]
[ProducesResponseType(StatusCodes.Status204NoContent)]
[ProducesResponseType(StatusCodes.Status400BadRequest)]
0 references
public async Task<ActionResult<CallResults>> Ubdate( Books books)
{
    CallResults results = new CallResults();
    int startTick = Environment.TickCount;

    context.Entry(books).State = EntityState.Modified;

    var book = await context.SaveChangesAsync();

    int endTick = Environment.TickCount;
    int elapsedTime = endTick - startTick;
    results.stats.milliseconds = elapsedTime ;

    results.data = new List<Books> { books };

    return results;

}
```

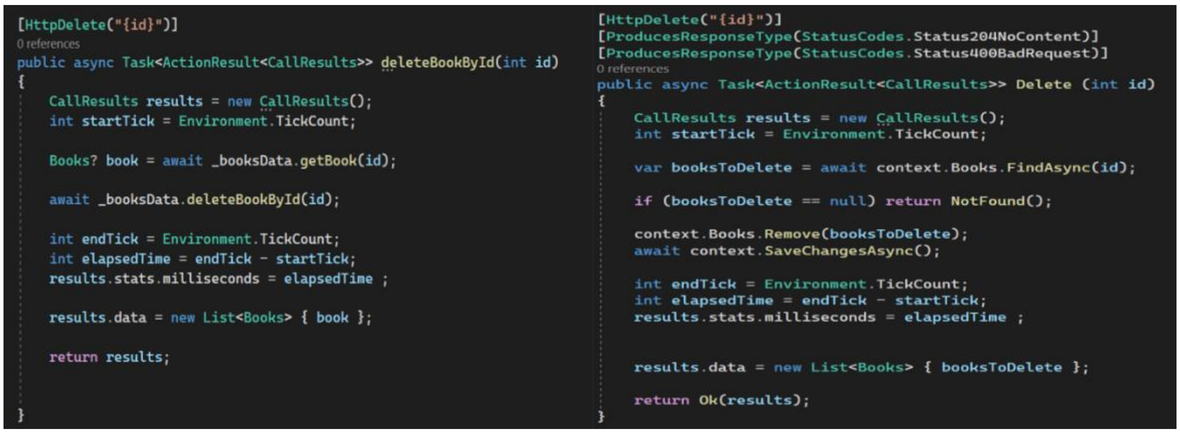
Figure 8 : Endpoint for PUT method updating book info for Dapper and Entity Framework.



```
[HttpGet("{id}")]
0 references
public async Task<ActionResult<CallResults>> getBookById(int id)
{

    CallResults results = new CallResults();
    int startTick = Environment.TickCount;

    Books? book = await _booksData.getBook(id);

    int endTick = Environment.TickCount;
    int elapsedTime = endTick - startTick;
    results.stats.milliseconds = elapsedTime ;

    if (book == null || book == new Books())
    {
        return NotFound($"book with id: {id} was not found");
    }
    results.data = new List<Books> { book };

    return results;
}
```

```
[HttpGet("{id}")]
[ProducesResponseType(typeof(CallResults), StatusCodes.Status200OK)]
[ProducesResponseType(StatusCodes.Status404NotFound)]
1 reference
public async Task<ActionResult<CallResults>> GetById(int id)
{
    CallResults results = new CallResults();
    int startTick = Environment.TickCount;

    var book = await context.Books.FindAsync(id);

    int endTick = Environment.TickCount;
    int elapsedTime = endTick - startTick;
    results.stats.milliseconds = elapsedTime;

    if (book == null || book == new Books())
    {
        return NotFound($"book with id: {id} was not found");
    }

    results.data = new List<Books> { book };

    return Ok(results);
}
```

Figure 9 : Endpoint for GETById method getting book by id for Dapper and Entity Framework.



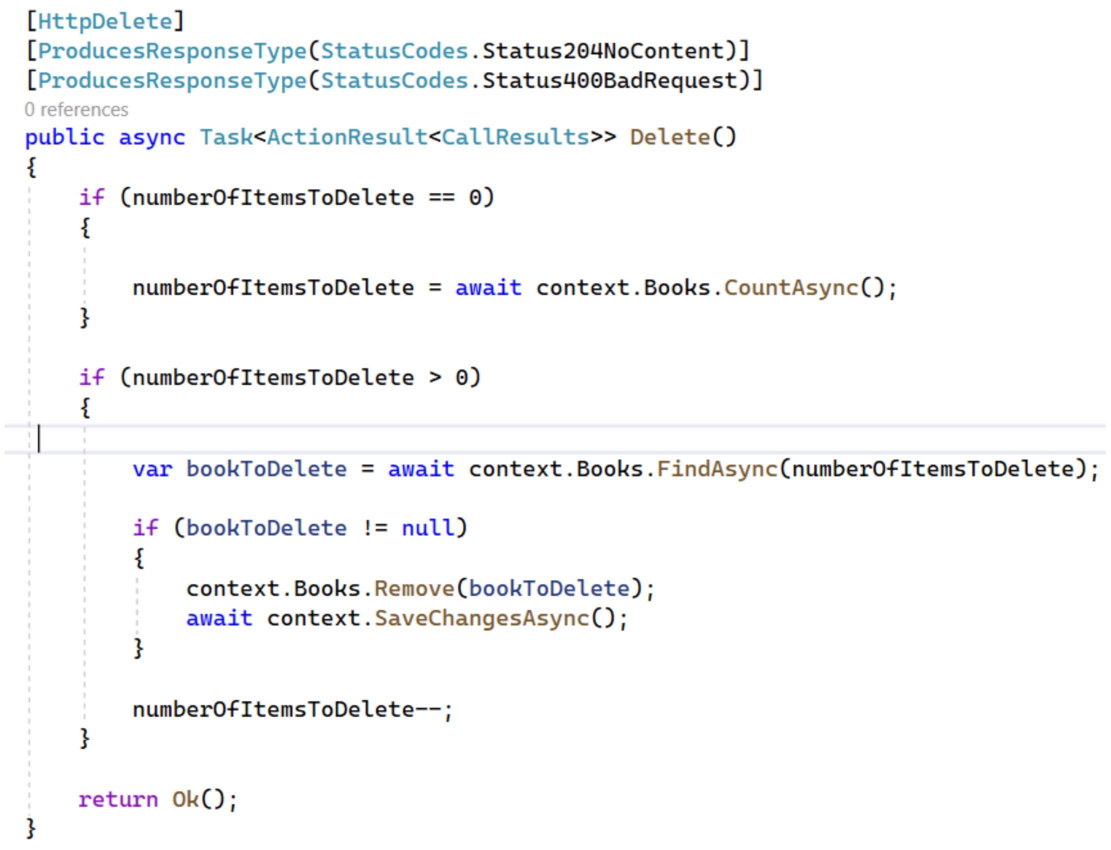
```
[HttpDelete("{id}")]
0 references
public async Task<ActionResult<CallResults>> deleteBookById(int id)
{
    CallResults results = new CallResults();
    int startTick = Environment.TickCount;

    Books? book = await _booksData.getBook(id);

    await _booksData.deleteBookById(id);

    int endTick = Environment.TickCount;
    int elapsedTime = endTick - startTick;
    results.stats.milliseconds = elapsedTime ;

    results.data = new List<Books> { book };

    return results;


}
```

```
[HttpDelete("{id}")]
[ProducesResponseType(StatusCodes.Status204NoContent)]
[ProducesResponseType(StatusCodes.Status400BadRequest)]
0 references
public async Task<ActionResult<CallResults>> Delete (int id)
{
    CallResults results = new CallResults();
    int startTick = Environment.TickCount;

    var booksToDelete = await context.Books.FindAsync(id);

    if (booksToDelete == null) return NotFound();

    context.Books.Remove(booksToDelete);
    await context.SaveChangesAsync();

    int endTick = Environment.TickCount;
    int elapsedTime = endTick - startTick;
    results.stats.milliseconds = elapsedTime ;


    results.data = new List<Books> { booksToDelete };

    return Ok(results);
}
```

Figure 10 : Endpoint for DELETE method for Dapper and Entity Framework.

The implementations of the endpoints below, are the second version of the APIs that are only used for the JMeter. The reason for this is that the new version does not require an id for each item in the database.

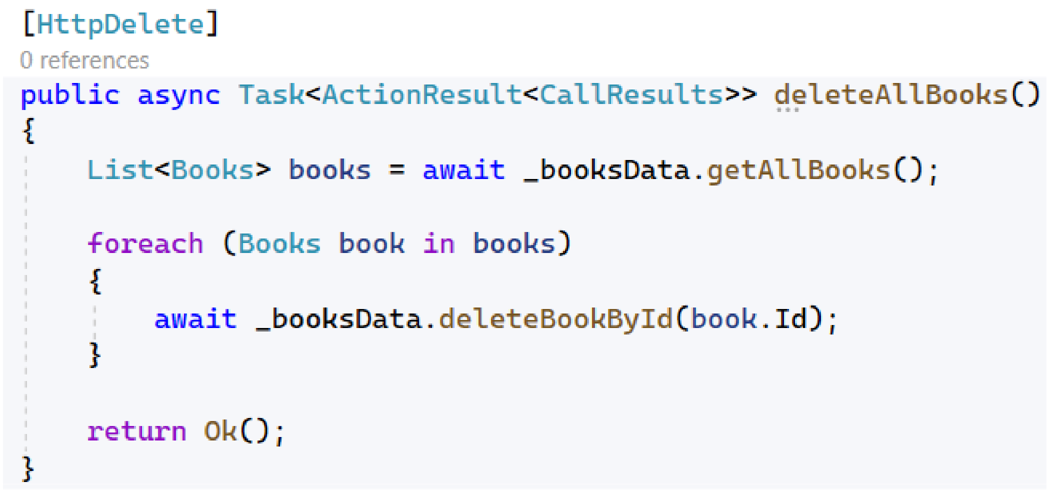
The ASP.NET Core class for EF core method CountAsync() is an example of an ASP.NET Core class method written in C#. The HttpDelete property indicates that the function's role is to handle HTTP DELETE requests, and the ProducesResponseType parameters provide many forms of HTTP responses as the implementation can e observed in Figure 11. The function returns an object of the type ActionResult CallResults, and checks the current value of the numberOfItemsToDelete variable. The employment of Ef Core in conjunction with the execution of the CountAsync() function is a standard procedure in relation to the administration of databases, allowing for the asynchronous retrieval of the count of books. The software will delete a book if the value of numberOfItemsToDelete is larger than zero. To do this, the FindAsync() function is used to obtain the book and provide the argument numberOfItemsToDelete. The removal of books is accomplished by using the Remove() function and the asynchronous saving of changes to the database by utilising the SaveChangesAsync() method. The value of numberOfItemsToDelete is decreased and the function produces an Ok() result. It is important to note that in the absence of the whole class and its elements, it may be difficult to appreciate the aim and behaviour of this code in its totality.



```
[HttpDelete]
[ProducesResponseType(StatusCodes.Status204NoContent)]
[ProducesResponseType(StatusCodes.Status400BadRequest)]
0 references
public async Task<ActionResult<CallResults>> Delete()
{
    if (numberOfItemsToDelete == 0)
    {

        numberOfItemsToDelete = await context.Books.CountAsync();
    }

    if (numberOfItemsToDelete > 0)
    {

        var bookToDelete = await context.Books.FindAsync(numberOfItemsToDelete);

        if (bookToDelete != null)
        {
            context.Books.Remove(bookToDelete);
            await context.SaveChangesAsync();
        }

        numberOfItemsToDelete--;
    }

    return Ok();
}
```

Figure 11 : Endpoint for deleting books for entity.

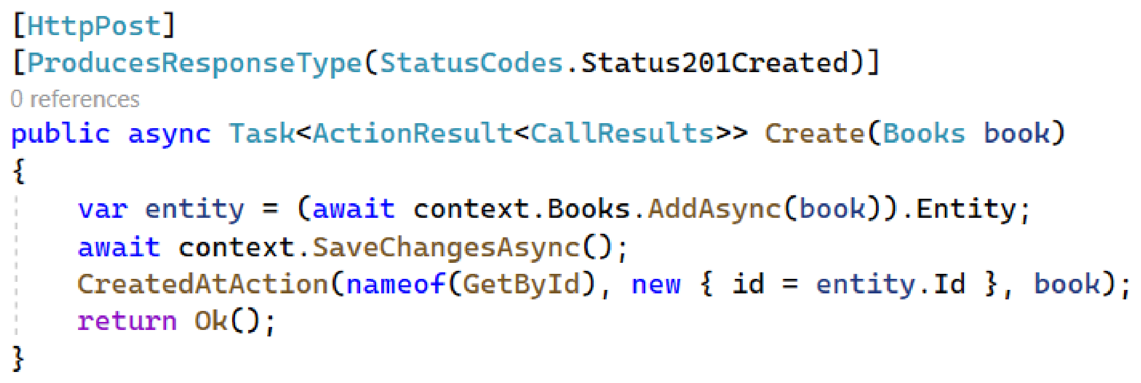
The method called deleteAllBooks(), which has the [HttpDelete] property and is used to handle HTTP DELETE requests. It makes an asynchronous call to the _booksData.getAllBooks() method and uses the "await" keyword to wait for the completion of the operation as shown in Figure 12. An iteration using the foreach statement is carried out to go through each book in the list of books. The _booksData.deleteBookById(book.Id) function is used to delete a book based on its ID. The await keyword allows the developer to suspend the program's execution until the operation is finished. The method produces an output of type ok(), which indicates that the operation was successful. When invoked by the framework as an HTTP DELETE request, the result will be turned into an appropriate HTTP response.



```
[HttpDelete]
0 references
public async Task<ActionResult<CallResults>> deleteAllBooks()
{
    List<Books> books = await _booksData.getAllBooks();

    foreach (Books book in books)
    {
        await _booksData.deleteBookById(book.Id);
    }

    return Ok();
}
```

Figure 12 : Endpoint for deleting all books for dapper.

The class method that has the HttpPost property as illustrated in Figure 13, which indicates its function as an action method that handles requests made through HTTP POST. This method generates a new record in a database that pertains to a literary publication, using an argument with the name 'book' and the type 'Books'. The 'Add Async' method is used to include the book object into the context, while the 'await' keyword is used to perform an asynchronous wait for the completion of the database activity. The 'SaveChangesAsync' function is used to preserve the novel book entry. The 'CreatedAtAction' function is called upon to generate a response that includes a location header that contains the Uniform Resource Locator URL of the newly generated book resource. The 'GetById' action method is passed along as a route value, but the results of this method are not included in the code snippet. The 'Ok' result indicates a successful HTTP response with a status code of 200.



```
[HttpPost]
[ProducesResponseType(StatusCodes.Status201Created)]
0 references
public async Task<ActionResult<CallResults>> Create(Books book)
{
    var entity = (await context.Books.AddAsync(book)).Entity;
    await context.SaveChangesAsync();
    CreatedAtAction(nameof(GetById), new { id = entity.Id }, book);
    return Ok();
}
```

Figure 13 : Endpoint for inserting books for entity.

In Figure 14. It show The value "All Book insert" is used to kick off the "m" string variable and create a new instance of the "CallResults" class. The current system tick count is the beginning time of the process. Invoking the method "_booksData.insertBook" in an asynchronous fashion makes the insertion of the "book" object into a particular data source possible. The "id" variable receives the value that has been returned and the final time is reported as being equal to the tick count of the system at the current moment. The amount of time that has elapsed is stored in the "milliseconds" property of the "stats" property of the "results" object. The "data" property of the "results" entity has been given a collection that consists of the "book" object as its component. The "results

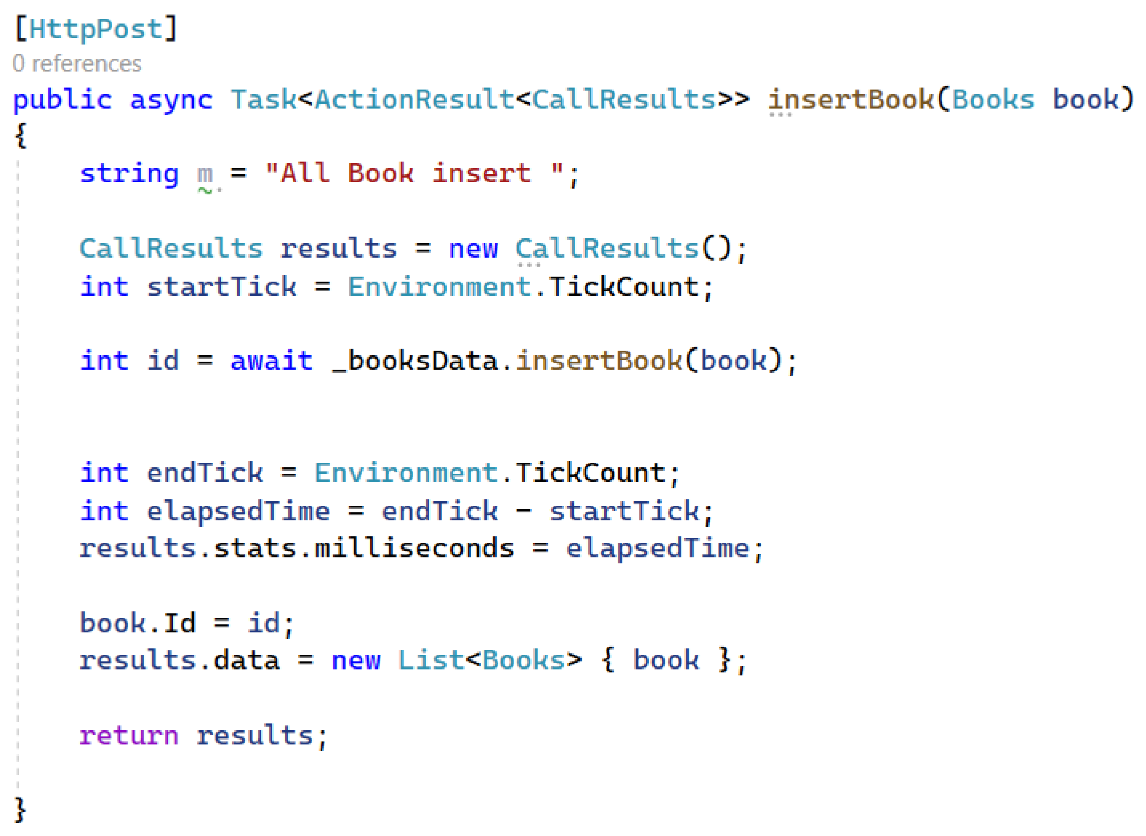
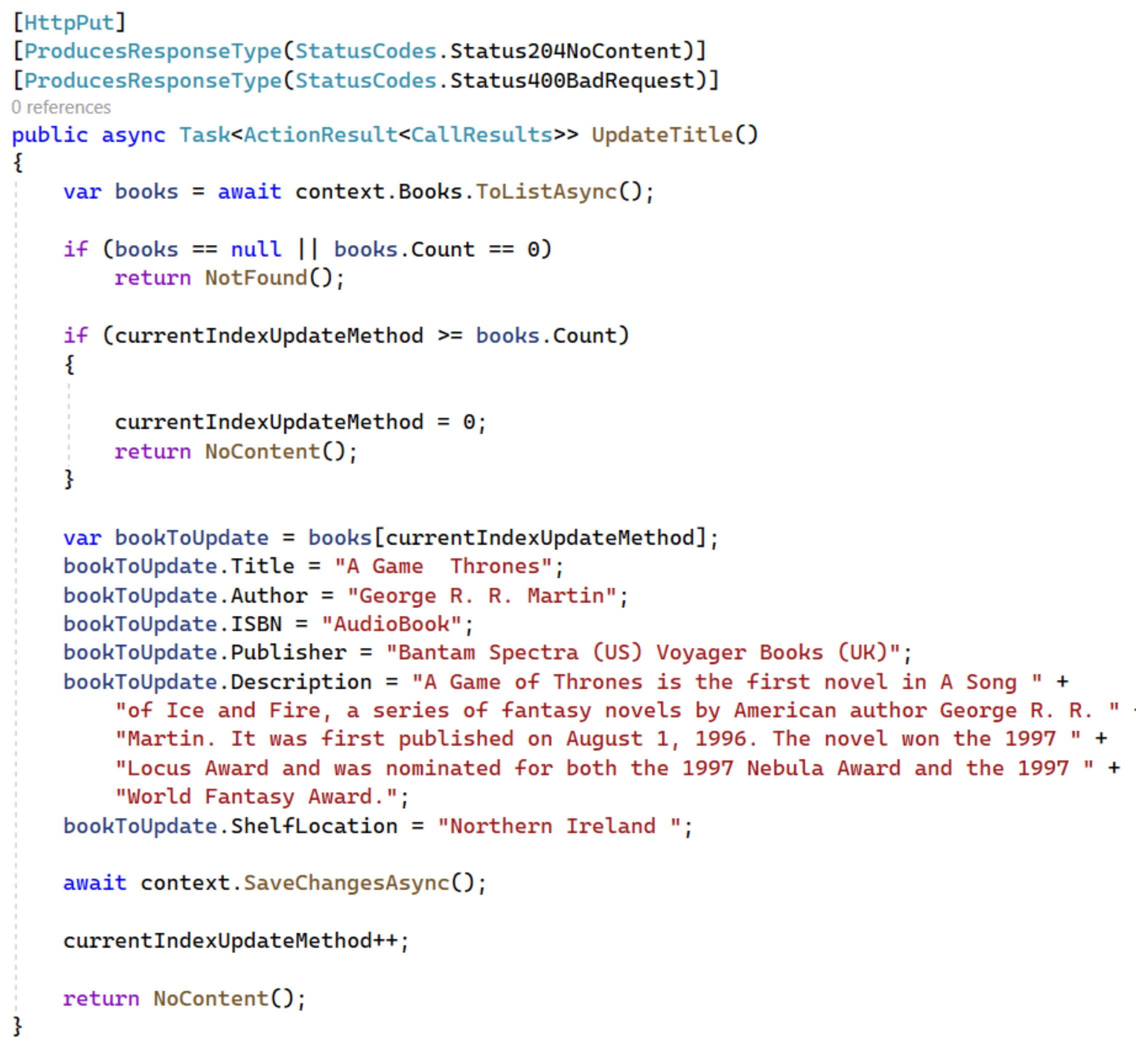

```
[HttpPost]
0 references
public async Task<ActionResult<CallResults>> insertBook(Books book)
{
    string m = "All Book insert ";

    CallResults results = new CallResults();
    int startTick = Environment.TickCount;

    int id = await _booksData.insertBook(book);


    int endTick = Environment.TickCount;
    int elapsedTime = endTick - startTick;
    results.stats.milliseconds = elapsedTime;

    book.Id = id;
    results.data = new List<Books> { book };

    return results;

}
```

Figure 14 : Endpoint for inserting books for dapper.

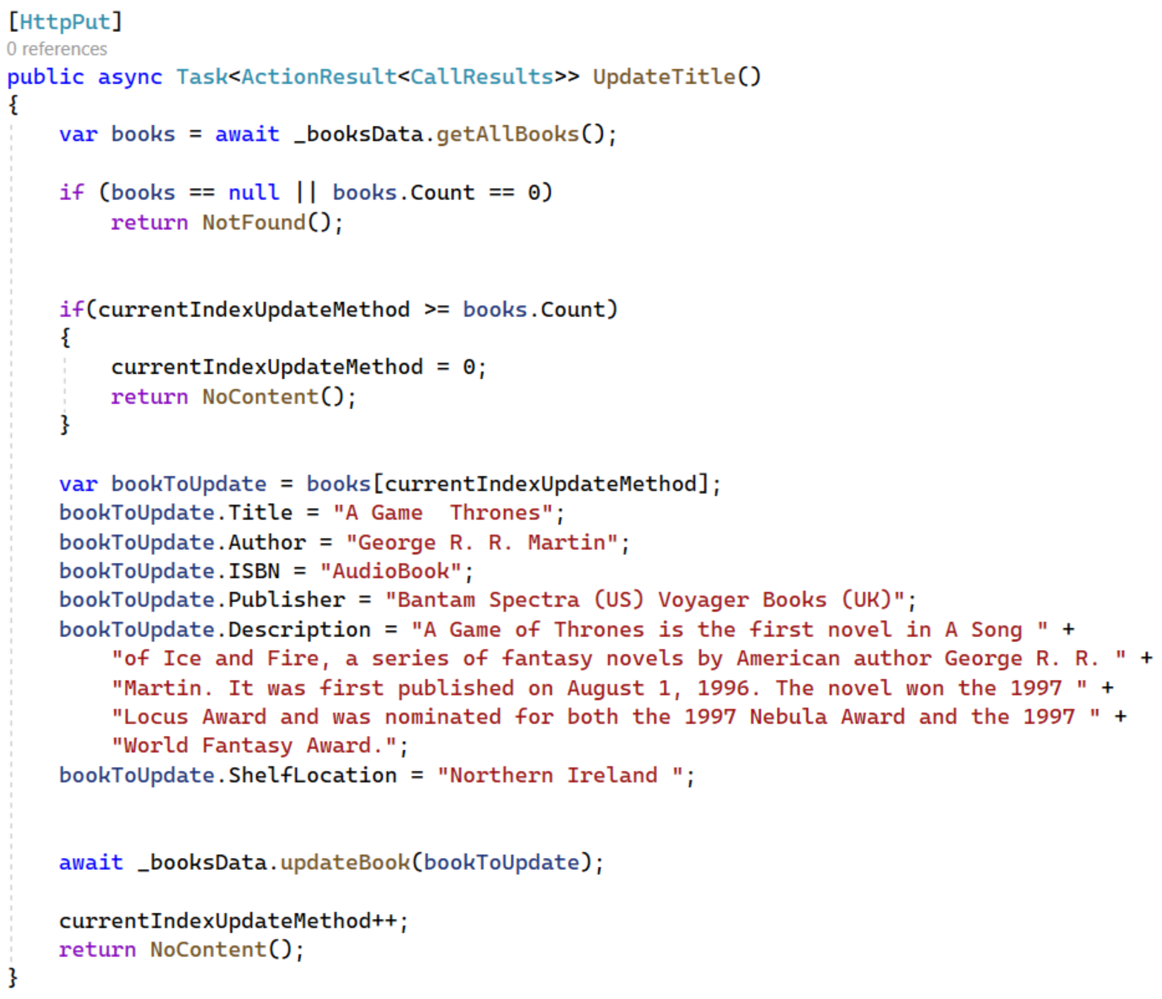
The [HttpPut] property in the function's annotations specifies the function's duty as a PUT request handler in Figure 15. It proves The execution of the method will result in the production of a TaskActionResultCallResults outcome, which identifies an asynchronous process as the source of an ActionResult belonging to the CallResults category. The HTTP status codes that the method is able to return are outlined in the [ProducesResponseType] annotations that are placed on the method. The process starts off by getting all of the books from the context asynchronously using the implementation of the ToListAsync() function. A response code of 404 (NotFound) is sent if the collection of books does not include any items. In the case that the condition is not satisfied, the computer system will choose the book that is currently situated at the current index and will continue to modify the book's properties. Invoking the SaveChanges



```
[HttpPut]
[ProducesResponseType(StatusCodes.Status204NoContent)]
[ProducesResponseType(StatusCodes.Status400BadRequest)]
0 references
public async Task<ActionResult<CallResults>> UpdateTitle()
{
    var books = await context.Books.ToListAsync();

    if (books == null || books.Count == 0)
        return NotFound();

    if (currentIndexUpdateMethod >= books.Count)
    {

        currentIndexUpdateMethod = 0;
        return NoContent();
    }

    var bookToUpdate = books[currentIndexUpdateMethod];
    bookToUpdate.Title = "A Game  Thrones";
    bookToUpdate.Author = "George R. R. Martin";
    bookToUpdate.ISBN = "AudioBook";
    bookToUpdate.Publisher = "Bantam Spectra (US) Voyager Books (UK)";
    bookToUpdate.Description = "A Game of Thrones is the first novel in A Song " +
        "of Ice and Fire, a series of fantasy novels by American author George R. R. " +
        "Martin. It was first published on August 1, 1996. The novel won the 1997 " +
        "Locus Award and was nominated for both the 1997 Nebula Award and the 1997 " +
        "World Fantasy Award.";
    bookToUpdate.ShelfLocation = "Northern Ireland ";

    await context.SaveChangesAsync();

    currentIndexUpdateMethod++;

    return NoContent();
}
```

Figure 15 : Endpoint for updating title of the books for entity.

The function UpdateTitle in a class is used to carry out asynchronous operations. This is indicated by the use of the async keyword in the function's definition. To acquire a collection of books, the method involves an asynchronous call of the _booksData.getAllBooks() function as can be seen in Figure 16. If the collection of books being searched for does not exist or is null, the function will provide a return of NotFound() and a NoContent() result. If the value of the variable currentIndexUpdateMethod is equal to or greater than the total number of books, the method will reset the value of currentIndexUpdateMethod to 0 and produce a NoContent() result. The _booksData.updateBook(bookToUpdate) method is used to retrieve a collection of books, adjust the characteristics of a specific book, maintain the changes, and move on to the next book in the collection until all of the books have been worked with. This method entails the retrieval of a collection of books, the adjustment of the characteristics of a specific book, the maintenance of the changes, and the movement on to the next book in the collection until all of the books have been worked with.



