Introduction

This article will explain how to perform **CRUD (Create, Read, Update and Delete)** operations in ASP.NET Core Web API using Entity Framework Core. We will see step by step instructions about CRUD operations in ASP.NET Core Web API. In this demonstration, we will use the Database First Approach where our database will be ready before creating an actual code.

Let's move to this demonstration and create CRUD operations. But before moving to the practical demonstration of CRUD operation in ASP.NET Core Web API. Let's understand the objective of this demonstration which tells what exactly will be covered in this article. So, here is the objective as follows:

1. Create the Database and Tables
2. Create ASP.NET Core Web API Project
3. Install Nuget Packages for Entity Framework
4. Generate Model Classes
5. Setup Dependency Injection
6. Create Repository and implement CRUD operations
7. Create a Controller and Create API call
8. Test the API End Points in Postman
9. Enable CORS

Create the Database and Tables

First, we will open SQL Server Management Studio and create a database name as "BlogDB" along with two tables as "Category" and "Post". Category table will keep the information about the category of the blog post and Post table will actually keep the details about the post. So, you can create database and tables using the following SQL snippet code.

Hide   Copy Code

USE BlogDB

GO

CREATE TABLE Category(ID INT PRIMARY KEY IDENTITY(1,1), NAME VARCHAR(255), SLUG VARCHAR(255));

INSERT INTO Category(NAME, SLUG) VALUES('CSHARP', 'csharp');

INSERT INTO Category(NAME, SLUG) VALUES('VISUAL STUDIO', 'visualstudio');

INSERT INTO Category(NAME, SLUG) VALUES('ASP.NET CORE', 'aspnetcore');

INSERT INTO Category(NAME, SLUG) VALUES('SQL SERVER','sqlserver');

CREATE TABLE Post(POST\_ID INT PRIMARY KEY IDENTITY(1,1), TITLE VARCHAR(2000),

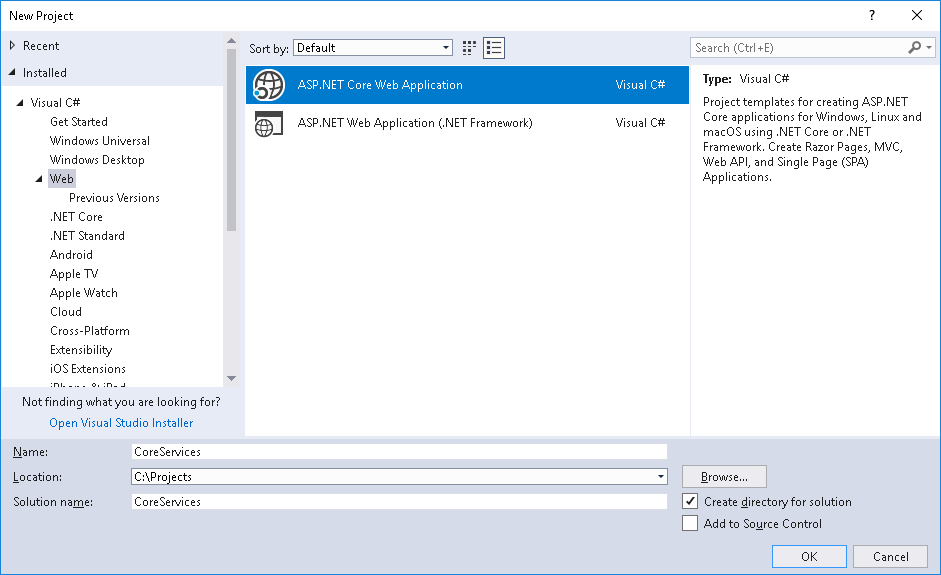
DESCRIPTION VARCHAR(MAX), CATEGORY\_ID INT FOREIGN KEY REFERENCES Category(ID),

CREATED\_DATE DATETIME);

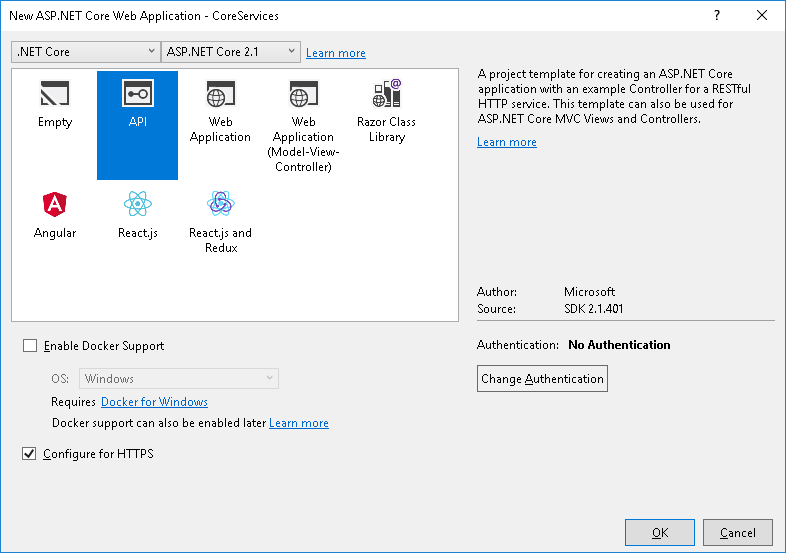
Create ASP.NET Core Web API Project

Now, we have database and tables are ready. So, let's move to create **ASP.NET Core Web API** project, but be sure your development environment is ready for .NET Core application. You should require some of the prerequisites to start this demonstration with **Visual Studio 2017**. First, you should have installed**.NET Core 2.0.0**or above SDK and second, you should have installed the latest version of **Visual Studio 2017, i.e., VS 2017 15.7+** version. Once your environment is ready, you are good to go. We should follow the below steps while creating new ASP.NET Core Web API.

