

Building a secure ASP.NET web API for MongoDB with identity and JWT authentication

1. Introduction	1
2. Run MongoDb ReplicaSet in a container	3
3. Develop a web API with ASP.NET Core and MongoDB.....	14
4. We provide Authentication and Authorization to the web API.	24
5. Testing the API	34
6. Conclusion.....	40
7. Future work.....	40
8. References	41

1. Introduction

In this article we are going to implement a web API ASP.Net Core that runs Create, Read, Update, and Delete (CRUD) operations on a MongoDB NoSQL database. Also, we provide Authentication and Authorization capabilities. We protect the Web API by implementing JWT authentication. We use authorization in ASP.NET Core to supply access to various functionalities of the application. We store user credentials in a MongoDb database.

We start our implementation developing a CRUD API with MongoDB ReplicaSet as database. In this figure we show in swagger the CRUD functions (get, post, get by id, put and delete).

The screenshot shows the BookStoreApi v1 Swagger interface. At the top, there's a navigation bar with the Swagger logo, a dropdown for 'Select a definition' set to 'BookStoreApi v1', and an 'Authorize' button with a lock icon.

The main content area has a title 'BookStoreApi 1.0 OAS3' and a URL 'https://localhost:7037/swagger/v1/swagger.json'. Below this, there's a section titled 'Authenticate' with a collapse arrow.

The 'Books' section is expanded, showing the following methods:

- GET /api/Books (blue button)
- POST /api/Books (green button)
- GET /api/Books/{id} (blue button)
- PUT /api/Books/{id} (orange button)
- DELETE /api/Books/{id} (red button)

Each method row has a collapse arrow and a lock icon.

After, we integrate the authorization and authentication features. We also store the user data in the MongoDB ReplicaSet in the “Identity” database. The API will include the following methods:

The screenshot shows the BookStoreApi v1 Swagger interface. At the top, there's a navigation bar with the Swagger logo, a dropdown for 'Select a definition' set to 'BookStoreApi v1', and an 'Authorize' button with a lock icon.

The main content area has a title 'BookStoreApi 1.0 OAS3' and a URL 'https://localhost:7037/swagger/v1/swagger.json'. Below this, there's a section titled 'Authenticate' with a collapse arrow.

The 'Authenticate' section is expanded, showing the following POST methods:

- POST /api/Authenticate/RegisterUser (green button)
- POST /api/Authenticate/RegisterAdmin (green button)
- POST /api/Authenticate/Login (green button)
- POST /api/Authenticate/RefreshToken (green button)
- POST /api/Authenticate/Revoke/{user_name} (green button)
- POST /api/Authenticate/RevokeAll (green button)

Each method row has a collapse arrow and a lock icon.

Below the Authenticate section, there's a collapsed section titled 'Books'.

2. Run MongoDb ReplicaSet in a container

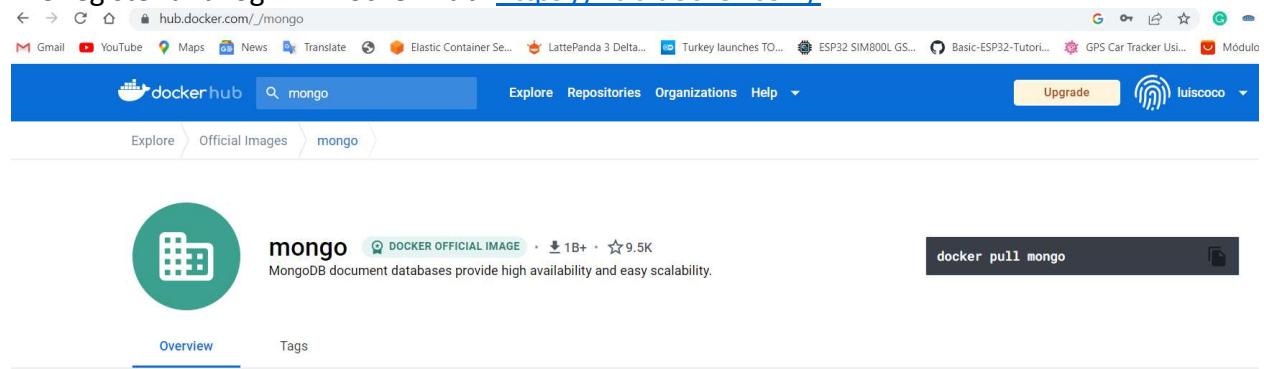
Before starting to develop the web API we have to run the MongoDb ReplicaSet in a Docker container. We have to follow these steps:

We install Docker Desktop: <https://docs.docker.com/desktop/install/windows-install/>



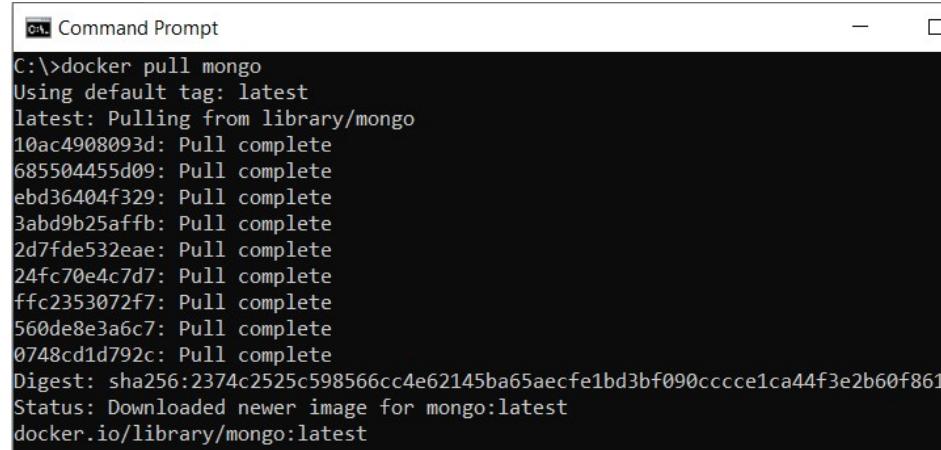
The screenshot shows the Docker Docs website with the URL <https://docs.docker.com/desktop/install/windows-install/>. The main content is titled "Install Docker Desktop on Windows". It includes a sidebar with links for "Install on Windows", "Understand permission requirements for Mac", "Understand permission requirements for Windows", "Install on Linux", and "Installation per Linux distro". A large blue button labeled "Docker Desktop for Windows" is prominently displayed.

We register and login in Docker Hub: <https://hub.docker.com/>



The screenshot shows the Docker Hub search results for "mongo". The top navigation bar includes links for "Explore", "Repositories", "Organizations", "Help", and "Upgrade". The search bar shows "mongo". Below the search bar, there is a card for the "mongo" image, which is labeled as a "DOCKER OFFICIAL IMAGE" with 1B+ pulls and 9.5K stars. The card describes MongoDB document databases as providing high availability and easy scalability. At the bottom right of the card is a "docker pull mongo" button.

We run Docker Desktop application, and after that, we pull the MongoDb official image from Docker Hub to our local host. Open a command window and run the command “docker pull mongo”.



The screenshot shows a Command Prompt window with the following text output:

```
C:\>docker pull mongo
Using default tag: latest
latest: Pulling from library/mongo
10ac4908093d: Pull complete
685504455d09: Pull complete
ebd36404f329: Pull complete
3abd9b25affb: Pull complete
2d7fde532eae: Pull complete
24fc70e4c7d7: Pull complete
ffc2353072f7: Pull complete
560de8e3a6c7: Pull complete
0748cd1d792c: Pull complete
Digest: sha256:2374c2525c598566cc4e62145ba65aecfe1bd3bf090cccce1ca44f3e2b60f861
Status: Downloaded newer image for mongo:latest
docker.io/library/mongo:latest
```

See the mongo image in your local host. Run the command “docker images”

```
Command Prompt  
C:\>docker images  
REPOSITORY TAG IMAGE ID CREATED SIZE  
mongo latest a440572ac3c1 2 weeks ago 639MB
```

Also, we see the mongo image in you Docker Desktop.

Docker Desktop Upgrade plan Search Ctrl+K Sign in

Images Give feedback

An image is a read-only template with instructions for creating a Docker container. [Learn more](#)

Local Hub

0 Bytes / 639.31 MB in use 1 images Last refresh: 4 minutes ago

Search

Name	Tag	Status	Created	Size	Actions
mongo a440572ac3c1	latest	Unused	18 days ago	639.31 MB	...

We create a new network. Run the command “docker network create mongoCluster”. Run MongoDB in three containers.

```
docker run -d --rm -p 27017:27017 --name mongo1 --network mongoCluster mongo:latest  
mongod --replSet myReplicaSet --bind_ip localhost,mongo1
```

```
docker run -d --rm -p 27018:27017 --name mongo2 --network mongoCluster mongo:latest  
mongod --replSet myReplicaSet --bind_ip localhost,mongo2
```

```
docker run -d --rm -p 27019:27017 --name mongo3 --network mongoCluster mongo:latest  
mongod --replSet myReplicaSet --bind_ip localhost,mongo3
```

```
C:\>docker run -d --rm -p 27017:27017 --name mongo1 --network mongoCluster mongo:latest mongod --replSet myReplicaSet --bind_ip localhost,mongo1
07c89fc19770cfb960d3576ac00f0afe7c6549193b1603f66c0a2ab564d03dea

C:\>docker run -d --rm -p 27018:27017 --name mongo2 --network mongoCluster mongo:latest mongod --replSet myReplicaSet --bind_ip localhost,mongo2
dd2337cbc992a8a3d4bcc9233d7cdb240d375d06a9036ae88e980d70f12200f4

C:\>docker run -d --rm -p 27019:27017 --name mongo3 --network mongoCluster mongo:latest mongod --replSet myReplicaSet --bind_ip localhost,mongo3
30aa9a4093bbe6d9f9e0f3136853ee20a4f70814b56b26fc138ae768f0566c00
```

We can see the three containers running in your Docker Desktop.

