ASPNET Core Web API Notes

1. Introduction
   1. This courses covers C#, .NET and ASP.NET.
   2. Requirements
      1. Microsoft .NET SDK 9.0.303
      2. Visual Studio
2. Creating the API

* 1. [Creating the Visual Studio program](#S01)

* 1. [Creating the Domain model](#S02)

* 1. [Creating the DBContext Class](#S03)

* 1. [Adding Connection to appsettings.json](#S04)

* 1. [Dependency Injection](#S05)

* 1. [Running Entity Framework Core Migrations](#S06)

* 1. [Creating New Controllers](#S07)

* 1. [Connecting our Controllers to DbContext](#S08)

* 1. [The ID Action Method](#S09)

* 1. [DTO and Domain Model](#S10)

* 1. [Changing Methods to Use DTOs](#S11)

* 1. [Creating a New Table Entry](#S12)

* 1. [Updating a Table Item](#S13)

* 1. [Deleting a Table Item](#S14)

* 1. [Asynchronous Programming](#S15)

* 1. [Repository Pattern and Benefits](#S16)

* 1. [AutoMapper](#S17)

* 1. [Creating the Walks Controller](#S18)

* 1. [Validations in ASP.NET](#S19)

* 1. [Filtering, Sorting and Pagination](#S20)

* 1. [Securing the ASP.NET API](#S21)

* 1. [Defining Control Authorization](#S22)

* 1. [Setting Up Identity in the API](#S23)

* 1. [Login Functionality using Passwords](#S24)

* 1. [Creating JWT Tokens](#S25)

* 1. [Role Based Authorization](#S26)

* 1. [Adding Authorization to Swagger](#S27)

* 1. [Testing the JWT Token Environment](#S28)

* 1. [Image Uploading in ASP.NET](#S29)

* 1. [Serving Static Files through ASP.NET](#S30)

* 1. [Logging in ASP.NET](#S31)

* 1. [Implementing Global Exception Handling](#S32)

1. Creating the UI
   1. Overview
   2. Creating the new UI Controller

1. [Creating the Visual Studio Program](#TS01)
   1. Open visual Studio
   2. Select the filters C#, All Platforms and Web at the top of the screen
   3. Select the ASP.NET Core Web API template and click next.
   4. Configure the new Project:
      1. Use these names:
         1. Project name: NZWalks.API
         2. Location: C:\Udemy\ASP\_NET\Sameer Saini
         3. Solution Name: NZWalks
      2. Additional Information
         1. Use these selections:
            1. Framework: .NET 9.0
            2. Authentication Type: None
            3. Configure for HTTPS: Yes
            4. Enable container support: No
            5. Enable OpenAPI support: Yes
            6. Do not use Top Level Statements: Yes
            7. Use Controllers: Yes
            8. Enlist in .NET Aspire Orchestration: No
         2. Click the create button
   5. Packages to install dependencies via NuGet
      1. Microsoft.EntityFrameworkCore.SqlServer
      2. Microsoft.EntityFrameworkCore.Tools
      3. Automapper
      4. Microsoft.AspNetCore.Authentication.JwtBearer
      5. Microsoft.IdentityModel.Tokens
      6. System.IdentityModel.Tokens.Jwt
      7. Microsoft.AspNetCore.Identity.EntityFrameworkCore

1. [Creating the Domain model](#TS02)
   1. There will be three Domain classes. These are:

|  |  |  |
| --- | --- | --- |
| Walk.cs | Region.cs | Difficulty.cs |
| namespace NZWalks.API.Controllers.Models.Domain  {  public class Walk  {  public Guid Id { get; set; }  public string Name { get; set; }  public string Description { get; set; }  public double LengthInKm { get; set; }  public string? WalkImageUrl { get; set; }  public Guid DifficultyId { get; set; }  public Guid RegionId { get; set; }  // Navigation properties  public Difficulty Difficulty { get; set; }  public Region Region { get; set; }  }  } | namespace NZWalks.API.Controllers.Models.Domain  {  public class Region  {  public Guid Id { get; set; }  public string Code { get; set; }  public string Name { get; set; }  public string? RegionImageUrl { get; set; }  }  } | namespace NZWalks.API.Controllers.Models.Domain  {  public class Difficulty  {  public Guid Id { get; set; }  public string? Name { get; set; }  }  } |

* 1. In the Walk domain, we have relationships between Walk.cs and the other two domains.
     1. Explanation of the Primary Key relationships:
        1. public Guid DifficultyId { get; set; } tells Entity FrameWork Walk.cs will have a difficulty, and this is the difficulty Id for it
        2. public Guid RegionId { get; set; } tells Entity FrameWork Walk.cs will have a Region, and this is the region Id for it.
        3. public Difficulty Difficulty { get; set; } is a navigation property and defines an one-to-one relationship between a walk and a difficulty
        4. public Region Region { get; set; } is a navigation property and defines a one-to-one relationship between a walk and a region.
     2. To help grasp what is happening from the SQL standpoint, here is how you would reproduce these table in SMS:

Setting up the Same Table Key Relationship Manually in SQL SMS

|  |
| --- |
| First, Create the Walks table with a primary key CREATE TABLE Walks(Id UniqueIdentifier,Name nvarchar(50),Description nvarchar(100),LengthInKm float, WalkImageUrl nvarchar(255),DifficultyId UniqueIdentifier, RegionId UniqueIdentifier, CONSTRAINT PK\_Walks PRIMARY KEY NONCLUSTERED (Id))  Secondly, create the Difficulties table with the Primary key CREATE TABLE Difficulties(Id UniqueIdentifier, Name nvarchar(100),  CONSTRAINT PK\_Difficulties PRIMARY KEY NONCLUSTERED (Id))  Thirdly, create the Regions table with the Primary key CREATE TABLE Regions(Id UniqueIdentifier,Code NVARCHAR(100),Name NVARCHAR(100),RegionImageUrl NVARCHAR(255),  CONSTRAINT PK\_Regions PRIMARY KEY NONCLUSTERED (Id))  Lastly, alter the Walks table and add the Foreign keys pointing back to the Difficulties and Regions table ALTER TABLE Walks ADD CONSTRAINT FK\_Walks\_Regions\_RegionId FOREIGN Key (RegionId) REFERENCES Regions(Id)  ALTER TABLE Walks ADD CONSTRAINT FK\_Walks\_Difficulties\_DifficultyId FOREIGN Key (DifficultyId) REFERENCES Difficulties(Id) |

* + 1. So you can see there is a Primary Key/Foreign Key relationship between the Walks table and the Difficulties/Regions tables.
       1. The RegionId field in the Walks table has a Foreign/Primary key relationship with the Id field in the Regions table.
       2. The DifficultyId field in the Walks table has a Foreign/Primary key relationship with the Id field in the Difficulties table.

1. [Creating the DBContext Class](#TS03)
   1. The DbContext class has these attributes:
      1. Maintaining connection to the database
      2. Track changes
      3. Perform CRUD operations
      4. Bridge between domain models and the database  
           
         DbContext Provides a separation between the Controllers and the Database

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Controller |  | DbContext |  | Database |

* + 1. The DbContext class will be named NZWalksDbContext.cs and will look like this:

The NZWalksDbContext.cs Class

|  |
| --- |
| using Microsoft.EntityFrameworkCore;  using NZWalks.API.Controllers.Models.Domain;  namespace NZWalks.API.Controllers.Data  {  public class NZWalksDbContext : DbContext  {  public NZWalksDbContext(DbContextOptions<NZWalksDbContext> dbContextOptions) : base(dbContextOptions)  { }  public DbSet<Difficulty> Difficulties { get; set; }  public DbSet<Region> Regions { get; set; }  public DbSet<Walk> Walks { get; set; }  }  } |

1. [Adding Connection to appsettings.json](#TS04)
   1. This connection string allows Entity Framework to connect to the database.
   2. To create this, follow these steps.
      1. Open the appsettings.json file.
      2. Add the connection string. The new syntax for the connection string will be in blue:

The appsettings.json File

|  |
| --- |
| {  "Logging": {  "LogLevel": {  "Default": "Information",  "Microsoft.AspNetCore": "Warning"  }  },  "AllowedHosts": "\*",  "ConnectionStrings": {  "NZWalksConnectionString": "Server=DAVES\_PC;Database=NZWalksDb;TrustServerCertificate=true;Trusted\_Connection=true;"  }  } |

1. [Dependency Injection](#TS05)
   1. These are the attributes of Dependency Injection
      1. Design pattern to increase maintainability and testability
      2. Dependency Injection is built into ASP.NET core
      3. Dependency Injection is responsible for creating and maintaining instances
   2. Injecting DbContext into our Application
      1. The first thing we need to do is ensure we have installed the SQL dependency package for SQLServer.
      2. Install these two package if it is not already installed:
         1. Microsoft.EntityFrameworkCore.SqlServer
         2. Microsoft.EntityFrameworkCore.Tools
      3. Next, update the Program.cs file by adding this line just above the var app = builder.Build(); line:
         1. builder.Services.AddDbContext<NZWalksDbContext>(options => options.UseSqlServer(builder.Configuration.GetConnectionString("NZWalksConnectionString")));
   3. Now that we have done this, we are ready to run EF Core Migrations.

1. [Running Entity Framework Core Migrations](#TS06)
   1. This will create the tables we need.
   2. To set up EF Core Migrations, follow these steps:
      1. In the top menu line, click on Tools, NuGet Package Manager and Package Manager Console.
      2. In the bottom console, run these commands:
         1. Add-Migration “Initial Migration”
            1. Press enter to create the migration.
            2. This will create the migration plan but it will not execute it.
            3. If you look into the migration plan, you will see it is creating DbSets.

DbSets are the equivalent to tables in SQL, transposed into objects in EF. They are piers to SQL tables.

* + - 1. Update-Database
         1. This will run the migration plan.
         2. You will see several lines scroll up in the Package Manager console window as it executes each command.
         3. Now if you go to SMS, you will see the new **NZWalksDB** database along with these tables:

Difficulties

Regions

Walks

1. [Creating New Controllers](#TS07)
   1. Now we are going to create a new controller.
   2. This will be an entry point into the application which is listening on localhost at the port set up in the launchingSettings.json file:

LaunchSettings.Json file

|  |
| --- |
| "profiles": {  "http": {  "commandName": "Project",  "dotnetRunMessages": true,  "launchBrowser": true,  "launchUrl": "swagger",  "applicationUrl": **"http://localhost:5102",**  "environmentVariables": {  "ASPNETCORE\_ENVIRONMENT": "Development"  }  },  "https": {  "commandName": "Project",  "dotnetRunMessages": true,  "launchBrowser": true,  "launchUrl": "swagger",  "applicationUrl": **"https://localhost:7257;http://localhost:5102",**  "environmentVariables": {  "ASPNETCORE\_ENVIRONMENT": "Development"  } |