1. Open **Visual Studio 2017**
2. Click to **File** > **New** > **Project**from the Menu
3. In New Project windows, from the left panel, select **Installed**> **Visual C#** > **Web**
4. Select the**ASP.NET Core Web Application**project template from the middle panel
5. Enter **CoreServices** as the name of the project and click **OK**



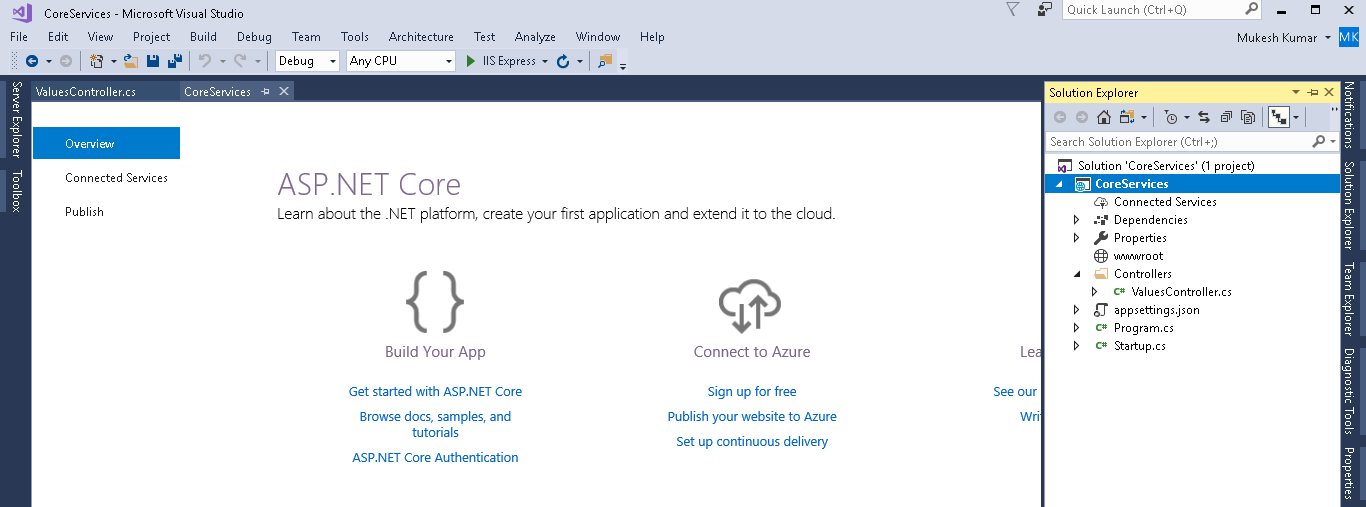
1. Next dialog will appear for the **New ASP.NET Core Web Application**.
2. Choose the **target framework** as **.NET Core**and select the version from the dropdown as**ASP.NET Core 2.1**
3. Select **API**as a template
4. Select the Authentication as **'No Authentication'**
5. Click **OK**



As we have selected '**Configure for HTTPS**' while creating the project, it will ask to configure SSL for this project. To configure this, click to **YES**.



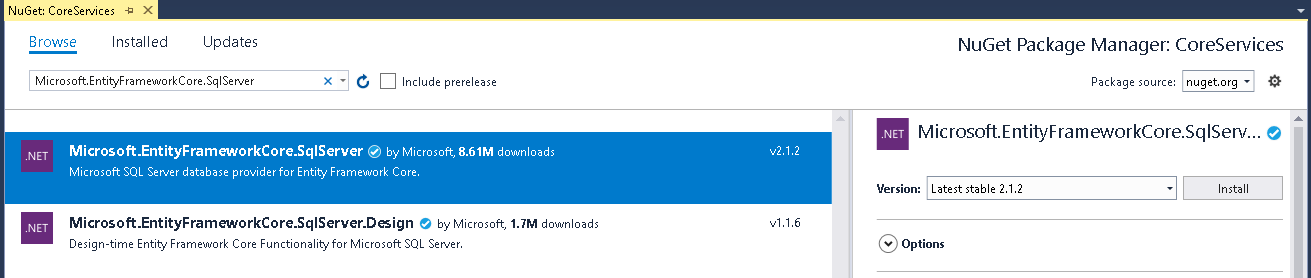
If we will do each step correctly, our project will be ready within a few seconds and it will look like follows. Here You can see the project structure with following images for the ASP.NET Core Web API. Although all the components are important for this application, mainly we should focus on ***Startup.cs***, ***Appsettings.json*, Dependencies and API Controller**.



