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

Source Code here

In this article, we will show you how to build a web API using ASP.Net Core (.Net7) that can perform CRUD (Create, Read, Update and Delete) operations on a MongoDB NoSQL database. To ensure security, we will protect the API using JWT (JSON web tokens) authentication. In addition, we will demonstrate how to store user credentials in a MongoDB database.

Run MongoDb ReplicaSet in a container

Before starting to develop the web API we must run the MongoDb ReplicaSet in a Docker container. For this purpose, we first install Docker Desktop. We register and login in Docker Hub, we pull the MongoDb official image. We create a new network running the command: docker network create mongoCluster. Then we run the MongoDb three Docker containers: docker run -d --rm -p 27017:27017 --name mongo1 --network mongoCluster mongo:latest mongod --replSet myReplicaSet --bind_ip localhost,mongo1 (for running first container), docker run -d --rm -p 27018:27017 --name mongo2 --network mongoCluster mongo:latest mongod --replSet myReplicaSet --bind_ip localhost,mongo2 (for running the second container), and docker run -d --rm -p 27019:27017 --name mongo3 --network mongoCluster mongo:latest mongod --replSet myReplicaSet --bind_ip localhost,mongo3 (for running the third container). We initialize the MongoDb ReplicaSet with the command: 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}]})".

We install Studio 3T for MongoDb. We connect to the MongoDb ReplicaSet from Studio 3T. We can create the "BookStore" database and inside it, we create a new collection "Books." We populate the collection with several documents. We enter in a container bash with the command: docker exec -it mongo1 bash. Then we enter in the MongoShell: mongsh. To list the databases run the command: Show databases. 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. To insert may 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"}]). If you would like to show the items/ documents inside the "Books" collection then use the command: db.Books.find().pretty().

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 "BookStoreApi" and the solution path "C:\Net Chapter\src_article". We select the Framework (.Net 7) and rest of the options: Configure https, use controllers and enable openAPI support. After the solution is created, load the NuGet libraries: Microsoft.AspNetCore.OpenApi(7.0.2), Swashbuckle. AspNetCore(6.4.0) and MongoDB.Driver. We add the entity model, we create a "Models" folder/directory to the project root. We add a "Book.cs" class to the "Entities" directory/folder.

In the "Book" class, we require the "Id" property for mapping the 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 collectionname:"ConnectionString":"mongodb:// localhost :2027", "DatabaseName":"Bookstore", "BookCollectionName": "Books". We add a "BookStoreDbSettings.cs" class to the "Configuration" directory/folder. We configure the "BookStoreSettings" in the middleware "Program.cs" file. 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. We implement the interface with the "MongoDbBooksRepository.cs" class. We configure the repository in the middleware.

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. It is time to test the web API. For this purpose, we are going to use Swagger. Swagger UI offers a web-based UI OpenAPI specification that provides information about the API functions. For more information see **this link**.

In Visual Studio 2022 when we create the API project, if we select the OpenAPI option, automatically integrates Swagger into the project. To check it go to the "Solution Explorer" and in "Dependencies" inside "Packages" we can see the "Swashbuckle.AspNetCore(6.5.0)" package already referenced. Also, inside the middleware we include and configure the Swagger service. Now if we run the app, we can interact with BooksController via **Swagger page.**

We provide authentication and authorization to the web API

Authentication is the process of verifying user credentials, while authorization is the process of determining whether a user has permission to access specific modules within an application. In order to secure an ASP.NET Core Web API application, we can implement JWT (JSON web token) authentication. Additionally, we can use authorization in ASP.NET Core to grant access to various functionalities of the application based on a user's privileges. To store user credentials securely, we utilize a MongoDB database.

JSON web token (JWT) is an open standard (RFC 7519) that defines a secure and compact way to transmit information between different parties as a JSON object. Since JWTs are digitally signed, the transmitted data can be trusted and verified. JWTs can be signed using a secret key (with the HMAC algorithm) or a public/private key pair using RSA or ECDSA. The compact form of a JSON web token consists of three parts separated by dots (.), namely the Header, Payload, and Signature. This structure makes JWTs easily transferable and straightforward to process. Typically, a JWT looks like this: xxxx.yyyy.zzzz. If you require more information about JSON web tokens, you Tokens, you can find additional details through various resources online **here**.