The screenshot shows the Docker Desktop interface. On the left, there's a sidebar with 'Containers', 'Images', 'Volumes', and 'Dev Environments' (BETA). Below that is an 'Extensions' section with 'Add Extensions'. The main area is titled 'Containers' and contains a table with columns: Name, Image, Status, Port(s), Started, and Actions. Three containers are listed:

Name	Image	Status	Port(s)	Started	Actions
mongo1	mongo:latest	Running	27017:27017	1 minute ago	⋮
mongo2	mongo:latest	Running	27018:27017	1 minute ago	⋮
mongo3	mongo:latest	Running	27019:27017	1 minute ago	⋮

We initialize new MongoDB ReplicaSet.

```
docker exec -it mongo1 mongosh --eval "rs.initiate({ _id: 'myReplicaSet', members: [{_id: 0, host: 'mongo1', priority: 3}, {_id: 1, host: 'mongo2', priority: 2}, {_id: 2, host: 'mongo3', priority: 1}]} )"
```

```
C:\>docker exec -it mongo1 mongosh --eval "rs.initiate({ _id: 'myReplicaSet', members: [{_id: 0, host: 'mongo1', priority: 3}, {_id: 1, host: 'mongo2', priority: 2}, {_id: 2, host: 'mongo3', priority: 1}]} )"
Current Mongos Log ID: 63f377da516de05057f81059
Connecting to: mongodb://127.0.0.1:27017/?directConnection=true&serverSelectionTimeoutMS=2000&appName=mongosh+1.6.2
Using MongoDB: 6.0.4
Using Mongosh: 1.6.2

For mongosh info see: https://docs.mongodb.com/mongodb-shell/

To help improve our products, anonymous usage data is collected and sent to MongoDB periodically (https://www.mongodb.com/legal/privacy-policy).
You can opt-out by running the disableTelemetry() command.

-----
The server generated these startup warnings when booting
2023-02-20T13:33:51.981+00:00: Using the XFS filesystem is strongly recommended with the WiredTiger storage engine. See http://dochub.mongodb.org/core/prodnotes-filesystem
2023-02-20T13:33:53.386+00:00: Access control is not enabled for the database. Read and write access to data and configuration is unrestricted
2023-02-20T13:33:53.386+00:00: /sys/kernel/mm/transparent_hugepage/enabled is 'always'. We suggest setting it to 'never'
2023-02-20T13:33:53.386+00:00: vm.max_map_count is too low
-----

-----
Enable MongoDB's free cloud-based monitoring service, which will then receive and display
metrics about your deployment (disk utilization, CPU, operation statistics, etc).

The monitoring data will be available on a MongoDB website with a unique URL accessible to you
and anyone you share the URL with. MongoDB may use this information to make product
improvements and to suggest MongoDB products and deployment options to you.

To enable free monitoring, run the following command: db.enableFreeMonitoring()
To permanently disable this reminder, run the following command: db.disableFreeMonitoring()

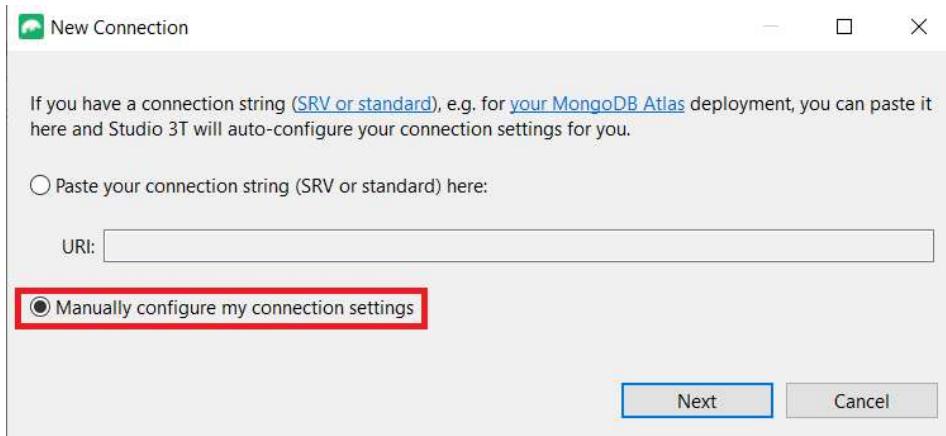
{ ok: 1 }
```

We install Studio 3T for MongoDB.

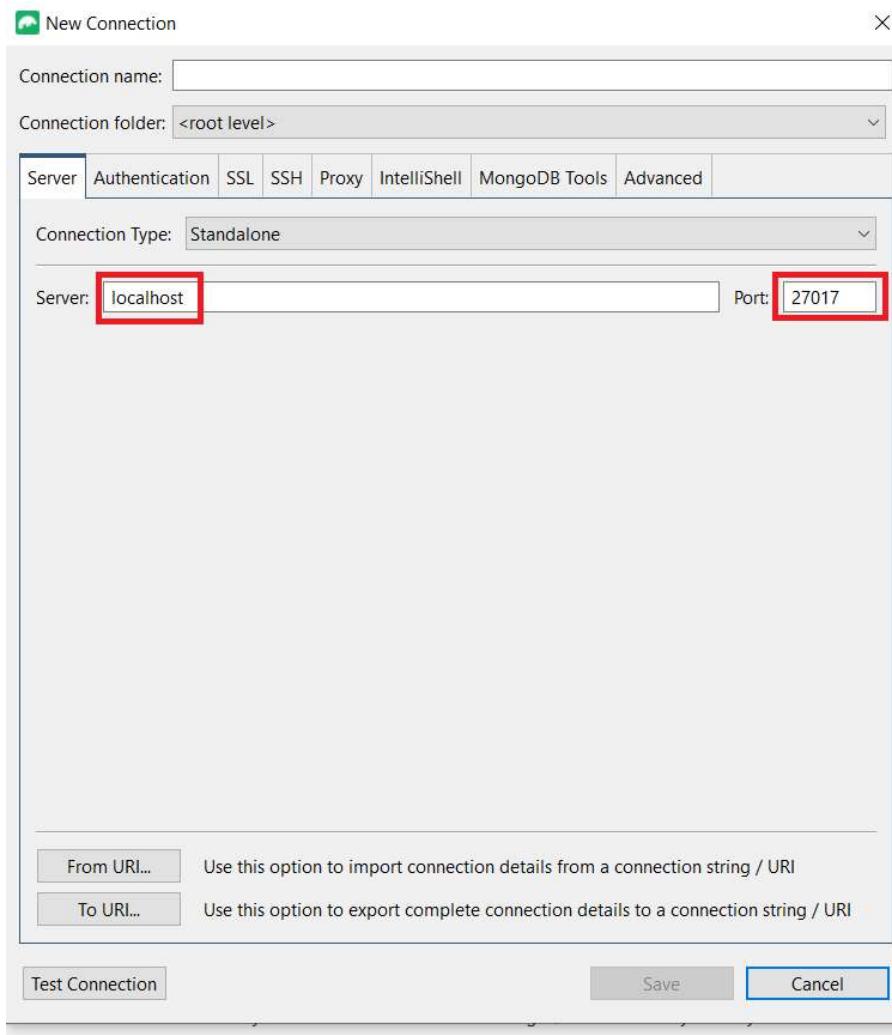
The screenshot shows the official website for Studio 3T. At the top, there's a navigation bar with links like 'TOOLS', 'SOLUTIONS', 'RESOURCES', 'CONTACT US', 'STORE', and a prominent green 'DOWNLOAD' button. Below the header, a large green title reads 'Studio 3T, the professional GUI for MongoDB.' A subtext below it says 'All your IDE, client and GUI tools for MongoDB – on Atlas, or anywhere.' A green 'Try Studio 3T for free' button is centered. Below the button is a screenshot of the Studio 3T application interface, which includes a toolbar with various icons, a left sidebar for connection management, and a main panel showing a MongoDB query shell with some code and results.

We connect to the MongoDB ReplicaSet from Studio 3T. Run Studio 3T and click on the “New Connection” button.

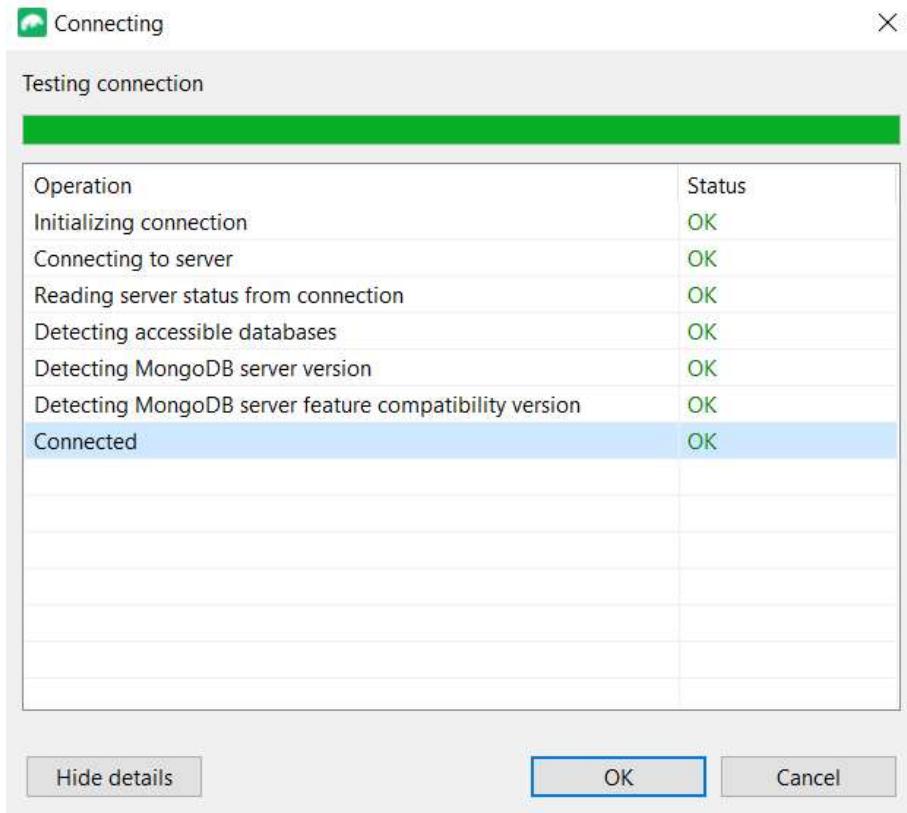
This screenshot shows the Studio 3T application window. On the left, there's a toolbar with a 'Connect' button, which is highlighted with a red box. Below the toolbar is a search bar and a 'Quickstart' tab. The main area is titled 'Connection Manager' and contains a sub-section for 'New Connection'. It features a 'New Connection' button, also highlighted with a red box, and other buttons for 'New Folder', 'Edit', 'Delete', 'Duplicate', 'Import', 'Export', and 'To URL'. At the bottom of the manager, there's a table with columns for 'Name', 'DB Server', 'Security', 'Last Accessed', and 'Shortcut'. A checkbox for 'Show on startup' is at the bottom left, and 'Connect' and 'Close' buttons are at the bottom right.