```
[HttpPut]
0 references
public async Task<ActionResult<CallResults>> UpdateTitle()
{
    var books = await _booksData.getAllBooks();

    if (books == null || books.Count == 0)
        return NotFound();


    if(currentIndexUpdateMethod >= books.Count)
    {
        currentIndexUpdateMethod = 0;
        return NoContent();
    }

    var bookToUpdate = books[currentIndexUpdateMethod];
    bookToUpdate.Title = "A Game  Thrones";
    bookToUpdate.Author = "George R. R. Martin";
    bookToUpdate.ISBN = "AudioBook";
    bookToUpdate.Publisher = "Bantam Spectra (US) Voyager Books (UK)";
    bookToUpdate.Description = "A Game of Thrones is the first novel in A Song " +
        "of Ice and Fire, a series of fantasy novels by American author George R. R. " +
        "Martin. It was first published on August 1, 1996. The novel won the 1997 " +
        "Locus Award and was nominated for both the 1997 Nebula Award and the 1997 " +
        "World Fantasy Award.";
    bookToUpdate.ShelfLocation = "Northern Ireland ";


    await _booksData.updateBook(bookToUpdate);

    currentIndexUpdateMethod++;
    return NoContent();
}
```

Figure 16 : Endpoint for updating title of the books for dapper.

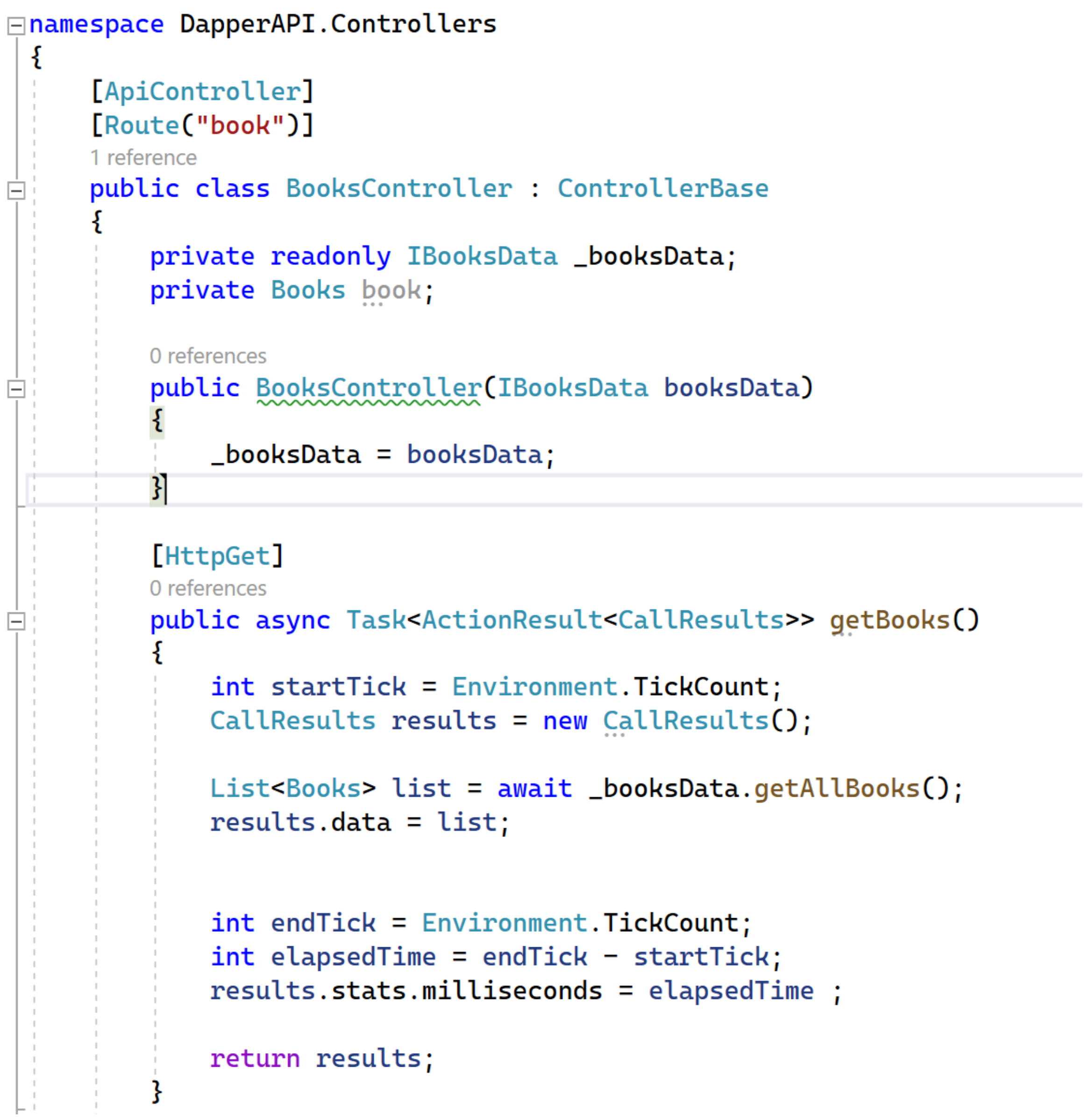
2.2 Classification

2.2.1 Controllers

2.2.1.1 Dapper Controllers

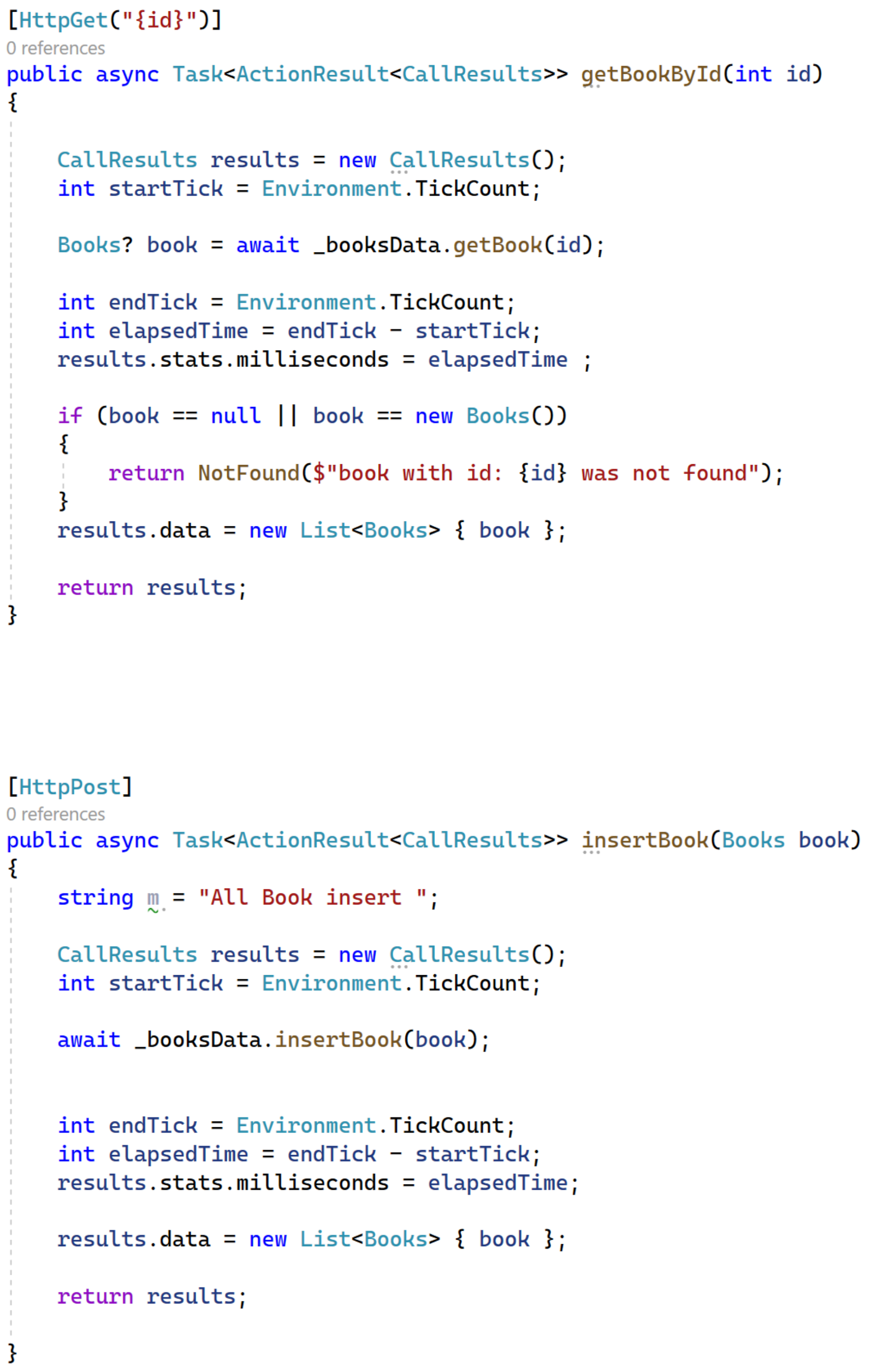
The "BooksController" is a class that handles HTTP requests specific to books. It is found in the namespace "DapperAPI.Controllers" and is identified by the [ApiController] and [Route("book")] properties. Multiple HTTP action methods, such as [HttpGet], [HttpPost], [HttpDelete], and [HttpPut].

- The getBooks() method manages a GET request to obtain all books, while the aggregated data is combined in a ListBooks object and the amount of time that the operation took as shown in Figure 17.
- GetBookById(int id) is used to get a book based on its specific identifier in response to a GET request as illustrated in Figure 18.
- InsertBook(Books book) is used to add a new book to the database.
- DeleteBookById(int id) is used to find a specific book by id and then delete the book from the system, as can be seen Figure 19. The _booksData.getBook(id) and _booksData.deleteBookById(id) functions are used to remove books from the database.
- The Update() function handles any PUT requests that are sent in with the intention of making changes to an existing book. An API endpoint is provided by the class that makes it easier to do CRUD operations on books, which engages in communication with a data service responsible for the implementation of the IBooksData interface.



```
namespace DapperAPI.Controllers
{
    [ApiController]
    [Route("book")]
    1 reference
    public class BooksController : ControllerBase
    {
        private readonly IBooksData _booksData;
        private Books book;

        0 references
        public BooksController(IBooksData booksData)
        {
            _booksData = booksData;
        }

        [HttpGet]
        0 references
        public async Task<ActionResult<CallResults>> getBooks()
        {
            int startTick = Environment.TickCount;
            CallResults results = new CallResults();

            List<Books> list = await _booksData.getAllBooks();
            results.data = list;


            int endTick = Environment.TickCount;
            int elapsedTime = endTick - startTick;
            results.stats.milliseconds = elapsedTime ;

            return results;
        }
```

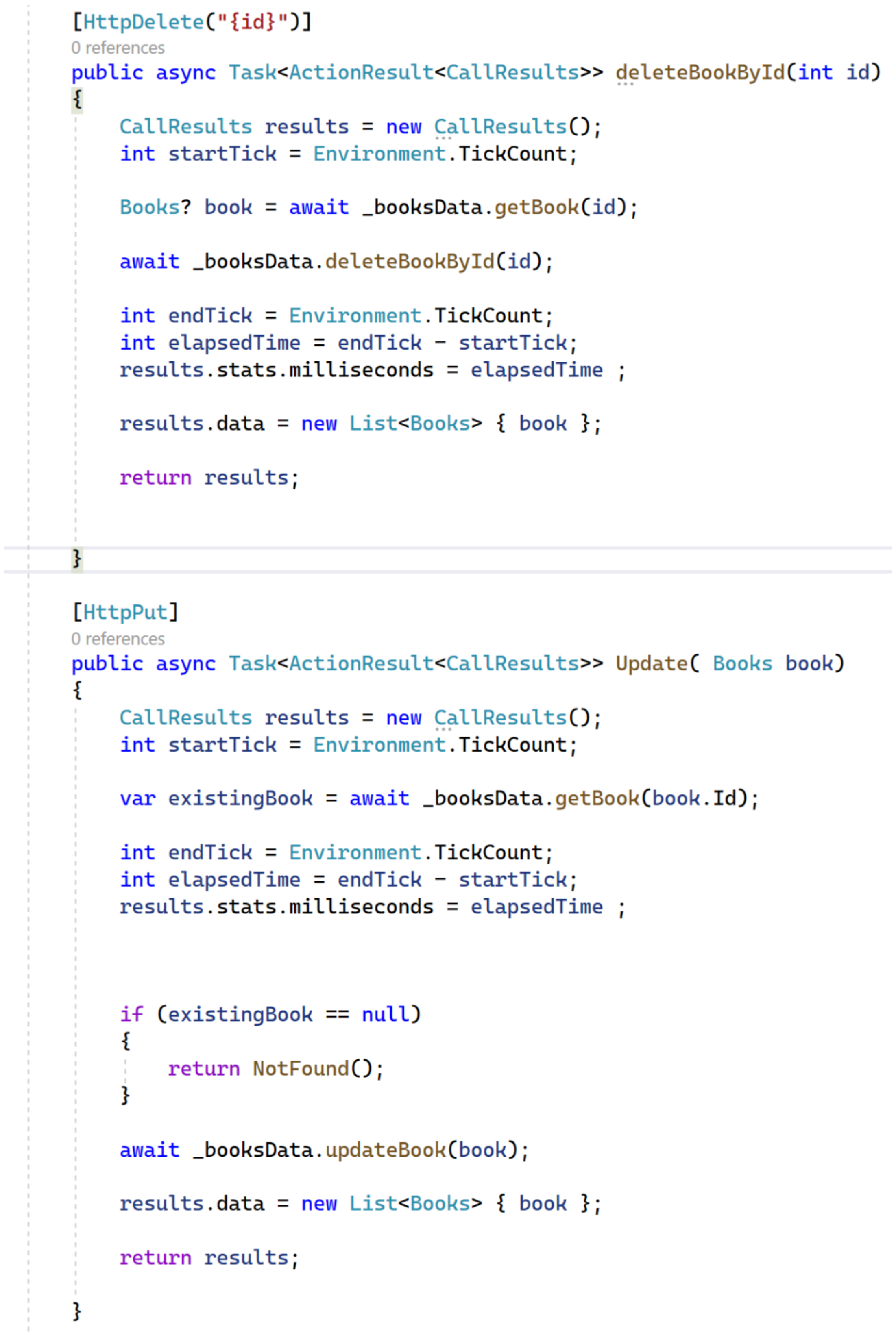
Figure 17 : Endpoint for retrieving books in Dapper Framework.



```
[HttpGet("{id}")]
0 references
public async Task<ActionResult<CallResults>> getBookById(int id)
{

    CallResults results = new CallResults();
    int startTick = Environment.TickCount;

    Books? book = await _booksData.getBook(id);

    int endTick = Environment.TickCount;
    int elapsedTime = endTick - startTick;
    results.stats.milliseconds = elapsedTime ;

    if (book == null || book == new Books())
    {
        return NotFound($"book with id: {id} was not found");
    }
    results.data = new List<Books> { book };

    return results;
}
```

```
[HttpPost]
0 references
public async Task<ActionResult<CallResults>> insertBook(Books book)
{
    string m = "All Book insert ";

    CallResults results = new CallResults();
    int startTick = Environment.TickCount;

    await _booksData.insertBook(book);


    int endTick = Environment.TickCount;
    int elapsedTime = endTick - startTick;
    results.stats.milliseconds = elapsedTime;

    results.data = new List<Books> { book };

    return results;

}
```

Figure 18 : Endpoints for retrieving book by id and sending new book to database in Dapper Framework.



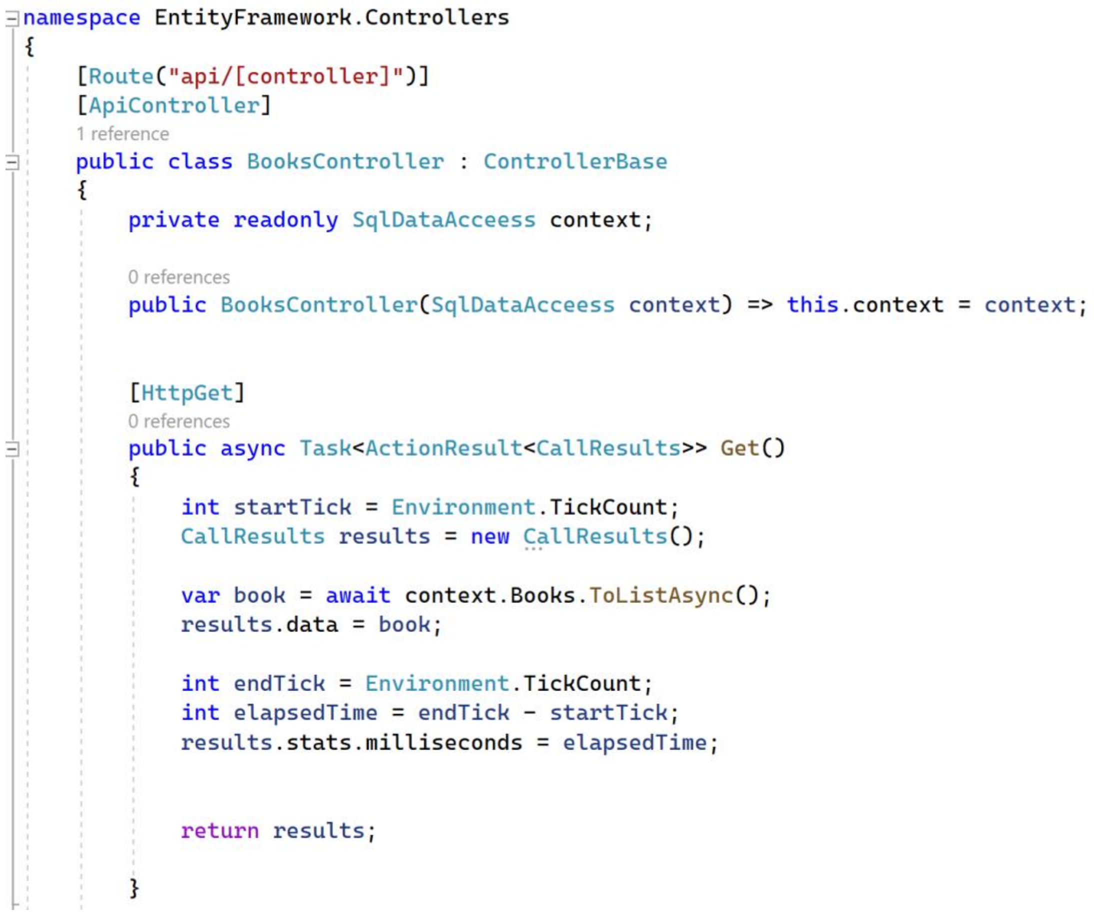
```
[HttpDelete("{id}")]
0 references
public async Task<ActionResult<CallResults>> deleteBookById(int id)
{
    CallResults results = new CallResults();
    int startTick = Environment.TickCount;

    Books? book = await _booksData.getBook(id);

    await _booksData.deleteBookById(id);

    int endTick = Environment.TickCount;
    int elapsedTime = endTick - startTick;
    results.stats.milliseconds = elapsedTime ;

    results.data = new List<Books> { book };

    return results;


}

[HttpPut]
0 references
public async Task<ActionResult<CallResults>> Update( Books book)
{
    CallResults results = new CallResults();
    int startTick = Environment.TickCount;

    var existingBook = await _booksData.getBook(book.Id);

    int endTick = Environment.TickCount;
    int elapsedTime = endTick - startTick;
    results.stats.milliseconds = elapsedTime ;



    if (existingBook == null)
    {
        return NotFound();
    }

    await _booksData.updateBook(book);

    results.data = new List<Books> { book };

    return results;

}
```

Figure 19 : Endpoints for deleting book by id and updating book in Dapper Framework.

2.2.1.2 Entity Framework Core’s Controller

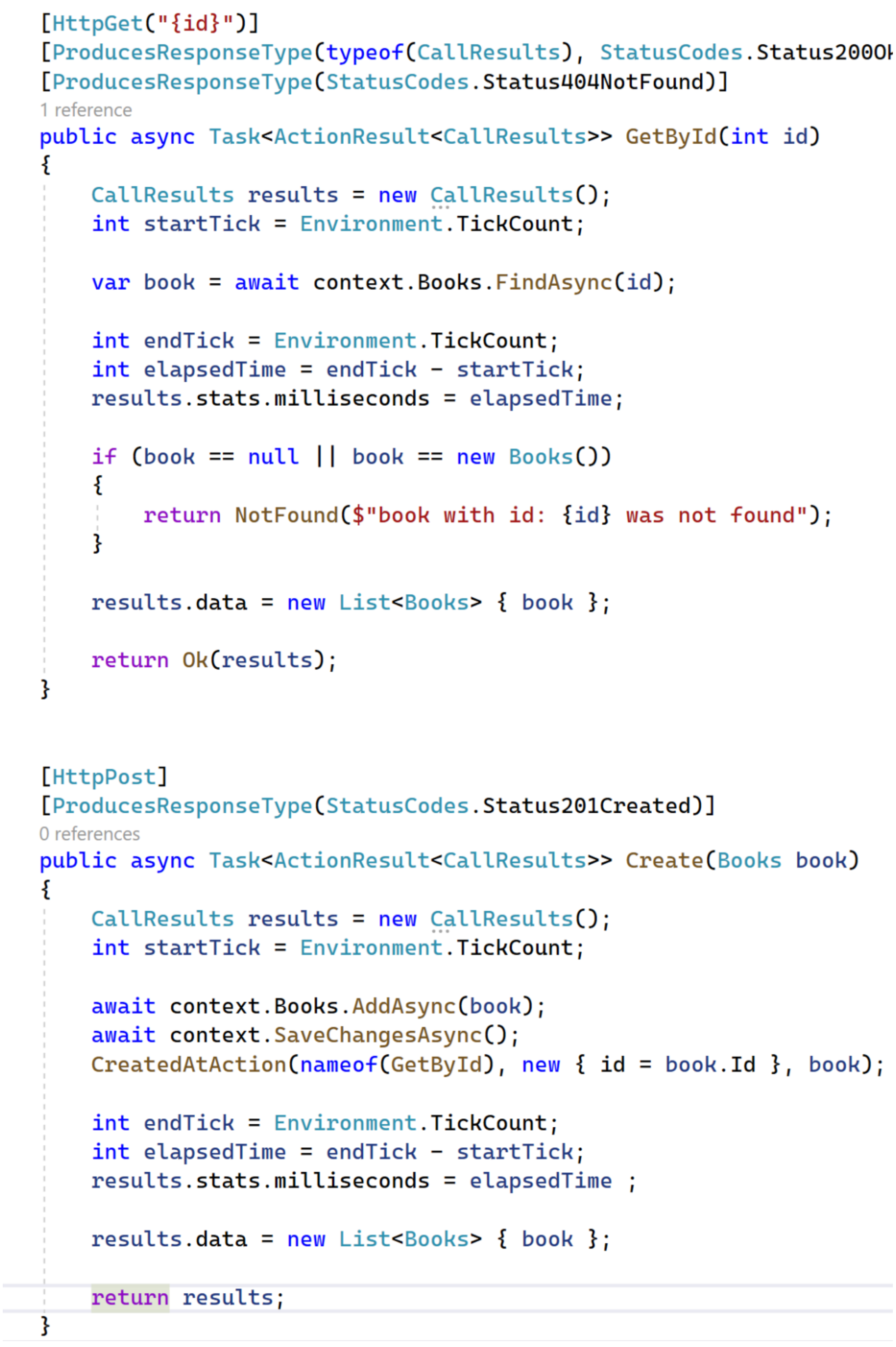
The BooksController class is a derivation of ControllerBase, which is an essential base class for all ASP.NET Core controllers. It contains a constructor that receives an argument of type SqlDataAccess and assigns that parameter to a private field called context.

- The Get() function is used to obtail a list of books from the database, which is an implementation of the HTTP GET method as can be seen in Figure 20.
- The GetById function is an HTTP GET operation that obtains a specific book from the database based on the unique identifier for that book. If the book cannot be found, the system will respond with "404 Not Found". The BooksController class acts as an API controller by providing endpoints for performing CRUD capabilities on books through the usage of Entity Framework for data retrieval. This includes creating new books, reading existing books, updating existing books, and deleting books.
- The Create (Books book) function of the HTTP POST protocol is applied to create an entry for a novel book as can be seen in Figure 21.
- the Update (Books books) function makes changes to the state of the books argument and saves those changes to the Books table.
- The Delete (int id) method is used to delete a book from the database that has been assigned a certain ID as can be seen in Figure 22. The information and metrics that are acquired from each API method are saved in the Call Results class. The implementation makes use of a variety of namespaces, such EntityFramework.dbServer, EntityFramework.Models, and EntityFramework.ThesisTools, amongst others. The Books Controller class acts as an API controller by providing endpoints for performing CRUD.



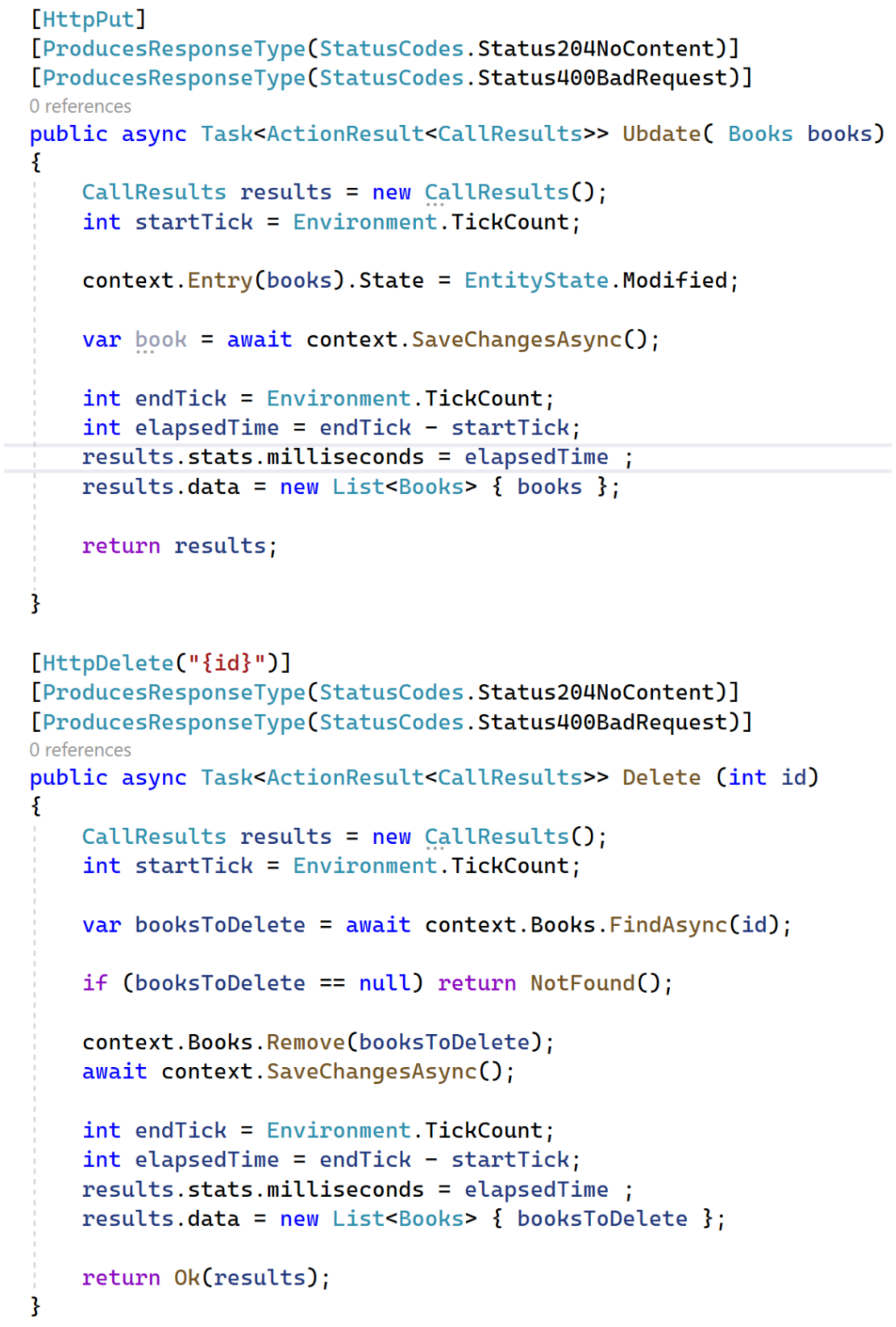
```
namespace EntityFramework.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    1 reference
    public class BooksController : ControllerBase
    {
        private readonly SqlDataAcceess context;

        0 references
        public BooksController(SqlDataAcceess context) => this.context = context;


        [HttpGet]
        0 references
        public async Task<ActionResult<CallResults>> Get()
        {
            int startTick = Environment.TickCount;
            CallResults results = new CallResults();

            var book = await context.Books.ToListAsync();
            results.data = book;

            int endTick = Environment.TickCount;
            int elapsedTime = endTick - startTick;
            results.stats.milliseconds = elapsedTime;


            return results;

        }
```

Figure 20 : Endpoint for getting in Entity framework.