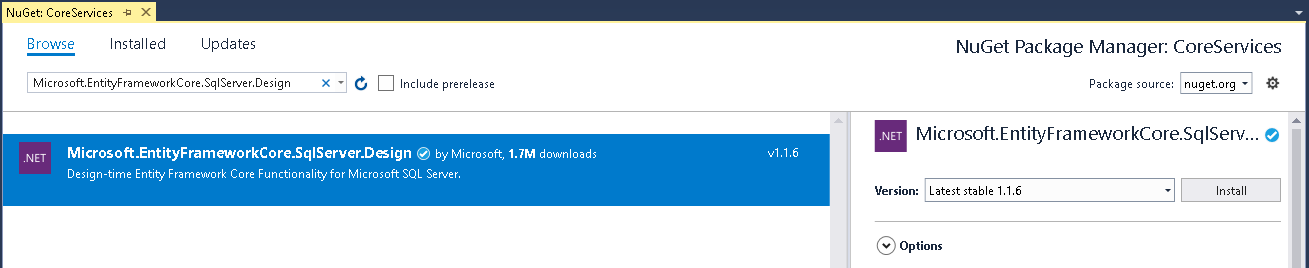
Install Nuget Packages for Entity Framework

As we already know, we will use **Entity Framework Core** as ORM for this demonstration. To use EF Core, we have to install some of the packages which will provide classes to work with EF Core. We can install these packages either using **Package Manager Console** to execute a command to install specific package or directly search on **NuGet Package Manager**and install it. So, let's install one by one required packages with this solution. So, right click to project and choose**Manage NuGet Packages**. It will open **Nuget Packages Manager**from where we can search any required ASP.NET Core package and install the specific version.

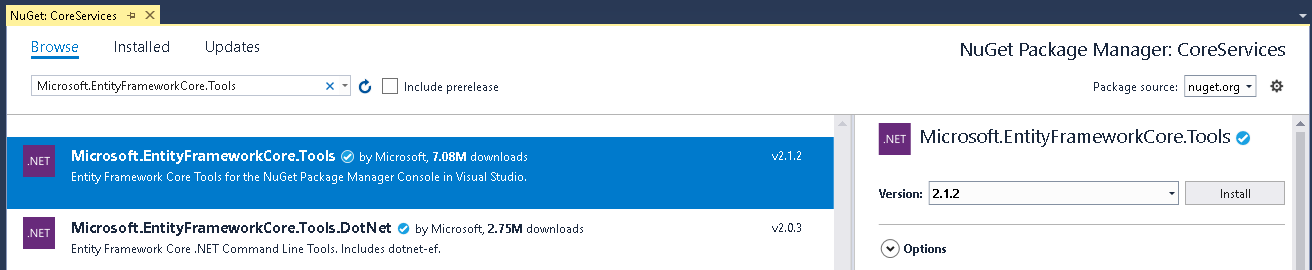
The first one is whichMicrosoft.EntityFrameworkCore.SqlServer will provide classes to connect with SQL Server for CRUD Operation to Entity Framework Core.



The next one is whichMicrosoft.EntityFrameworkCore.SqlServer.Design will provide design-time core functionality to Entity Framework Core for SQL Server.



And the last one is whichMicrosoft.EntityFrameworkCore.Tools will help to work with database related activity like add migration, script migration, get dbcontext, update the database, etc.



Generates Model Classes

As above, we have installed all the required packages which are required to work with Entity Framework Core. So, generating Entity Framework Core Models Classes from the database as we have a database already exists; we should run the following command from the Package Manager Console. To open it, go to **Tools** > **NuGet Package Manager** > **Package Manager Console** and execute the following commands.

Hide   Copy Code

Scaffold-DbContext "Server=DESKTOP-XYZ;Database=BlogDB;UID=sa;PWD=££££££££;"

Microsoft.EntityFrameworkCore.SqlServer -OutputDir Models

With the above command, we have provided SQL Server details like server name, username and password to access that server and existing database which contains tables. Apart from this, we have given *Output* directory where Models classes should be generated. This above command creates three classes inside the *Model* folder as BlogDBContextwhich is nothing but DbContext class for application and other two classes are Model classes which are related to database's tables.

Just for now, comment the OnConfiguring() method of BlogDBContext class because later, we will configure our Dependency Injection inside the *Startup.cs* class.

Hide   Shrink Image 8 for CRUD Operation in ASP.NET Core Web API with Entity Framework Core   Copy Code

using System;

using Microsoft.EntityFrameworkCore;

using Microsoft.EntityFrameworkCore.Metadata;

namespace CoreServices.Models

{

public partial class BlogDBContext : DbContext

{

public BlogDBContext()

{

}

public BlogDBContext(DbContextOptions<BlogDBContext> options)

: base(options)

{

}

public virtual DbSet<Category> Category { get; set; }

public virtual DbSet<Post> Post { get; set; }

*// protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)*

*// {*

*// if (!optionsBuilder.IsConfigured)*

*// {*

*//#warning To protect potentially sensitive information in your connection string,*

*//you should move it out of source code. See http://go.microsoft.com/fwlink/?LinkId=723263*

*//for guidance on storing connection strings.*

*// optionsBuilder.UseSqlServer("Server=DESKTOP-XYZ;Database=BlogDB;*

*// UID=sa;PWD=££££££;");*

*// }*

*// }*

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Category>(entity =>

{

entity.Property(e => e.Id).HasColumnName("ID");

entity.Property(e => e.Name)

.HasColumnName("NAME")

.HasMaxLength(255)

.IsUnicode(false);

entity.Property(e => e.Slug)

.HasColumnName("SLUG")

.HasMaxLength(255)

.IsUnicode(false);

});

modelBuilder.Entity<Post>(entity =>

{

entity.Property(e => e.PostId).HasColumnName("POST\_ID");

entity.Property(e => e.CategoryId).HasColumnName("CATEGORY\_ID");

entity.Property(e => e.CreatedDate)

.HasColumnName("CREATED\_DATE")

.HasColumnType("datetime");

entity.Property(e => e.Description)

.HasColumnName("DESCRIPTION")

.IsUnicode(false);

entity.Property(e => e.Title)

.HasColumnName("TITLE")

.HasMaxLength(2000)

.IsUnicode(false);

entity.HasOne(d => d.Category)

.WithMany(p => p.Post)

.HasForeignKey(d => d.CategoryId)

.HasConstraintName("FK\_\_Post\_\_CATEGORY\_I\_\_1273C1CD");

});

}

}

}

This is a Category Model class which is auto-generated from the database.