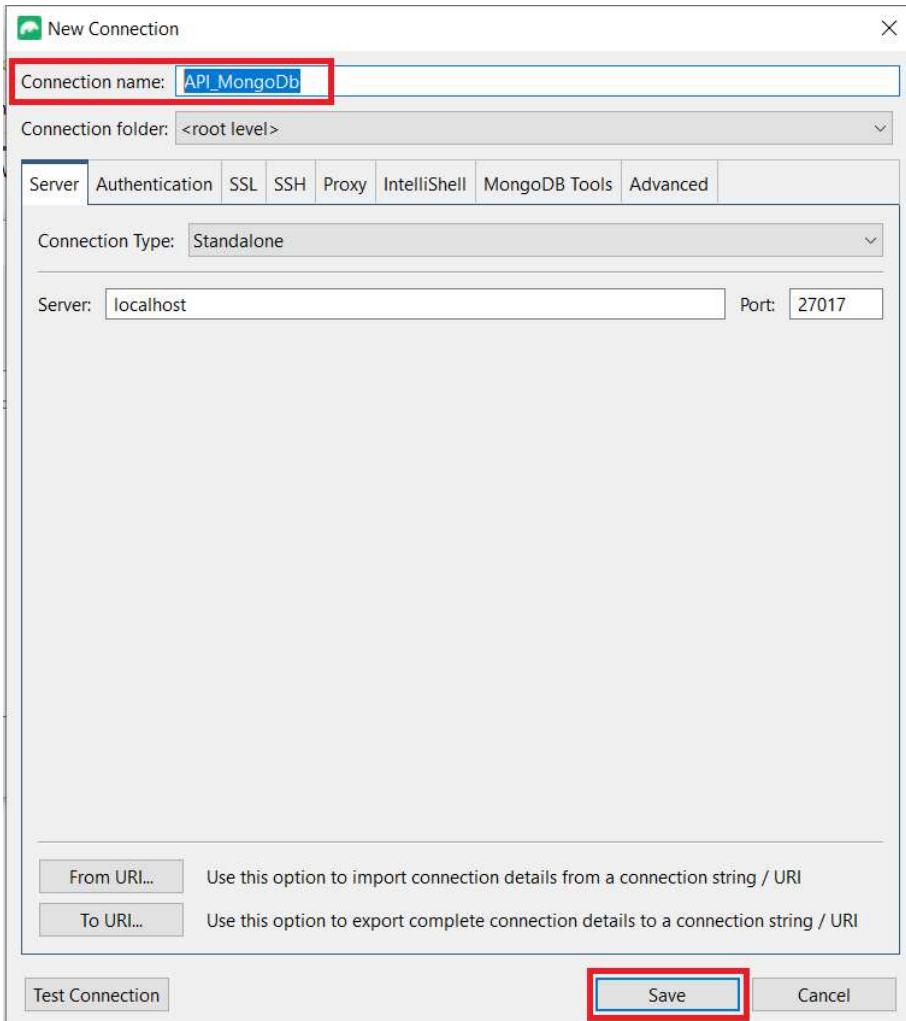
We set the hostname and the port number.



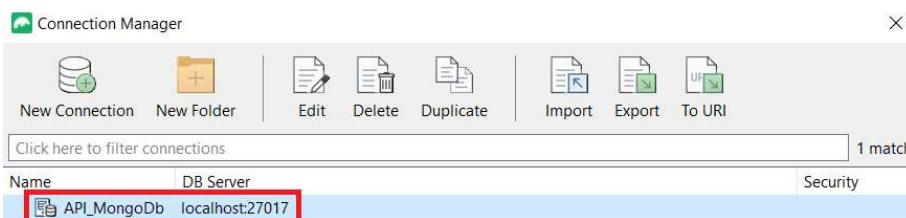
We press the “Test Connection” button.



Finally, we set the connection name and press the “Save” button.



Now the new connection appears in the Connection Manager list. Select it and press the "Connect" button.



We are now connected to the MongoDB Primary localhost in port 27017.

The screenshot shows the Studio 3T Free interface. The top menu bar includes File, Edit, Database, Collection, Index, Document, View, and Help. Below the menu is a toolbar with icons for Connect, IntelliShell, Export, Import, Feedback, and Want More?. A sidebar on the left labeled 'Open connections' shows a single entry: 'API_MongoDb [replica set: myReplicaSet] [direct]'. This entry has three sub-options: admin, config, and local. To the right of the connection list, a 'Connection' status bar displays 'Connection: API_MongoDb [replica set: myReplicaSet] [direct]', 'Server(s): localhost:27017 - Online [PRIMARY]', and 'Server version: 6.0.4'. A yellow callout box highlights the 'localhost:27017 - Online [PRIMARY]' text. At the bottom right of the interface, the text 'Welcome to Studio 3T Free' is visible.

The same operation we could repeat for the other two nodes. The result is illustrated in the following image:

This screenshot shows the Studio 3T Free interface with three open connections listed under 'Open connections': 'API_MongoDb [replica set: myReplicaSet] [direct]', 'API_MongoDb_second [replica set: myReplicaSet] [direct]', and 'API_MongoDb_third [replica set: myReplicaSet] [direct]'. Each of these connections has three sub-options: admin, config, and local. To the right of the connections, a 'Recent Connections' section lists three entries: 'API_MongoDb_second (localhost:27018)' (last accessed moments ago), 'API_MongoDb_third (localhost:27019)' (last accessed moments ago), and 'API_MongoDb (localhost:27017)' (last accessed 4 minutes ago). A yellow callout box highlights the 'localhost:27017' text in the recent connections list.

We can create the “BookStore” database and inside it, we create a new collection “Books”. We populate the collection with several documents.

As we can see in the above screen, there are three mongo docker containers running (the MongoDB ReplicaSet). Run the command “docker ps” to see the running containers

A screenshot of a Windows Command Prompt window titled 'Command Prompt'. The window shows the output of the 'docker ps' command. The table lists three Docker containers:

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
30aa9a4093bb	mongo:latest	"docker-entrypoint.s..."	23 hours ago	Up 23 hours	0.0.0.0:27019->27017/tcp	mongo3
dd2337cbc992	mongo:latest	"docker-entrypoint.s..."	23 hours ago	Up 23 hours	0.0.0.0:27018->27017/tcp	mongo2
07c89fc19770	mongo:latest	"docker-entrypoint.s..."	23 hours ago	Up 23 hours	0.0.0.0:27017->27017/tcp	mongo1

We enter in a container bash running the command: docker exec -it mongo1 bash

```
C:\ Command Prompt - docker exec -it mongo1 bash

C:\>docker ps
CONTAINER ID   IMAGE      COMMAND       CREATED     STATUS      PORTS          NAMES
9bcb933ccb01   mongo:latest "docker-entrypoint.s..."  20 seconds ago Up 19 seconds  0.0.0.0:27019->27017/tcp  mongo3
06bc89d5ee22   mongo:latest "docker-entrypoint.s..."  25 seconds ago Up 24 seconds  0.0.0.0:27018->27017/tcp  mongo2
f2ff92fc392d   mongo:latest "docker-entrypoint.s..."  32 seconds ago Up 31 seconds  0.0.0.0:27017->27017/tcp  mongo1

C:\>docker exec -it mongo1 bash
root@f2ff92fc392d:/#
```

Now we enter in the MongoDB Shell (<https://www.mongodb.com/docs/mongodb-shell/>) executing this command: mongsh

```
C:\ mongosh mongodb://127.0.0.1:27017/?directConnection=true&serverSelectionTimeoutMS=2000

C:\>docker exec -it mongo1 bash
root@f2ff92fc392d:/# mongosh
Current Mongosh Log ID: 63ff07fd0ce066ee4b458220
Connecting to:      mongodb://127.0.0.1:27017/?directConnection=true&serverSelectionTimeoutMS=2000&appName=mongosh+1.6.2
Using MongoDB:      6.0.4
Using Mongosh:      1.6.2

For mongosh info see: https://docs.mongodb.com/mongodb-shell/

-----
The server generated these startup warnings when booting
2023-03-01T07:58:15.299+00:00: Using the XFS filesystem is strongly recommended with the WiredTiger storage engine. See http://dochub.mongodb.org/core/prodnotes-filesystem
2023-03-01T07:58:15.879+00:00: Access control is not enabled for the database. Read and write access to data and configuration is unrestricted
2023-03-01T07:58:15.879+00:00: /sys/kernel/mm/transparent_hugepage/enabled is 'always'. We suggest setting it to 'never'
2023-03-01T07:58:15.879+00:00: vm.max_map_count is too low
-----

-----
Enable MongoDB's free cloud-based monitoring service, which will then receive and display metrics about your deployment (disk utilization, CPU, operation statistics, etc).

The monitoring data will be available on a MongoDB website with a unique URL accessible to you and anyone you share the URL with. MongoDB may use this information to make product improvements and to suggest MongoDB products and deployment options to you.

To enable free monitoring, run the following command: db.enableFreeMonitoring()
To permanently disable this reminder, run the following command: db.disableFreeMonitoring()

-----
myReplicaSet [direct: primary] test
```

The **-it** flag is short for **--interactive + --tty**. When you docker run with this command it takes you straight inside the container.

--interactive flag, allows you to send commands to the container via standard input ("STDIN"), which means you can "interactively" type commands to the pseudo-tty/terminal
--tty flag, tells Docker to allocate a virtual terminal session within the container

In this case is required the mongo1 docker container is previously running.

```
c:\ mongosh mongodb://127.0.0.1:27017/?directConnection=true&serverSelectionTimeoutMS=2000
C:\ docker exec -it mongo1 mongosh
Current Mongosh Log ID: 63f4bb8fcc7abcb969949ef1
Connecting to:      mongodb://127.0.0.1:27017/?directConnection=true&serverSelectionTimeoutMS=2000&appName=mongosh+1.6.2
Using MongoDB:      6.0.4
Using Mongosh:      1.6.2

For mongosh info see: https://docs.mongodb.com/mongodb-shell/

-----
The server generated these startup warnings when booting
2023-02-20T13:33:51.981+00:00: Using the XFS filesystem is strongly recommended with the WiredTiger storage engine. See http://dochub.mongodb.org/core/prodnotes-filesystem
2023-02-20T13:33:53.386+00:00: Access control is not enabled for the database. Read and write access to data and configuration is unrestricted
2023-02-20T13:33:53.386+00:00: /sys/kernel/mm/transparent_hugepage/enabled is 'always'. We suggest setting it to 'never'
2023-02-20T13:33:53.386+00:00: vm.max_map_count is too low
-----

Enable MongoDB's free cloud-based monitoring service, which will then receive and display metrics about your deployment (disk utilization, CPU, operation statistics, etc).

The monitoring data will be available on a MongoDB website with a unique URL accessible to you and anyone you share the URL with. MongoDB may use this information to make product improvements and to suggest MongoDB products and deployment options to you.

To enable free monitoring, run the following command: db.enableFreeMonitoring()
To permanently disable this reminder, run the following command: db.disableFreeMonitoring()

-----
myReplicaSet [direct: primary] test> 
```

Now we are inside the MongoDb Shell, we can type the following mongo commands:

To list the databases run the command:

```
>show databases
```

```
myReplicaSet [direct: primary] test> show databases
admin   80.00 KiB
config  160.00 KiB
local   404.00 KiB
```

To switch or create the database “BookStore” run the command:

```
>use BookStore
```

To create a new collection “Books” run the command

```
>db.createCollection('Books')
```