```
[HttpGet("{id}")]
[ProducesResponseType(typeof(CallResults), StatusCodes.Status200OK
[ProducesResponseType(StatusCodes.Status404NotFound)]
1 reference
public async Task<ActionResult<CallResults>> GetById(int id)
{
    CallResults results = new CallResults();
    int startTick = Environment.TickCount;

    var book = await context.Books.FindAsync(id);

    int endTick = Environment.TickCount;
    int elapsedTime = endTick - startTick;
    results.stats.milliseconds = elapsedTime;

    if (book == null || book == new Books())
    {
        return NotFound($"book with id: {id} was not found");
    }

    results.data = new List<Books> { book };

    return Ok(results);
}


[HttpPost]
[ProducesResponseType(StatusCodes.Status201Created)]
0 references
public async Task<ActionResult<CallResults>> Create(Books book)
{
    CallResults results = new CallResults();
    int startTick = Environment.TickCount;

    await context.Books.AddAsync(book);
    await context.SaveChangesAsync();
    CreatedAtAction(nameof(GetById), new { id = book.Id }, book);

    int endTick = Environment.TickCount;
    int elapsedTime = endTick - startTick;
    results.stats.milliseconds = elapsedTime ;

    results.data = new List<Books> { book };

    return results;
}
```

Figure 21 : Endpoints for retrieve book by Id and send new books to database in Entity Framework.



```
[HttpPut]
[ProducesResponseType(StatusCodes.Status204NoContent)]
[ProducesResponseType(StatusCodes.Status400BadRequest)]
0 references
public async Task<ActionResult<CallResults>> Ubdate( Books books)
{
    CallResults results = new CallResults();
    int startTick = Environment.TickCount;

    context.Entry(books).State = EntityState.Modified;

    var book = await context.SaveChangesAsync();

    int endTick = Environment.TickCount;
    int elapsedTime = endTick - startTick;
    results.stats.milliseconds = elapsedTime ;
    results.data = new List<Books> { books };

    return results;

}

[HttpDelete("{id}")]
[ProducesResponseType(StatusCodes.Status204NoContent)]
[ProducesResponseType(StatusCodes.Status400BadRequest)]
0 references
public async Task<ActionResult<CallResults>> Delete (int id)
{
    CallResults results = new CallResults();
    int startTick = Environment.TickCount;

    var booksToDelete = await context.Books.FindAsync(id);

    if (booksToDelete == null) return NotFound();

    context.Books.Remove(booksToDelete);
    await context.SaveChangesAsync();

    int endTick = Environment.TickCount;
    int elapsedTime = endTick - startTick;
    results.stats.milliseconds = elapsedTime ;
    results.data = new List<Books> { booksToDelete };

    return Ok(results);
}
```

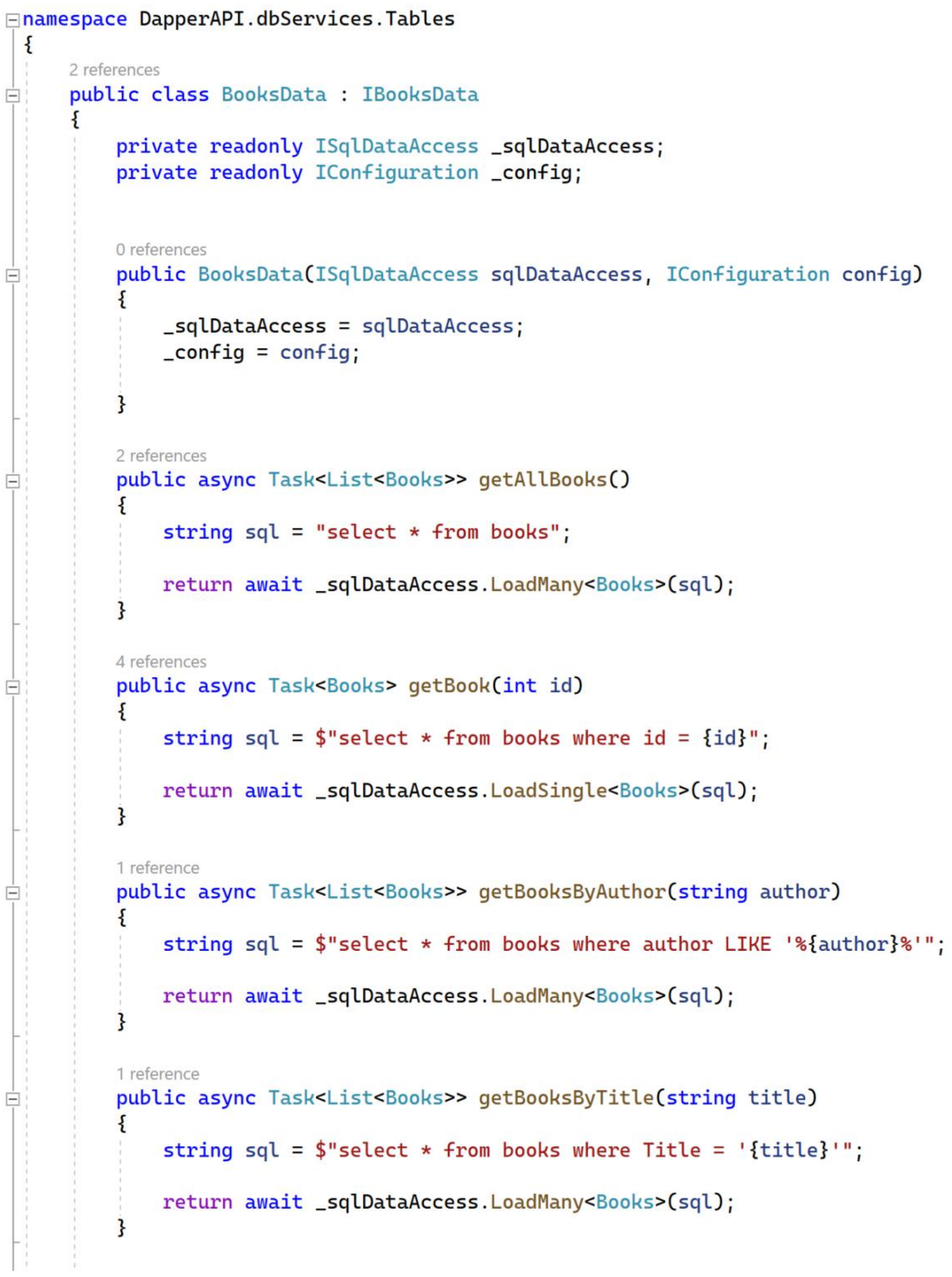
Figure 22 : Endpoints for updating books and deleting book by Id in Entity Framework.

2.2.2 Database Servers

2.2.2.1 Dapper's Book Data

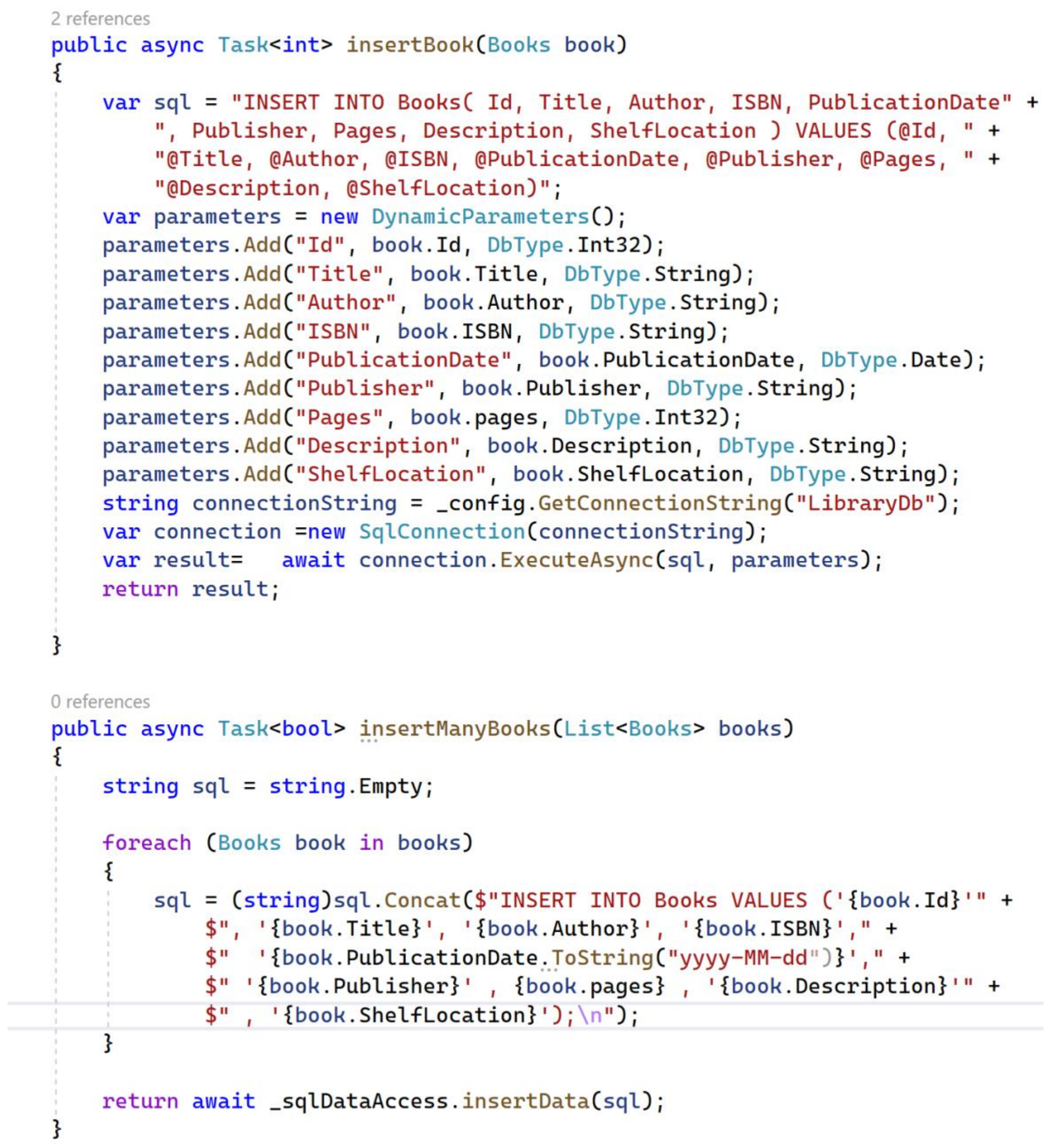
The BooksData class is part of the Dapper ORM and is used to manage namespaces and dependencies. The constructor is intended to take in two arguments, ISqlDataAccess and IConfiguration respectively, to initiate the sqlDataAccess and config fields in the database. The SQL query and the _sqlDataAccess object used in the getAllBooks() function enable the user to obtain all the books stored in the database. A list of Books objects is returned in an asynchronous fashion by the procedure.

- The "getBook(int id)" function is used to obtain a book from a database.
- The "getBooksByAuthor" function is used to obtain books from a database that correspond to the name of a certain author.
- The "insertBook" method is used to add a new book to the database.
- The "insertManyBooks" function is used to enter many books into the database as observed in Figure 24.
- The "deleteBookById(int id)" function is used to delete a book from the database according to a certain identifier.
- The "updateBook(Books book)" function is used to update the information about a book that is saved in the database as can be seen in Figure 25.



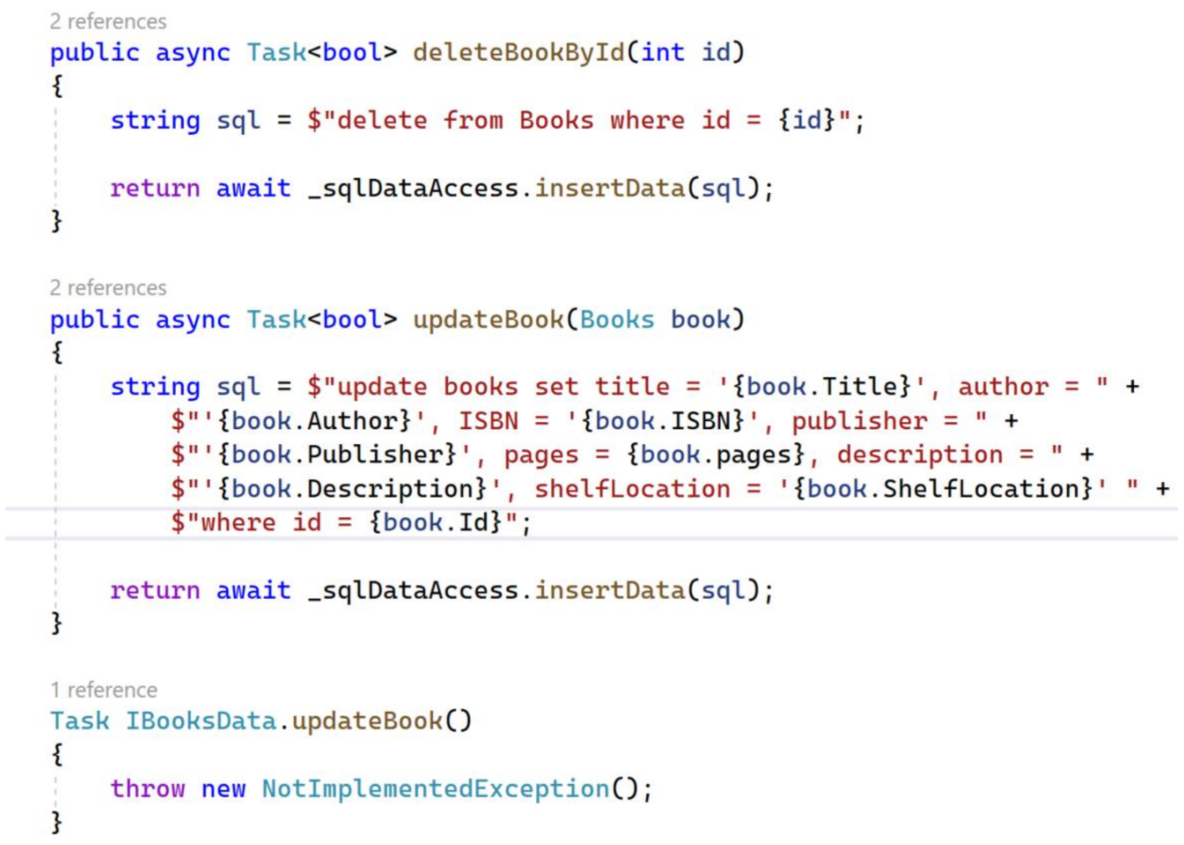
```
namespace DapperAPI.dbServices.Tables
{
    2 references
    public class BooksData : IBooksData
    {
        private readonly ISqlDataAccess _sqlDataAccess;
        private readonly IConfiguration _config;


        0 references
        public BooksData(ISqlDataAccess sqlDataAccess, IConfiguration config)
        {
            _sqlDataAccess = sqlDataAccess;
            _config = config;

        }

        2 references
        public async Task<List<Books>> getAllBooks()
        {
            string sql = "select * from books";

            return await _sqlDataAccess.LoadMany<Books>(sql);
        }

        4 references
        public async Task<Books> getBook(int id)
        {
            string sql = $"select * from books where id = {id}";

            return await _sqlDataAccess.LoadSingle<Books>(sql);
        }

        1 reference
        public async Task<List<Books>> getBooksByAuthor(string author)
        {
            string sql = $"select * from books where author LIKE '%{author}%'";

            return await _sqlDataAccess.LoadMany<Books>(sql);
        }

        1 reference
        public async Task<List<Books>> getBooksByTitle(string title)
        {
            string sql = $"select * from books where Title = '{title}'";

            return await _sqlDataAccess.LoadMany<Books>(sql);
        }
```

Figure 23 : Implementation of getting book from database in Dapper framework.



```
2 references
public async Task<int> insertBook(Books book)
{
    var sql = "INSERT INTO Books( Id, Title, Author, ISBN, PublicationDate" +
        ", Publisher, Pages, Description, ShelfLocation ) VALUES (@Id, " +
        "@Title, @Author, @ISBN, @PublicationDate, @Publisher, @Pages, " +
        "@Description, @ShelfLocation)";
    var parameters = new DynamicParameters();
    parameters.Add("Id", book.Id, DbType.Int32);
    parameters.Add("Title", book.Title, DbType.String);
    parameters.Add("Author", book.Author, DbType.String);
    parameters.Add("ISBN", book.ISBN, DbType.String);
    parameters.Add("PublicationDate", book.PublicationDate, DbType.Date);
    parameters.Add("Publisher", book.Publisher, DbType.String);
    parameters.Add("Pages", book.pages, DbType.Int32);
    parameters.Add("Description", book.Description, DbType.String);
    parameters.Add("ShelfLocation", book.ShelfLocation, DbType.String);
    string connectionString = _config.GetConnectionString("LibraryDb");
    var connection =new SqlConnection(connectionString);
    var result=   await connection.ExecuteAsync(sql, parameters);
    return result;

}

0 references
public async Task<bool> insertManyBooks(List<Books> books)
{
    string sql = string.Empty;

    foreach (Books book in books)
    {
        sql = (string)sql.Concat($"INSERT INTO Books VALUES ('{book.Id}'" +
            $"', '{book.Title}', '{book.Author}', '{book.ISBN}'," +
            $"  '{book.PublicationDate.ToString("yyyy-MM-dd")}'," +
            $" '{book.Publisher}' , {book.pages} , '{book.Description}'" +
            $" , '{book.ShelfLocation}');\n");
    }

    return await _sqlDataAccess.insertData(sql);
}
```

Figure 24: Implementation for POST method and inserting book into database in Dapper framework.

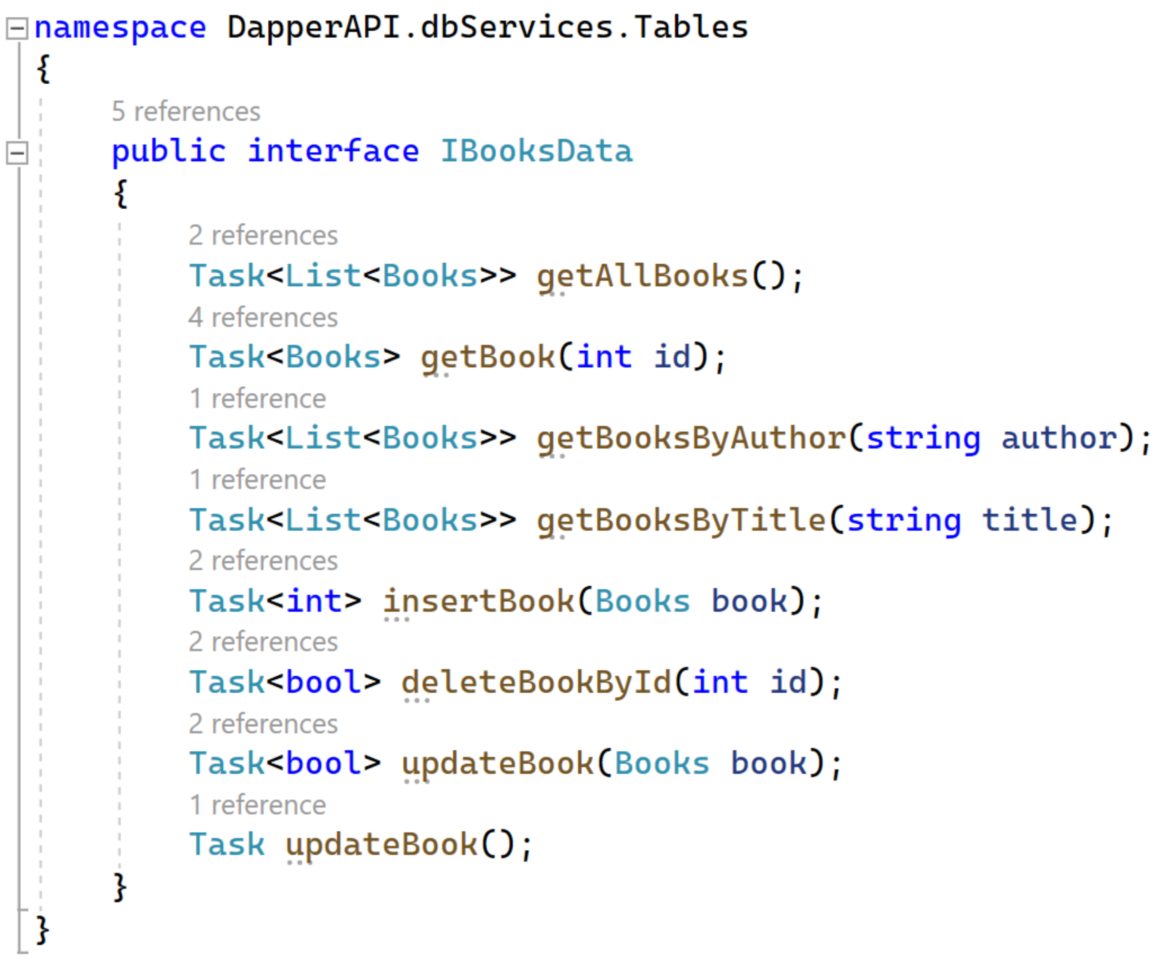


```
2 references
public async Task<bool> deleteBookById(int id)
{
    string sql = $"delete from Books where id = {id}";

    return await _sqlDataAccess.insertData(sql);
}

2 references
public async Task<bool> updateBook(Books book)
{
    string sql = $"update books set title = '{book.Title}', author = " +
        $"'{book.Author}', ISBN = '{book.ISBN}', publisher = " +
        $"'{book.Publisher}', pages = {book.pages}, description = " +
        $"'{book.Description}', shelfLocation = '{book.ShelfLocation}' " +
        $"where id = {book.Id}";

    return await _sqlDataAccess.insertData(sql);
}

1 reference
Task IBooksData.updateBook()
{
    throw new NotImplementedException();
}
```

Figure 25 : Implementations of DELEET method for deleting book by id and UPDATE method for updating book info in database in Dapper framework.

2.2.2.2 *Dapper's Book Data Interface*

The IBooksData interface is a series of procedures that make communication with a database management system easier as illustrated in Figure 26. It provides the necessary methods for carrying out CRUD activities on a collection of books in the underlying database by utilizing Dapper ORM. These methods include getAllBooks(), getBooksByAuthor(), getBooksByTitle(), insertBook(), deleteBookById(int id), updateBook(Books book) and updateBook().

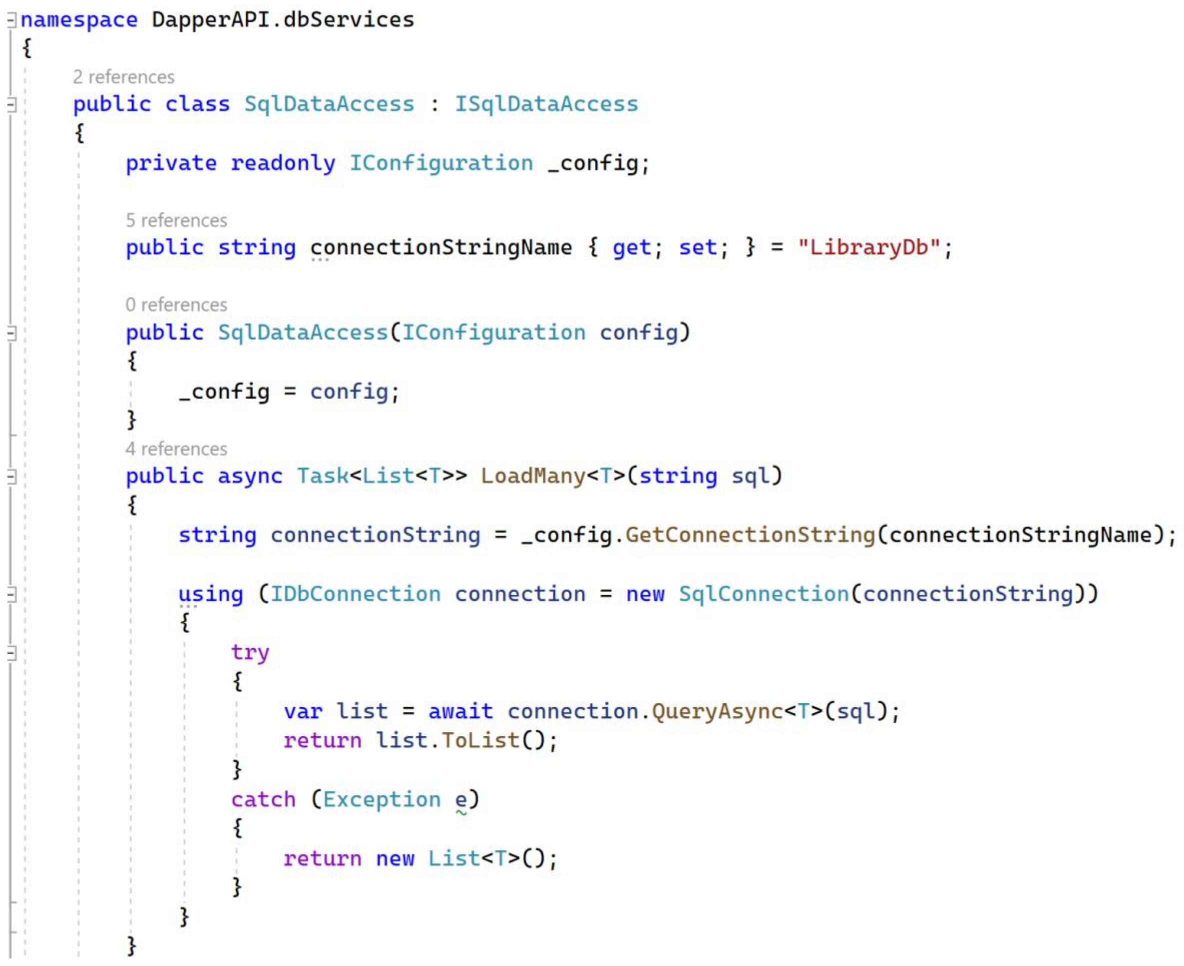