* 1. To create the new controller, follow these steps:
     1. Right-Click on the controllers folder in the Solution Explorer window
     2. Select Add and then Controller.
     3. In the New Scaffolded Item window, Click on API on the left and then API Controller – Empty in the middle pane.
     4. Name the controller RegionsController.cs.
     5. Now enter this code into the RegionsController.cs file:

RegionsController.cs File

|  |
| --- |
| using Microsoft.AspNetCore.Http;  using Microsoft.AspNetCore.Mvc;  using NZWalks.API.Controllers.Models.Domain;  namespace NZWalks.API.Controllers  {  // Routes to https://localhost:xxxx/api/Regions  **[Route("api/[controller]")]**  **[ApiController]**  public class RegionsController : ControllerBase  {  [HttpGet]  public IActionResult GetAll()  {  var regions = new List<Region>  {  new Region  {  Id = Guid.NewGuid(),  Name = "Auckland Region",  Code = "AKL",  RegionImageUrl = "xxxxx"  },  new Region  {  Id = Guid.NewGuid(),  Name = "Wellington Region",  Code = "WLG",  RegionImageUrl = "xxxxx"  }  };  return Ok(regions);  }  }  } |

1. [Connecting our Controllers to DbContext](#TS08)
   1. Now we will replace the code of displaying data via a list with Dependency Injection of DbContext as shown below
      1. The URL for this Regions method is <https://localhost:xxxx/api/Regions>
      2. This is how it is broken down:
         1. <https://localhost:xxxx> comes from the fact the application is listening on port 5182 on my laptop.
         2. /api comes from the Route command: [Route("api/**[controller]**")]
         3. /Regions comes from the method: public class RegionsController : ControllerBase
            1. The word controller is striped off just leaving Regions.

|  |
| --- |
| using Microsoft.AspNetCore.Mvc;  using NZWalks.API.Controllers.Data;  using NZWalks.API.Controllers.Models.Domain;  namespace NZWalks.API.Controllers  {  // https://localhost:5182/api/Regions  [Route("api/**[controller]**")]  [ApiController]  public class RegionsController : ControllerBase  {  // Dependency Injection of DBContext into the class  private readonly NZWalksDbContext dbContext;  public RegionsController(NZWalksDbContext dbContext)  {  this.dbContext = dbContext;  }  [HttpGet]  public IActionResult GetAll()  {  var regions = dbContext.Regions.ToList();  return Ok(regions);  }  }  } |

* 1. Here we inject NZWalksDbContext from the Program.cs class and inject it as a property of the RegionsController class.
  2. DbContext has a built-in method to connect to a SQL table through the DbSet and convert it to a list.
  3. Analyzing the line var regions = dbContext.Regions.ToList();
     1. dbContext is using the DbSet Regions to communicate directly with the SQL table Regions.
     2. In the line var regions = dbContext.Regions.ToList(); Regions is not referring to the SQL table Regions, but rather to the DbSet Regions.
        1. The DbSet Regions has a direct liaison to the SQL table Regions, they are piers and communicate directly with each other.
        2. There is no way C# could read rad SQL table data. That is what DbSets do, they ingest SQL data and reformat it so Class objects can store it.
     3. Finally the regions variable (Which is a listing of the Regions Class) is given the memory address of where this data is stored in memory.
        1. The return line transmits that data stored in memory in a streaming fashion to the users laptop, and more specifically their browser where it is displayed.

1. [The ID Action Method](#TS09)
   1. There are times we may not want all the records, but just one.
   2. We will need a key field that we can execute a search on.
   3. In the case of our NZWalks database, we have a Walks table which has an Id field which is of type Guid.
      1. The method we will be using there will be called GetById.
   4. In SQL terms, we would form our search something like this:
      1. Select \* from Walks where Id = @Id;
   5. How we search for the Id field in our Method however is to supply a Route command which routes the search Guid from the URL itself into the GetById method.
   6. Below is the GetById method that we will talk about:

The GetById Method

|  |
| --- |
| [HttpGet]  [Route("{id:Guid}")]  public IActionResult GetBtId([FromRoute] Guid id)  {  var region = dbContext.Regions.FirstOrDefault(x => x.Id == id);  if(region == null)  return NotFound();  return Ok(region);  } |

* 1. We can see by the Http annotation that this is a Get Request ( [HttpGet].
  2. We can also see that it will be routing the Id in the Route Statement: [Route("{id:Guid}")] to the FromRoute statement in the Method line: public IActionResult GetBtId([FromRoute] Guid id).
     1. Here is an example of what the URL would look like when calling this method:
        1. https://localhost:7257/api/Regions/d7edf702-fee9-4df4-a6ec-11a93f1b37b6
     2. Whatever value that is assigned to the id variable in the route statement (Highlighted in blue above) is inserted in the id variable in the method line.
     3. Now we have our syntax for the where statement if this were an SQL select statement.
  3. Finally the dbContext line executes the FirstOrDefault method to look for the id match.
     1. In FirstOrDefault, X will sift through all the tables, picking the first match it finds.
     2. If no result was found, the NotFound() statement will be returned.
     3. If the entry in the Regions table was found, a 200 response will be returned by the Ok() method along with the region variable.