Hide   Copy Code

using System;

using System.Collections.Generic;

namespace CoreServices.Models

{

public partial class Category

{

public Category()

{

Post = new HashSet<Post>();

}

public int Id { get; set; }

public string Name { get; set; }

public string Slug { get; set; }

public ICollection<Post> Post { get; set; }

}

}

This is a Post Model class which is auto-generated from the database.

Hide   Copy Code

using System;

using System.Collections.Generic;

namespace CoreServices.Models

{

public partial class Post

{

public int PostId { get; set; }

public string Title { get; set; }

public string Description { get; set; }

public int? CategoryId { get; set; }

public DateTime? CreatedDate { get; set; }

public Category Category { get; set; }

}

}

As we have commented OnConfiguring() method of BlogDBContext and are now going to create our mechanism to access database and perform CRUD operation on this. First, we will define our connection stringinside the *appsettings.json* as follows:

Hide   Copy Code

{

"Logging": {

"LogLevel": {

"Default": "Warning"

}

},

"ConnectionStrings": {

"BlogDBConnection": "Server=DESKTOP-XYZ;Database=BlogDB;UID=sa;PWD=#########;"

},

"AllowedHosts": "\*"

}

Setup Dependency Injection

Now, let's create a Repositoryclass as the middle layer of this API. So, just create a folder name with '*Repository*' and create two files as the name with '*IPostRepository*' and '*PostRepository*. Just leave these interface and class as for now. We will implement CRUD implementation code with later. For now, just jump to *Startup.cs*class and add dependency injection for BlogDBContext class and PostRepository using the following code:

Hide   Copy Code

services.AddDbContext<BlogDBContext>(item => item.UseSqlServer

(Configuration.GetConnectionString("BlogDBConnection")));

services.AddScoped<IPostRepository, PostRepository>();

Whole code for the *Startup.cs* class is as follows:

Hide   Shrink Image 9 for CRUD Operation in ASP.NET Core Web API with Entity Framework Core   Copy Code

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using CoreServices.Models;

using CoreServices.Repository;

using Microsoft.AspNetCore.Builder;

using Microsoft.AspNetCore.Hosting;

using Microsoft.AspNetCore.HttpsPolicy;

using Microsoft.AspNetCore.Mvc;

using Microsoft.EntityFrameworkCore;

using Microsoft.Extensions.Configuration;

using Microsoft.Extensions.DependencyInjection;

using Microsoft.Extensions.Logging;

using Microsoft.Extensions.Options;

namespace CoreServices

{

public class Startup

{

public Startup(IConfiguration configuration)

{

Configuration = configuration;

}

public IConfiguration Configuration { get; }

*// This method gets called by the runtime. Use this method to add services to the container.*

public void ConfigureServices(IServiceCollection services)

{

services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version\_2\_1);

services.AddDbContext<BlogDBContext>(item => item.UseSqlServer

(Configuration.GetConnectionString("BlogDBConnection")));

services.AddScoped<IPostRepository, PostRepository>();

}

*// This method gets called by the runtime. Use this method*

*// to configure the HTTP request pipeline.*

public void Configure(IApplicationBuilder app, IHostingEnvironment env)

{

if (env.IsDevelopment())

{

app.UseDeveloperExceptionPage();

}

else

{

app.UseHsts();

}

app.UseHttpsRedirection();

app.UseMvc();

}

}

}

Now let's create a folder name with *ViewModel* and create one class as PostViewModel. This is nothing but a model class which is responsible for getting the data from multiple sources as we have to show data together for category and related post. So, that's why we have created this PostViewModel.

Hide   Copy Code

using System;

namespace CoreServices.ViewModel

{

public class PostViewModel

{

public int PostId { get; set; }

public string Title { get; set; }

public string Description { get; set; }

public int? CategoryId { get; set; }

public DateTime? CreatedDate { get; set; }

public string CategoryName { get; set; }

}

}

Create Repository and Implement CRUD Operations

Now jump to actual **CRUD** implementation with the code. So, open IPostRepository and add the required methods for the CRUD operations. So, we can see with the following IPostRepository interface, we have defined different methods for a different purpose. GetCategories will get the list of available category, GetPosts will get the list of available posts, GetPost will get the individual post for specific Post Id, AddPostwill add new post detail, DeletePost will delete the individual post based of Post Id and last UpdatePost will update the existing post. As we are returning Task-specific data, it means, data will return asynchronously.

Hide   Copy Code

using CoreServices.Models;

using CoreServices.ViewModel;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

namespace CoreServices.Repository

{

    public interface IPostRepository

    {

        Task<List<Category>> GetCategories();

        Task<List<PostViewModel>> GetPosts();

        Task<PostViewModel> GetPost(int? postId);

        Task<int> AddPost(Post post);

        Task<int> DeletePost(int? postId);

        Task UpdatePost(Post post);

    }

}

Now, let's jump to PostRepository and implement IPostRepository. First, we will get the instance of the BlogDBContextusing the **Constructor Dependency Injection**. Once an instance will be available, then we will move to further implementation. So, implement **CRUD operations (Create the Post, Read the Post, Update the Post and Delete the Post)** using the following codes. In this PostRepository class, we will interact with the database using **Entity Framework Core and perform the CRUD operations**.