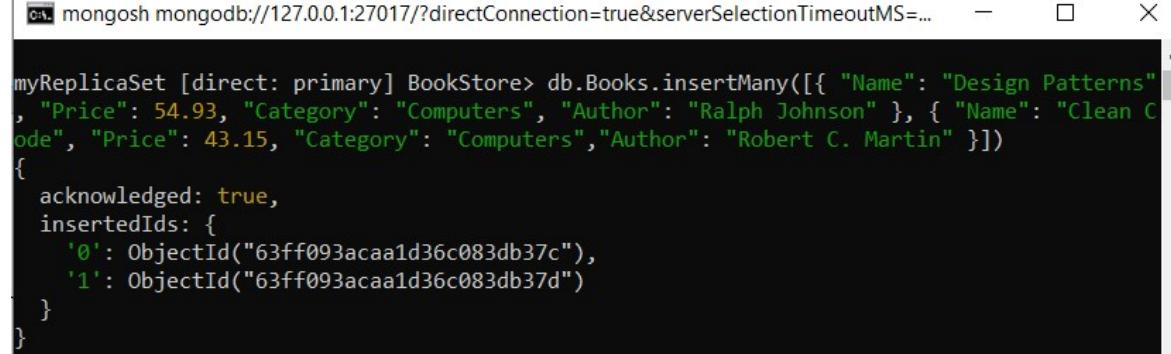
You can also show the collections inside the database running the command:

```
>show collections
```

```
myReplicaSet [direct: primary] BookStore> db.createCollection('Books')
{ ok: 1 }
myReplicaSet [direct: primary] BookStore> show collections
Books
```

To insert many items/documents inside the “Books” collections then type the command:

```
>db.Books.insertMany([{"Name": "Design Patterns", "Price": 54.93, "Category": "Computers", "Author": "Ralph Johnson"}, {"Name": "Clean Code", "Price": 43.15, "Category": "Computers", "Author": "Robert C. Martin"}])
```

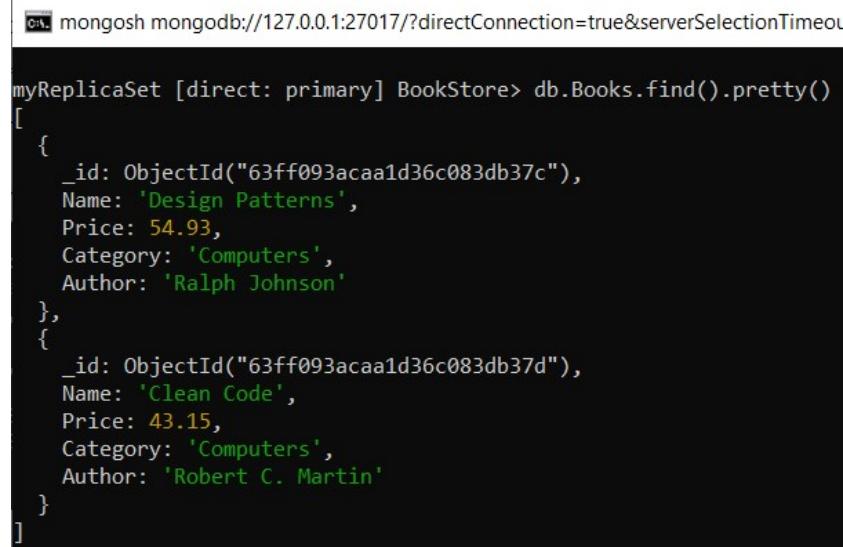


```
mongosh mongodb://127.0.0.1:27017/?directConnection=true&serverSelectionTimeoutMS=...
```

```
myReplicaSet [direct: primary] BookStore> db.Books.insertMany([{"Name": "Design Patterns", "Price": 54.93, "Category": "Computers", "Author": "Ralph Johnson"}, {"Name": "Clean Code", "Price": 43.15, "Category": "Computers", "Author": "Robert C. Martin"}])
{
  acknowledged: true,
  insertedIds: {
    '0': ObjectId("63ff093acaa1d36c083db37c"),
    '1': ObjectId("63ff093acaa1d36c083db37d")
  }
}
```

Finally, if you would like to show the items/documents inside the “Books” collection then type the command:

```
> db.Books.find().pretty()
```

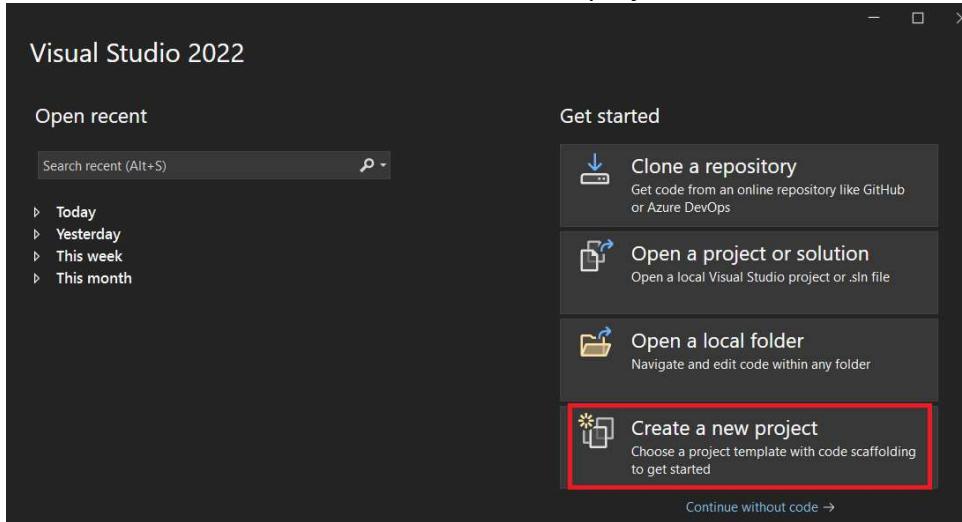


```
mongosh mongodb://127.0.0.1:27017/?directConnection=true&serverSelectionTimeoutMS=...
```

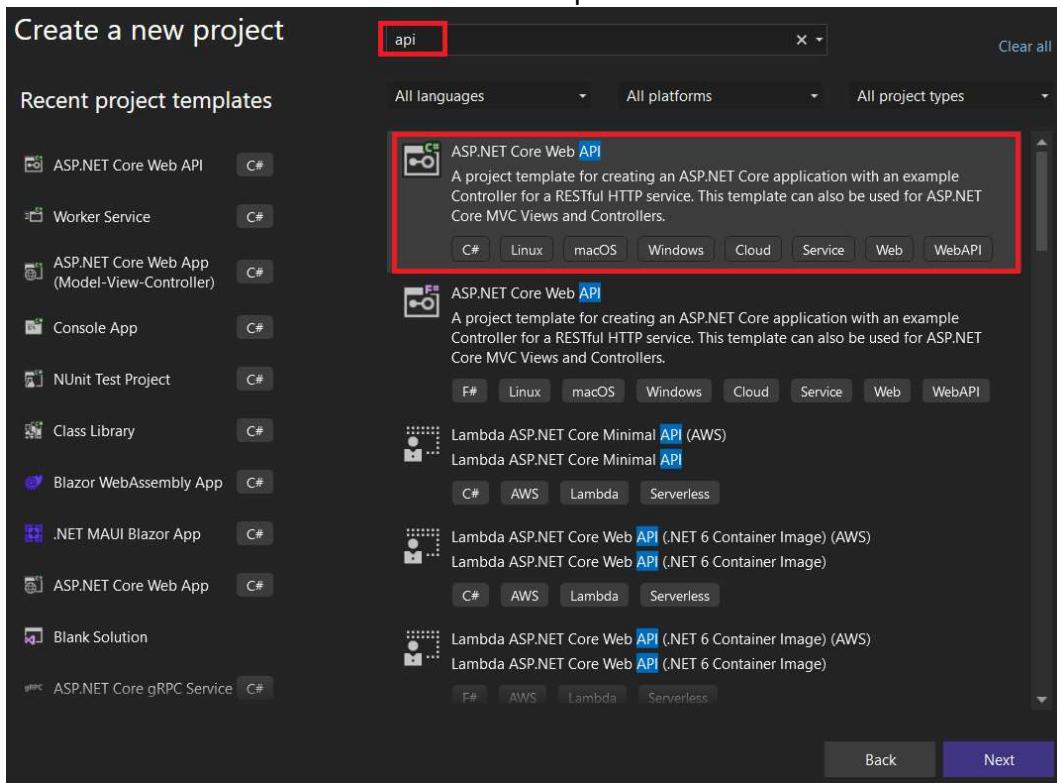
```
myReplicaSet [direct: primary] BookStore> db.Books.find().pretty()
[
  {
    _id: ObjectId("63ff093acaa1d36c083db37c"),
    Name: 'Design Patterns',
    Price: 54.93,
    Category: 'Computers',
    Author: 'Ralph Johnson'
  },
  {
    _id: ObjectId("63ff093acaa1d36c083db37d"),
    Name: 'Clean Code',
    Price: 43.15,
    Category: 'Computers',
    Author: 'Robert C. Martin'
  }
]
```

3. Develop a web API with ASP.NET Core and MongoDB

We run Visual Studio 2022 and create a new project.



We select the “ASP.NET Core Web API” template.



We set the Project name and the solution path.

Configure your new project

ASP.NET Core Web API C# Linux macOS Windows Cloud Service Web WebAPI

Project name

BookStoreApi

Location

C:\Net Chapter\src_article\



Solution name ⓘ

BookStoreApi

Place solution and project in the same directory

Back

Next

We select the Framework (.Net 7) and rest of the options: configure https and use controllers, enable openAPI support.