```
namespace DapperAPI.dbServices.Tables
{
    5 references
    public interface IBooksData
    {
        2 references
        Task<List<Books>> getAllBooks();
        4 references
        Task<Books> getBook(int id);
        1 reference
        Task<List<Books>> getBooksByAuthor(string author);
        1 reference
        Task<List<Books>> getBooksByTitle(string title);
        2 references
        Task<int> insertBook(Books book);
        2 references
        Task<bool> deleteBookById(int id);
        2 references
        Task<bool> updateBook(Books book);
        1 reference
        Task updateBook();
    }
}
```

Figure 26: Interface for communicating with the database.

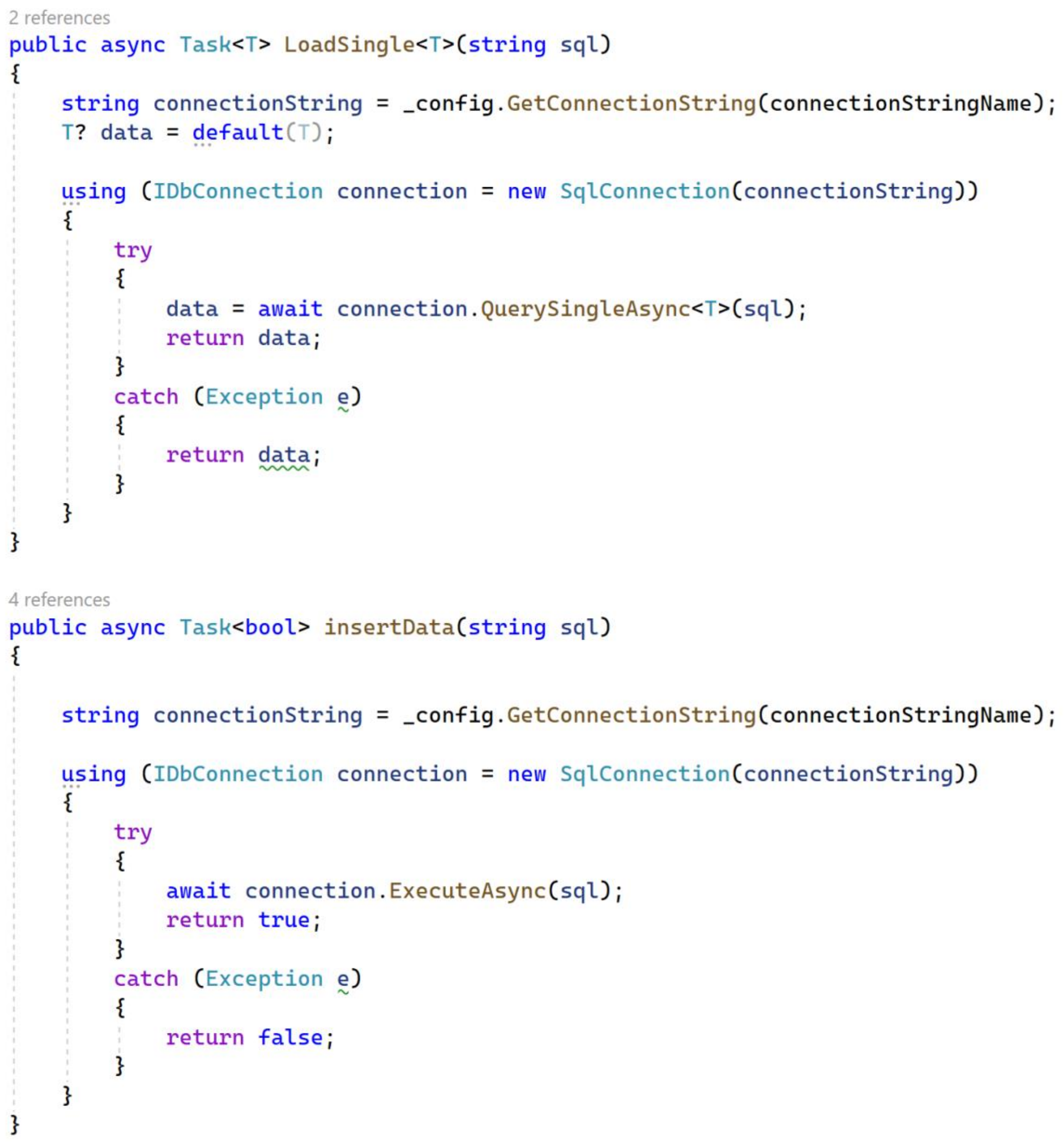
2.2.2.3 Dapper's SQL Data Access

This class provides its own implementation of the ISqlDataAccess interface and uses the Dapper ORM library to provide data access capabilities for the purpose of interacting with a SQL Server database as illustrated in Figure 27. The class makes use of the private field _config, which is of type IConfiguration, to store the necessary configuration settings for establishing a connection with the database. The class has a constructor that takes in an object of type IConfiguration and assigns it to the _config field after receiving it. The LoadMany<T> function executes a SQL query to get a collection of objects of type T, which is named after the type T. The query is carried out in an asynchronous fashion by using the QueryAsync function provided by Dapper. The LoadSingle function is used to retrieve a single instance of type T as illustrated in Figure 28. It is executed using Dapper's QuerySingleAsync function, which allows for the execution of the query in an asynchronous manner. The insertDataWithObjectReturn<T> function is responsible for the execution of a SQL statement designed to insert data into the database. It then delivers a return value that is an object of type T. The ExecuteScalarAsync function then typecasts the output to T, allowing for the statement to be executed asynchronously. The SqlDataAccess class provides a simplified and efficient method for dealing with a SQL Server database by using Dapper. It encapsulates the underlying database connection and query execution functionality and handles any exceptions that may occur.



```
namespace DapperAPI.dbServices
{
    2 references
    public class SqlDataAccess : ISqlDataAccess
    {
        private readonly IConfiguration _config;

        5 references
        public string connectionStringName { get; set; } = "LibraryDb";

        0 references
        public SqlDataAccess(IConfiguration config)
        {
            _config = config;
        }
        4 references
        public async Task<List<T>> LoadMany<T>(string sql)
        {
            string connectionString = _config.GetConnectionString(connectionStringName);

            using (IDbConnection connection = new SqlConnection(connectionString))
            {
                try
                {
                    var list = await connection.QueryAsync<T>(sql);
                    return list.ToList();
                }
                catch (Exception e)
                {
                    return new List<T>();
                }
            }
        }
```

Figure 27 : Uses ISqlDataAccess interface and Dapper ORM library for SQL Server database access.

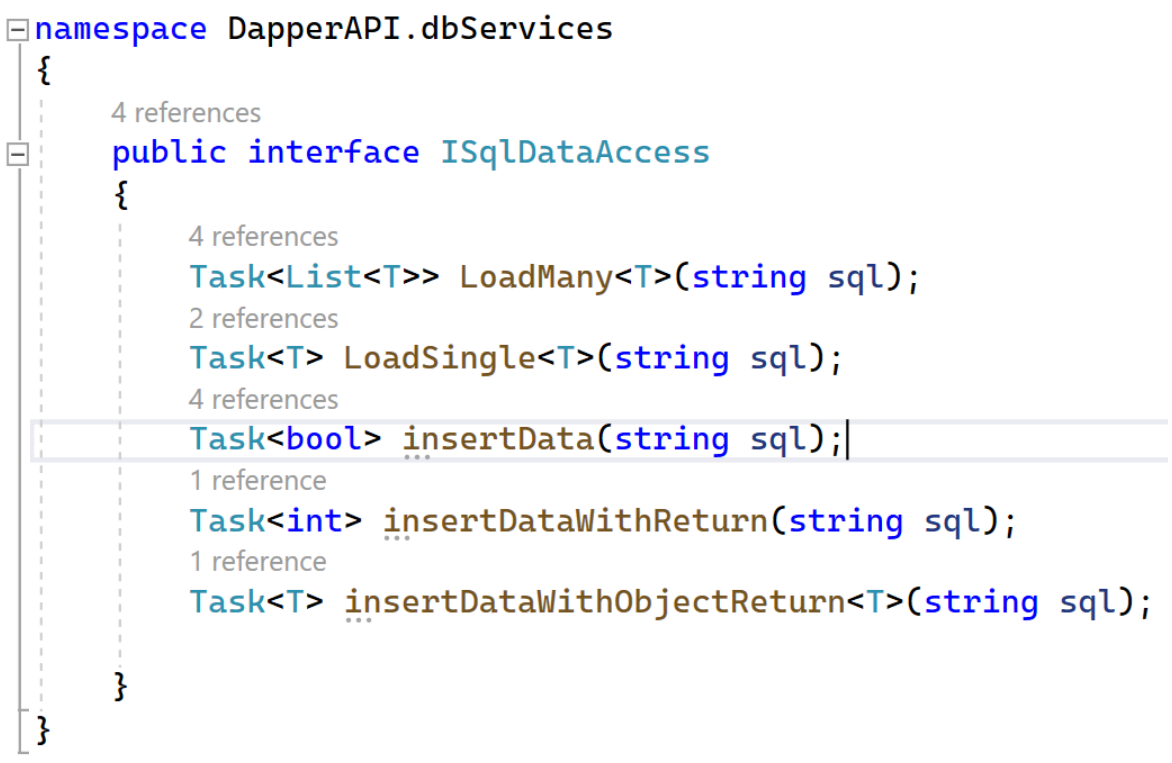


```
2 references
public async Task<T> LoadSingle<T>(string sql)
{
    string connectionString = _config.GetConnectionString(connectionStringName);
    T? data = default(T);

    using (IDbConnection connection = new SqlConnection(connectionString))
    {
        try
        {
            data = await connection.QuerySingleAsync<T>(sql);
            return data;
        }
        catch (Exception e)
        {
            return data;
        }
    }
}

4 references
public async Task<bool> insertData(string sql)
{

    string connectionString = _config.GetConnectionString(connectionStringName);

    using (IDbConnection connection = new SqlConnection(connectionString))
    {
        try
        {
            await connection.ExecuteAsync(sql);
            return true;
        }
        catch (Exception e)
        {
            return false;
        }
    }
}
```

Figure 28: Constructor assigns IConfiguration to _config. LoadMany<T> executes async SQL query. LoadSingle retrieves T instance.

2.2.2.4 Dapper's SQL Data Access Interface

ISqlDataAccess is an interface that can be found within the DapperAPI.dbServices namespace as can be seen in Figure 29. It outlines a selection of procedures that make it simpler to deal with a database. These include LoadMany, LoadSingle, InsertDataWithReturn, and InsertDataWithObjectReturn. LoadMany uses a SQL query to obtain several instances of objects belonging to type T from the database. LoadSingle uses a SQL query to obtain a single instance of an object of type T. InsertDataWithReturn is responsible for the execution of a SQL query that inserts data into the database and returns the number of rows that were modified as a result. InsertDataWithObjectReturn is intended to carry out a SQL query that runs an insertion operation in the database and returns an object of type T. The execution of the required methods enables a class that has this interface implemented to provide the necessary functionality to interact with a database using Dapper.

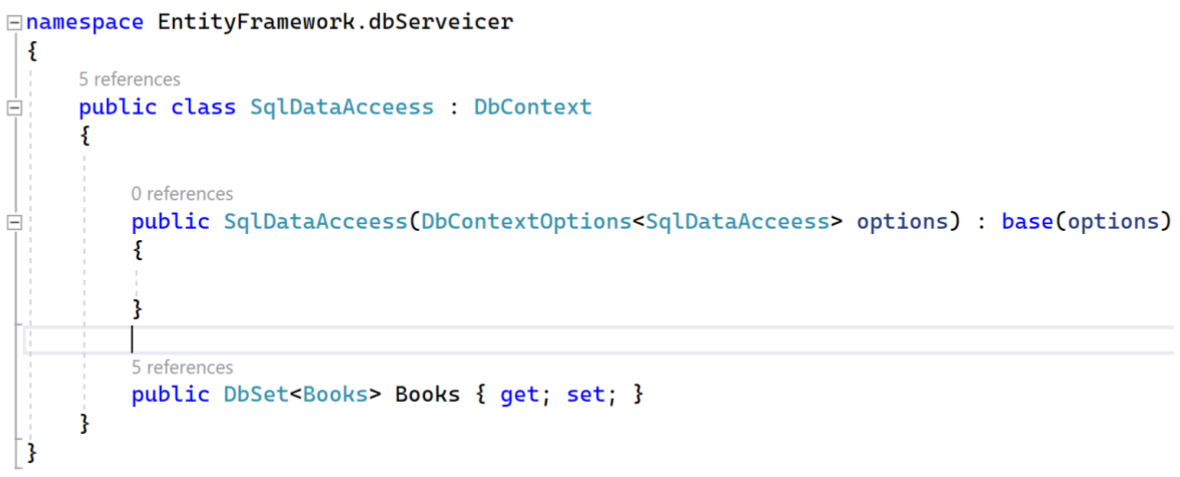


```
namespace DapperAPI.dbServices
{
    4 references
    public interface ISqlDataAccess
    {
        4 references
        Task<List<T>> LoadMany<T>(string sql);
        2 references
        Task<T> LoadSingle<T>(string sql);
        4 references
        Task<bool> insertData(string sql);
        1 reference
        Task<int> insertDataWithReturn(string sql);
        1 reference
        Task<T> insertDataWithObjectReturn<T>(string sql);

    }
}
```

Figure 29 : ISqlDataAccess is a DapperAPI.dbServices interface.

2.2.2.5 *Entity Framework Core's SQL Data Access*

SqlDataAccess sets the definition of a class, which is located inside the Entity Framework library and derives from the DbContext class as can be seen in Figure 30. This class is responsible for setting a connection to a SQL database and interacting with it by means of EF. It also manages the associated access tokens. An instance of the DbContext OptionsSqlDataAccess class must be sent along to the SqlDataAccess class's constructor for it to be possible to instantiate the class. The course also includes a characteristic known as Books, which has the type DbSet<Books>. This characteristic allows for a variety of actions to be performed on the Books table that is included inside the linked SQL database. The SqlDataAccess class acts as a mediator between the SQL database and the application, providing an abstraction layer.



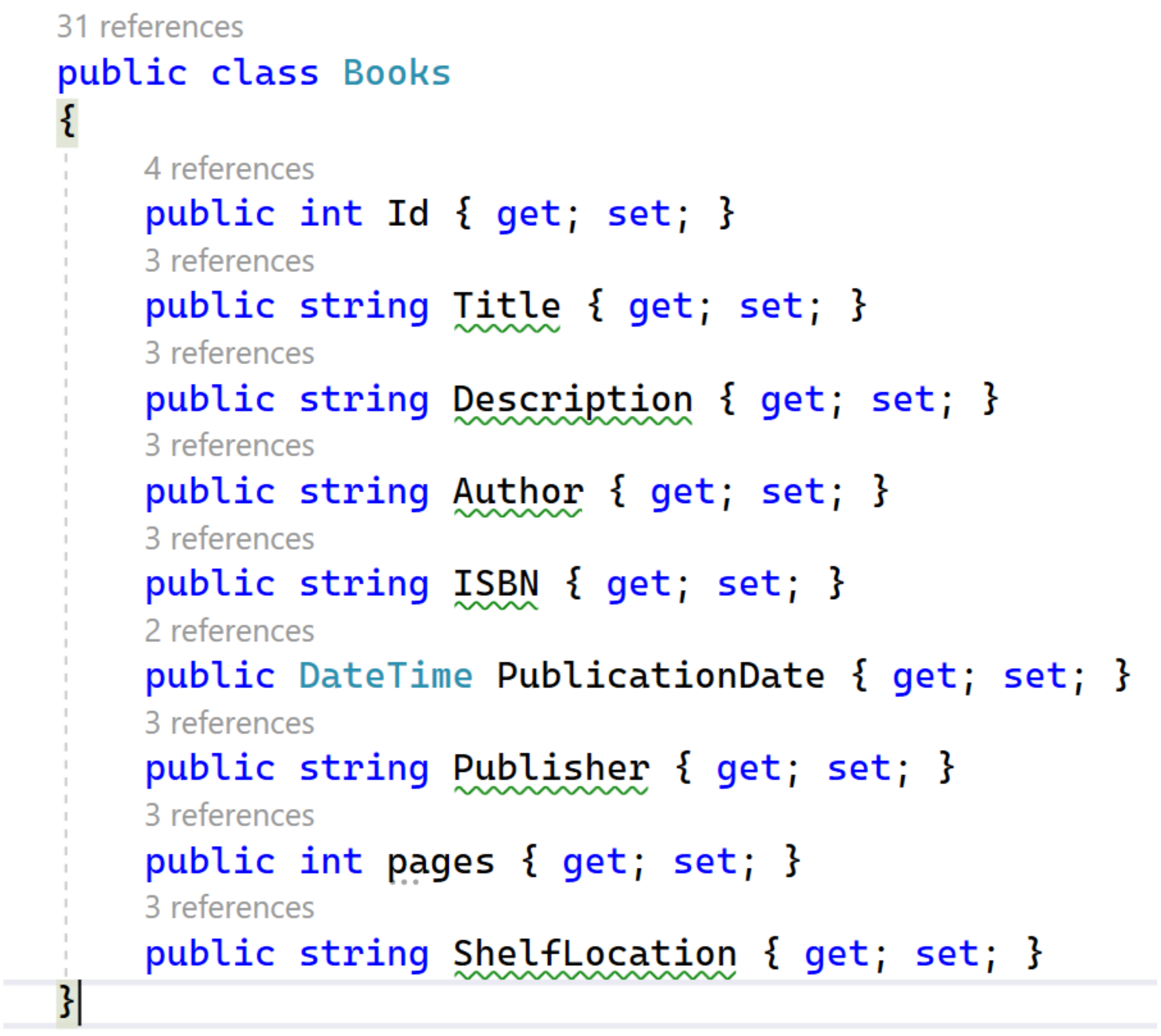
```
namespace EntityFramework.dbServeicer
{
    5 references
    public class SqlDataAcceess : DbContext
    {

        0 references
        public SqlDataAcceess(DbContextOptions<SqlDataAcceess> options) : base(options)
        {

        }

        5 references
        public DbSet<Books> Books { get; set; }
    }
}
```

Figure 30 : User's code defines a class called SqlDataAccess in Entity Framework, derived from DbContext.

2.2.3 Model

2.2.3.1 Dapper's Models

The ” DapperAPI.Models" includes the definition of a class known as "Books", as observed in Figure 31. These books include a unique identifier, a title, a description, an author, an International Standard Book Number (ISBN), a PublicationDate element, a Publisher name, a total number of pages, and a ShelfLocation string. These characteristics enable the storing and retrieval of data relevant to a literary work, which is facilitated by the characteristics discussed so far. The Id is a unique identifier for the book, the Title is a property of the literary work, the Description is a concise overview or synopsis of the book, the Author is a string that specifies the name of the person who wrote the book, the International Standard Book Number (ISBN) is associated with a certain book, the PublicationDate is a DateTime data type, the Publisher is a string that contains the name of the publishing house, the Pages is an integer, and the ShelfLocation string represents the specific location of the book on

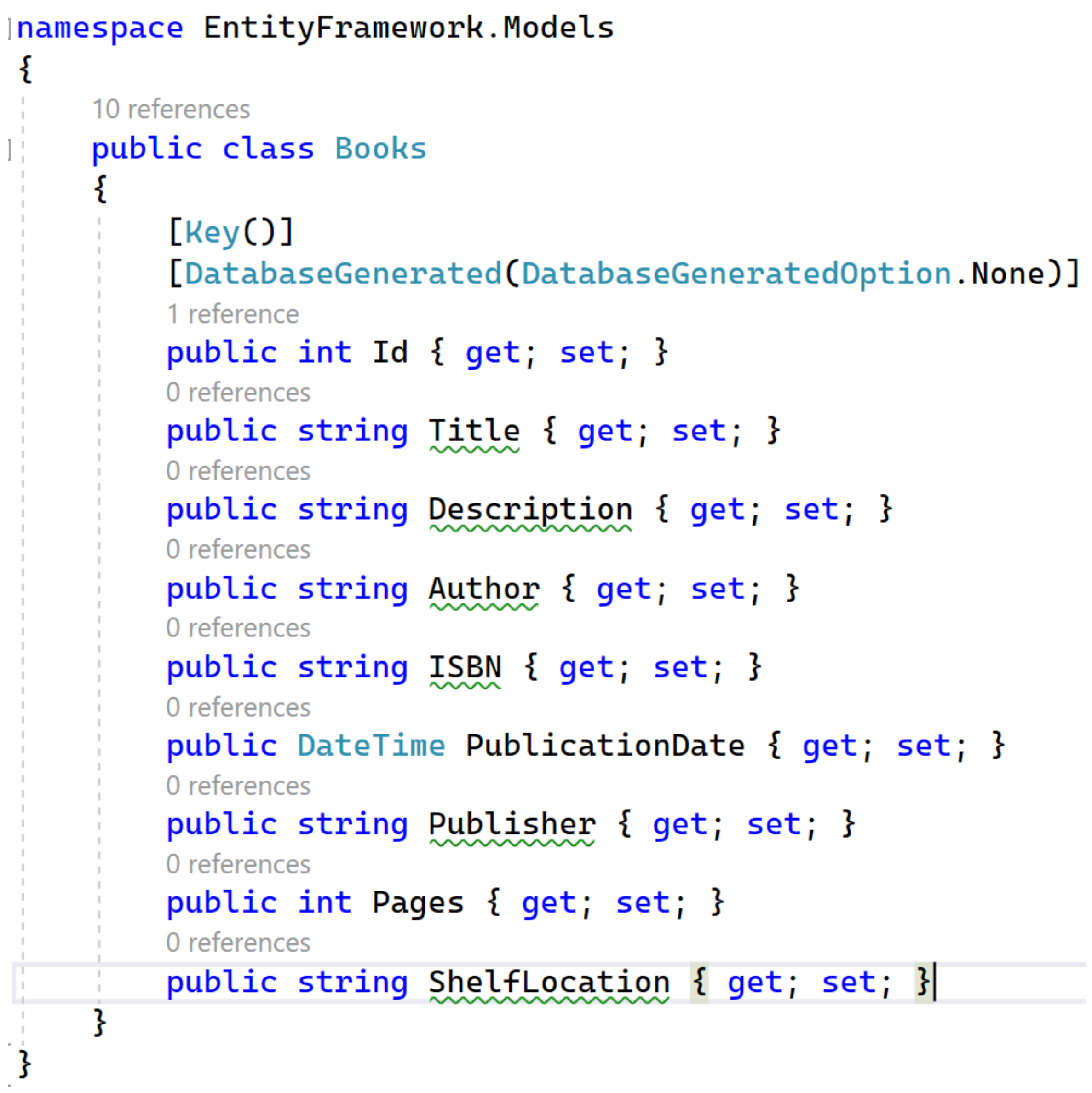


```
31 references
public class Books
{
    4 references
    public int Id { get; set; }
    3 references
    public string Title { get; set; }
    3 references
    public string Description { get; set; }
    3 references
    public string Author { get; set; }
    3 references
    public string ISBN { get; set; }
    2 references
    public DateTime PublicationDate { get; set; }
    3 references
    public string Publisher { get; set; }
    3 references
    public int pages { get; set; }
    3 references
    public string ShelfLocation { get; set; }
}
```

Figure 31 : The code has a class called "Books" in the "DapperAPI.Models" namespace.

2.2.3.2 *Entity Framework Core's Models*

This class acts as a model for books stored in a database as shown in Figure 32. It includes an integer identifier, title, description, author, ISBN, publication date, and publisher. The Id, Title, Description, Author, ISBN, Publication Date, and Publisher attributes are used to identify the book. It uses the "Pages" property, "ShelfLocation" string, System.Utilisation of the ComponentModel platform, schema namespaces, the "Id" property, and Entity Framework to represent book data inside a database. The "Pages" property indicates the total number of pages, while the "ShelfLocation" string represents the precise location of a book on a shelf.



```
namespace EntityFramework.Models
{
    10 references
    public class Books
    {
        [Key()]
        [DatabaseGenerated(DatabaseGeneratedOption.None)]
        1 reference
        public int Id { get; set; }
        0 references
        public string Title { get; set; }
        0 references
        public string Description { get; set; }
        0 references
        public string Author { get; set; }
        0 references
        public string ISBN { get; set; }
        0 references
        public DateTime PublicationDate { get; set; }
        0 references
        public string Publisher { get; set; }
        0 references
        public int Pages { get; set; }
        0 references
        public string ShelfLocation { get; set; }
    }
}
```

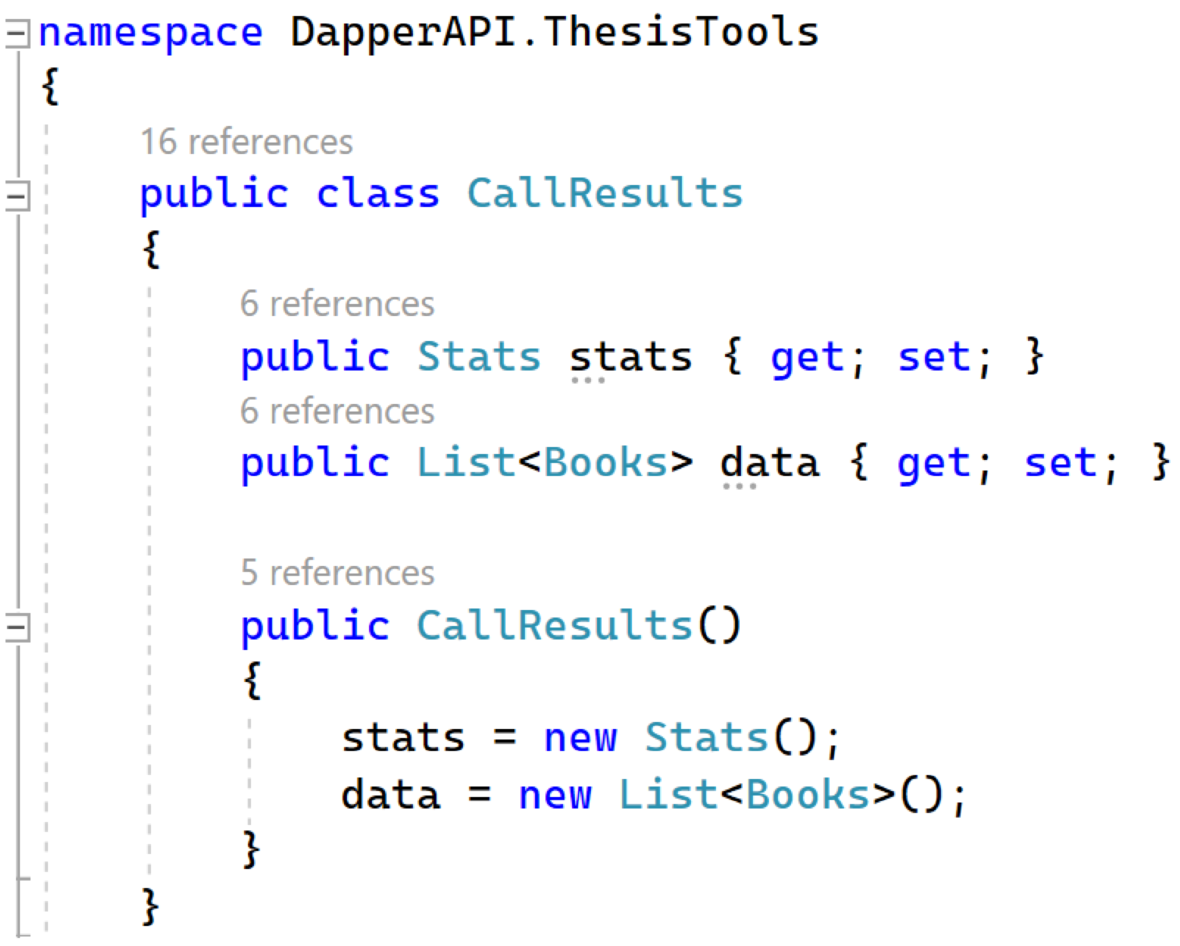
Figure 32: The code creates a "Books" class using Entity Framework as a model for database-stored books.

2.2.4 Call Results

2.2.4.1 Dapper's Call Results

The CallResults class is a construct created by DapperAPI. It is composed of two attributes, stats, and data as can be observed in Figure 33. A list of objects of type Books is included inside the data attribute of the object. The CallResults class has a constructor called

CallResults () that initializes the stats property and the data property with a list of Books that does not yet include any entries. The major function of the CallResults class is to act as a repository for call results, storing statistical data in the form of stats and a collection of information pertaining to books in the form of data.