Hide   Shrink Image 10 for CRUD Operation in ASP.NET Core Web API with Entity Framework Core   Copy Code

using CoreServices.Models;

using CoreServices.ViewModel;

using Microsoft.EntityFrameworkCore;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

namespace CoreServices.Repository

{

    public class PostRepository : IPostRepository

    {

        BlogDBContext db;

        public PostRepository(BlogDBContext \_db)

        {

            db = \_db;

        }

        public async Task<List<Category>> GetCategories()

        {

            if (db != null)

            {

                return await db.Category.ToListAsync();

            }

            return null;

        }

        public async Task<List<PostViewModel>> GetPosts()

        {

            if (db != null)

            {

                return await (from p in db.Post

                              from c in db.Category

                              where p.CategoryId == c.Id

                              select new PostViewModel

                              {

                                  PostId = p.PostId,

                                  Title = p.Title,

                                  Description = p.Description,

                                  CategoryId = p.CategoryId,

                                  CategoryName = c.Name,

                                  CreatedDate = p.CreatedDate

                              }).ToListAsync();

            }

            return null;

        }

        public async Task<PostViewModel> GetPost(int? postId)

        {

            if (db != null)

            {

                return await (from p in db.Post

                              from c in db.Category

                              where p.PostId == postId

                              select new PostViewModel

                              {

                                  PostId = p.PostId,

                                  Title = p.Title,

                                  Description = p.Description,

                                  CategoryId = p.CategoryId,

                                  CategoryName = c.Name,

                                  CreatedDate = p.CreatedDate

                              }).FirstOrDefaultAsync();

            }

            return null;

        }

        public async Task<int> AddPost(Post post)

        {

            if (db != null)

            {

                await db.Post.AddAsync(post);

                await db.SaveChangesAsync();

                return post.PostId;

            }

            return 0;

        }

        public async Task<int> DeletePost(int? postId)

        {

            int result = 0;

            if (db != null)

            {

*//Find the post for specific post id*

                var post = await db.Post.FirstOrDefaultAsync(x => x.PostId == postId);

                if (post != null)

                {

*//Delete that post*

                    db.Post.Remove(post);

*//Commit the transaction*

                    result = await db.SaveChangesAsync();

                }

                return result;

            }

            return result;

        }

        public async Task UpdatePost(Post post)

        {

            if (db != null)

            {

*//Delete that post*

                db.Post.Update(post);

*//Commit the transaction*

                await db.SaveChangesAsync();

            }

        }

    }

}

Create a Controller and Create API Calls

Now, we will do the actual implementation with the **API**controller to expose the **END-POINT**. So, first add a new API controller name as 'PostController' to right click on *Controller* folder and choose **Add**> **New Item**. So, again, we will get the instance of PostRepository using Constructor Dependency Injection and then first get the list of the category using the GetCategories() End Point, similarly, we will define rest of the End Point for CRUD operations.

Hide   Shrink Image 11 for CRUD Operation in ASP.NET Core Web API with Entity Framework Core   Copy Code

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using CoreServices.Models;

using CoreServices.Repository;

using Microsoft.AspNetCore.Mvc;

namespace CoreServices.Controllers

{

[Route("api/[controller]")]

[ApiController]

public class PostController : ControllerBase

    {

        IPostRepository postRepository;

        public PostController(IPostRepository \_postRepository)

        {

            postRepository = \_postRepository;

        }

        [HttpGet]

        [Route("GetCategories")]

        public async Task<IActionResult> GetCategories()

        {

            try

            {

                var categories = await postRepository.GetCategories();

                if (categories == null)

                {

                    return NotFound();

                }

                return Ok(categories);

            }

            catch (Exception)

            {

                return BadRequest();

            }

        }

        [HttpGet]

        [Route("GetPosts")]

        public async Task<IActionResult> GetPosts()

        {

            try

            {

                var posts = await postRepository.GetPosts();

                if (posts == null)

                {

                    return NotFound();

                }

                return Ok(posts);

            }

            catch (Exception)

            {

                return BadRequest();

            }

        }

        [HttpGet]

        [Route("GetPost")]

        public async Task<IActionResult> GetPost(int? postId)

        {

            if (postId == null)

            {

                return BadRequest();

            }

            try

            {

                var post = await postRepository.GetPost(postId);

                if (post == null)

                {

                    return NotFound();

                }

                return Ok(post);

            }

            catch (Exception)

            {

                return BadRequest();

            }

        }

        [HttpPost]

        [Route("AddPost")]

        public async Task<IActionResult> AddPost([FromBody]Post model)

        {

            if (ModelState.IsValid)

            {

                try

                {

                    var postId = await postRepository.AddPost(model);

                    if (postId > 0)

                    {

                        return Ok(postId);

                    }

                    else

                    {

                        return NotFound();

                    }

                }

                catch (Exception)

                {

                    return BadRequest();

                }

            }

            return BadRequest();

        }

        [HttpDelete]

        [Route("DeletePost")]

        public async Task<IActionResult> DeletePost(int? postId)

        {

            int result = 0;

            if (postId == null)

            {

                return BadRequest();

            }

            try

            {

                result = await postRepository.DeletePost(postId);

                if (result == 0)

                {

                    return NotFound();

                }

                return Ok();

            }

            catch (Exception)

            {

                return BadRequest();

            }

        }

        [HttpPut]

        [Route("UpdatePost")]

        public async Task<IActionResult> UpdatePost([FromBody]Post model)