Additional information

ASP.NET Core Web API C# Linux macOS Windows Cloud Service Web WebAPI

Framework ⓘ

.NET 7.0 (Standard Term Support)

Authentication type ⓘ

None

Configure for HTTPS ⓘ

Enable Docker ⓘ

Docker OS ⓘ

Linux

Use controllers (uncheck to use minimal APIs) ⓘ

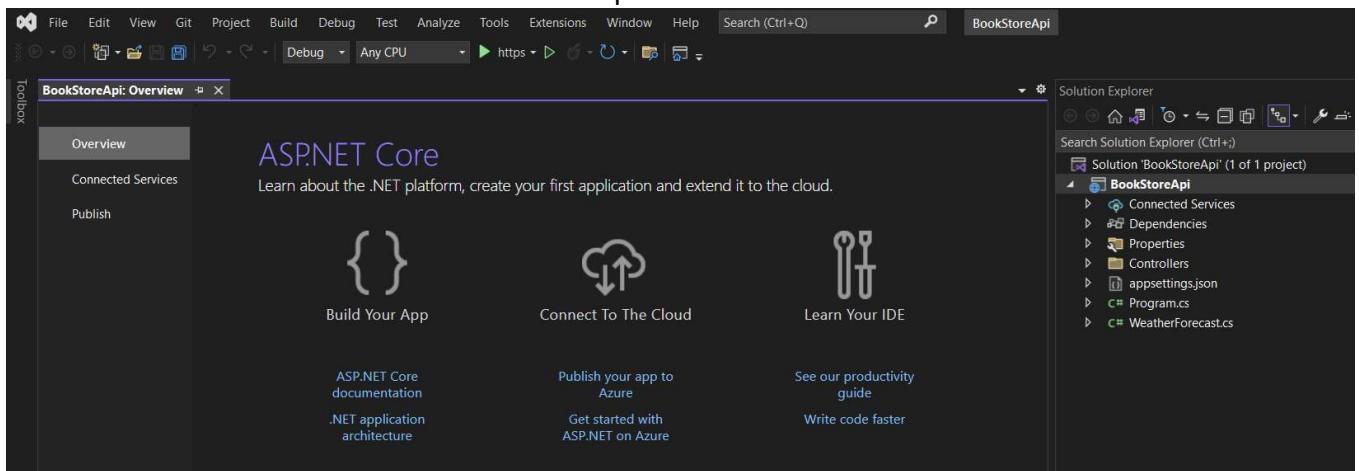
Enable OpenAPI support ⓘ

Do not use top-level statements ⓘ

Back

Create

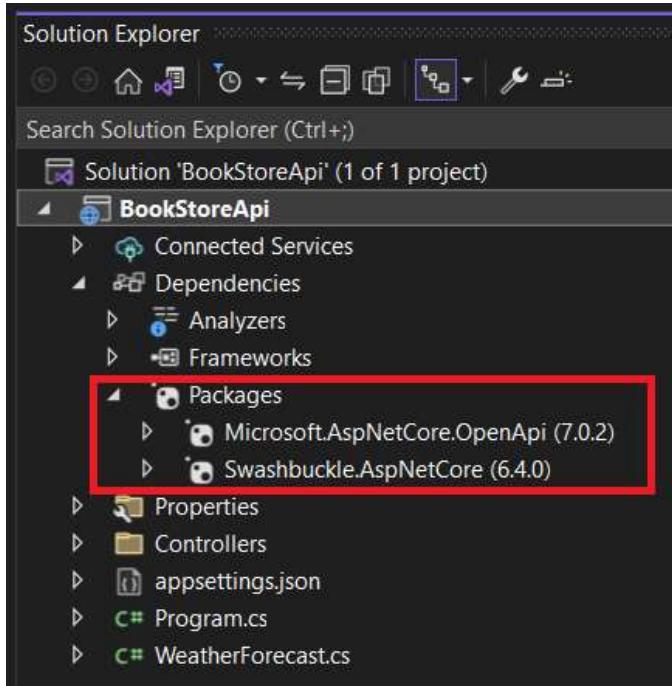
We can see the new solution in the Solution Explorer.



We run the solution to see that it is working properly.

A screenshot of a web browser displaying the Swagger UI for the BookStoreApi v1 API. The address bar shows "localhost:7063/swagger/index.html". The main page title is "BookStoreApi 1.0 OAS3". Below it, the "WeatherForecast" endpoint is listed under "GET /WeatherForecast". The "Parameters" section indicates "No parameters". At the bottom are "Execute" and "Clear" buttons. The "Responses" section shows a "Curl" command to execute the API call: "curl -X 'GET' \ 'https://localhost:7063/WeatherForecast' \ -H 'accept: text/plain'" and the "Request URL" "https://localhost:7063/WeatherForecast". The "Server response" section displays the JSON response body: [{"date": "2023-02-21", "temperatureC": -20, "temperatureF": -3, "summary": "Scorching"}, {"date": "2023-02-22", "temperatureC": 49, "temperatureF": 128, "summary": "Scorching"}].

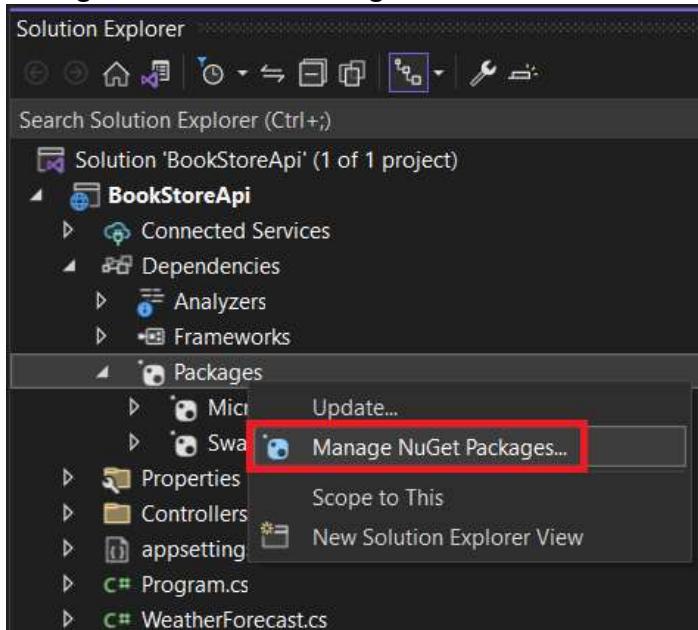
The first step is to load the NuGet libraries. Now we have only to dependencies.



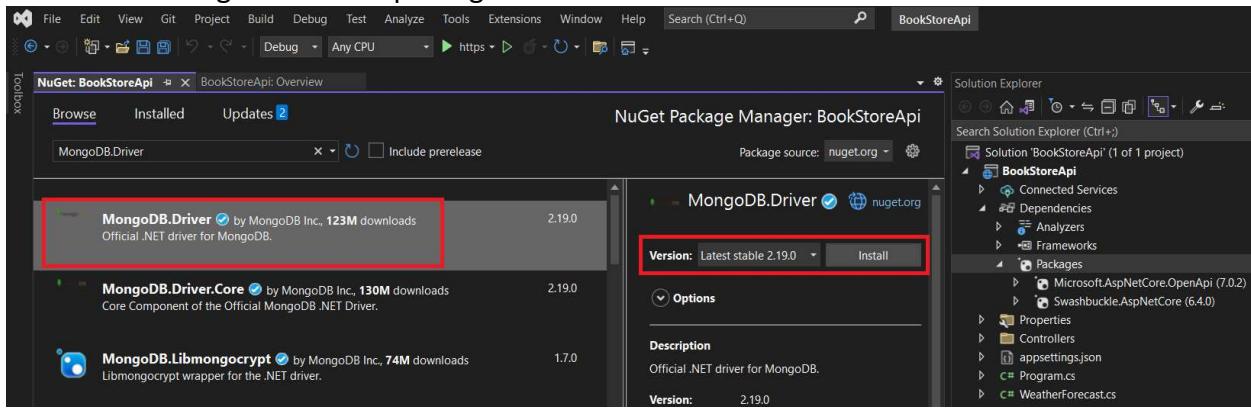
In order to access the MongoDB database from our API we have to install the following NuGet package:

MongoDB.Driver

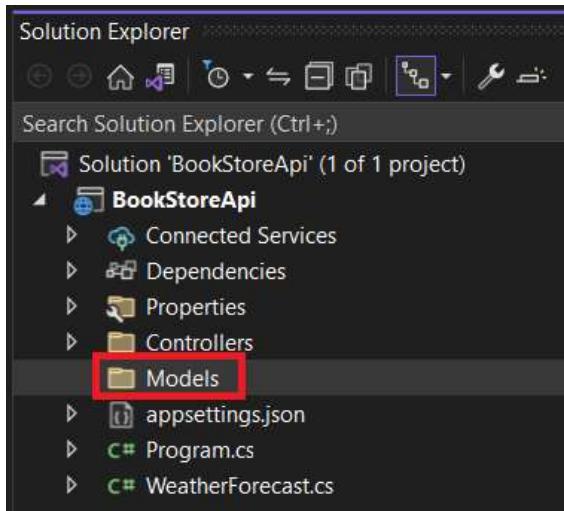
We right click on the “Packages” and select the “Manage NuGet Packages...” option.



Select the “MongoDB.Driver” package and install it.



We add the entity model. For this purpose, we add a “Models” folder/directory to the project root.



We add a “Book.cs” class to the “Entities” directory/folder with the following code:

```

using MongoDB.Bson;
using MongoDB.Bson.Serialization.Attributes;

namespace BookStoreApi.Entities;

public record Book
{
    [BsonId]//to mark as a PK for Mongo db
    [BsonRepresentation(BsonType.ObjectId)]//to allow passing as string instead of object
    id...?
    public string? Id { get; init; }

    //#[BsonElement("Name")]
    public string Name { get; init; } = null!;
    public decimal Price { get; init; }
    public string Category { get; init; } = null!;
    public string Author { get; init; } = null!;
}

```

In the “Book” class, we require the Id property for mapping the Common Language Runtime (CLR) object to the MongoDB collection. We use the [BsonId] annotation to make this property the document's primary key. We also use the [BsonRepresentation(BsonType.ObjectId)] annotation to allow passing the parameter as type string instead of an ObjectId structure.

Mongo handles the conversion from string to ObjectId. The BookName property is annotated with the [BsonElement] attribute. The attribute's value of Name represents the property name in the MongoDB collection. It is optional to use this attribute to rename the C# model property to a MongoDB column name.

We configure the MongoDB connection information in the appsettings.json file. We set the connection string, the database name and the books collection name:

```

...
"BookStoreDbSettings": {
    "ConnectionString": "mongodb://localhost:27017",
    "DatabaseName": "BookStore",
    "BooksCollectionName": "Books"
},
...

```

We add a “BookStoreDbSettings.cs” class to the “Configuration” directory/folder with the following code:

```

namespace BookStoreApi.Configuration;

public record BookStoreDbSettings
{
    public const string SectionName = "BookStoreDbSettings";
    public string ConnectionString { get; init; } = null!;
    public string DatabaseName { get; init; } = null!;
    public string BooksCollectionName { get; init; } = null!;
}

```

We configure the “BookStoreSettings” in the middleware (Program.cs file):

```
builder.Services.Configure<BookStoreDbSettings>(builder.Configuration
    .GetSection(BookStoreDbSettings.SectionName));

builder.Services.AddSingleton<BookStoreDbSettings>(sp => sp
    .GetRequiredService<IOptions<BookStoreDbSettings>>().Value);
```

Now we add a CRUD operations to the project. We add a “Services” directory/folder to the project root. Inside this folder we implement the repository interface and class.