1. [DTO and Domain Model](#TS10)
   1. Data Transfer Objects (DTO) is an interface to the client that provides a layer of separation between the client and the actual data.
      1. Its purpose is for security so attacks to your application will be thwarted as the DTO objects prevent access to the data.
   2. Domain Models are tightly coupled to the database and provides a mean of pulling data via the DbSets.
      1. The DbSets normally have the same naming standards as the Database tables themselves to avoid confusion.
   3. Below is a diagram that displays the relationship of the API to the client and the data using DTOs and Domain Models
   4. Advantages of DTOs
      1. Separation of Concerns
         1. Limits the amount of data sent to the client instead of just sending all of the table data
      2. Performance
      3. Security
         1. Limits the amount of data exposed to the client such as PPH/HIPPA data.
      4. Versioning
         1. We can update the Domain Model and DTO versioning separately.

The DTO/Domain Model Architecture

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Client |  | DTO | API | Domain Model |  | Database |

1. [Changing Methods to Use DTOs](#TS11)
   1. In our examples of HttpGet, we will change the methods in incorporate DTOs.
   2. Follow these steps to accomplish this:
      1. Right-Click on the Models folder and add a folder called DTO
      2. Inside the DTO folder, create a new class called RegionDto.cs
      3. The body of the RegionDto.cs file will be identical to that of the Region.cs file as shown below

Comparison of the Region.cs file to the RegionDto.cs file

|  |  |
| --- | --- |
| Region.cs File | RegionDto.cs File |
| namespace NZWalks.API.Controllers.Models.Domain  {  public class Region  {  public Guid Id { get; set; }  public string Code { get; set; }  public string Name { get; set; }  public string? RegionImageUrl { get; set; }  }  } | namespace NZWalks.API.Controllers.Models.DTO  {  public class RegionDto  {  public Guid Id { get; set; }  public string Code { get; set; }  public string Name { get; set; }  public string? RegionImageUrl { get; set; }  }  } |

* + 1. This is where we create the separation between the database and the client by transferring the individual data from the Domain Model to the DTO

|  |
| --- |
| [HttpGet]  public IActionResult GetAll()  {  // Get data directly from database via DbSets      // Create a new regionsDto object to transfer the DM data from.  /\*  When pulling in all the rows in the database table, we will need to store several objects.  We can use either IEnumerable or a List to store them.  It really doesn't matter to the ForEach loop since ForEach can process either one simply fine.  Here are the examples of using both IEnumerable and ToList below:  var regionsDomain = dbContext.Regions.ToList();  IEnumerable<Region> regionsDomain = dbContext.Regions;  Again, both will load up the variable *regionsDomain* with all the database table data.  Below we will use the ToList() method, but we could just as easily have used IEnumerable. \*/  var regionsDomain = dbContext.Regions.ToList();  // Create the mew DTO object *regionsDto* which will ultimate contain all the data that *regionsDomain* contains now.  var regionsDto = new List<RegionDto>();    /\*  Mapping the Domain Model to the DTO  -----------------------------------  Loop through each row in the database table and  map the individual columns from the DM to the DTO. \*/  foreach (var regionDomain in regionsDomain)  {  regionsDto.Add(new RegionDto()  {  Id = regionDomain.Id,  Code = regionDomain.Code,  Name = regionDomain.Name,  RegionImageUrl = regionDomain.RegionImageUrl  });  }  // Now we return the DTO to the user instead of the Domain Model.  return Ok(regionsDto);  } |

* + 1. Now we can apply our DTO transfer to just one record as well in the GetById method

|  |
| --- |
| [HttpGet]  [Route("{id:Guid}")]  public IActionResult GetById([FromRoute] Guid id)  {  var regionDomain = dbContext.Regions.FirstOrDefault(x => x.Id == id);  if(regionDomain == null)  {  return NotFound();  }  else  {  var regionsDto = new RegionDto  {  Id = regionDomain.Id,  Code = regionDomain.Code,  Name = regionDomain.Name,  RegionImageUrl = regionDomain.RegionImageUrl  };  return Ok(regionsDto);  }  } |

1. [Creating a New Table Entry](#TS12)
   1. When entering a new region entry, we will be performing an HttpPost method.
      1. Since we will be accepting information from the user, this information will be embedded into the body of the request.
      2. For that reason, we use the [FromBody] annotation in the method list to tell the API to look in the body for the new record information.
   2. This method name will be Create()
   3. First, let’s look at the Region class. If has four properties:
      1. Id
      2. Code
      3. Name
      4. RegionImageUrl
   4. We will be receiving 3 pieces of information from the client for this add request, those being:
      1. Code
      2. Name
      3. RegionImageUrl
   5. The Id value will be generated automatically by the API so the client will not be supplying it.
   6. To set up the new add request, follow these steps:
      1. In the DTO folder, create a new class called AddRegionRequestDto.cs.
      2. Populate the file so it looks like this:

The AddRegionRequestDto.cs File

|  |
| --- |
| namespace NZWalks.API.Controllers.Models.DTO  {  public class AddRegionRequestDto  {  public string Code { get; set; }  public string Name { get; set; }  public string? RegionImageUrl { get; set; }  }  } |

* + 1. Now we will create a new method in the RegionsController.cs file to add a new record into the Regions SQL table.
    2. Below is the code for this new method named Create().