```
namespace DapperAPI.ThesisTools
{
    16 references
    public class CallResults
    {
        6 references
        public Stats stats { get; set; }
        6 references
        public List<Books> data { get; set; }

        5 references
        public CallResults()
        {
            stats = new Stats();
            data = new List<Books>();
        }
    }
```

Figure 33:CallResults class in DapperAPI consists of two attributes: stats and data.

2.2.4.2 *Dapper's Stats*

There are three different attributes available for the "Stats" class: the "start" property, the "end" attribute, and the "milliseconds" attribute. The keywords "get", and "set" indicate that every property comes equipped with a getter and setter that may be accessed by the public. The goal of this class is to provide statistical information that is relevant to a particular process or activity, including the starting and ending values, as well as the total amount of time, which will be measured in milliseconds.

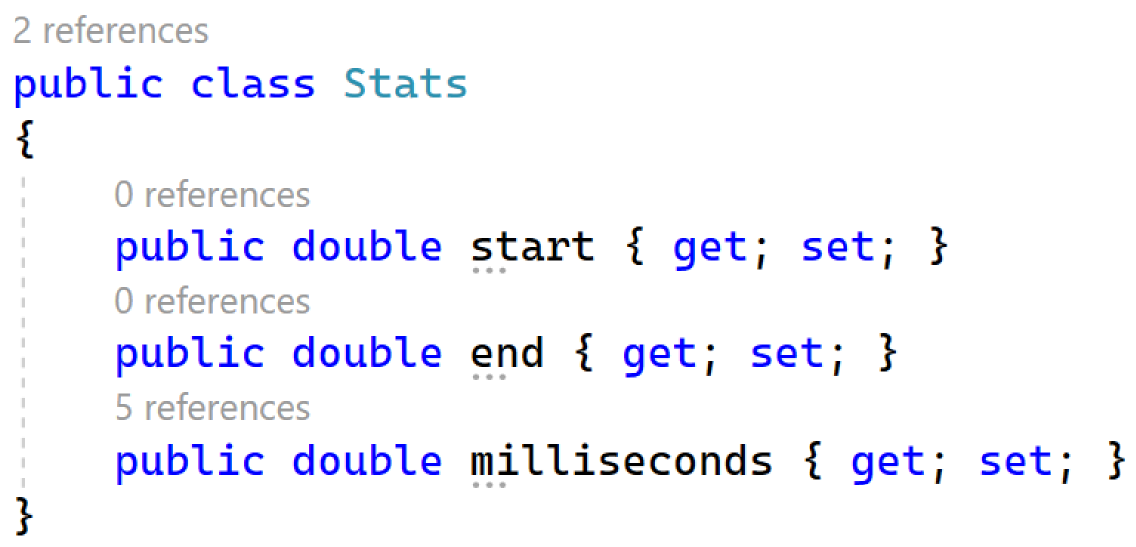
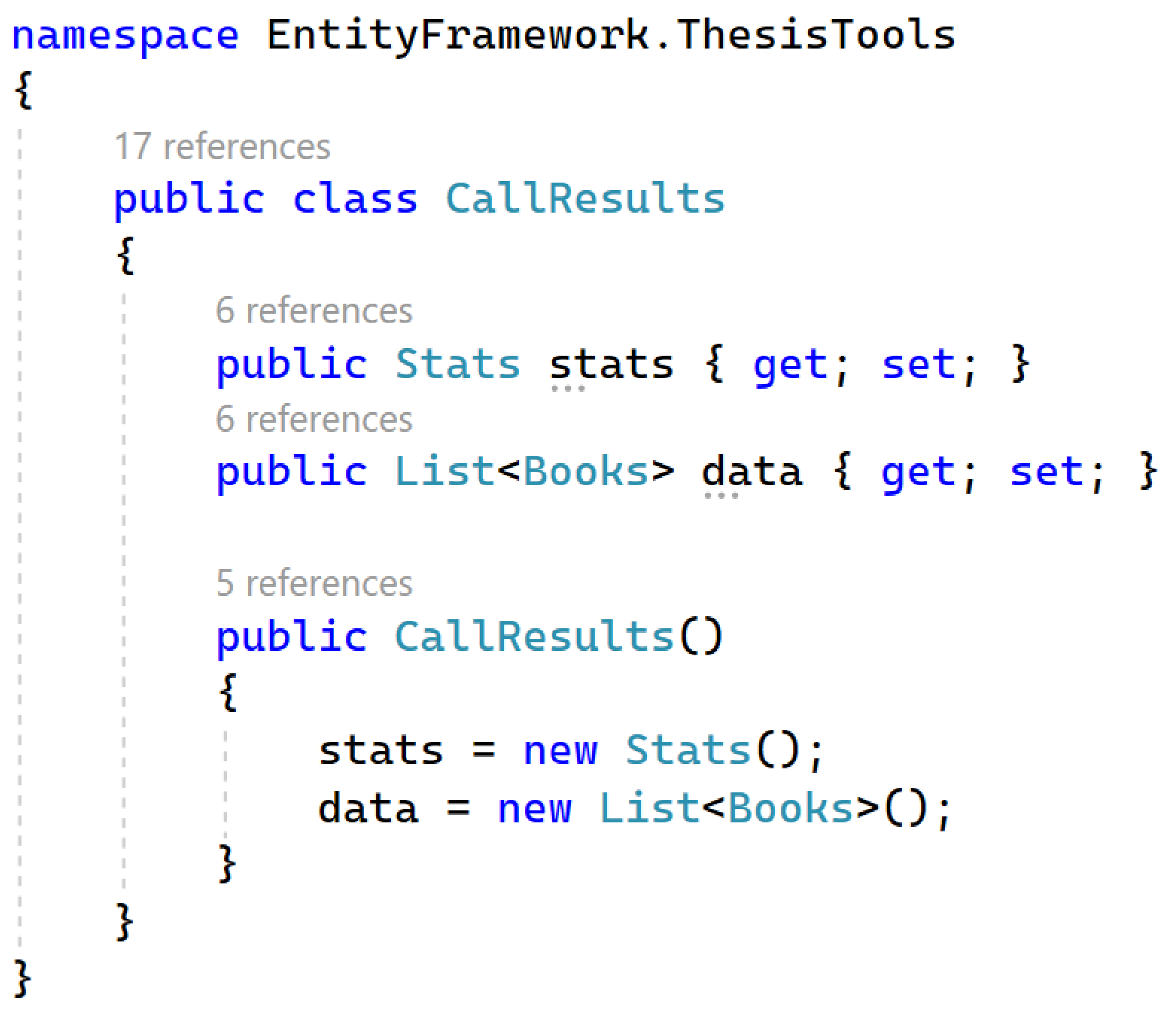

```
2 references
public class Stats
{
    0 references
    public double start { get; set; }
    0 references
    public double end { get; set; }
    5 references
    public double milliseconds { get; set; }
}
```

Figure 34: Code instantiates "Stats" class in "DapperAPI.ThesisTools" namespace.

2.2.4.3 Entity Framework Core's Call Results Stats

The "EntityFramework.ThesisTools" namespace contains a class called "CallResults" which is distinguished by two qualities: statistics and information. The "stats" property is classed as a type of "Stats" and the "data" property is of the type "List<Books>" AS SHOWN IN Figure 35. The "stats" property is initialised with a new instance of the "Stats" class and the "data" property is initialised with an empty list of "Books". The default constructor of the class is provided for initialising the attributes with values that are appropriate by default.

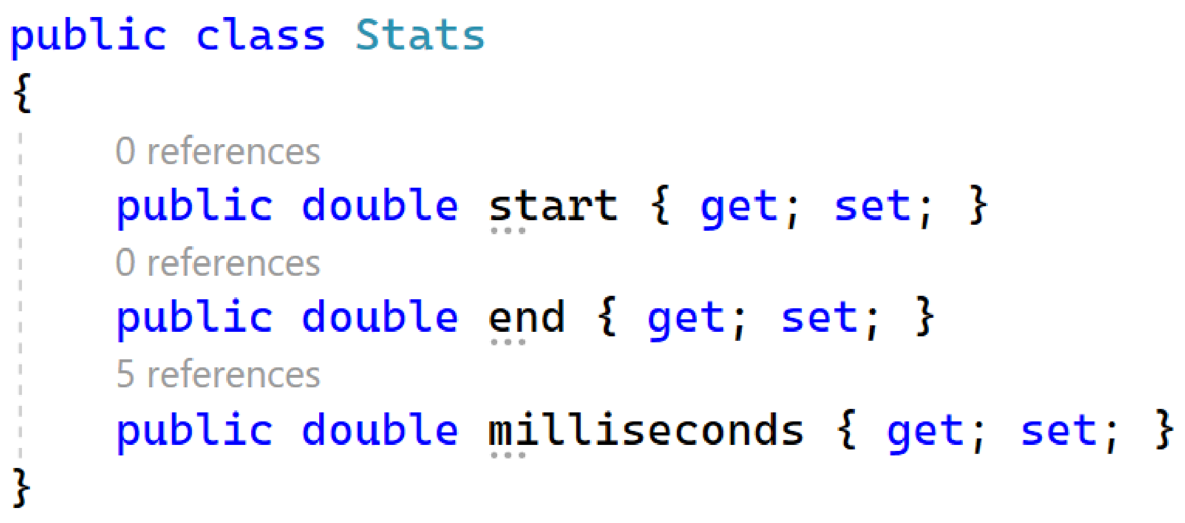


```
namespace EntityFramework.ThesisTools
{
    17 references
    public class CallResults
    {
        6 references
        public Stats stats { get; set; }
        6 references
        public List<Books> data { get; set; }

        5 references
        public CallResults()
        {
            stats = new Stats();
            data = new List<Books>();
        }
    }
}
```

Figure 35 : The "EntityFramework.ThesisTools" namespace has a class called "CallResults" with statistics and information qualities.

2.2.4.4 Entity Framework Core's Stats

An instance of a class called "Stats" under the "EntityFramework.ThesisTools" namespace as observed in Figure 36. The class has three properties, namely "start," "end," and "milliseconds," which are all the double data types. The properties are accessible to the public and have a getter accessor in addition to a setter accessor. The values of the characteristics may be assigned or retrieved in a manner analogous to the way that traditional variables are handled. The "Stats" course is an elementary kind of data construction that is composed of statistical information. It has a trio of properties that can store numerical data, referred to as "start," "end," and "milliseconds," respectively.

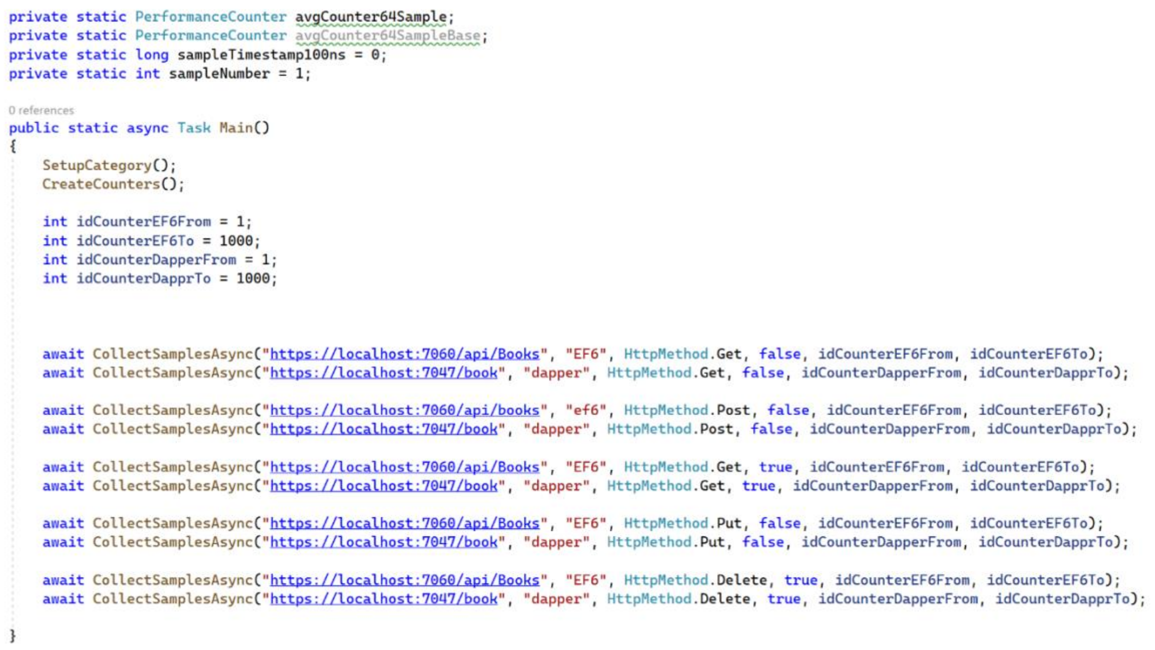


```
public class Stats
{
    0 references
    public double start { get; set; }
    0 references
    public double end { get; set; }
    5 references
    public double milliseconds { get; set; }
}
```

Figure 36 : The code creates an instance of "Stats" class in the "EntityFramework.ThesisTools" namespace.

2.3 Performance Counter and HTTP Request

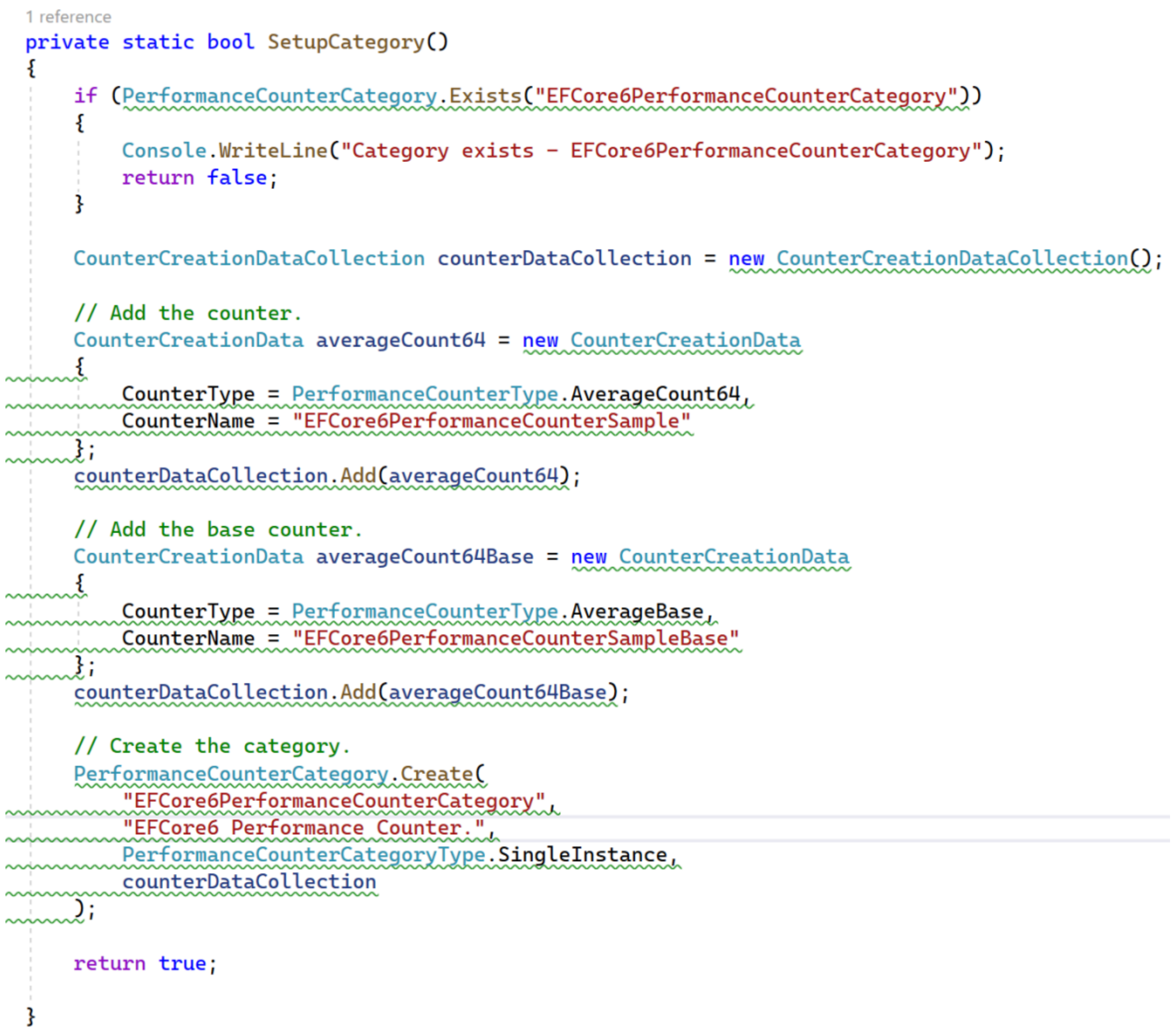
The class incorporates several private static variables, such as the Performance Counter class and the SetupCategory and Create Counters methods as shown in Figure 37. The Main function calls the CollectSamplesAsync procedure several times, and each invocation passes a unique set of inputs to the system. The CollectSamplesAsync function is used to acquire performance samples from a variety of URLs and HTTP protocols. The procedure requires the consideration of several factors, including the Uniform Resource Locator URL, a defined label (for example, "EF6" or "dapper"), the Hypertext Transfer Protocol (HTTP) method, a binary indicator, and a series of numerical values indicating a range of counters. In summary, the course entails the collecting of performance data from a variety of API endpoints via the use of various HTTP methods, while also applying performance counters to assess and analyze the performance metrics.



```
private static PerformanceCounter avgCounter64Sample;
private static PerformanceCounter avgCounter64SampleBase;
private static long sampleTimestamp100ns = 0;
private static int sampleNumber = 1;

0 references
public static async Task Main()
{
    SetupCategory();
    CreateCounters();

    int idCounterEF6From = 1;
    int idCounterEF6To = 1000;
    int idCounterDapperFrom = 1;
    int idCounterDapprTo = 1000;


    await CollectSamplesAsync("https://localhost:7060/api/Books", "EF6", HttpMethod.Get, false, idCounterEF6From, idCounterEF6To);
    await CollectSamplesAsync("https://localhost:7047/book", "dapper", HttpMethod.Get, false, idCounterDapperFrom, idCounterDapprTo);

    await CollectSamplesAsync("https://localhost:7060/api/books", "ef6", HttpMethod.Post, false, idCounterEF6From, idCounterEF6To);
    await CollectSamplesAsync("https://localhost:7047/book", "dapper", HttpMethod.Post, false, idCounterDapperFrom, idCounterDapprTo);

    await CollectSamplesAsync("https://localhost:7060/api/Books", "EF6", HttpMethod.Get, true, idCounterEF6From, idCounterEF6To);
    await CollectSamplesAsync("https://localhost:7047/book", "dapper", HttpMethod.Get, true, idCounterDapperFrom, idCounterDapprTo);

    await CollectSamplesAsync("https://localhost:7060/api/Books", "EF6", HttpMethod.Put, false, idCounterEF6From, idCounterEF6To);
    await CollectSamplesAsync("https://localhost:7047/book", "dapper", HttpMethod.Put, false, idCounterDapperFrom, idCounterDapprTo);

    await CollectSamplesAsync("https://localhost:7060/api/Books", "EF6", HttpMethod.Delete, true, idCounterEF6From, idCounterEF6To);
    await CollectSamplesAsync("https://localhost:7047/book", "dapper", HttpMethod.Delete, true, idCounterDapperFrom, idCounterDapprTo);

}
```

Figure 37: Curriculum includes private static variables like Performance Counter, SetupCategory, and Create Counters methods.

SetupCategory() is used to create a performance counter category with the name "EFCore6PerformanceCounterCategory". If the category does not exist, a new CounterCreationDataCollection object is created. The averageCount64 counter is used to represent the average count and uses the PerformanceCounterType. The AverageCount64 counter type is used in computer systems, and the EFCore6PerformanceCounterSample and EFCore6PerformanceCounterSampleBase are used to track how many times the averageCount64Base has been used. The code creates a performance counter category known as "EFCore6PerformanceCounterCategory" to encapsulate the performance metrics of the EFCore6 application, composed of two counters known as "EFCore6PerformanceCounterSample" and "EFCore6PerformanceCounterSampleBase" as can be observed in Figure 38.

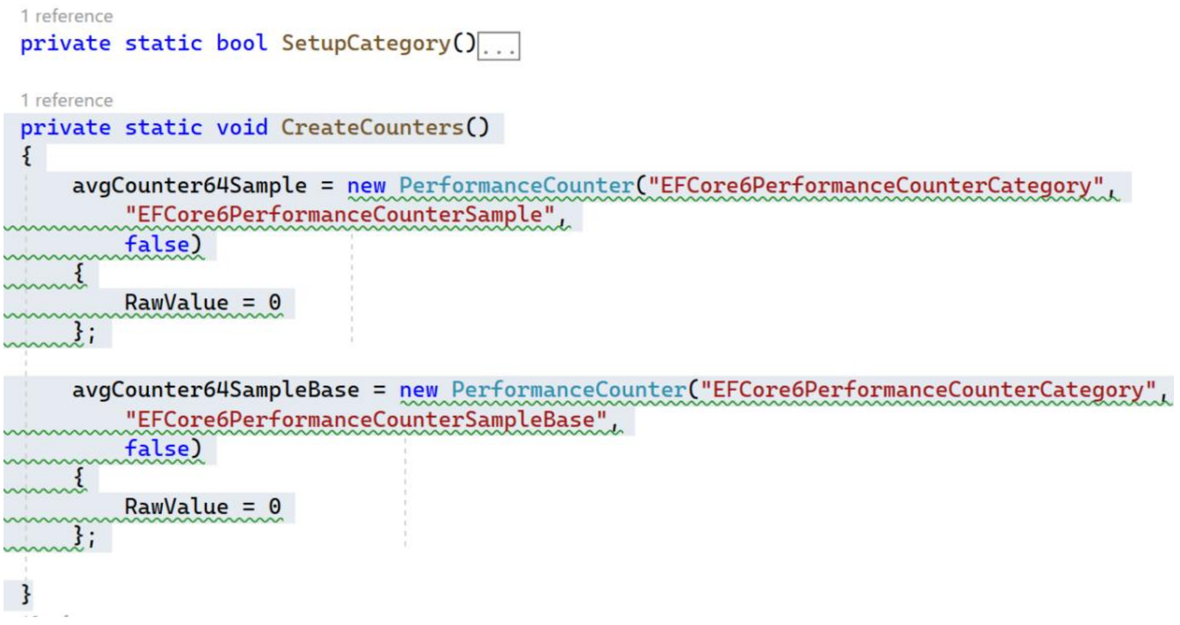


```
1 reference
private static bool SetupCategory()
{
    if (PerformanceCounterCategory.Exists("EFCore6PerformanceCounterCategory"))
    {
        Console.WriteLine("Category exists - EFCore6PerformanceCounterCategory");
        return false;
    }

    CounterCreationDataCollection counterDataCollection = new CounterCreationDataCollection();

    // Add the counter.
    CounterCreationData averageCount64 = new CounterCreationData
    {
        CounterType = PerformanceCounterType.AverageCount64,
        CounterName = "EFCore6PerformanceCounterSample"
    };
    counterDataCollection.Add(averageCount64);

    // Add the base counter.
    CounterCreationData averageCount64Base = new CounterCreationData
    {
        CounterType = PerformanceCounterType.AverageBase,
        CounterName = "EFCore6PerformanceCounterSampleBase"
    };
    counterDataCollection.Add(averageCount64Base);

    // Create the category.
    PerformanceCounterCategory.Create(
        "EFCore6PerformanceCounterCategory",
        "EFCore6 Performance Counter.",
        PerformanceCounterCategoryType.SingleInstance,
        counterDataCollection
    );

    return true;

}
```

Figure 38 : EFCore6PerformanceCounterSampleBase tracks usage.

"EFCore6PerformanceCounterCategory" has two counters for EFCore6 metrics.

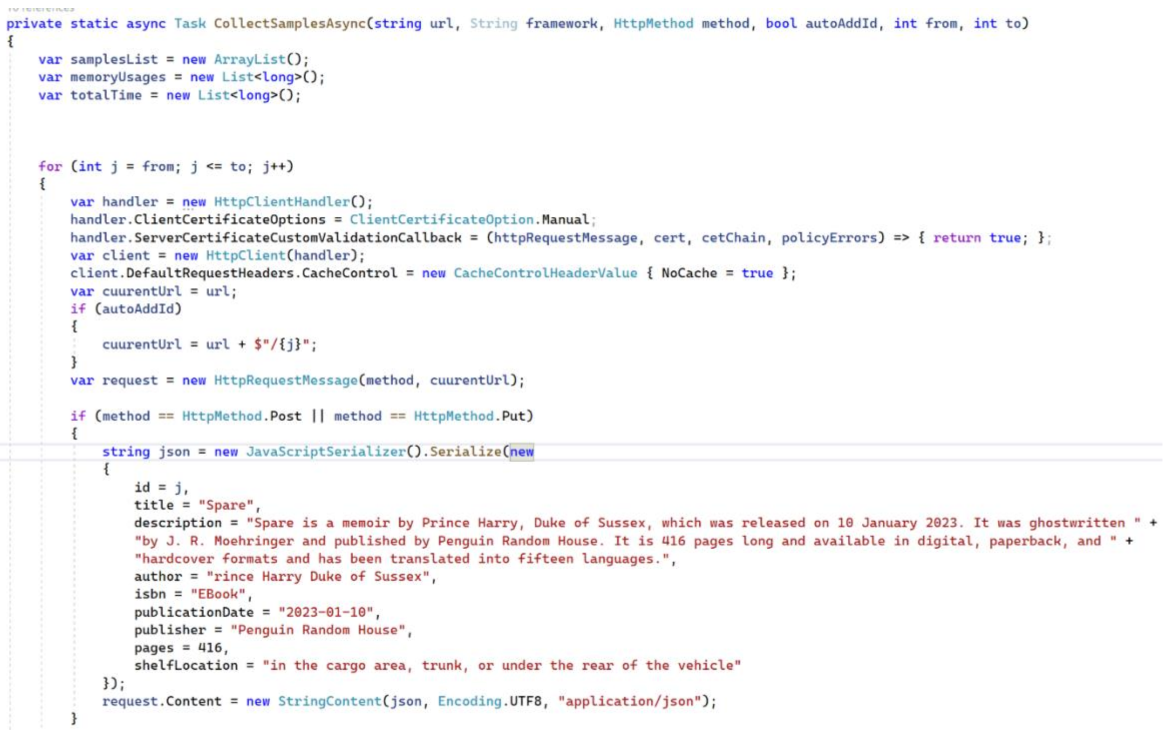
Ther is private static function called "CreateCounters". This function is responsible for the production of a pair of performance gauges, which are connected to a performance monitoring group concerning EF Core 6. The first counter, which has the notation "EFCore6PerformanceCounterSample," is an example of the category known as PerformanceCounter. The second counter, whose notation is "EFCore6PerformanceCounterSampleBase," stands for a further instance of the PerformanceCounter class. When calculating the mean value of the performance metric that is being assessed by the main counter, the beginning counter is used as the fundamental unit that serves as the basis for the calculation as shown in Figure 39. The two counters are created by using the same category denomination, which is referred to as the "EFCore6PerformanceCounterCategory". Additionally, the RawValue of both counters is set to zero when they are first created, which indicates that the monitoring of performance metrics will begin with zero.