We create the “IBookRepository.cs” interface:

```
using BookStoreApi.Entities;

namespace BookStoreApi.Services;

public interface IBooksRepository
{
    Task<IReadOnlyCollection<Book>> GetAll();
    Task<Book?> GetById(string id);

    Task Create(Book newBook);
    Task Update(string id, Book updatedBook);
    Task Remove(string id);
}
```

We implement the interface with the “MongoDbBooksRepository.cs” class:

```

using BookStoreApi.Configuration;
using BookStoreApi.Entities;
using MongoDB.Driver;

namespace BookStoreApi.Services;

public class MongoDBBooksRepository : IBooksRepository
{
    private readonly IMongoCollection<Book> _booksCollection;

    public MongoDBBooksRepository(BookStoreDbSettings settings)
    {
        MongoClient client = new(settings.ConnectionString);
        var database = client.GetDatabase(settings.DatabaseName);
        _booksCollection = database.GetCollection<Book>(settings.BooksCollectionName);
    }

    public async Task<IReadOnlyCollection<Book>> GetAll() => await _booksCollection
        .Find(_ => true)
        .ToListAsync();

    public async Task<Book?> GetById(string id) => await _booksCollection
        .Find(book => book.Id == id)
        .FirstOrDefaultAsync();

    public async Task Create(Book newBook) =>
        await _booksCollection.InsertOneAsync(newBook);

    public async Task Update(string id, Book updatedBook)
    {
        var ub = updatedBook with { Id = id };
        await _booksCollection.ReplaceOneAsync(book => book.Id == id, ub);
    }

    public async Task Remove(string id) => await _booksCollection
        .DeleteOneAsync(book => book.Id == id);
}

```

We configure the repository in the middleware (program.cs file).

```
builder.Services.AddSingleton<IBooksRepository, MongoDBBooksRepository>();
```

To finish this section 3, we add the controller. For this purpose, we create the “BooksController.cs” class inside the “Controllers” directory/folder. The controller includes the “IBooksRepository” dependency injection. By means of this service we access the functions (GetAsync, CreateAsync, UpdateAsync, RemoveAsync), now we can interchange information with the MongoDB database.

```
using BookStoreApi.Authorization;
using BookStoreApi.Entities;
using BookStoreApi.Services;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;

namespace BookStoreApi.Controllers;

[ApiController]
[Route("api/[controller]")]
public class BooksController : ControllerBase
{
    private const string IdUrlTemplate = "{id:length(24)}";
    private readonly IBooksRepository _booksRepository;

    public BooksController(IBooksRepository booksRepository)
    {
        _booksRepository = booksRepository;
    }

    [HttpGet]
    public async Task<IReadOnlyCollection<Book>> Get() => await _booksRepository.GetAll();

    [HttpGet(IdUrlTemplate)]
    public async Task<IActionResult> Get([FromRoute] string id)
    {
        var book = await _booksRepository.GetById(id);
        return book is null ? NotFound() : Ok(book);
    }

    [HttpPost]
    public async Task<IActionResult> Create([FromBody] Book book)
    {
        await _booksRepository.Create(book);
        return CreatedAtAction(nameof(Get), new { Id = book.Id }, book);
    }

    [HttpPut(IdUrlTemplate)]
    public async Task<IActionResult> Update([FromRoute] string id, [FromBody] Book book)
    {
        var existing = await _booksRepository.GetById(id);
        if (existing is null) return NotFound($"There is no book with id '{id}'");

        await _booksRepository.Update(id, book);
        return NoContent();
    }
}
```

```

[HttpDelete(IdUrlTemplate)]
public async Task<IActionResult> Remove([FromRoute] string id)
{
    var existing = await _booksRepository.GetById(id);
    if (existing is null) return NotFound($"There is no book with id '{id}'");

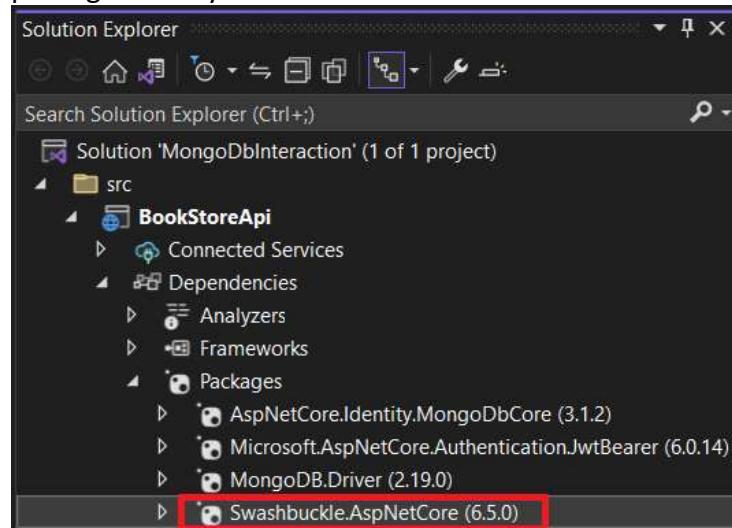
    await _booksRepository.Remove(id);
    return NoContent();
}

}//Controller

```

It is time to test the web API. For this purpose, we are going to use Swagger. Swagger UI offers a web-based UI that provides information about the service (API functions), using the generated OpenAPI specification. For more information see this link (ASP.NET Core web API documentation with Swagger/OpenAPI): <https://learn.microsoft.com/en-us/aspnet/core/tutorials/web-api-help-pages-using-swagger?view=aspnetcore-7.0>

Visual Studio 2022 when we create the API project, if we select the OpenAPI option, automatically integrates Swagger into our project. We can check it. In the “Solution Explorer” go to the “Dependencies” and inside “Packages” we can see the “Swashbuckle.AspNetCore(6.5.0)” package already referenced.



Also inside the middleware (Program.cs) we included the following code for implementing the Swagger service:

```

builder.Services.AddSwaggerGen(opt =>
{
...
});
...

```

```

if (app.Environment.IsDevelopment())
{
    app.UseSwagger();
    app.UseSwaggerUI(opt =>
    {
        opt.ConfigObject.TryItOutEnabled = true;
        opt.ConfigObject.DisplayRequestDuration = true;
    });
}

```

Now if we run the app, we can interact with BooksController via Swagger page: <https://localhost:7075/swagger/index.html>

4. We provide Authentication and Authorization to the web API

Authentication is the process of validating user credentials and authorization is the process of checking privileges for a user to access specific modules in an application. We protect an ASP.NET Core Web API application by implementing JWT authentication. We use authorization in ASP.NET Core to supply access to various functionalities of the application. We store user credentials in a MongoDB database.

JSON Web Token (JWT) is an open standard (RFC 7519) that defines a compact and self-contained way for securely transmitting information between parties as a JSON object. This information can be verified and trusted because it is digitally signed. JWTs can be signed using a secret (with the HMAC algorithm) or a public/private key pair using RSA or ECDSA.

In its compact form, JSON Web Tokens consist of three parts separated by dots (.), which are:

- Header
- Payload
- Signature

Therefore, a JWT typically looks like the following.

xxxx.yyyy.zzzz

You can find more details about JSON Web Tokens [here](#).

We install the Nugget packages:

- “AspNetCore.Identity.MongoDbCore”
This is a MongoDB UserStore and RoleStore adapter for Microsoft.Extension.Identity.Core 3.1.
- “Microsoft.AspNetCore.Authentication.JwtBearer”
This package enables the application to receive an OpenID Connect bearer token.

After loading the Nugget packages, we start our implementation creating a new directory/folder called “Authorization” where to store all the models to manage the authentication and authorization process.

We first define our Identity “User” and “Role” classes. The Identity User class inherits from “MongoIdentityUser” class and is mapped to “Users” collections.

```
using AspNetCore.Identity.MongoDbCore.Models;
using MongoDBGenericRepository.Attributes;

namespace BookStoreApi.Authorization;

[CollectionName("Users")]
public class ApplicationUser : MongoIdentityUser<Guid>
{
    public string? RefreshToken { get; set; }
    public DateTime RefreshTokenExpiryTime { get; set; }
}
```

The Identity Role class inherits from “MongoIdentityRole” class and is mapped to “Roles” collections. The code is given below.

```
[CollectionName("Roles")]
public class ApplicationRole : MongoIdentityRole<Guid>
{}
```

Also, we create a static class “UserRoles to add two constant values “Admin” and “User” as roles. You can add as many roles as you wish.

```
public class UserRoles
{
    public const string Admin = "Admin";
    public const string User = "User";
}
```

We create a “User.cs” class for new user registration. In this class we store the personal information for each user (Name, Email and Password).

```
using System.ComponentModel.DataAnnotations;

namespace BookStoreApi.Authorization;

public record User
{
    [Required] public string Name { get; init; } = null!;

    [Required]
    [EmailAddress(ErrorMessage = "Invalid Email")]
    public string Email { get; init; } = null!;

    [Required]
    public string Password { get; init; } = null!;
}
```

We store the login user credentials (name and password) inside the “LoginModel.cs” class. See the code below.

```
using System.ComponentModel.DataAnnotations;

namespace BookStoreApi.Authorization;
public record LoginModel
{
    [Required(ErrorMessage = "User name is needed")]
    public string UserName { get; init; } = null!;

    [Required(ErrorMessage = "Password is needed")]
    public string Password { get; init; } = null!;
}
```

To store the information required by the JWT authorization we create the “TokenModel.cs” class. In this class we store the AccessToken and the RefreshToken. The access token is valid for the time period we establish in the appsettings.json file. After that period of time, we have to create a new token with the refreshtoken. We will explain later in more detail.

```
public record TokenModel
{
    public string? AccessToken { get; set; }
    public string? RefreshToken { get; set; }
}
```

To manage the http requests responses, we create the “Response.cs” class. In this class we define the response message and the response status code.

```

namespace BookStoreApi.Authorization;

public record Response
{
    public string? Status { get; init; }
    public string? Message { get; init; }

    public static Response Success { get; } = new Response() { Status = "Success" };
    public static Response Error { get; } = new Response() { Status = "Error" };
}

```

After creating all the authentication and authorization models classes inside the “Authorization” directory/folder, we proceed with the appsettings.json file modifications. We have to include a new section “JWT”. Inside this new section in the appsettings.json file we define five new variable (ValidAudience, ValidIssuer, Secret, TokenValidityInMinutes and RefreshTokenValidityInDays).

```

...
"JwtSettings": {
    "ValidAudience": "http://localhost:4200",
    "ValidIssuer": "http://localhost:5094",
    "Secret": "JwtRefreshTokenSecuredString019~!#Password",
    "TokenValidity": "00:10:00",
    "RefreshTokenValidity": "7.00:00:00"
}

```

ValidAudience: What is the target of this token. In other words which services should accept this token an access token for the service. They may be many valid tokens in the world, but not all of those tokens have been granted by the user to allow access to the resources saved in the product services. A token valid for Google drive should not be accepted for GMail, even if both of them have the same issuer, they'll have different audiences.