The Create() Method

|  |
| --- |
| [HttpPost]  public IActionResult Create([FromBody] AddRegionRequestDto addRegionRequestDto)  {  var regionDomainModel = new Region()  {  Code = addRegionRequestDto.Code,  Name = addRegionRequestDto.Name,  RegionImageUrl = addRegionRequestDto.RegionImageUrl  };  dbContext.Regions.Add(regionDomainModel);  dbContext.SaveChanges();  var regionDto = new Region  {  Id = regionDomainModel.Id,  Code = regionDomainModel.Code,  Name = regionDomainModel.Name,  RegionImageUrl = regionDomainModel.RegionImageUrl  };  return CreatedAtAction(nameof(GetById), new { id = regionDto.Id }, regionDto);  } |

* + 1. Analyzing what is happening in the method above, it is basically broken down into two areas:  
       1. Writing the user input to the SQL table:

|  |
| --- |
| var regionDomainModel = new Region()  {  Code = addRegionRequestDto.Code,  Name = addRegionRequestDto.Name,  RegionImageUrl = addRegionRequestDto.RegionImageUrl  };  dbContext.Regions.Add(regionDomainModel);  dbContext.SaveChanges(); |

* + - * 1. The Add() function of dbContext auto generates the Id and creates the commands for the SQL insert. It, however, does NOT perform the write.
        2. The SaveChanges() method is what actually sends to write request to the Regions DbSet which in turn sends the insert request to SQL.
      1. Returning the processed request to the user:

|  |
| --- |
| var regionDto = new Region  {  Id = regionDomainModel.Id,  Code = regionDomainModel.Code,  Name = regionDomainModel.Name,  RegionImageUrl = regionDomainModel.RegionImageUrl  };  return CreatedAtAction(nameof(GetById), new { id = regionDto.Id }, regionDto |

* + - * 1. The CreatedAtAction() method leverages the GetById() method to send the id to it so GetById() will return the newly added record.
        2. Note that the return code will come back as 201, not 200 for a successful insert.

1. [Updating a Table Item](#TS13)
   1. When updating an existing region entry, we will be performing an HttpPut method.
      1. In the GetById method, we needed the [FromRoute] annotation.
      2. In the Update() method, we needed the ([FromBody] annotation.
      3. With the Update() method, we will need both, one to request the specific record and the other to replace the existing record data with the user supplied changes.
   2. This method name will be Create()
   3. First, let’s again look at the Region class. If has four properties:
      1. Id
      2. Code
      3. Name
      4. RegionImageUrl
   4. For this operation, we will need another DTO class. We will put it in the DTO folder and call this class UpdateRegionRequestDto.cs.
      1. Now you have to determine what information you want the client to update. Obviously, we do not want them to update the Guid
      2. The question remains, what fields within the Region table *do you* want them up update? This will be a business requirements question
   5. For this example we will allow them to update the Code, Name and RegionImageUrl.
      1. If we didn’t want the client to update the code, we would not include the Code line.
   6. To set up the new add request, follow these steps:
      1. In the DTO folder, create a new class called UpdateRegionRequestDto.cs.
      2. Populate the file so it looks like this

The UpdateRegionRequestDto.cs File

|  |
| --- |
| namespace NZWalks.API.Controllers.Models.DTO  {  public class UpdateRegionRequestDto  {  public string Code { get; set; }  public string Name { get; set; }  public string? RegionImageUrl { get; set; }  }  } |

* + 1. Now we will create a new method in the RegionsController.cs file to edit a record into the Regions SQL table.
    2. Below is the code for this new method named Update().

The Update() Method

|  |
| --- |
| [HttpPut]  [Route("{id:Guid}")]  public IActionResult Update([FromRoute] Guid id, [FromBody] UpdateRegionRequestDto updateRegionRequestDto)  {  var regionDomainModel = dbContext.Regions.FirstOrDefault(x => x.Id == id);  if (regionDomainModel == null)  return NotFound();  regionDomainModel.Code = updateRegionRequestDto.Code;  regionDomainModel.Name = updateRegionRequestDto.Name;  regionDomainModel.RegionImageUrl = updateRegionRequestDto.RegionImageUrl;  dbContext.SaveChanges();  var regionDto = new RegionDto()  {  Id = regionDomainModel.Id,  Code = regionDomainModel.Code,  Name = regionDomainModel.Name,  RegionImageUrl = regionDomainModel.RegionImageUrl  };  return Ok(regionDto);  } |

* + 1. Analyzing what is happening in the method above, it is basically broken down into three areas:  
       1. Checking to see if the record exist in the database and return a 404 Not Found error if it is not found:

|  |
| --- |
| var regionDomainModel = dbContext.Regions.FirstOrDefault(x => x.Id == id);  if (regionDomainModel == null)  return NotFound(); |

* + - 1. Transferring the user inputs from the updateRegionRequestDto object to overwrite the existing values in the regionDomainModel object.
         1. The new values are then saved to the SQL table:

|  |
| --- |
| regionDomainModel.Code = updateRegionRequestDto.Code;  regionDomainModel.Name = updateRegionRequestDto.Name;  regionDomainModel.RegionImageUrl = updateRegionRequestDto.RegionImageUrl;  dbContext.SaveChanges(); |

* + - 1. Transfer the values from the regionDomainModel object to the regionDto object for display in the browser:

|  |
| --- |
| var regionDto = new RegionDto()  {  Id = regionDomainModel.Id,  Code = regionDomainModel.Code,  Name = regionDomainModel.Name,  RegionImageUrl = regionDomainModel.RegionImageUrl  };  return Ok(regionDto); |

* + - 1. A 200 response will be return upon successful update of the SQL record.