        {

            if (ModelState.IsValid)

            {

                try

                {

                    await postRepository.UpdatePost(model);

                    return Ok();

                }

                catch (Exception ex)

                {

                    if (ex.GetType().FullName ==

  "Microsoft.EntityFrameworkCore.DbUpdateConcurrencyException")

                    {

                        return NotFound();

                    }

                    return BadRequest();

                }

            }

            return BadRequest();

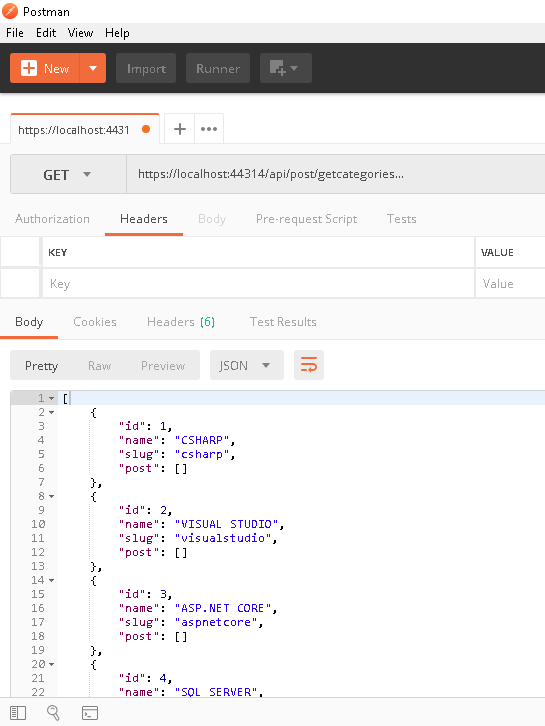
        }

    }

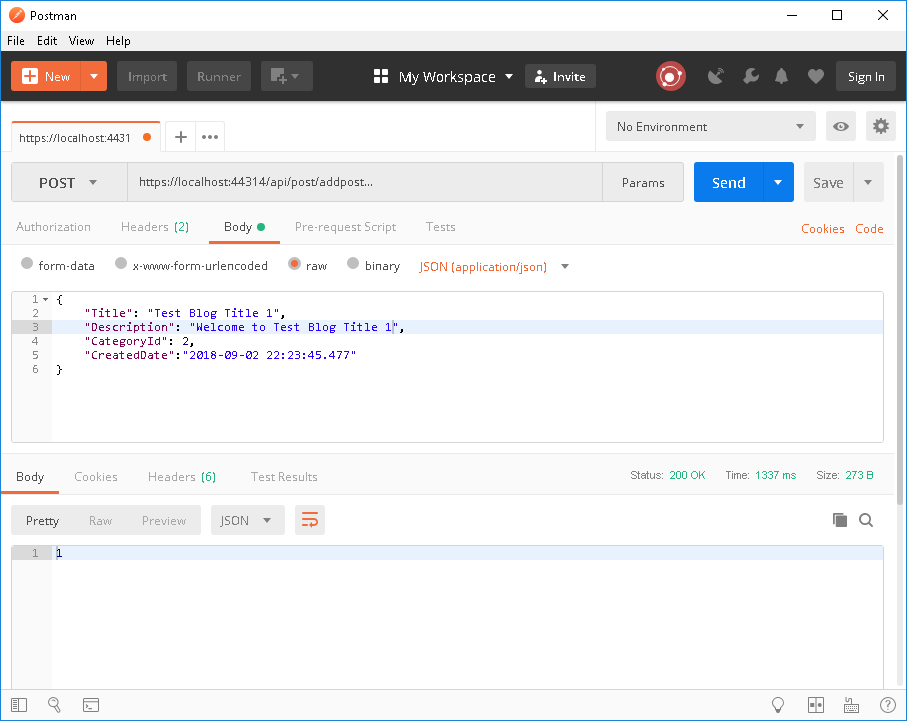
}

Test the API End-Points in Postman

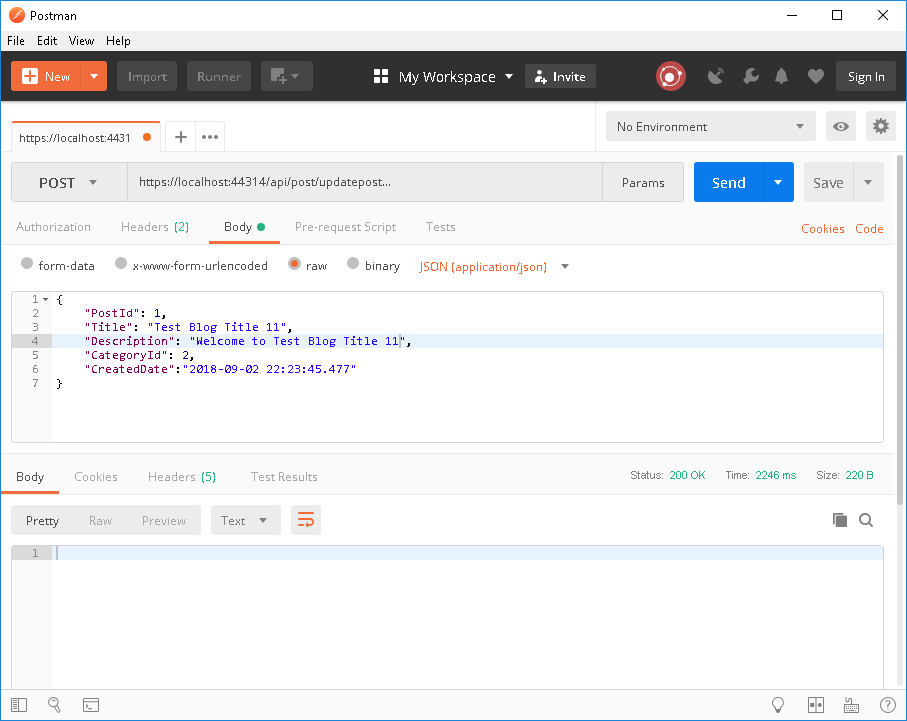
**Finally, we are done with the implementation of CRUD operations with ASP.NET Core Web API using Entity Framework Core and Repository**. So, let's move to Postman and test our API End Points. So, let's first test the GetCategories End Point as follows. Choose the Get as a method and pass the End Point inside the URL and define a header as'Content-Type': 'application/json'and click to Send button. It will return the list of available categories.