ValidIssuer: Who created the token. In this case is our localhost web API. When we launch the API, the URL is set in the launchSettings.json.

```

...
"profiles": {
    "http": {
        "commandName": "Project",
        "dotnetRunMessages": true,
        "launchBrowser": true,
        "launchUrl": "swagger",
        "applicationUrl": "http://localhost:5000",
        "environmentVariables": {
            "ASPNETCORE_ENVIRONMENT": "Development"
        }
    },
}

```

Secret: The secret will be used for setting the JWT “IssuerSigningKey”. It is the public key used for validating incoming JWT tokens. By specifying a key here, the token can be validated without any need for the issuing server. What is needed, instead, is the location of the public key. The certLocation parameter, for example a string pointing to a .cer certificate file containing the

public key corresponding to the private key used by the issuing authentication server. Of course, this certificate could just as easily (and more likely) come from a certificate store instead of a file.

TokenValidityInMinutes: the access token is valid for the period of time established in this variable. After that time the token expires and the user should request a new token with the aid of the refresh token. In this case to have enough time to test the application we set the expiration time to 10 minutes. But in a production environment the most common practice is to set 1 minute for the expiration time to increase the system security.

RefreshTokenValidityInDays: as an additionally functionality we can set the refresh token expiration time. We create this variable in the appsettings.json file to track in the Authenticate Controller, but in this article is not implemented.

After setting the JWT configuration variables we have to modify the middleware (Program.cs) for including the authentication and authorization scenario. Let's explain which the main modifications are required.

We define the MongoDb, the "Identity" database, as the database for storing the users' credential (usernames, roles, etc). The MongoDb is running in three Docker containers our localhost in port 27017, 27018 and 27019.

```
builder.Services
    .AddIdentity<ApplicationUser, ApplicationRole>()
    .AddMongoDbStores<ApplicationUser, ApplicationRole, Guid>
    (
        bookStoreSettings.ConnectionString, "Identity"
    )
    .AddDefaultTokenProviders();
```

We also conifure the JWT Authentication system with the following code. For more information and a detail explanation take a look [here](#) and [here](#).

```
builder.Services
    .AddAuthentication(opt =>
    {
        opt.DefaultAuthenticateScheme = JwtBearerDefaults.AuthenticationScheme;
        opt.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme;
        opt.DefaultScheme = JwtBearerDefaults.AuthenticationScheme;
    })
    .AddJwtBearer(opt =>
    {
        opt.SaveToken = true;
        opt.RequireHttpsMetadata = false;
        opt.TokenValidationParameters = new TokenValidationParameters()
        {
            ValidateIssuer = true,
            ValidateAudience = true,
            ValidAudience = jwtSettings.ValidAudience,
            ValidIssuer = jwtSettings.ValidIssuer,
            IssuerSigningKey = jwtSettings.GetKey()
        };
    });
});
```

After introducing the required modification in the middleware, we are going to create an API controller “AuthenticateController.cs” inside the “Controllers” folder and add below code.

```
using System.IdentityModel.Tokens.Jwt;
using System.Security;
using System.Security.Claims;
using BookStoreApi.Authorization;
using BookStoreApi.Configuration;
using Microsoft.AspNetCore.Authentication;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Mvc;
using Microsoft.IdentityModel.Tokens;

namespace BookStoreApi.Controllers;

[ApiController]
[Route("api/[controller]")]
public class AuthenticateController : ControllerBase
{
    private readonly UserManager<ApplicationUser> _userManager;
    private readonly RoleManager<ApplicationRole> _roleManager;
    private readonly SignInManager<ApplicationUser> _signInManager;
    private readonly JwtSettings _jwtSettings;
    private readonly ISystemClock _systemClock;

    public AuthenticateController(
        UserManager<ApplicationUser> userManager
        , RoleManager<ApplicationRole> roleManager
        , SignInManager<ApplicationUser> signInManager
        , JwtSettings jwtSettings
        , ISystemClock systemClock)
    {
        _userManager = userManager;
        _roleManager = roleManager;
        _signInManager = signInManager;
        _jwtSettings = jwtSettings;
        _systemClock = systemClock;
    }
}
```

RegisterUser, RegisterAdmin: this function receives as a parameter the new user info (username, password and email). The action first looks for that user info in the MongoDb database, if the user already exists, send a message “User already exists!”, otherwise create the new user in the “Identity” database and assign the new user with the “User” or “Admin” role.

POST /api/Authenticate/RegisterUser

POST /api/Authenticate/RegisterAdmin

```

[HttpPost("[action]")]
public async Task<IActionResult> RegisterUser([FromBody] User user) =>
    await RegisterWithRole(user, UserRoles.User);

[HttpPost("[action]")]
public async Task<IActionResult> RegisterAdmin([FromBody] User user) =>
    await RegisterWithRole(user, UserRoles.Admin);

```

Login: after registering a new user, we have stored the user personal information in the Mongo database. The login action receives as a parameter the user's name and password and look for that user in the Mongo collection. If the user is found the CreateToken function is called to create an access token for that user.

POST /api/Authenticate/Login

```

[HttpPost("[action]")]
public async Task<IActionResult> Login([FromBody] LoginModel loginModel)
{
    var user = await _userManager.FindByNameAsync(loginModel.UserName);
    if (user == null
        || !await _userManager.CheckPasswordAsync(user, loginModel.Password))
        return Unauthorized();

    var roles = await _userManager.GetRolesAsync(user);
    var claims = new Claim[]
    {
        new(ClaimTypes.Name, user.UserName),
        new(JwtRegisteredClaimNames.Jti, Guid.NewGuid().ToString()),
    }
    .Concat(roles.Select(role => new Claim(ClaimTypes.Role, role)))
    .ToArray();

    var token = CreateToken(claims);

    var refreshToken = GenerateRefreshToken();
    user.RefreshToken = refreshToken;

    user.RefreshTokenExpiryTime = _systemClock.UtcNow.LocalDateTime
        .Add(_jwtSettings.RefreshTokenValidity);

    var signInResult = await _signInManager.PasswordSignInAsync
    (
        user, loginModel.Password, isPersistent: false, lockoutOnFailure: false
    );
    if (!signInResult.Succeeded)
        return Unauthorized();

    await _userManager.UpdateAsync(user);
    return Ok(new
    {
        Token = new JwtSecurityTokenHandler().WriteToken(token),
        RefreshToken = refreshToken,
        Expiration = _systemClock.UtcNowToLocalTime()
            .Add(_jwtSettings.TokenValidity)
    });
}

```

We create a token with the function “JwtSecurityToken”, and we attach the following properties: issuer, audience, claims, expires and signingCredentials:

```
private SecurityToken CreateToken(IEnumerable<Claim> claims) => new JwtSecurityToken
(
    issuer: _jwtSettings.ValidIssuer,
    audience: _jwtSettings.ValidAudience,
    claims: claims,
    expires: _systemClock.UtcNow.LocalDateTime.Add(_jwtSettings.TokenValidity),
    signingCredentials: new SigningCredentials
    (
        key: _jwtSettings.GetKey(),
        algorithm: SecurityAlgorithms.HmacSha256
    )
);
```

After the token expires, we need to create a new token. For this purpose we have to call the “RefreshToken” function. Remember in the appSettings.json we set the token expiration time.

POST /api/Authenticate/RefreshToken

```
[HttpPost("[action]")]
public async Task<IActionResult> RefreshToken([FromBody] TokenModel tokenModel)
{
    var principal = GetPrincipalFromExpiredToken(tokenModel.AccessToken);
    if (principal is null)
        return BadRequest();

    var user = await _userManager.FindByNameAsync(principal.Identity.Name);
    if (user is null
        || user.RefreshToken != tokenModel.RefreshToken
        || user.RefreshTokenExpiryTime <= _systemClock.UtcNow.LocalDateTime)
        return BadRequest();

    user.RefreshToken = GenerateRefreshToken();
    await _userManager.UpdateAsync(user);

    return Ok(new
    {
        AccessToken = new JwtSecurityTokenHandler()
            .WriteToken(CreateToken(principal.Claims)),
        RefreshToken = user.RefreshToken,
    });
}

private static string GenerateRefreshToken()
{
    var randomNumber = new byte[64];
    Random.Shared.NextBytes(randomNumber);
    return Convert.ToString(randomNumber);
}
```

In case we would like to revoke the permissions granted to a user, we can call the “Revoke” an user or “RevokeAll” users. These functions set to “null” the RefreshToken, then the user cannot create a new token after the expiration time:

POST /api/Authenticate/Revoke/{userName}

POST /api/Authenticate/RevokeAll

```
[Authorize]
[HttpPost("[action]/{userName}")]
public async Task<IActionResult> Revoke([FromRoute] string userName)
{
    var user = await _userManager.FindByNameAsync(userName);
    if (user is null)
        return BadRequest($"There is no user with name '{userName}'.");

    user.RefreshToken = null;
    await _userManager.UpdateAsync(user);

    return NoContent();
}

[Authorize]
[HttpPost("[action]")]
public async Task<IActionResult> RevokeAll()
{
    var users = _userManager.Users.ToList();
    foreach (var user in users)
    {
        user.RefreshToken = null;
        await _userManager.UpdateAsync(user);
    }

    return NoContent();
}
```

After the Authentication process (register user, login, create token, etc) defined in the “AuthenticateController.cs”, we have to authorize users to access the actions in the “BooksController.cs” and “WeatherForecastController.cs”. For this purpose, we include the [Authorize] attribute in all the actions where we require a granted permission to be accessed.

In our API we granted permission to the users with “Admin” role to access the actions inside the “BooksController.cs”.

```
using BookStoreApi.Authorization;
using BookStoreApi.Entities;
using BookStoreApi.Services;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;

namespace BookStoreApi.Controllers;

[Authorize(Roles = UserRoles.Admin)]
[ApiController]
[Route("api/[controller]")]
public class BooksController : ControllerBase
{
    private const string IdUrlTemplate = "{id:length(24)}";
    private readonly IBooksRepository _booksRepository;

    public BooksController(IBooksRepository booksRepository)
    {
        _booksRepository = booksRepository;
    }

    [AllowAnonymous]
    [HttpGet]
    public async Task<IReadOnlyCollection<Book>> Get() => await _booksRepository.GetAll();

    [AllowAnonymous]
    [HttpGet(IdUrlTemplate)]
    public async Task<IActionResult> Get([FromRoute] string id)
    {
        var book = await _booksRepository.GetById(id);
        return book is null ? NotFound() : Ok(book);
    }

    [HttpPost]
    public async Task<IActionResult> Create([FromBody] Book book)
    {
        await _booksRepository.Create(book);
        return CreatedAtAction(nameof(Get), new { Id = book.Id }, book);
    }

    [HttpPut(IdUrlTemplate)]
    public async Task<IActionResult> Update([FromRoute] string id, [FromBody] Book book)
    {
        var existing = await _booksRepository.GetById(id);
        if (existing is null) return NotFound($"There is no book with id '{id}'");

        await _booksRepository.Update(id, book);
        return NoContent();
    }
}
```

```
[HttpDelete(IdUrlTemplate)]
public async Task<IActionResult> Remove([FromRoute] string id)
{
    var existing = await _booksRepository.GetById(id);
    if (existing is null) return NotFound($"There is no book with id '{id}'");

    await _booksRepository.Remove(id);
    return NoContent();
}
}//Controller
```

5. Testing the API

We run the application in Visual Studio 2022.