```
1 reference
private static bool SetupCategory()[...]

1 reference
private static void CreateCounters()
{
    avgCounter64Sample = new PerformanceCounter("EFCore6PerformanceCounterCategory",
        "EFCore6PerformanceCounterSample",
        false)
    {
        RawValue = 0
    };

    avgCounter64SampleBase = new PerformanceCounter("EFCore6PerformanceCounterCategory",
        "EFCore6PerformanceCounterSampleBase",
        false)
    {
        RawValue = 0
    };

}
```

Figure 39 : Mean value of performance metric is calculated using the beginning counter as the basis.

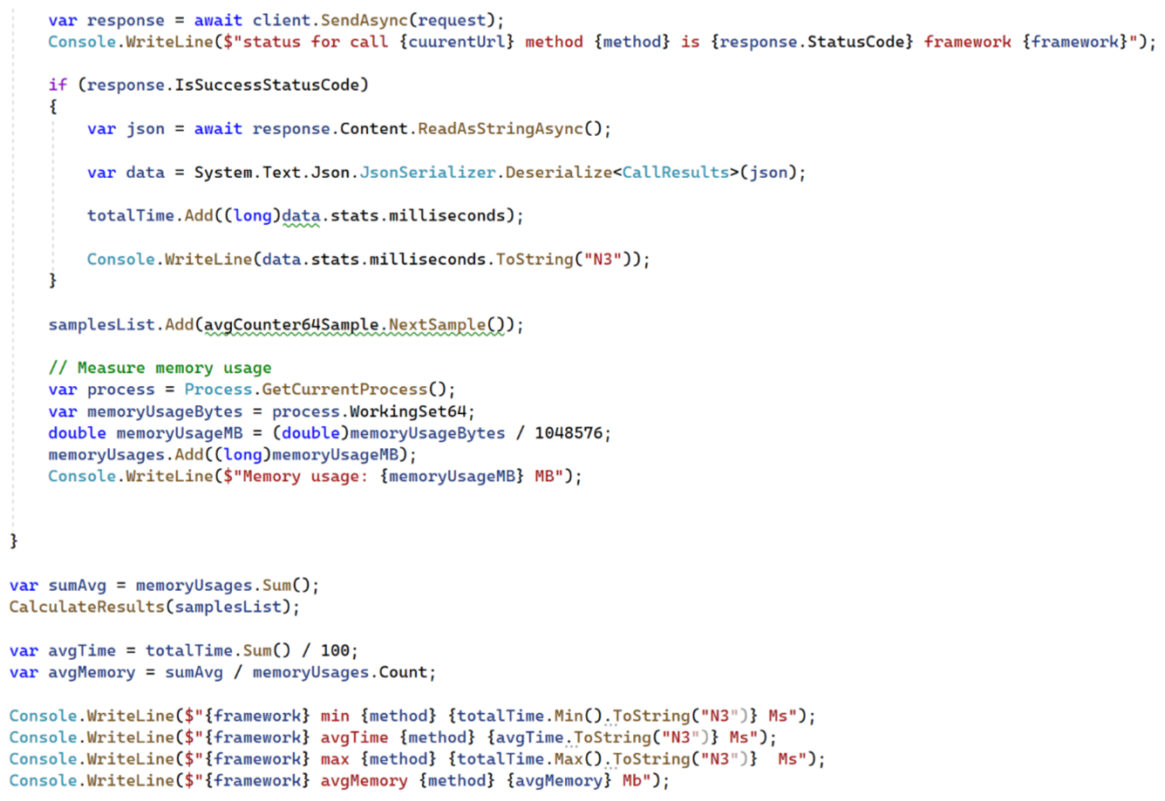
The CollectSamplesAsync class is responsible for retrieving samples by sending HTTP requests to a certain URL. It requires multiple input parameters, including the base URL, the framework, the HTTP method, the boolean value indicating whether an ID should be automatically appended to the URL, and the starting and ending points for a loop. The procedure sets up three data structures: an ArrayList with the name samplesList, a Listlong with the name memoryUsages, and a Listlong with the name total time. During each cycle of the loop, an instance of the HttpClientHandler class is created and set up to receive server certificates without any limitations. The Post method is the subject of this conversation, which involves the serialization of a JSON object that is then allocated as the content of the request once the serialization process has been completed. The program uses the HttpClient to wait for a response and sends the answer status code and any further data to the console. If the answer is successful, a JSON string is used to represent the content of the answer and is transformed into an instance of the CallResults type.Text.Json.JsonSerializer. A new entry is added to the total-time list that contains the time value that was retrieved from the deserialized object. A model is added to the samples list by using an avgCounter64Sample object. The function determines how much memory is being used by the current process and adds that information to the list of functions that use memory as can be observed in Figure 40.



```
private static async Task CollectSamplesAsync(string url, String framework, HttpMethod method, bool autoAddId, int from, int to)
{
    var samplesList = new ArrayList();
    var memoryUsages = new List<long>();
    var totalTime = new List<long>();



    for (int j = from; j <= to; j++)
    {
        var handler = new HttpClientHandler();
        handler.ClientCertificateOptions = ClientCertificateOption.Manual;
        handler.ServerCertificateCustomValidationCallback = (httpRequestMessage, cert, cetChain, policyErrors) => { return true; };
        var client = new HttpClient(handler);
        client.DefaultRequestHeaders.CacheControl = new CacheControlHeaderValue { NoCache = true };
        var cuurentUrl = url;
        if (autoAddId)
        {
            cuurentUrl = url + $"/{j}";
        }
        var request = new HttpRequestMessage(method, cuurentUrl);

        if (method == HttpMethod.Post || method == HttpMethod.Put)
        {
            string json = new JavaScriptSerializer().Serialize(new
            {
                id = j,
                title = "Spare",
                description = "Spare is a memoir by Prince Harry, Duke of Sussex, which was released on 10 January 2023. It was ghostwritten " +
                "by J. R. Moehringer and published by Penguin Random House. It is 416 pages long and available in digital, paperback, and " +
                "hardcover formats and has been translated into fifteen languages.",
                author = "rince Harry Duke of Sussex",
                isbn = "EBook",
                publicationDate = "2023-01-10",
                publisher = "Penguin Random House",
                pages = 416,
                shelfLocation = "in the cargo area, trunk, or under the rear of the vehicle"
            });
            request.Content = new StringContent(json, Encoding.UTF8, "application/json");
        }
```

Figure 40 : New entry added to total-time list. Sample added to samples list. Function tracks and updates memory usage.

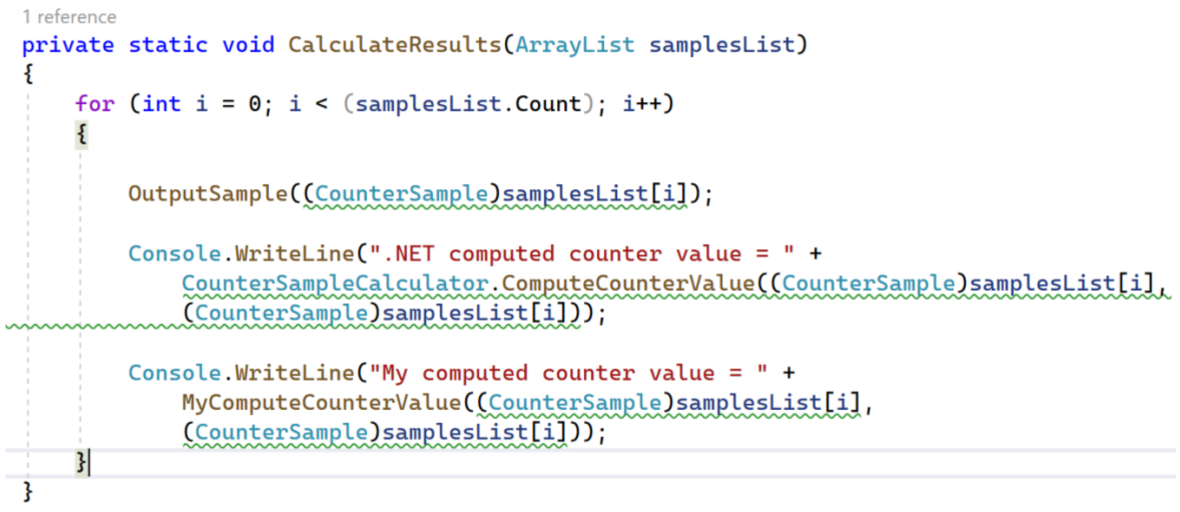
After the loop has been completed, the total of the memory use values is calculated, and a method called CalculateResults is called. The console shows the lowest, average, and maximum values for time, as well as the average value for memory, and it also indicates the framework and technique that is being used as shown in Figure 41.



```
        var response = await client.SendAsync(request);
        Console.WriteLine($"status for call {cuurentUrl} method {method} is {response.StatusCode} framework {framework}");

        if (response.IsSuccessStatusCode)
        {
            var json = await response.Content.ReadAsStringAsync();

            var data = System.Text.Json.JsonSerializer.Deserialize<CallResults>(json);

            totalTime.Add((long)data.stats.milliseconds);

            Console.WriteLine(data.stats.milliseconds.ToString("N3"));
        }

        samplesList.Add(avgCounter64Sample.NextSample());

        // Measure memory usage
        var process = Process.GetCurrentProcess();
        var memoryUsageBytes = process.WorkingSet64;
        double memoryUsageMB = (double)memoryUsageBytes / 1048576;
        memoryUsages.Add((long)memoryUsageMB);
        Console.WriteLine($"Memory usage: {memoryUsageMB} MB");


    }

    var sumAvg = memoryUsages.Sum();
    CalculateResults(samplesList);

    var avgTime = totalTime.Sum() / 100;
    var avgMemory = sumAvg / memoryUsages.Count;

    Console.WriteLine($"{framework} min {method} {totalTime.Min().ToString("N3")} Ms");
    Console.WriteLine($"{framework} avgTime {method} {avgTime.ToString("N3")} Ms");
    Console.WriteLine($"{framework} max {method} {totalTime.Max().ToString("N3")}  Ms");
    Console.WriteLine($"{framework} avgMemory {method} {avgMemory} Mb");
```

Figure 41 : Loop completes, CalculateResults calculates memory total. Console shows time and memory stats, framework/technique indicated.

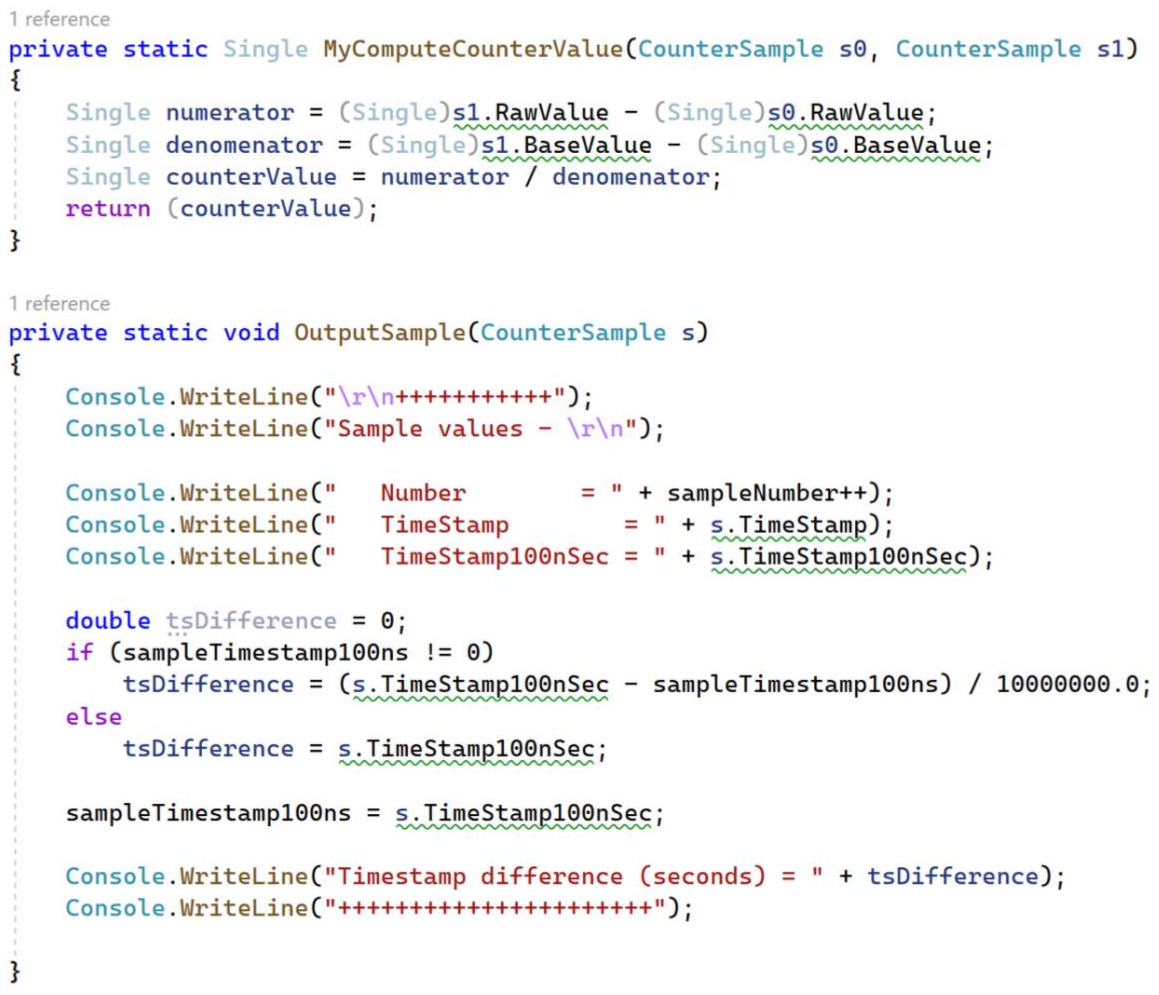
The code snippet demonstrates a private static function known as CalculateResults, which includes a for loop and three key functions. It is called by typecasting the currently active member of the samples list into an instance of the CounterSample class, and the CounterSampleCalculator is used to compute and display the calculated counter value. The MyComputeCounterValue method is used to determine the calculated counter value, and the results are displayed using the.NET calculated counter value and my computed counter value statements as shown in Figure 42.



```
1 reference
private static void CalculateResults(ArrayList samplesList)
{
    for (int i = 0; i < (samplesList.Count); i++)
    {

        OutputSample((CounterSample)samplesList[i]);

        Console.WriteLine(".NET computed counter value = " +
            CounterSampleCalculator.ComputeCounterValue((CounterSample)samplesList[i],
            (CounterSample)samplesList[i]));

        Console.WriteLine("My computed counter value = " +
            MyComputeCounterValue((CounterSample)samplesList[i],
            (CounterSample)samplesList[i]));
    }
}
```

Figure 42 : Code has CalculateResults function that computes and displays counter values using CounterSample and CounterSampleCalculator.

This code snippet outlines a class that consists of two procedures, MyComputeCounterValue and OutputSample. MyComputeCounterValue has two input arguments, both of which are instances of the CounterSample class. The procedure entails carrying out a calculation by making use of the numerical data collected from the samples, which ultimately results in a single output. The numerator is calculated using the method described above, while the denominator is found by computing the difference between the BaseValues of s0 and s1 and then subtracting that value from the first BaseValue. The counter value is calculated by dividing the numerator by the denominator. The program compiles a wide variety of data about the sample and presents it to the user through the console. It is important to note that the shown code segment only demonstrates the class's methods; it is impossible to infer the class specification, or any more particulars based purely on this code segment as shown in Figure 43

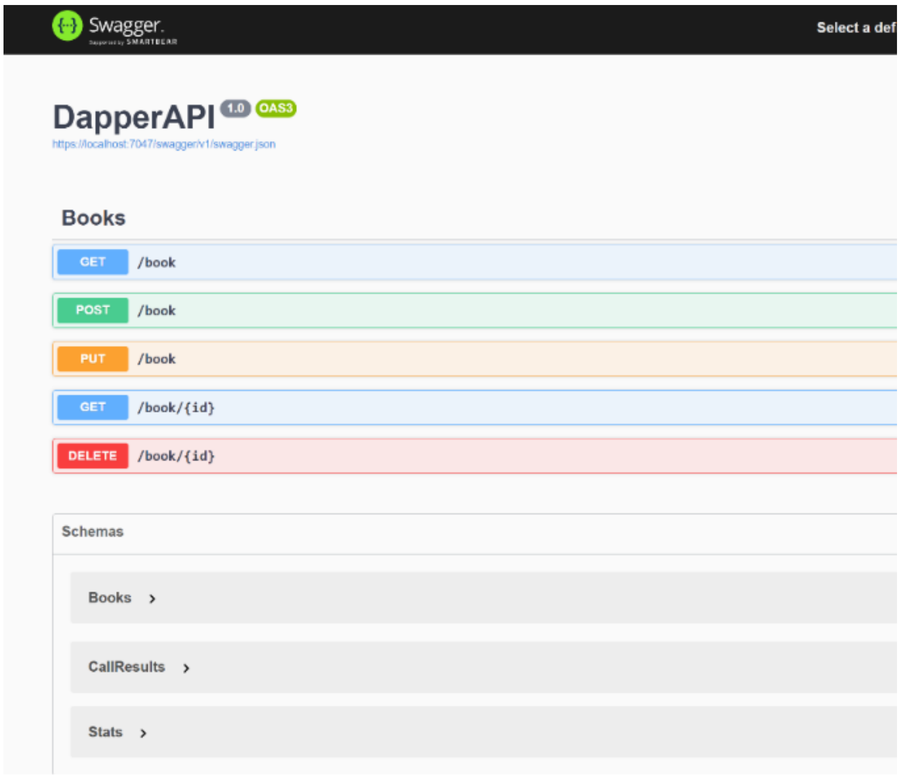


```
1 reference
private static Single MyComputeCounterValue(CounterSample s0, CounterSample s1)
{
    Single numerator = (Single)s1.RawValue - (Single)s0.RawValue;
    Single denomenator = (Single)s1.BaseValue - (Single)s0.BaseValue;
    Single counterValue = numerator / denomenator;
    return (counterValue);
}

1 reference
private static void OutputSample(CounterSample s)
{
    Console.WriteLine("\r\n+++++++++++");
    Console.WriteLine("Sample values - \r\n");

    Console.WriteLine("   Number        = " + sampleNumber++);
    Console.WriteLine("   TimeStamp        = " + s.TimeStamp);
    Console.WriteLine("   TimeStamp100nSec = " + s.TimeStamp100nSec);

    double tsDifference = 0;
    if (sampleTimestamp100ns != 0)
        tsDifference = (s.TimeStamp100nSec - sampleTimestamp100ns) / 10000000.0;
    else
        tsDifference = s.TimeStamp100nSec;

    sampleTimestamp100ns = s.TimeStamp100nSec;

    Console.WriteLine("Timestamp difference (seconds) = " + tsDifference);
    Console.WriteLine("++++++++++++++++++++++");

}
```

Figure 43 : Code snippet: Class with procedures, input arguments of CounterSample class.

2.4 Screenshot of app

After the application is run successfully, As the supplied example demonstrates, it is possible to retrieve the Swagger documentation that is associated with the Dapper API.. This UI helps users to access the endpoints through an interface. The interface shows what the application has in the specific API. Each endpoint call gives the details of the call and the timing of the endpoint. In Figure 44, the endpoints that are used are by dapper API, and Figure 45 is for Entity Framework.



Swagger.

Select a def

DapperAPI 1.0 OAS3

https://localhost:7047/swagger/v1/swagger.json

Books

GET /book

POST /book

PUT /book

GET /book/{id}

DELETE /book/{id}

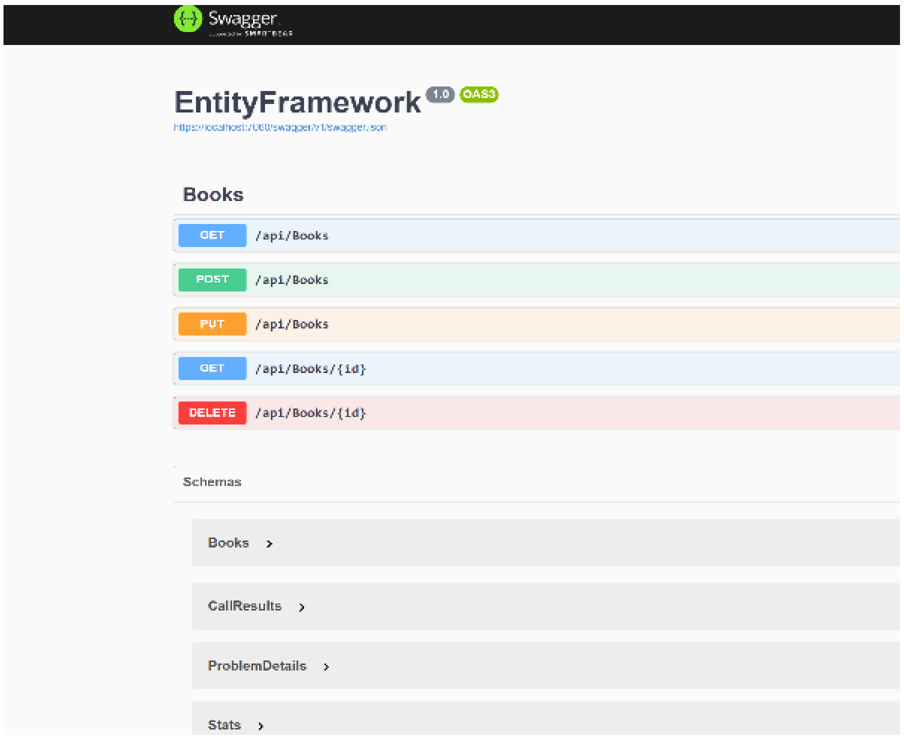
Schemas

Books ›

CallResults ›

Stats ›

Figure 44 : Swager dapper API.



Swagger.

EntityFramework 1.0 OAS3

Books

GET /api/Books

POST /api/Books

PUT /api/Books

GET /api/Books/{id}

DELETE /api/Books/{id}

Schemas

Books ›

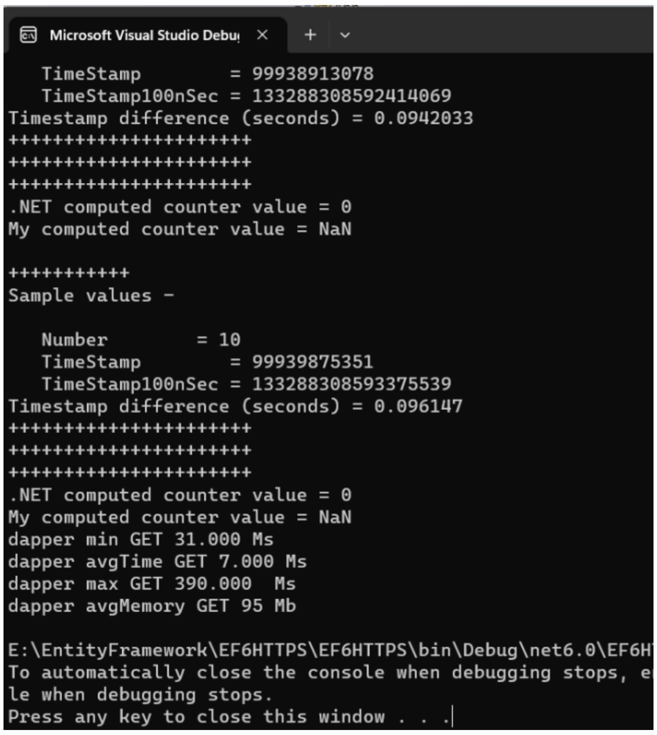
CallResults ›

ProblemDetails ›

Stats ›

Figure 45 : Swagger Entity Framework API.

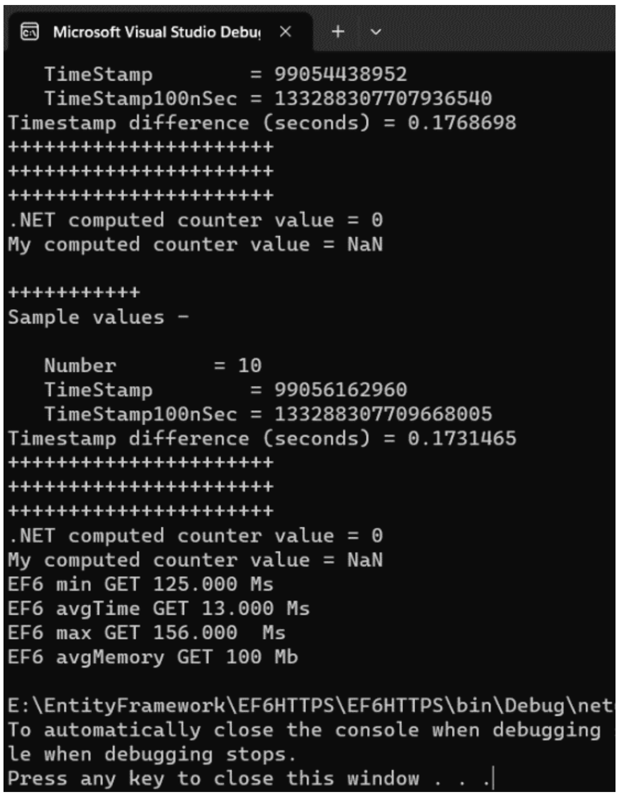
In Figure 46 and Figure 47 a console application is shown, that includes the performance counter. The application calculates the minimum time, maximum time, average time, and the average memory. Thousands of requests that are sent to and from the database are collected and put for the user to see and have a visual on how long the thousand transactions took.



Microsoft Visual Studio Debu

```
   TimeStamp        = 99938913078
   TimeStamp100nSec = 133288308592414069
Timestamp difference (seconds) = 0.0942033
++++++++++++++++++++++
++++++++++++++++++++++
++++++++++++++++++++++
.NET computed counter value = 0
My computed counter value = NaN

+++++++++++
Sample values -

   Number        = 10
   TimeStamp        = 99939875351
   TimeStamp100nSec = 133288308593375539
Timestamp difference (seconds) = 0.096147
++++++++++++++++++++++
++++++++++++++++++++++
++++++++++++++++++++++
.NET computed counter value = 0
My computed counter value = NaN
dapper min GET 31.000 Ms
dapper avgTime GET 7.000 Ms
dapper max GET 390.000  Ms
dapper avgMemory GET 95 Mb