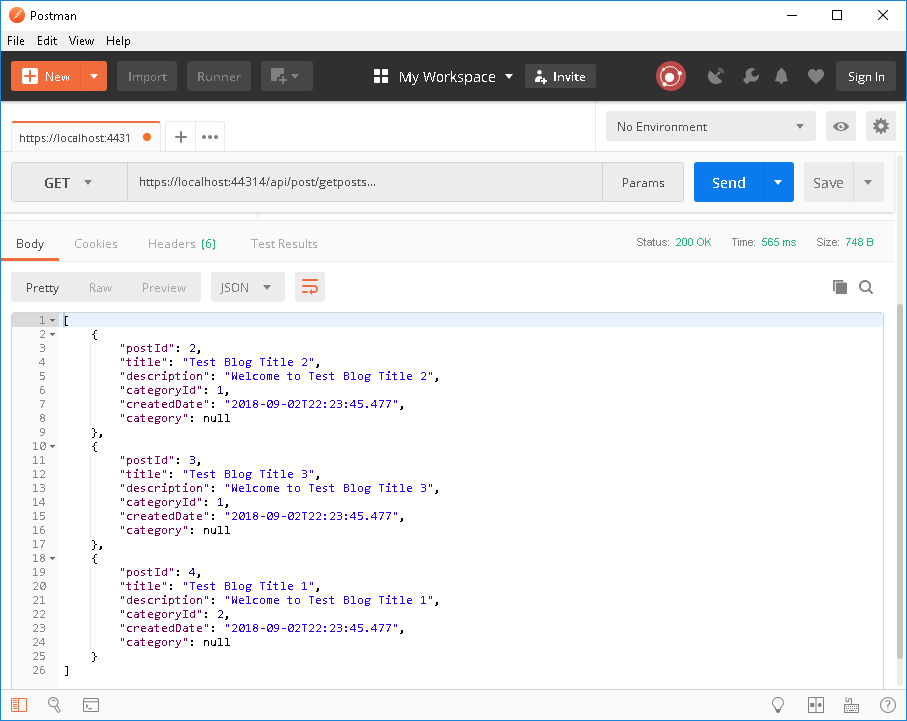
Next test for adding new post using Postman, so, first choose the method as a POST and pass the addpost End Point, add a header similar to the above for Content-Type and provide the post details which will add inside the RAW section. After setting this, click to SEND. It will add your record to the database.



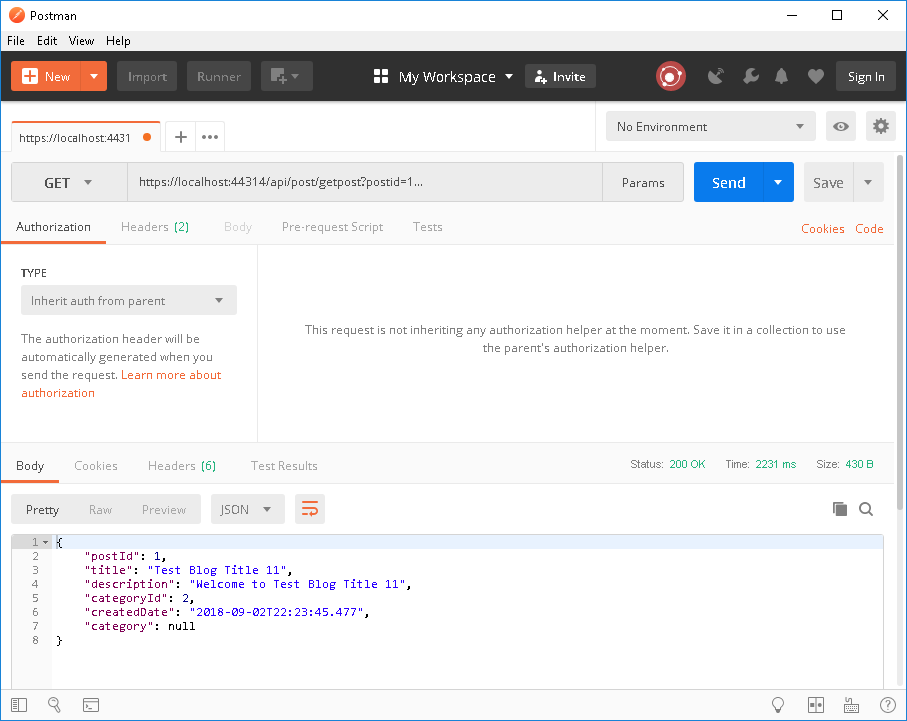
Next test for updating the existing post using Postman, so, first choose the method as a POST and pass the updatepost End Point, add a header similar to the above for Content-Type and provide the post details which need to be updated inside the RAW section. After setting this, click to SEND. It will update your existing record to the database.