<https://localhost:5001/swagger/index.html>

The screenshot shows the Swagger UI for the BookStoreApi v1. At the top, there is a navigation bar with the title "BookStoreApi 1.0 OAS3" and a dropdown menu "Select a definition" set to "BookStoreApi v1". Below the header, the main content area is divided into sections:

- Authenticate**: This section lists several POST methods:
 - /api/Authenticate/RegisterUser
 - /api/Authenticate/RegisterAdmin
 - /api/Authenticate/Login
 - /api/Authenticate/RefreshToken
 - /api/Authenticate/Revoke/{userName}
 - /api/Authenticate/RevokeAll
- Books**: This section lists various HTTP methods for the /api/Books endpoint:
 - GET /api/Books
 - POST /api/Books
 - GET /api/Books/{id}
 - PUT /api/Books/{id}
 - DELETE /api/Books/{id}

We create a new user with “Admin” role.

Swagger
Supported by SMARTBEAR

Select a definition BookStoreApi v1

BookStoreApi 1.0 (OpenAPI 3)

<https://localhost:5001/swagger/v1/swagger.json>

Authenticate

POST /api/Authenticate/registerUser

POST /api/Authenticate/registerAdmin

Parameters

No parameters

Request body

application/json

```
{
  "Name": "user2",
  "Email": "user2@example.com",
  "Password": "test123@"
}
```

Cancel Reset

Execute

Responses

Curl

```
curl -X 'POST' \
'https://localhost:5001/api/Authenticate/registerAdmin' \
-H 'Accept: */*' \
-H 'Content-Type: application/json' \
-d '{
  "Name": "user2",
  "Email": "user2@example.com",
  "Password": "test123@"
}'
```

Request URL

<https://localhost:5001/api/Authenticate/registerAdmin>

Server response

Code	Details	Links
200	<p>Response body</p> <pre>{ "Status": "Success", "Message": "User created successfully!" }</pre> <p>Download</p> <p>Response headers</p> <pre>access-control-allow-origin: * content-type: application/json; charset=utf-8 date: Wed,22 Feb 2023 11:32:49 GMT server: Kestrel</pre>	
Responses		
Code	Description	Links
200	Success	No links

Also, we create a new user with the "User" role.

The screenshot shows the BookStoreApi v1 documentation generated by Swagger. The main navigation bar includes the Swagger logo, the title 'BookStoreApi 1.0 OAS3', and a dropdown menu 'Select a definition' set to 'BookStoreApi v1'. Below the title, the URL 'https://localhost:5001/swagger/v1/swagger.json' is displayed.

The 'Authenticate' section is currently selected. It contains a 'POST /api/Authenticate/registerUser' request configuration. The 'Parameters' tab is active, showing 'No parameters'. The 'Request body' tab shows a JSON payload:

```
{
  "Name": "user1",
  "Email": "user1@example.com",
  "Password": "Test123@"
}
```

Below the request configuration is a large blue 'Execute' button. The status bar at the bottom of the interface shows the URL 'https://localhost:5001/api/Authenticate/registerUser'.

On the right side of the interface, there is a 'Responses' section. It includes a 'Curl' block with the command:

```
curl -X 'POST' \
'https://localhost:5001/api/Authenticate/registerUser' \
-H 'accept: */*' \
-H 'Content-Type: application/json' \
-d '{
  "Name": "user1",
  "Email": "user1@example.com",
  "Password": "Test123@"
}'
```

Below the curl command is a 'Request URL' field containing 'https://localhost:5001/api/Authenticate/registerUser'. The 'Server response' section shows a successful 200 status code response. The 'Details' table has one row for code 200, which includes a 'Response body' table with a single row:

{	"Status": "Success", "Message": "User created successfully!"		
---	---	--	--

Below the response body is a 'Response headers' table:

access-control-allow-origin: *
content-type: application/json; charset=utf-8
date: Wed,22 Feb 2023 11:31:05 GMT
server: Kestrel

The 'Responses' section also lists a 'Code' table for code 200, which shows the description 'Success' and a 'Links' column with 'No links'.

After we login and obtain the access token for each user: "user1" and "user2".

We proceed in the same way for the “user2”

The screenshot shows the Postman interface with a green 'POST' button and the URL '/api/Authenticate/login'. Under 'Parameters', there are no parameters listed. In the 'Request body' section, the content type is set to 'application/json' and the body contains the following JSON:

```
{
  "Username": "user2",
  "Password": "Test123@"
}
```

At the bottom, there are 'Execute' and 'Clear' buttons.

The screenshot shows the Postman interface with a blue 'Execute' button and a 'Responses' tab. Under 'Responses', there is a 'Curl' section with the command:

```
curl -X 'POST' \
'https://localhost:5001/api/Authenticate/login' \
-H 'accept: */*' \
-H 'Content-Type: application/json' \
-d '{
  "Username": "user2",
  "Password": "Test123@"
}'
```

Below it is a 'Request URL' field with the value 'https://localhost:5001/api/Authenticate/login'. The 'Server response' section shows a status code of 200. The response body is a JSON object containing a token, refresh token, and expiration date:

```
{
  "Token": "eyJhbGciOiJIUzI1NiIsInR5cIi6IkpxVCJ9.cy2odHlwO18vc2NoZn1hcy54bmxzb2FwLm9yZy93cy8yMDA1zA1L21k2h59aX85L2NsYm1tcy9uYm11joiidXN1cjl1LChdgk1013NgbyYTQxMS1m#W2LTQxZm0tYjAxMyjkN2NjYz1Nj3YvYm11lC3odhba01Bvc2No7d1hcv5faNbhb3NvZnQyZ9H13dzLzTzDqg0IDVahR1bnRp#lkvY2chah1l1L37vbGU101JB2Glpb1Ts1wAcC16MTY3NzA2NjY0NCvi0XN1jjoialR0cDov2rY2fsaG9zdd01MDAwIix1YXVK1joialR0cDov12xvY2F5aG9zdB00tjwfn0.F1LdOKKfMn20g.DhJuq4.5f2RQ0bbkIVRSkpzQ91Fc",
  "RefreshToken": "Hh12e5dalJPhNj2GGu+ujqFnUQm1Scajx2XcmJhM1ft81gsahZZivK1dxTaDG9BGFzBFxKKoNPFq0Fw71Mw==",
  "Expiration": "2023-02-22T12:50:40Z"
}
```

There is also a 'Download' button next to the response body. Below the response body is a 'Response headers' section with the following content:

```
access-control-allow-origin: *
cache-control: no-cache,no-store
content-type: application/json; charset=utf-8
date: Wed,22 Feb 2023 11:40:40 GMT
expires: Thu,01 Jan 1970 00:00:00 GMT
pragma: no-cache
server: Kestrel
```

We run Postman and we send a GET requests. As we populated, in previous section in this article, the “BookStore” database and “Books” collection with documents we receive as response the requested data.

For the “user1” with the “User” role we receive a “403 Forbidden” response status code, because as we mentioned in this article, we only authorized the “Admin” role user to access the actions inside the “BooksController”, as we explained in section 4.

The screenshot shows a Postman interface with a single request in the list. The request is a GET to `https://localhost:5001/api/Books`. In the 'Authorization' tab, the 'Type' is set to 'Bearer Token' and the token field contains a long, encoded string. The response status is 403 Forbidden, with a status message, time taken (144 ms), and size (99 B). The response body is empty.

However, for the “user2” we receive a response status code “200 OK” because the “user2” has the “Admin” role, and hence he is authorized to access all the actions inside the “BooksController”, as defined in section 4.

The screenshot shows a Postman interface with a single request in the list. The request is a GET to `https://localhost:5001/api/Books`. In the 'Authorization' tab, the 'Type' is set to 'Bearer Token' and the token field contains a long, encoded string. The response status is 200 OK, with a status message, time taken (140 ms), and size (150 B). The response body is empty.

In this first test I did not populated the MongoDB database containers, due to this issue I receive a blank array as response.

But if we populate the Mongo database with the following commands, we receive the JSON response with two books items retrieve from the “Books” collections inside the “BookStore” database.

```
C:/>docker exec -it mongo1 mongosh
>use BookStore
>db.createCollection('Books')
```

```
>db.Books.insertMany([{"Name": "Design Patterns", "Price": 54.93, "Category": "Computers", "Author": "Ralph Johnson"}, {"Name": "Clean Code", "Price": 43.15, "Category": "Computers", "Author": "Robert C. Martin"}])
```

The screenshot shows a Postman interface with a GET request to `https://localhost:5001/api/Books`. The response status is 200 OK with a time of 79 ms and a size of 389 B. The response body is displayed in JSON format, showing two book documents:

```

1
{
  "Id": "63f602ca4c99ee8be63794e1",
  "Name": "Design Patterns",
  "Price": 54.93,
  "Category": "Computers",
  "Author": "Ralph Johnson"
},
{
  "Id": "63f602ca4c99ee8be63794e2",
  "Name": "Clean Code",
  "Price": 43.15,
  "Category": "Computers",
  "Author": "Robert C. Martin"
}

```

6. Conclusion

In this article we define a methodology/procedure to create a .Net 7 web API with Mongo database, and we also provided authentication and authorization to access the API functions by means of JWT.

7. Future work

We propose the reader to achieve the integration test creating a Client Web API to send and receive data from the API developed in this article.

Also we recomend to deploy the API and the MongoDb Replicaset on an Elastic Container Instance (Azure ACI, AWS ECS, etc) or on Kubernetes hosted in the Cloud (Azure AKS, AWS EKS, Google GKE, etc).

Also it would be a good practice to redo this article instead with MongoDb with an Azure CosmosDb, a AWS DynamoDb , a Google Cloud DataStore or an Oracle NoSQL Database Cloud Service. These are NoSQL databases hosted in the Cloud. In these cases we can study to simplify the authentication and authorizations process and use the identity services available in the Clouds Providers.

8. References

[1] Create a web API with ASP.NET Core and MongoDB.

<https://learn.microsoft.com/en-us/aspnet/core/tutorials/first-mongo-app?view=aspnetcore-7.0&tabs=visual-studio>

[2] ASP.NET Core Identity with MongoDB as Database.

<https://www.yogihosting.com/aspnet-core-identity-mongodb/>

[3] JWT Authentication And Authorization In .NET 6.0 With Identity Framework.

<https://www.c-sharpcorner.com/article/jwt-authentication-and-authorization-in-net-6-0-with-identity-framework/>