We install the Nugget packages: "AspNetCore.Identity.MongoDbCore" and "Microsoft.AspNetCore.Authentication.JwtBearer". 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. The Identity Role class inherits from "MongoIdentityRole" class and is mapped to "Roles" collections. 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. 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). We store the login user credentials (name and password) inside the "LoginModel.cs" class.

To store the information required by the JWT authorization we create the "TokenModel.cs" class. In this class we store the access token and the refresh token. The access token is valid for the time period we establish in the "appsettings.json" file. After that period, we must create a new token with the refresh token. To manage the "http" responses we create the "Response.cs" class. In this class we define the response message and the response status code. After creating all the authentication and authorization models classes inside the "Authorization" directory/folder, we proceed with the "appsettings.json" file modifications. We must include a new section "JWT", inside this new section in the "appsettings.json" file we define five new variables (ValidAudience, ValidIssuer, Secret, TokenValidityInMinutes and RefreshTokenValidityInDays). After setting the JWT configuration variables we must modify the middleware (Program.cs file) to include the authentication and authorization scenario. Let's explain which main modifications are required.

We define the MongoDb “Identity” database, as the database for storing the users’ credentials (usernames, roles, etc). The MongoDb is running in three Docker containers our local host in port 27017, 27018 and 27019.

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

We also configure 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 create the API controller “AuthenticateController.cs” inside the “Controllers” folder.

RegisterUser, RegisterAdmin: These functions receive the new user’s information (username, password and email) as parameters. The first step is to check whether the user already exists in the MongoDb database. If the user is already registered, the function sends a message stating “User already exists!” If not, the function creates a new user in the “Identity” database and assigns them either the “User” or “Admin” role based on their access level.

```
[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 a new user registers, their personal information is stored in our Mongo database. When the user attempts to login, they provide their username and password, which we then use to search the Mongo collection for their corresponding user account. If a matching user account is found, we generate an access token for that user using the CreateToken function. This function creates a token using the JwtSecurityToken method and

attaches several properties to it, including the issuer, audience, claims, expiration time and signing credentials. By using this token-based authentication approach, we can ensure that only authorized users can access the protected resources of our system.

```
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
      )
);
```

Once the token expires, a new token must be created by calling the “RefreshToken” function. It is important to note that in the “appSettings.json”, the token expiration time was previously set.

```
private static string GenerateRefreshToken()
{
   var randomNumber = new byte[64];
   Random.Shared.NextBytes(randomNumber);
   return Convert.ToBase64String(randomNumber);
}
```

To revoke permissions granted to a user, we can use the “Revoke” or “RevokeAll” functions. These functions nullify the RefreshToken, preventing the user from creating a new token after it expires. After completing the authentication process (registering a user, logging in, creating a token, etc.), which is defined in the “AuthenticateController.cs”, we must authorize users to access the actions in the “BooksController.cs” and “WeatherForecastController.cs”. To do this, we add the [Authorize] attribute to all the actions that require a granted permission to be accessed. In our API, we grant permission to users with the “Admin” role to access the actions inside the “BooksController.cs”. To allow access to “Admin” role users, we provide the attribute [Authorize(Roles = UserRoles.Admin)], or [AllowAnonymous] if we grant access to all users.

Conclusion

This article outlines a comprehensive methodology for building a .NET7 web API with CRUD operations using a MongoDB database. In addition to providing step-by-step instructions for implementing these functionalities, we also cover how to add authentication and authorization protocols using JSON web tokens (JWT). To streamline the process further, we store both the users' roles and data within the same MongoDB database.

Future work

We suggest that readers test the integration by creating a Client Web API to send and receive data from the API developed in this article. We recommend deploying the API and MongoDb Replicaset on an Elastic Container Instance (such as Azure ACI, AWS ECS, or Kubernetes hosted in the cloud, such as Azure AKS, AWS EKS, or Google GKE). Additionally, it would be a good practice to update this article to include instructions for using NoSQL databases hosted in the cloud, such as Azure CosmosDb, AWS DynamoDb, Google Cloud DataStore, or Oracle NoSQL Database Cloud Service. In these cases, we can simplify the authentication and authorization processes by using the identity services provided by the cloud providers.

References

- Create a web API with ASP.NET Core and MongoDB
- ASP.NET Core Identity with MongoDB as Database
- JWT Authentication And Authorization In .NET 6.0 With Identity Framework

Author

Luis Coco Enríquez
Senior Software Engineer

Specialized in .NET
Azure Certified (IT administrator, associate developer and solutions architect)
PhD energy engineer