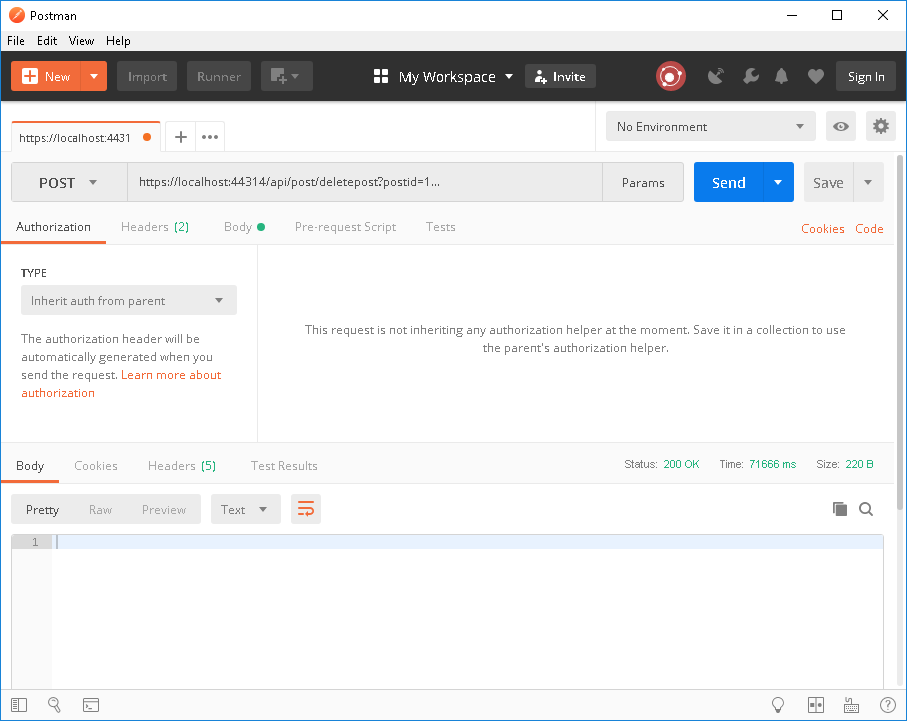
Next test for getting the list of available posts using Postman, so, first choose the method as a GET and pass the getposts End Point, add a header similar to the above for Content-Type. After setting this, click to SEND. It will give you a list of available post records from the database.



Next test for getting the individual post using Postman, so, first choose the method as a GET and pass the getpost along with Post Id End Point, add a header similar to the above for Content-Type. After setting this, click to SEND. It will give you individual post details from the database.



Next test for deleting the existing post using Postman, so, first choose the method as a POST and pass the deletepost End Point along with Post Id which needs to be deleted, adds a header similar to the above for Content-Type. After setting this, click to SEND. It will delete your existing record from the database.



Enable CORS

Wow... we have created an **ASP.NET Core Web API and added Entity Framework Core and implement CRUD operations**. In the end, we have also tested each End Point for the API and everything working fine. But one problem is persisting yet and that is CORS issue. If you will deploy this API somewhere and will use with some other application, then it will throw some **CORS** related exceptions. So, let's configure CORS also to make this API perfect. So, open *Startup.cs* and ConfigureService() method and add the following lines of code to create CORS policy.

Hide   Copy Code

services.AddCors(option => option.AddPolicy("MyBlogPolicy", builder => {

builder.AllowAnyOrigin().AllowAnyHeader().AllowAnyMethod();

}));

And then, use this CORS policy inside the Configure method as follows:

Hide   Copy Code

app.UseCors("MyBlogPolicy");

Following is the whole code for *Startup.cs* class along with CORS implementation.

Hide   Shrink Image 18 for CRUD Operation in ASP.NET Core Web API with Entity Framework Core   Copy Code

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using CoreServices.Models;

using CoreServices.Repository;

using Microsoft.AspNetCore.Builder;

using Microsoft.AspNetCore.Hosting;

using Microsoft.AspNetCore.HttpsPolicy;

using Microsoft.AspNetCore.Mvc;

using Microsoft.EntityFrameworkCore;

using Microsoft.Extensions.Configuration;

using Microsoft.Extensions.DependencyInjection;

using Microsoft.Extensions.Logging;

using Microsoft.Extensions.Options;

namespace CoreServices

{

public class Startup

{

public Startup(IConfiguration configuration)

{

Configuration = configuration;

}

public IConfiguration Configuration { get; }

*// This method gets called by the runtime. Use this method to add services to the container.*

public void ConfigureServices(IServiceCollection services)

{

services.AddCors(option => option.AddPolicy("MyBlogPolicy", builder => {

builder.AllowAnyOrigin().AllowAnyHeader().AllowAnyMethod();

}));

services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version\_2\_1);

services.AddDbContext<BlogDBContext>(item => item.UseSqlServer

(Configuration.GetConnectionString("BlogDBConnection")));

services.AddScoped<IPostRepository, PostRepository>();

}

*// This method gets called by the runtime.*

*// Use this method to configure the HTTP request pipeline.*

public void Configure(IApplicationBuilder app, IHostingEnvironment env)

{

if (env.IsDevelopment())

{

app.UseDeveloperExceptionPage();

}

else

{

app.UseHsts();

}

app.UseHttpsRedirection();

app.UseCors("MyBlogPolicy");

app.UseMvc();

}

}

}