E:\EntityFramework\EF6HTTPS\EF6HTTPS\bin\Debug\net6.0\EF6H
To automatically close the console when debugging stops, e
le when debugging stops.
Press any key to close this window . . .
```

Figure 46 : The console for Thousands of transactions for Dapper.

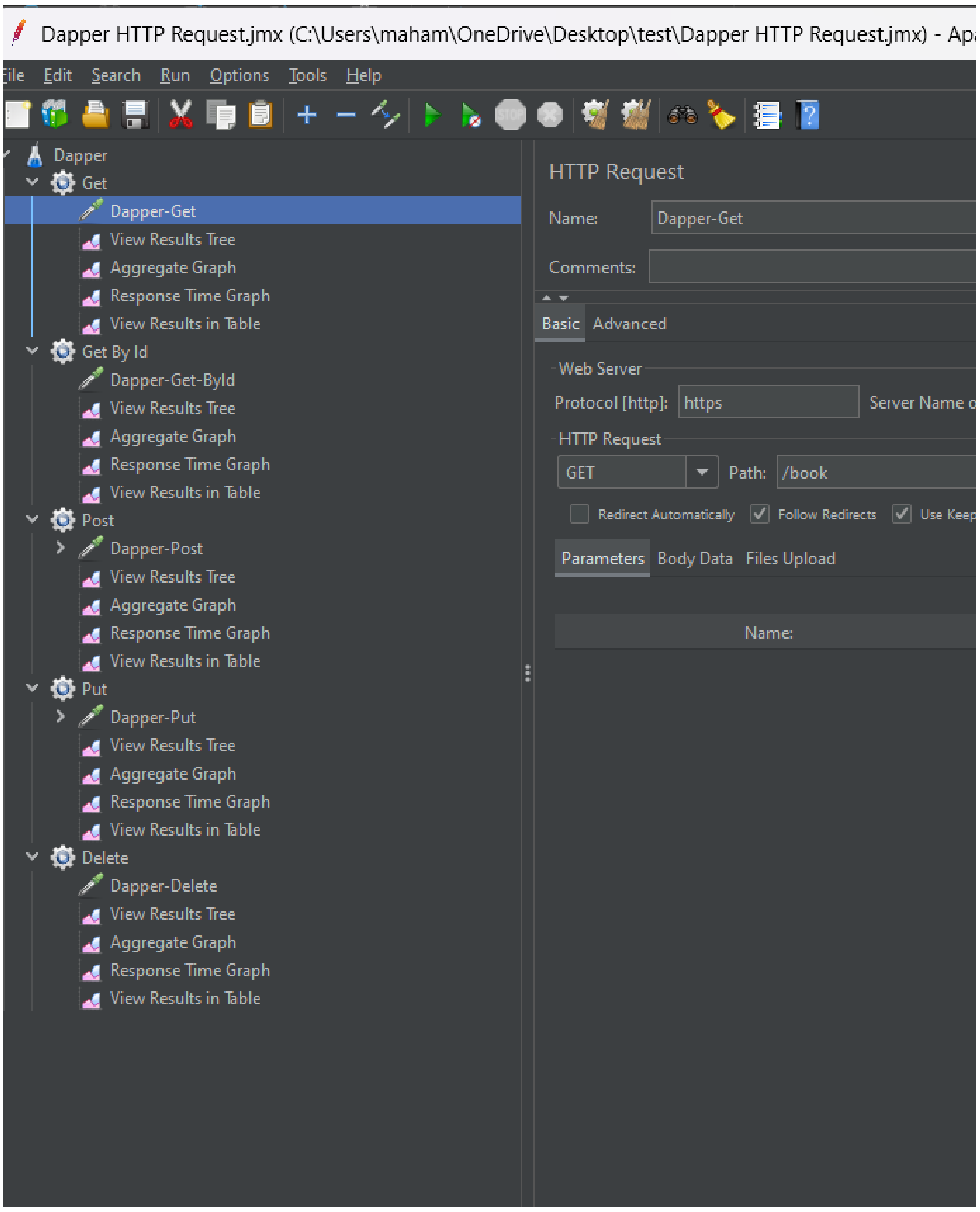


Microsoft Visual Studio Debu

```
   TimeStamp        = 99054438952
   TimeStamp100nSec = 133288307707936540
Timestamp difference (seconds) = 0.1768698
++++++++++++++++++++++
++++++++++++++++++++++
++++++++++++++++++++++
.NET computed counter value = 0
My computed counter value = NaN

+++++++++++
Sample values -

   Number        = 10
   TimeStamp        = 99056162960
   TimeStamp100nSec = 133288307709668005
Timestamp difference (seconds) = 0.1731465
++++++++++++++++++++++
++++++++++++++++++++++
++++++++++++++++++++++
.NET computed counter value = 0
My computed counter value = NaN
EF6 min GET 125.000 Ms
EF6 avgTime GET 13.000 Ms
EF6 max GET 156.000  Ms
EF6 avgMemory GET 100 Mb

E:\EntityFramework\EF6HTTPS\EF6HTTPS\bin\Debug\net
To automatically close the console when debugging
le when debugging stops.
Press any key to close this window . . .
```

Figure 47 : The console for Thousands of transactions for Entity Framework.

Apache JMeter is a library used for testing both APIs Entity Framework and Dapper. The library’s usage is to calculate the minimum time, maximum time, average time, and average memory of endpoints with already built in functionality as shown in Figure 48. Unlike the functionality inside the application, this library already has everything built-in for timing and estimating the calls memory. The only 2 things necessary for this application to work are the JSON file and the local host for the swagger as shown in Figure 49.



Dapper HTTP Request.jmx (C:\Users\maham\OneDrive\Desktop\test\Dapper HTTP Request.jmx) - Ap

- Dapper
  - Get
    - Dapper-Get
    - View Results Tree
    - Aggregate Graph
    - Response Time Graph
    - View Results in Table
  - Get By Id
    - Dapper-Get-ById
    - View Results Tree
    - Aggregate Graph
    - Response Time Graph
    - View Results in Table
  - Post
    - Dapper-Post
    - View Results Tree
    - Aggregate Graph
    - Response Time Graph
    - View Results in Table
  - Put
    - Dapper-Put
    - View Results Tree
    - Aggregate Graph
    - Response Time Graph
    - View Results in Table
  - Delete
    - Dapper-Delete
    - View Results Tree
    - Aggregate Graph
    - Response Time Graph
    - View Results in Table

HTTP Request

Name: Dapper-Get

Comments:

Basic Advanced

Web Server

Protocol [http]: https Server Name o

HTTP Request

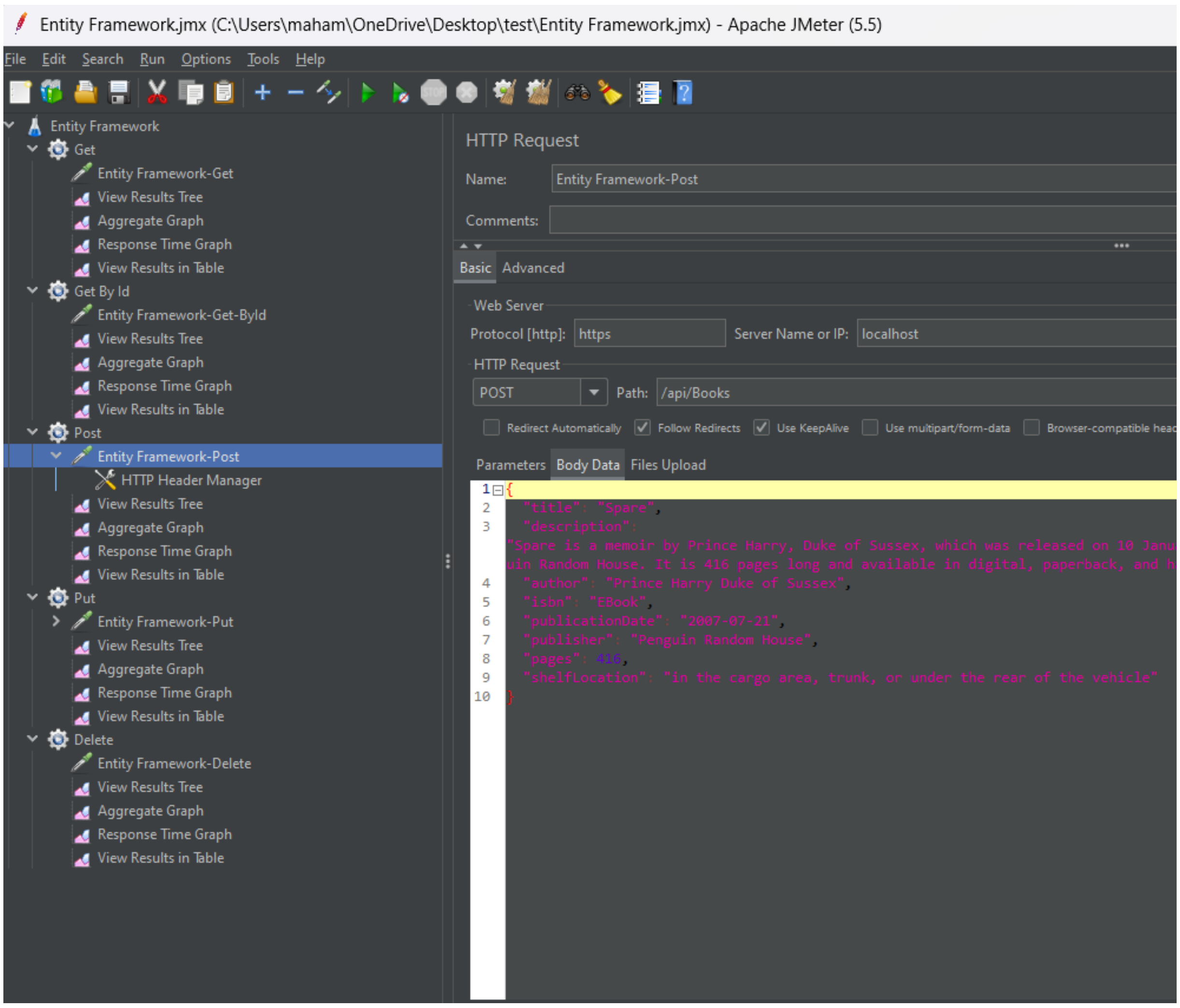
GET Path: /book

☐ Redirect Automatically ✓ Follow Redirects ✓ Use Keep

Parameters Body Data Files Upload

Name:

Figure 48 : JMeter application for testing Dapper Endpoints.



Name: Entity Framework-Post
Comments:
Protocol [http]: https
Server Name or IP: localhost
POST
Path: /api/Books
Redirect Automatically [ ] Follow Redirects [✓] Use KeepAlive [✓] Use multipart/form-data [ ] Browser-compatible head

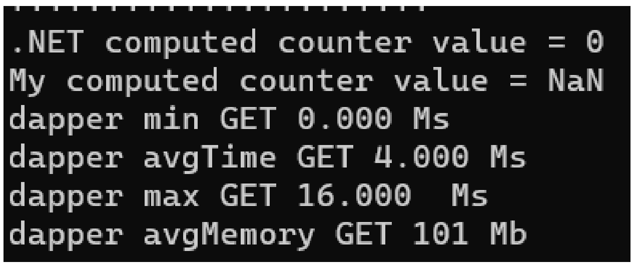
```
{
  "title": "Spare",
  "description":
"Spare is a memoir by Prince Harry, Duke of Sussex, which was released on 10 Janu
uin Random House. It is 416 pages long and available in digital, paperback, and h
  "author": "Prince Harry Duke of Sussex",
  "isbn": "EBook",
  "publicationDate": "2007-07-21",
  "publisher": "Penguin Random House",
  "pages": 416,
  "shelfLocation": "in the cargo area, trunk, or under the rear of the vehicle"
}
```

Figure 49 : JMeter application for testing Entity framework endpoints.

2.5 Benchmarks

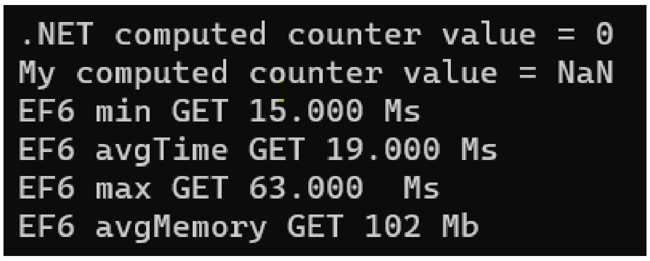
In our application, the main goal is to obtain our results and compare them at the end. It will execute each method over 1000 times and compare them to determine which ORM outperforms the other. The CRUD and get by id methods are utilized during the comparative analysis of EF Core 6 and Dapper. Benchmarks are used in order to evaluate the amount of time and memory that is consumed. Calculating the mean, minimum, and maximum duration of each operation is the first step in the process, which is then followed by a comparison of these values. as shown in Figure 50.

As it is shown from the figures below, in terms of the smallest amount of time, the largest amount of time, and the average amount of time, Dapper surpasses EF core. This illustrates that Dapper has a greater capacity to get data from the database in a more improper way compared to EF core, while requiring similar average memory as EF core. This is the case despite the fact that Dapper utilizes the same amount of memory on average as EF core. as shown in Figure 51.



```
.NET computed counter value = 0
My computed counter value = NaN
dapper min GET 0.000 Ms
dapper avgTime GET 4.000 Ms
dapper max GET 16.000  Ms
dapper avgMemory GET 101 Mb
```

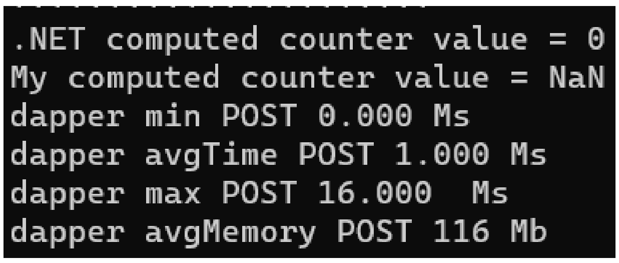
Figure 50 : Benchmark comparing time and memory performance Dapper for GET method.



```
.NET computed counter value = 0
My computed counter value = NaN
EF6 min GET 15.000 Ms
EF6 avgTime GET 19.000 Ms
EF6 max GET 63.000  Ms
EF6 avgMemory GET 102 Mb
```

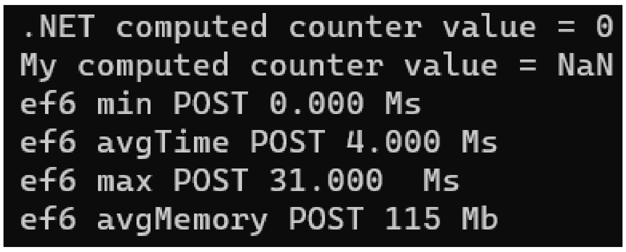
Figure 51 : Benchmark comparing time and memory performance Entity for GET method.

In Figure 52 and Figure 53 both results of post endpoint are shown and unlike the get endpoint, dapper is not faster in every aspect of the results. For minimum time of the post endpoint, dapper and Entity Framework have obtained the same results. In both the average time and the maximum time dapper is quicker than Entity Framework. On the other hand, Entity Framework, uses 1 Mb of average memory less than dapper.



```
.NET computed counter value = 0
My computed counter value = NaN
dapper min POST 0.000 Ms
dapper avgTime POST 1.000 Ms
dapper max POST 16.000  Ms
dapper avgMemory POST 116 Mb
```

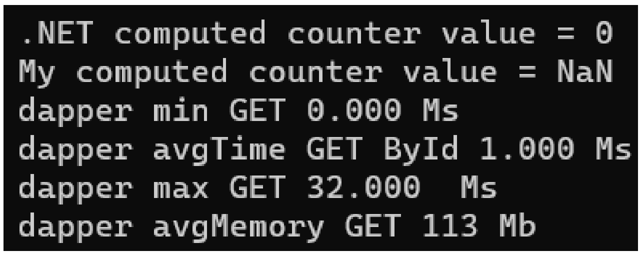
Figure 52 : Benchmark comparing time and memory performance Dapper for POST method.



```
.NET computed counter value = 0
My computed counter value = NaN
ef6 min POST 0.000 Ms
ef6 avgTime POST 4.000 Ms
ef6 max POST 31.000  Ms
ef6 avgMemory POST 115 Mb
```

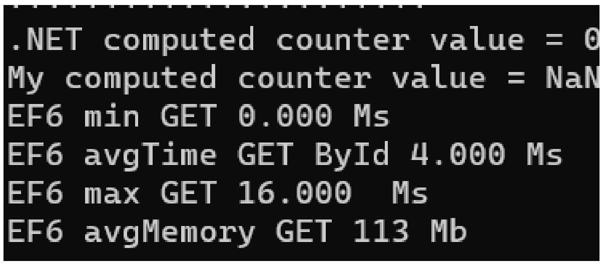
Figure 53 : Benchmark comparing time and memory performance Entity for POST method.

As is clear from the information provided before in the form of these numbers, It is clear to see that the minimum timeframes for any of the two entities are exactly the same. However, when looking at the timings on average, it is clear that the Dapper framework beats EF core by a wide margin. as shown in Figure 54 and Figure 55. And in terms of maximum time, Dapper is twice slower than EF core. This indicates that Dapper can get information by id from the database far less rapidly than EF core, even though it requires around the same amount of memory as EF core.



```
.NET computed counter value = 0
My computed counter value = NaN
dapper min GET 0.000 Ms
dapper avgTime GET ById 1.000 Ms
dapper max GET 32.000  Ms
dapper avgMemory GET 113 Mb
```

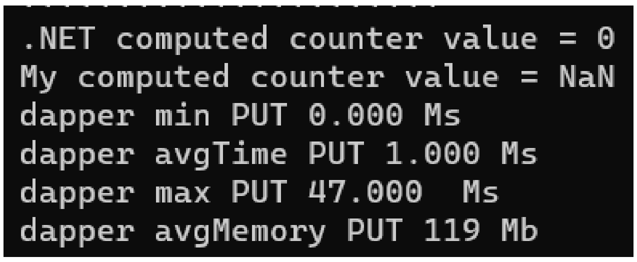
Figure 54 : Benchmark comparing time and memory performance Dapper for GET method.



```
.NET computed counter value = 0
My computed counter value = NaN
EF6 min GET 0.000 Ms
EF6 avgTime GET ById 4.000 Ms
EF6 max GET 16.000  Ms
EF6 avgMemory GET 113 Mb
```

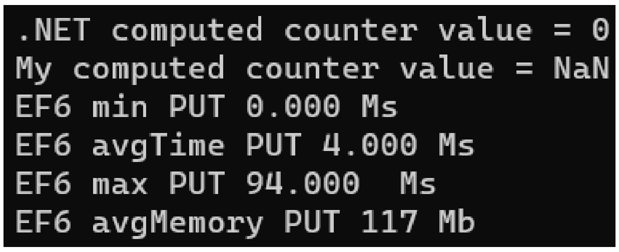
Figure 55 : Benchmark comparing time and memory performance Entity for GET method.

The statistical evidence that was provided suggests that the minimal timeframes required by both frameworks are comparable to one another. On the other hand, when looking at the timings on average, it is clear that the Dapper framework surpasses EF6 by a substantial amount. Dapper statistics is also lower in maximum timing, but this comes at a cost of using slightly more memory than Entity Framework as illustrated in Figure 56 and Figure 57.



```
.NET computed counter value = 0
My computed counter value = NaN
dapper min PUT 0.000 Ms
dapper avgTime PUT 1.000 Ms
dapper max PUT 47.000  Ms
dapper avgMemory PUT 119 Mb
```

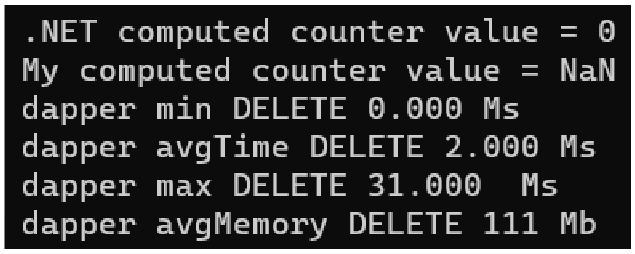
Figure 56 : Benchmark comparing time and memory performance Dapper for PUT method.



```
.NET computed counter value = 0
My computed counter value = NaN
EF6 min PUT 0.000 Ms
EF6 avgTime PUT 4.000 Ms
EF6 max PUT 94.000  Ms
EF6 avgMemory PUT 117 Mb
```

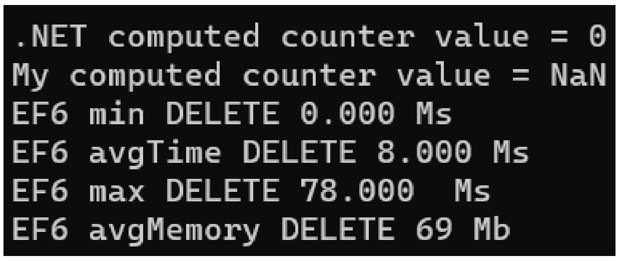
Figure 57 : Benchmark comparing time and memory performance Entity for PUT method.

There is a similar pattern again in the delete endpoint of dapper and Entity Framework in the numbers below in Figure 58 and Figure 59. All the timing are faster or at least similar between the two frameworks but again at a cost of memory usage for dapper. Dapper slightly sacrifices it memory usage to significantly boost its numbers.



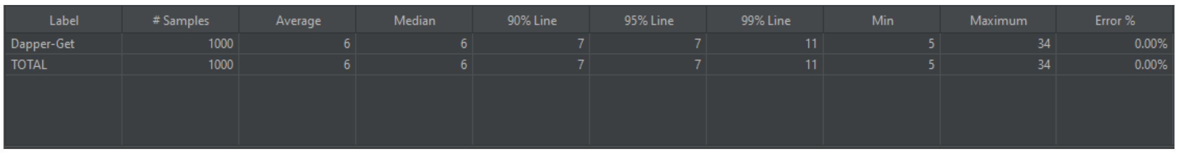
```
.NET computed counter value = 0
My computed counter value = NaN
dapper min DELETE 0.000 Ms
dapper avgTime DELETE 2.000 Ms
dapper max DELETE 31.000  Ms
dapper avgMemory DELETE 111 Mb
```

Figure 58 : Benchmark comparing time and memory performance Dapper for DELETE method.



```
.NET computed counter value = 0
My computed counter value = NaN
EF6 min DELETE 0.000 Ms
EF6 avgTime DELETE 8.000 Ms
EF6 max DELETE 78.000  Ms
EF6 avgMemory DELETE 69 Mb
```

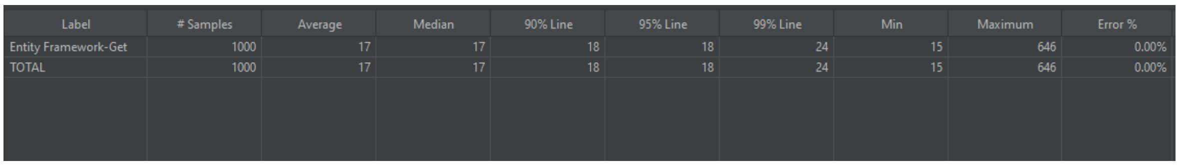
Figure 59 : Benchmark comparing time and memory performance Entity for DELETE method.



Label	# Samples	Average	Median	90% Line	95% Line	99% Line	Min	Maximum	Error %
Dapper-Get	1000	6	6	7	7	11	5	34	0.00%
TOTAL	1000	6	6	7	7	11	5	34	0.00%

In Figure 60 and Figure 61, the results of JMeter that correspond to the Get endpoints of Dapper and Entity Framework may be seen here. There are a lot of possible outcomes, but what we really need to know is the shortest, longest, and average amount of time it took. When compared against Entity Framework, it is very clear that Dapper has once again shown greater performance. The minimum time for dapper is exactly a third of Entity Frameworks minimum time. The average time for dapper is nearly half that of other frameworks, and the utmost time for Entity Framework exceeds that of dapper.

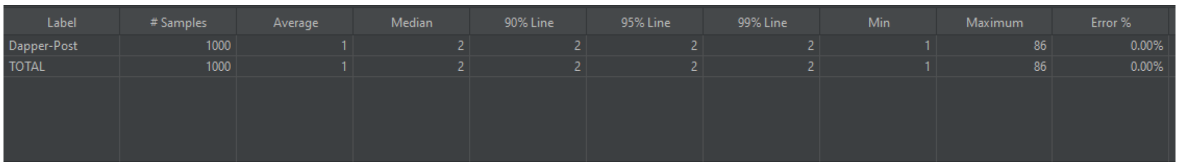
Figure 60 : JMeter Result for GET method endpoint for dapper.



Label	# Samples	Average	Median	90% Line	95% Line	99% Line	Min	Maximum	Error %
Entity Framework-Get	1000	17	17	18	18	24	15	646	0.00%
TOTAL	1000	17	17	18	18	24	15	646	0.00%

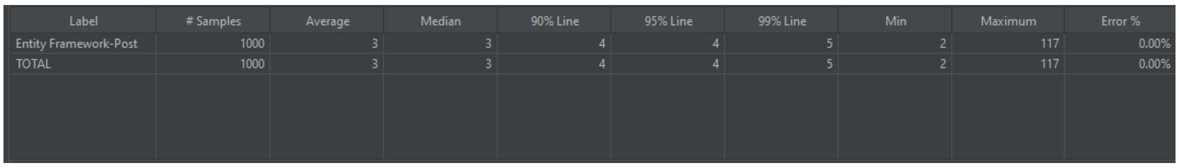
Figure 61 : JMeter Result for GET method endpoint for entity.

Unlike the outstanding numbers in Get endpoint, the results in the Post endpoint for Entity Framework are double what the results in dapper show as can be seen in Figure 62 and Figure 63. For almost all the timings, Entity Framework has double the time in minimum time maximum time and triple the time in average timing.



Label	# Samples	Average	Median	90% Line	95% Line	99% Line	Min	Maximum	Error %
Dapper-Post	1000	1	2	2	2	2	1	86	0.00%
TOTAL	1000	1	2	2	2	2	1	86	0.00%

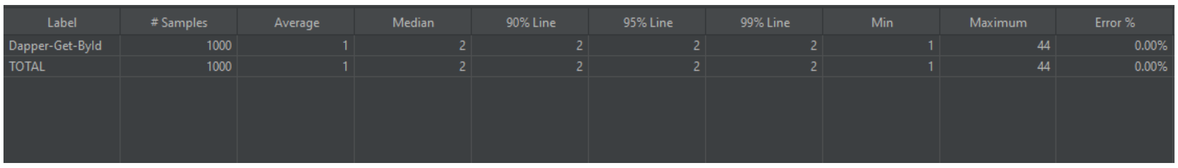
Figure 62 : JMeter Result for POST method endpoint for dapper.



Label	# Samples	Average	Median	90% Line	95% Line	99% Line	Min	Maximum	Error %
Entity Framework-Post	1000	3	3	4	4	5	2	117	0.00%
TOTAL	1000	3	3	4	4	5	2	117	0.00%

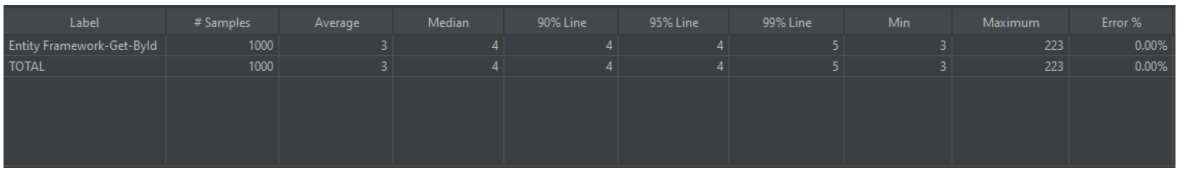
Figure 63: JMeter Result for POST method endpoint for entity.

Get by Id timing for Dapper is a third of Entity Frameworks timing as shown in Figure 64 and Figure 65. The minimum time and average time for dapper is a third of Entity Frameworks and the maximum time is almost a tenth of the others. This shows that in some cases they might be similar or close but in the maximum time, dapper outshines Entity Framework.



Label	# Samples	Average	Median	90% Line	95% Line	99% Line	Min	Maximum	Error %
Dapper-Get-ById	1000	1	2	2	2	2	1	44	0.00%
TOTAL	1000	1	2	2	2	2	1	44	0.00%

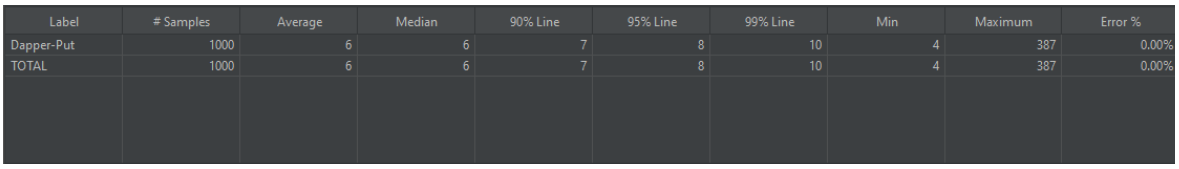
Figure 64 : JMeter Result for GET by Id method endpoint for dapper.



Label	# Samples	Average	Median	90% Line	95% Line	99% Line	Min	Maximum	Error %
Entity Framework-Get-ById	1000	3	4	4	4	5	3	223	0.00%
TOTAL	1000	3	4	4	4	5	3	223	0.00%

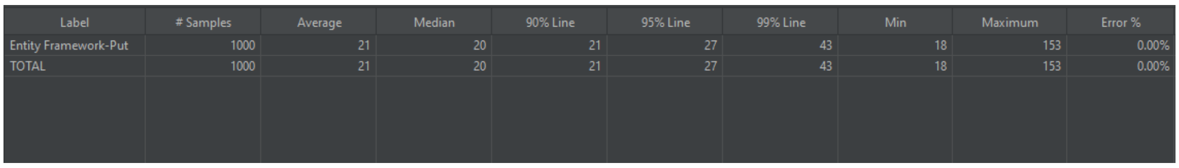
Figure 65 : JMeter Result for GET by Id method endpoint for entity.

Looking at the Figure 66 and Figure 67 it is seen that Dapper is much faster than EF6 when it comes to updating data in bulk. Dapper is approximately 3.5 times faster than EF6 when it comes to average timing and 4.5 times faster in minimum timing. However, Dapper struggles when it comes to maximum timing. EF6 appears to be 2.5 times faster than Dapper.



Label	# Samples	Average	Median	90% Line	95% Line	99% Line	Min	Maximum	Error %
Dapper-Put	1000	6	6	7	8	10	4	387	0.00%
TOTAL	1000	6	6	7	8	10	4	387	0.00%

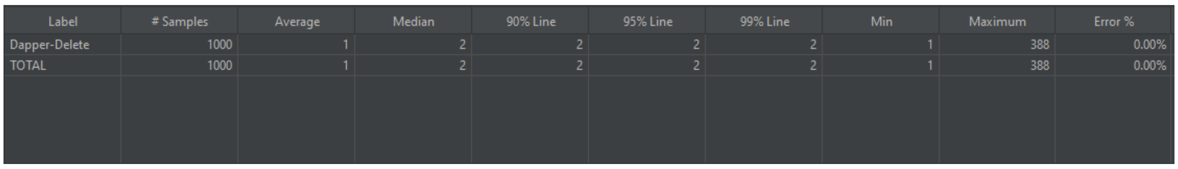
Figure 66 : JMeter Result for PUT method endpoint for Dapper.



Label	# Samples	Average	Median	90% Line	95% Line	99% Line	Min	Maximum	Error %
Entity Framework-Put	1000	21	20	21	27	43	18	153	0.00%
TOTAL	1000	21	20	21	27	43	18	153	0.00%

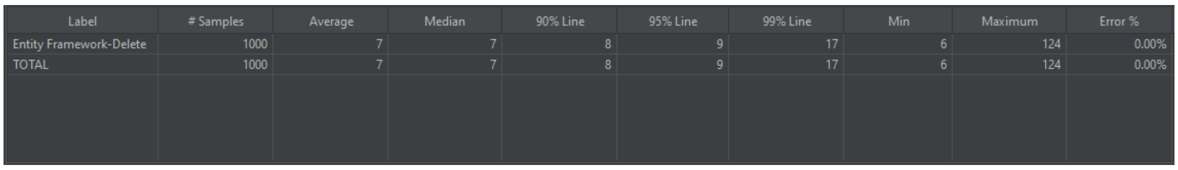
Figure 67 : JMeter Result for PUT method endpoint for entity.

While in Figure 68 dapper was extremely fast in the maximum times, in the delete endpoint it not the same as other endpoints. It obvious that dapper is uses less than half of the time of Entity Frameworks in maximum timing as can be seen in Figure 69, but it is six and seven times faster in minimum time and average time respectively.



Label	# Samples	Average	Median	90% Line	95% Line	99% Line	Min	Maximum	Error %
Dapper-Delete	1000	1	2	2	2	2	1	388	0.00%
TOTAL	1000	1	2	2	2	2	1	388	0.00%

Figure 68 : JMeter Result for DELETE method endpoint for dapper.



Label	# Samples	Average	Median	90% Line	95% Line	99% Line	Min	Maximum	Error %
Entity Framework-Delete	1000	7	7	8	9	17	6	124	0.00%
TOTAL	1000	7	7	8	9	17	6	124	0.00%

Figure 69 : JMeter Result for DELETE method endpoint for entity.

2.6 Result

In the figure below there are three different implementations which two of them have already been explained. One being the implementation done by the application and the other is the JMeter application. The third implementation is done by a programmer who has already compared both dapper and Entity Framework by using the .Net benchmark, which is a separate open-source benchmark [].

In Figure 70, are the results of minimum time required for dapper framework to execute an endpoint by all three of the implementations. The results show that in most of the endpoint executions, the programs implementation are better than the JMeter's timing and the DotNet[30]timing. With one exception of get by Id, which both the .Net and the programs timing are similar. On the other hand, JMeters implementation seems to be surpassing the .Net implementation in Post, Put, and Delete.

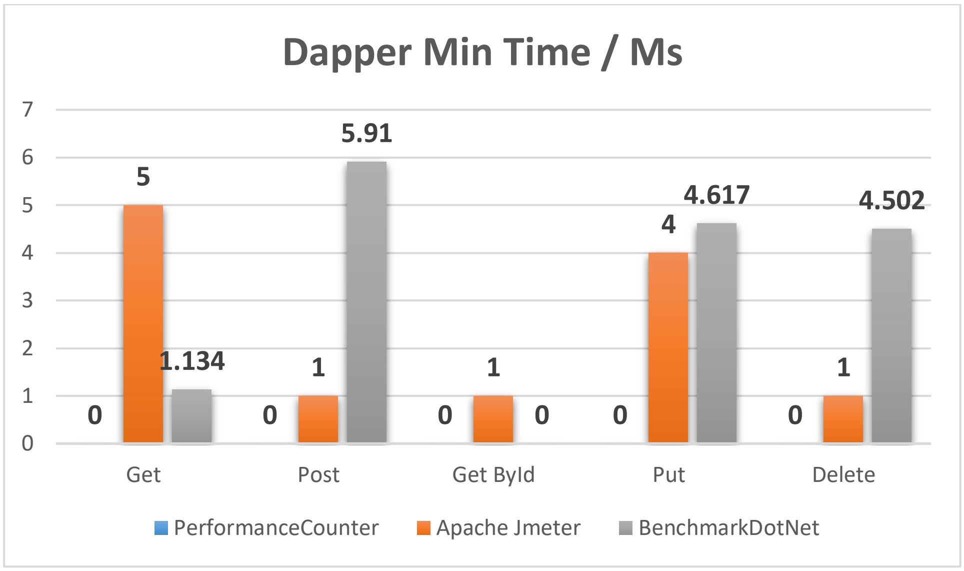


Figure 70 : Minimum time required for dapper framework to execute an endpoint by all three of the implementations.

In Figure 71, show results are for Entity Framework minimum time to execute an endpoint. The implementation of the program again is way less than the other two implementations except for get operation which is like JMeter but way more than DotNet[30] implementation. The JMeter takes more time in Put, get by Id, and get operations while the other impetrations are better on time.

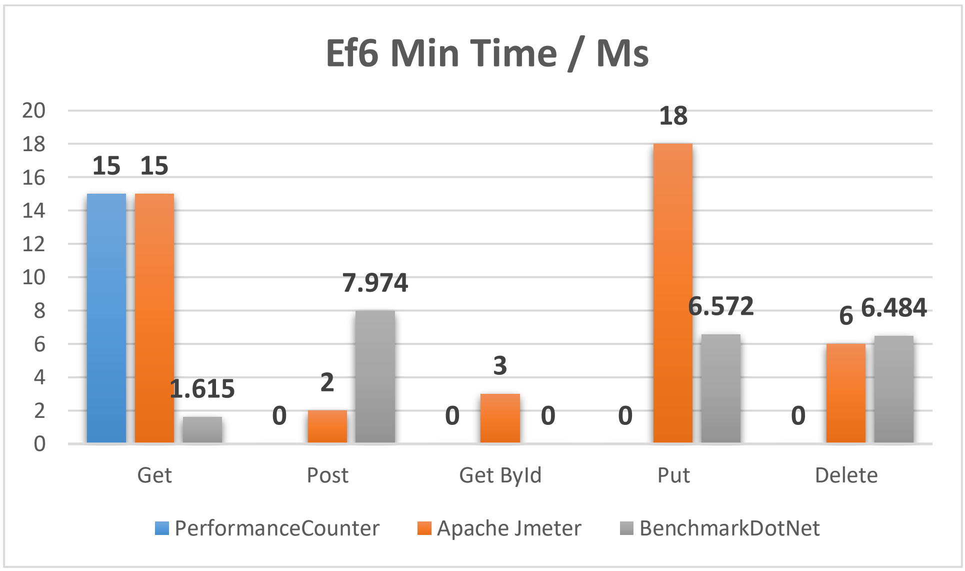


Figure 71 : Minimum time required for entity framework to execute an endpoint by all three of the implementations.

Dapper maximum time for the endpoints results shown in Figure 72 for all the three implementations. The most outstanding number from the rest is the delete and put timing by the JMeter implementation. The DotNet[30]implementation performs better than the programs implementation in put operation but vice versa in the delete operation. In get by id, .Net is close to zero while JMeter takes the most time almost reaching 150. On the contrary, .Net takes more time than both other implementations in both Post and Get endpoints with the program's implementation taking the least time in both.

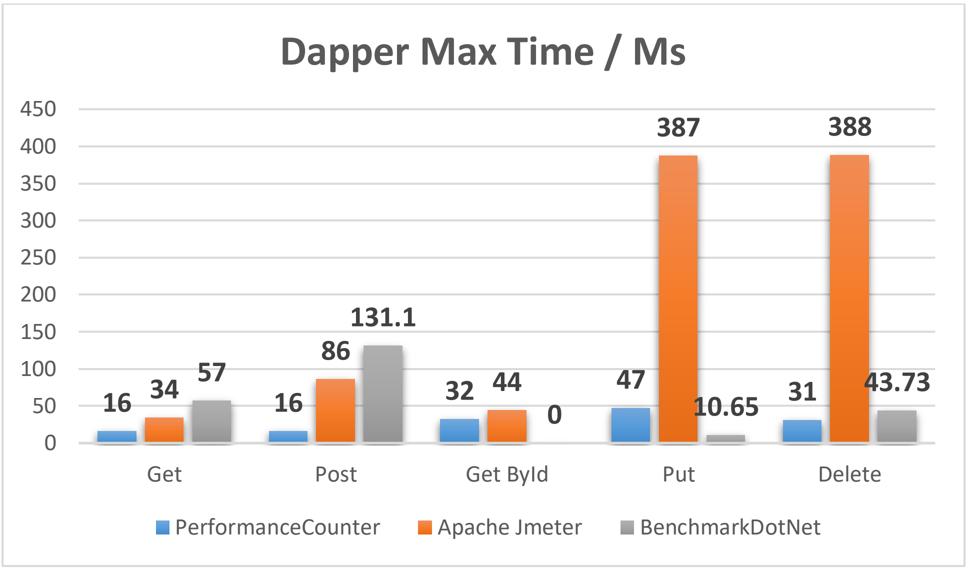


Figure 72 : Max average time for dapper framework.

Entity Framework maximum timing for all three implementations results shown below. JMeter again outstands in one operation than the rest, which is get operation and takes the most time, while both other operations have almost the same timing and this seems to be the case also in get by Id operation as illustrated in Figure 73. In the post operation, the DotNet[30]uses the most time then JMeter and the implementation of the program respectively. In both put and delete operation, the program and the JMeter are very close but the .Net performs the best with almost close to zero in timing.

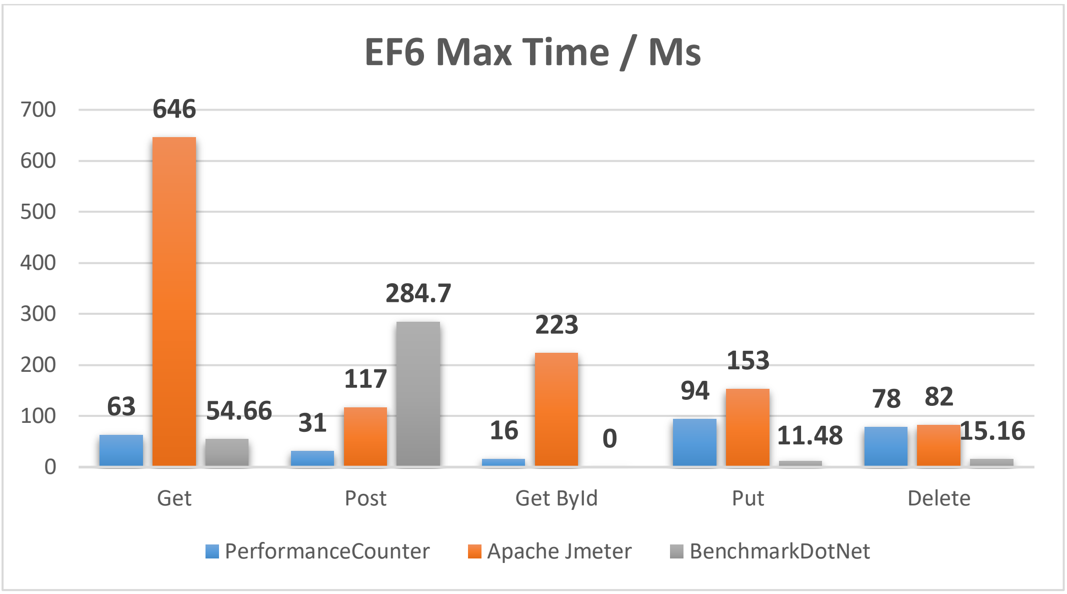


Figure 73 : Max average time for entity framework.

Dapper average timing for all three implementations as shown in Figure 74. In post and get by Id operations the programs implementations and the JMeter are the exact same while DotNet[30]uses the most timing in Put and least timing in get by id. In get and put operations, JMeter uses the same amount of time in both while .Net and the implementation in the program do the vice versa of each other.

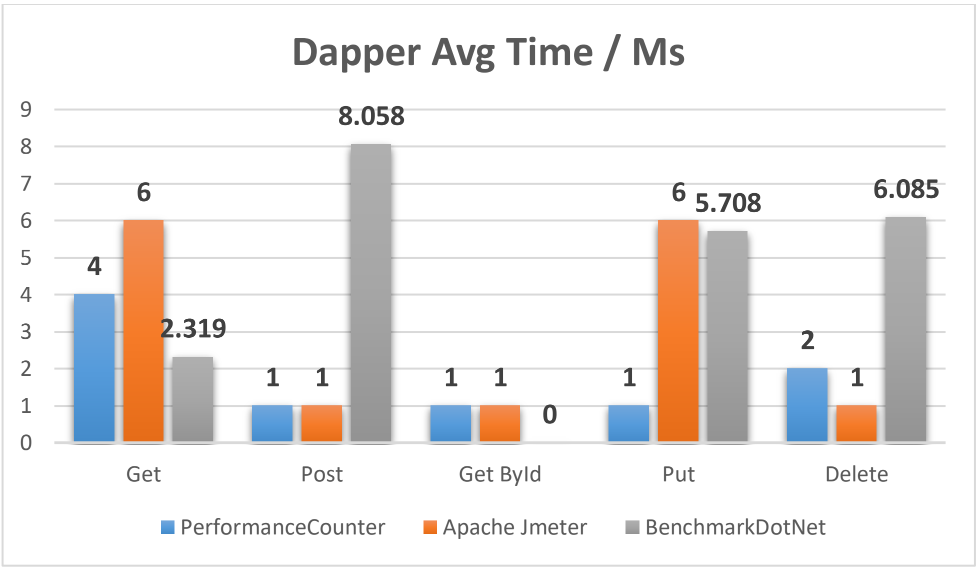


Figure 74 : Benchmark for average time for Dapper framework.

As shown in Figure 75, DotNet [30] has the least or equal amount of timing in almost all the operations except for Post endpoint. The two other implementations are the same in all the other endpoints with the outstanding average time in put operation that gives JMeter a bad sign, while the rest are all the same.

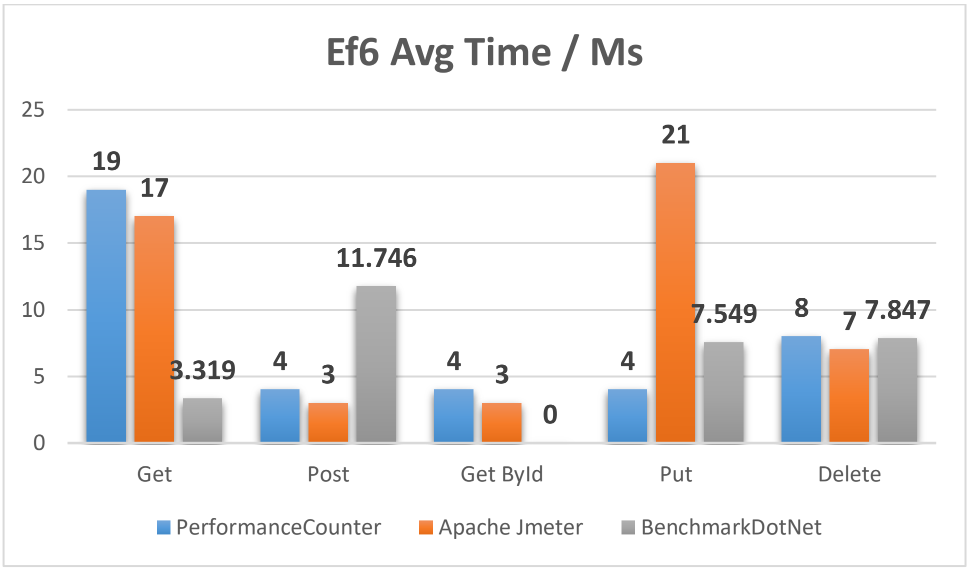


Figure 75 : Benchmark for average time for entity framework.

Average memory for dapper and Entity Framework are equal in memory usage but only significantly different in delete operations as shown in Figure 76. The other operations and endpoints are mostly the same and have no significant change.

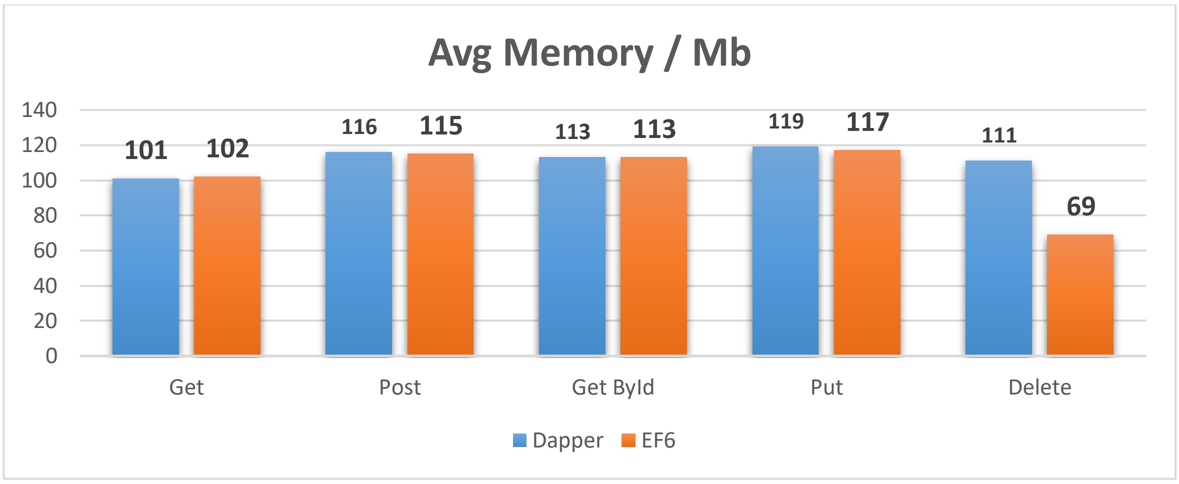


Figure 76 : Average memory for dapper and entity,

CONCLUSION

In conclusion, developers looking for the best ORM framework for their projects will find the performance comparison between Entity Framework Core 6 and Dapper to be an instructive reference. When sheer efficiency and simplicity are of crucial importance, Dapper emerges as the leader, whereas EF Core 6 delivers a rich variety of functionality and prioritizes developer productivity.

This study's benchmarks showed that Dapper regularly beat EF Core 6, demonstrating the benefits of the latter's lightweight design and direct mapping of queries to objects in terms of both query execution speed and resource use.

Performance is an important consideration when choosing an ORM framework, but it shouldn't be the only one. The developer's knowledge of the framework, the project's unique needs, and the project's long-term maintainability are all crucial factors. EF Core 6 is a great option for large-scale systems that value quick development and long-term maintainability because of its rich feature set, easy interaction with Microsoft technologies, and strong community support.

Dapper, on the other hand, excels when speed is more important, such as in applications that process large amounts of data or have complicated data manipulation needs. Developers may pick the most appropriate ORM framework by carefully considering the performance characteristics against other key variables. Developers may optimize the performance, efficiency, and overall success of their software applications in accordance with their unique project objectives and needs, whether they use the feature-rich environment of EF Core 6 or use the raw performance capabilities of Dapper.

BIBLIOGRAPHY

[1] CONTRIBUTOR, Staff. Why Do We Need Object-Relational Mapping? Software Reviews, Opinions, and Tips - DNSstuff [online]. 12 September 2022. Available from: https://www.dnsstuff.com/why-do-we-need-object-relational-mapping

[2] .NET Basics: ORM (Object Relational Mapping). Telerik Blogs [online]. 27 September 2022. Available from: https://www.telerik.com/blogs/dotnet-basics-orm-object-relational-mapping

[3] Understanding Object-Relational Mapping: Pros, Cons, and Types. AltexSoft [online]. 11 March 2021. Available from: https://www.altexsoft.com/blog/object-relational-mapping/

[4] .NET Basics: ORM (Object Relational Mapping). Telerik Blogs [online]. 27 September 2022. Available from: https://www.telerik.com/blogs/dotnet-basics-orm-object-relational-mapping

[5] What is an ORM – The Meaning of Object Relational Mapping Database Tools. freeCodeCamp.org [online]. 21 October 2022. Available from: https://www.freecodecamp.org/news/what-is-an-orm-the-meaning-of-object-relational-mapping-database-tools/

[6] ajcvickers. Overview of Entity Framework Core - EF Core. Overview of Entity Framework Core - EF Core | Microsoft Learn [online]. 25 May 2021. Available from: https://learn.microsoft.com/en-us/ef/core/

[7] Home - NHibernate. Home - NHibernate [online]. Available from: https://nhibernate.info/

[8] DapperLib. GitHub - DapperLib/Dapper: Dapper - a simple object mapper for .Net. GitHub [online]. 21 August 2022. Available from: https://github.com/DapperLib/Dapper

[9] Wikiwand - Base One Foundation Component Library. Wikiwand [online]. 11 November 2020. Available from: https://wikiwand.com/en/Base_One_Foundation_Component_Library

[10] iBATIS Home. iBATIS Home [online]. Available from: https://ibatis.apache.org/

[11] mcleblanc. LINQ to SQL - ADO.NET. LINQ to SQL - ADO.NET | Microsoft Learn [online]. 15 September 2021. Available from: https://learn.microsoft.com/en-us/dotnet/framework/data/adonet/sql/linq/

[12] What does nhydrate mean. What does nhydrate mean - Definition of nhydrate - Word finder [online]. Available from: https://findwords.info/term/nhydrate

[13] What is an ORM (Object Relational Mapper)? Prisma's Data Guide [online]. Available from: https://www.prisma.io/dataguide/types/relational/what-is-an-orm

[14] ORM Framework uses advantages and disadvantages of the post-frame. ORM Framework uses advantages and disadvantages of the post-frame [online]. Available from: https://topic.alibabacloud.com/a/orm-framework-uses-advantages-and-disadvantages-of-the-post-frame_8_8_20284586.html

[15] The advantages and disadvantages of using ORM. Stack Overflow [online]. 12 January 2011. Available from: https://stackoverflow.com/questions/4667906/the-advantages-and-disadvantages-of-using-orm

[16] KATHIRESAPILLAI, Bavanthini. Introduction to Dapper — A micro ORM , a simple Object Mapper for .NET. Medium [online]. 6 September 2020. Available from: https://medium.com/@k.bavanthini/introduction-to-dapper-a-micro-orm-a-simple-object-mapper-for-net-318201dc7030

[17] PROJECTS, ZZZ. Welcome To Learn Dapper ORM - A Dapper Tutorial for C# and .NET Core. Welcome To Learn Dapper ORM - A Dapper Tutorial for C# and .NET Core [online]. Available from: https://www.learndapper.com/

[18] Introduction to Dapper. Introduction to Dapper - ParTech [online]. Available from: https://www.partech.nl/nl/publicaties/2021/02/introduction-to-dapper

[19] BONELLO, Simon. Entity Framework VS Dapper. Side by side comparison - Chubby Developer. Chubby Developer [online]. 20 March 2022. Available from: https://www.chubbydeveloper.com/entity-framework-vs-dapper/

[20] Using Dapper Micro ORM in ASP.NET Core - Mind IT Systems. Mind IT Systems [online]. 16 March 2023. Available from: https://minditsystems.com/using-dapper-micro-orm-in-asp-net-core/

[21] Building a CRUD API With Dapper. Telerik Blogs [online]. 27 April 2023. Available from: https://www.telerik.com/blogs/building-crud-api-dapper

[22] Dapper 2.0.123. NuGet Gallery | Dapper 2.0.123 [online]. Available from: https://nuget.org/packages/Dapper/

[23] Audacia. Investigating the performance benefits of EF Core 6.0 compiled models feature. Medium [online]. 20 May 2022. Available from: https://medium.com/@audaciasoftware/investigating-the-performance-benefits-of-ef-core-6-0-compiled-models-feature-6f5acd750037

[24] Introduction to Entity Framework. Introduction to Entity Framework - ParTech [online]. Available from: https://www.partech.nl/nl/publicaties/2020/11/introduction-to-entity-framework

[25] Alam, Rashedul. “Advantages and Disadvantages of Entity Framework.” Cybarlab, 28 Sept. 2020, cybarlab.com/advantages-and-disadvantages-of-ef. Accessed 22 May 2023.

[26] mcleblanc. Entity Framework Overview - ADO.NET. Entity Framework Overview - ADO.NET | Microsoft Learn [online]. 15 September 2021. Available from: https://learn.microsoft.com/en-us/dotnet/framework/data/adonet/ef/overview

[27] KANJILAL, Joydip. How to use EF Core as an in-memory database in ASP.NET Core 6. InfoWorld [online]. Available from: https://www.infoworld.com/article/3672154/how-to-use-ef-core-as-an-in-memory-database-in-asp-net-core-6.html

[28] Microsoft.EntityFrameworkCore 7.0.5. NuGet Gallery | Microsoft.EntityFrameworkCore 7.0.5 [online]. Available from: https://nuget.org/packages/Microsoft.EntityFrameworkCore/

[29] What is Object Relational Mapping? Educative: Interactive Courses for Software Developers [online]. Available from: https://www.educative.io/answers/what-is-object-relational-mapping

[30] CANTEKIN, Salih. The Big Fight—Dapper vs Entity Framework Detailed Benchmark. Medium [online]. 23 February 2022. Available from: https://salihcantekin.medium.com/the-big-fight-dapper-vs-entity-framework-detailed-benchmark-2345af933382

LIST OF ABBREVIATIONS

API	Application Programming Interface
CRUD	CREATE, READ, UPDATE AND DELETE
EF	Entity Framework
HTTP	Hypertext Transfer Protocol
IDE	Integrated Development Environment
JSON	JavaScript Object Notation
LINQ	Language Integrated Query
OOP	Object Oriented Programming
ORM	Object Relational Mapping
OS	Operating System
RDB	Relational Database
RDBMS	Relational Database Management System
RDMS	Registered Diagnostic Medical Sonographer
SQL	Structured Query Language
URL	Uniform Resource Locator

LIST OF FIGURES

LIST OF TABLES

APPENDICES

APPENDIX P I: APPENDIX TITLE