1. [Deleting a Table Item](#TS14)
   1. When deleting an existing record, we will be performing an HttpDelete method.
   2. In the GetById method, we needed the [FromRoute] annotation
   3. Unlike the other methods, we will not be needing a DeleteRegionRequestDto.cs class since we are not adding or updating anything.
   4. Below is the code for the Delete() Method

The Delete() Method

|  |
| --- |
| [HttpDelete]  [Route("{id:Guid}")]  public IActionResult Get([FromRoute] Guid id)  {  var regionDomainModel = dbContext.Regions.FirstOrDefault(x => x.Id == id);  if (regionDomainModel == null)  return NotFound();  dbContext.Regions.Remove(regionDomainModel);  dbContext.SaveChanges();  var regionDto = new RegionDto()  {  Id = regionDomainModel.Id,  Code = regionDomainModel.Code,  Name = regionDomainModel.Name,  RegionImageUrl = regionDomainModel.RegionImageUrl  };  return Ok();  } |

* 1. Analyzing what is happening in the method above, it is basically broken down into three areas:  
     1. Searching for a matching record:

|  |
| --- |
| var regionDomainModel = dbContext.Regions.FirstOrDefault(x => x.Id == id);  if (regionDomainModel == null)  return NotFound(); |

* + 1. Deleting the record:

|  |
| --- |
| dbContext.Regions.Remove(regionDomainModel);  dbContext.SaveChanges(); |

* + 1. Returning the deleted record to the browser window:

|  |
| --- |
| var regionDto = new RegionDto()  {  Id = regionDomainModel.Id,  Code = regionDomainModel.Code,  Name = regionDomainModel.Name,  RegionImageUrl = regionDomainModel.RegionImageUrl  };  return Ok(); |

* 1. A 200 response will be return upon successful deletion of the record.

1. [Asynchronous Programming](#TS15)
   1. This refers to the ability for the API to submit a request and not have to wait for the process to finish.
   2. This leads to faster performance.
   3. Asynchronous methods require the Entity Framework Code package:
      1. using Microsoft.EntityFrameworkCore;
   4. Asynchronous programming uses the Async and Await keywords.
      1. It is actually quite easy to convert from Sync to Async programming by making these two changes:
         1. Surround IActionResult with async Task<IActionResult>
         2. Preceding the dbContext command with await dbContext
         3. Replacing the action words in the dbContext line with the async equivalent:
            1. ToList() to ToListAsync()
            2. Add() to AddAsync()
            3. FirstorDefault() to FirstOrDefaultAsync()
            4. SaveChanges() to SaveChangesAsync()
   5. Converting our CRUD operations to Async
      1. Converting the GetAll() method to Asynchronous

|  |
| --- |
| [HttpGet]  public **async Task<**IActionResult> GetAll()  {  var regionsDomain = **await** dbContext.Regions.ToListAsync();  var regionsDto = new List<RegionDto>();  foreach (var regionDomain in regionsDomain)  {  regionsDto.Add(new RegionDto()  {  Id = regionDomain.Id,  Code = regionDomain.Code,  Name = regionDomain.Name,  RegionImageUrl = regionDomain.RegionImageUrl  });  }  return Ok(regionsDto);  } |

* + 1. The changes made to convert the GetAll() method to async are in bold and underlined in the script above.
       1. We use async Task<> in the method line.
       2. We use await in the dbContext line.
  1. Now we can just follow suit on the remainder of the CRUD method to convert them to async as well.

1. [Repository Pattern and Benefits](#TS16)
   1. What is the Repository Pattern
      1. Design pattern to separate the data access layer from the application.
      2. Provides interface without exposing implementation
      3. Helps create abstraction
   2. Benefits of Repository Pattern
      1. Decoupling
      2. Consistency
      3. Performance
      4. Multiple data sources (Switching)
   3. Switching data access to the Interface
      1. Within our application, all our methods (GetAll, GetById, Create, Update and Delete) will be executed through the IRegionRepository.cs interface instead of directly by the RegionsController.cs file.
      2. There is a second class we will create called SQLRegionRepository.cs
         1. This is the class that will actually be doing the data loading and saving work.
         2. The RegionsController will ask IRegionRepository to get data, and IRegionRepository will engage whichever concrete class it is bounds to in the Scoped Services entry in Program.cs
         3. We will discuss modifying Scoped Services within Program.cs a few lines down in in section d.
      3. So what we have here is your typical Concrete class (SQLRegionRepository) being governed by the interface class IRegionRepository.cs.
         1. The Regions Controller basically says, ***Hey IRegionRepository, execute your GetAll() method and send me back some data. I don’t know where the data is coming from, I will just accept it as it is***.
         2. The Regions Controller is oblivious to the data source (SQL, Oracle, MySQL) and it doesn’t care, it’s just getting data back from the IRegionRepository without knowledge as to the data source.
      4. Now we can control the data source via the repository and the Regions Controller neither know nor cares about the data source, it just wants some data,.
      5. Essentially what the Regions Controller is doing is outsourcing access to the data to the interface.
   4. Implementing the Repository Pattern into our application
      1. Here is a summary of what we need to do implement the Repository and then we will discuss the details for each area:
         1. Create a folder called Repositories
         2. Create an Interface called IRegionRepository.cs
         3. Create a class called SQLRegionRepository.cs
         4. Update the Program.cs file to add RegionRepository and SQLRegionRepository into its scope.
         5. Update the Region Controller to call the Region Repository to execute a method to bring back data.
      2. Now we will discuss the detailed steps:
         1. Create a folder named Repositories
            1. This folder will be right off the API itself just likes the Controllers and Data folder are.
         2. Create an Interface called IRegionRepository.cs
            1. Next, click on add, add a class, but this time choose *Repository* instead of class.
            2. Name the new Interface IRegionRepository.cs and insert the following text:

The IRegionRepository.cs File

|  |
| --- |
| using NZWalks.API.Models.Domain;  namespace NZWalks.API.Repositories  {  public interface IRegionRepository  {  Task<List<Region>> GetAllAsync();  }  } |

* + - 1. Create a class called SQLRegionRepository.cs
         1. This is the concrete class that actually performs the work

The SQLRegionRepository.cs File

|  |
| --- |
| using Microsoft.EntityFrameworkCore;  using NZWalks.API.Data;  using NZWalks.API.Models.Domain;  namespace NZWalks.API.Repositories  {  public class SQLRegionRepository : IRegionRepository  {  private readonly NZWalksDbContext dbContext;  public SQLRegionRepository(NZWalksDbContext dbContext)  {  this.dbContext = dbContext;  }  public async Task<List<Region>> GetAllAsync()  {  return await dbContext.Regions.ToListAsync();  }  }  } |

* + - 1. Update the Program.cs file to add RegionRepository and SQLRegionRepository into its scope
         1. We need to include the line just above the var app builder.Build(); line in Program.cs:

builder.Services.AddScoped<IRegionRepository, SQLRegionRepository>();

* + - * 1. This line tells the IRegionRepository to use the SQLRegionRepository as its concrete class.
      1. Update the Region Controller to call the Region Repository to execute a method to bring back data
         1. There are a couple changes that need to be made in the RegionsController.cs file.

First we need to create a private variable instantiated after the NZWalksDbContext class

Then we need to make this variable an instance of the class

Finally we need to make the responsibility of the concrete class SQLRegionRepository.cs to extract the data.

Changes Made to RegionsController.cs to Make SQLRegionRepository Extract All Users

|  |
| --- |
| [Route("api/**[controller]**")]  [ApiController]  public class RegionsController : ControllerBase  {  // Dependency Injection of DBContext into the class  private readonly NZWalksDbContext dbContext;  **private readonly IRegionRepository regionRepository;**  public RegionsController(NZWalksDbContext dbContext, **IRegionRepository regionRepository**)  {  this.dbContext = dbContext;  **this.regionRepository = regionRepository**;  }  [HttpGet]  public async Task<IActionResult> GetAll()  {  **var regionsDomain = await regionRepository.GetAllAsync();**  var regionsDto = new List<RegionDto>();  foreach (var regionDomain in regionsDomain)  {  regionsDto.Add(new RegionDto()  {  Id = regionDomain.Id,  Code = regionDomain.Code,  Name = regionDomain.Name,  RegionImageUrl = regionDomain.RegionImageUrl  });  }  return Ok(regionsDto);  } |

* + - * 1. As we can see, the lines above in bold and underlined are the changes to the RegionController.cs file to make the GetAll() class use our Repository Pattern.
  1. Now let’s see how easy it is to implement another data source.
     1. In the Repository folder, we will make another repository class called InMemoryRegionRepository.
     2. This repository will create a manual List of items as shown below:

The InMemoryRegionRepository.cs file

|  |
| --- |
| using NZWalks.API.Models.Domain;  namespace NZWalks.API.Repositories  {  public class InMemmoryRegionRepository : IRegionRepository  {  public async Task<List<Region>> GetAllAsync()  {  return new List<Region>()  {  new Region()  {  Id = Guid.NewGuid(),  Code = "AKL",  Name = "Auckland",  RegionImageUrl = "Auckland Pictures"  },  new Region()  {  Id = Guid.NewGuid(),  Code = "NTL",  Name = "Northland",  RegionImageUrl = "Northland Pictures"  },  new Region()  {  Id = Guid.NewGuid(),  Code = "BOP",  Name = "Bay Of Plenty",  RegionImageUrl = "Bay Of Plenty Pictures"  },  new Region()  {  Id = Guid.NewGuid(),  Code = "WGN",  Name = "Wellington",  RegionImageUrl = "Wellington Pictures"  },  new Region()  {  Id = Guid.NewGuid(),  Code = "NSN",  Name = "Nelson",  RegionImageUrl = "Nelson Pictures"  },  new Region()  {  Id = Guid.NewGuid(),  Code = "STL",  Name = "Southland",  RegionImageUrl = "Southland Pictures"  }  };  }  }  } |

* + 1. Note that at this point the SQLRegionRepository.cs class and InMemoryRegionRepository.cs class are peers.
       1. They both are resources to point at two different data sources.
       2. We could just as easily spin up MySql, create the same data in MySql and create a third class called MySqlRegionRepository.cs.
    2. To use the InMemoryRegionRepository.cs repository file instead of the SQLRegionRepository.cs file, we simply need to make this change in the Program.cs file:
       1. Remove: builder.Services.AddScoped<IRegionRepository, SQLRegionRepository>();
       2. Add: builder.Services.AddScoped<IRegionRepository, InMemmoryRegionRepository>();
    3. This tells the IRegionRepository interface to point to InMemoryRegionRepository as its concrete working class.
    4. Wala! The RegionsController has absolutely no idea we are using a new data source, nor does it care.
    5. This is what it means by Dependency Injection. We are injecting a new data source in the Program.cs file and it works seamlessly.
  1. Finally, let see what the IRegionRepository, SQLRegionRepository and RegionsController look like now that they are using our new Repository Patterns:

The RegionsController.cs File

|  |
| --- |
| using Microsoft.AspNetCore.Mvc;  using NZWalks.API.Models.Domain;  using NZWalks.API.Models.DTO;  using NZWalks.API.Data;  using NZWalks.API.Repositories;  namespace NZWalks.API.Controllers  {  // https://localhost:xxxx/api/Regions  [Route("api/[controller]")]  [ApiController]  public class RegionsController : ControllerBase  {  // Dependency Injection of DBContext into the class  private readonly NZWalksDbContext dbContext;  private readonly IRegionRepository regionRepository;  public RegionsController(NZWalksDbContext dbContext, IRegionRepository regionRepository)  {  this.dbContext = dbContext;  this.regionRepository = regionRepository;  }  [HttpGet]  public async Task<IActionResult> GetAllAsync()  {  var regionsDomain = await regionRepository.GetAllAsync();  var regionsDto = new List<RegionDto>();  foreach (var regionDomain in regionsDomain)  {  regionsDto.Add(new RegionDto()  {  Id = regionDomain.Id,  Code = regionDomain.Code,  Name = regionDomain.Name,  RegionImageUrl = regionDomain.RegionImageUrl  });  }  return Ok(regionsDto);  }  [HttpGet]  [Route("{id:Guid}")]  public async Task<IActionResult> GetById([FromRoute] Guid id)  {  var regionDomain = await regionRepository.GetByIdAsync(id);  if (regionDomain == null)  {  return NotFound();  }  else  {  var regionsDto = new RegionDto  {  Id = regionDomain.Id,  Code = regionDomain.Code,  Name = regionDomain.Name,  RegionImageUrl = regionDomain.RegionImageUrl  };  return Ok(regionsDto);  }  }  [HttpPost]  public async Task<IActionResult> Create([FromBody] AddRegionRequestDto addRegionRequestDto)  {  var regionDomainModel = new Region()  {  Code = addRegionRequestDto.Code,  Name = addRegionRequestDto.Name,  RegionImageUrl = addRegionRequestDto.RegionImageUrl  };  await regionRepository.CreateAsync(regionDomainModel);  var regionDto = new Region  {  Id = regionDomainModel.Id,  Code = regionDomainModel.Code,  Name = regionDomainModel.Name,  RegionImageUrl = regionDomainModel.RegionImageUrl  };  return CreatedAtAction(nameof(GetById), new { id = regionDto.Id }, regionDto);  }  [HttpPut]  [Route("{id:Guid}")]  public async Task<IActionResult> Update([FromRoute] Guid id, [FromBody] UpdateRegionRequestDto updateRegionRequestDto)  {  // Map DTO to Domain Model  var regionDomainModel = new Region()  {  Code = updateRegionRequestDto.Code,  Name = updateRegionRequestDto.Name,  RegionImageUrl = updateRegionRequestDto.RegionImageUrl  };  regionDomainModel = await regionRepository.UpdateAsync(id, regionDomainModel);  if (regionDomainModel == null)  return NotFound();  var regionDto = new RegionDto()  {  Id = regionDomainModel.Id,  Code = regionDomainModel.Code,  Name = regionDomainModel.Name,  RegionImageUrl = regionDomainModel.RegionImageUrl  };  return Ok(regionDto);  }  [HttpDelete]  [Route("{id:guid}")]  public async Task<IActionResult> Get([FromRoute] Guid id)  {  var regionDomainModel = await regionRepository.DeleteAsync(id);  if (regionDomainModel == null)  return NotFound();  var regionDto = new RegionDto()  {  Id = regionDomainModel.Id,  Code = regionDomainModel.Code,  Name = regionDomainModel.Name,  RegionImageUrl = regionDomainModel.RegionImageUrl  };  return Ok(regionDto);  }  }  } |

The IRegionRepository.cs File

|  |
| --- |
| using NZWalks.API.Models.Domain;  namespace NZWalks.API.Repositories  {  public interface IRegionRepository  {  Task<List<Region>> GetAllAsync();  Task<Region> GetByIdAsync(Guid id);  Task<Region> CreateAsync(Region region);  Task<Region?> UpdateAsync(Guid id, Region region);  Task<Region?> DeleteAsync(Guid id);  }  } |

The SQLRegionRepository.cs File

|  |
| --- |
| using Microsoft.EntityFrameworkCore;  using NZWalks.API.Data;  using NZWalks.API.Models.Domain;  namespace NZWalks.API.Repositories  {  public class SQLRegionRepository : IRegionRepository  {  private readonly NZWalksDbContext dbContext;  public SQLRegionRepository(NZWalksDbContext dbContext)  {  this.dbContext = dbContext;  }  public async Task<List<Region>> GetAllAsync()  {  return await dbContext.Regions.ToListAsync();  }  public async Task<Region?> GetByIdAsync(Guid id)  {  return await dbContext.Regions.FirstOrDefaultAsync(x => x.Id == id);  }  public async Task<Region> CreateAsync(Region region)  {  await dbContext.Regions.AddAsync(region);  await dbContext.SaveChangesAsync();  return region;  }  public async Task<Region?> UpdateAsync(Guid id, Region region)  {  var existingRegion = await dbContext.Regions.FirstOrDefaultAsync(x => x.Id == id);  if (existingRegion == null)  return null;  existingRegion.Code = region.Code;  existingRegion.Name = region.Name;  existingRegion.RegionImageUrl = region.RegionImageUrl;  await dbContext.SaveChangesAsync();  return existingRegion;  }  public async Task<Region?> DeleteAsync(Guid id)  {  var existingRegion = await dbContext.Regions.FirstOrDefaultAsync(x => x.Id == id);  if(existingRegion == null)  return null;  dbContext.Regions.Remove(existingRegion);  await dbContext.SaveChangesAsync();  return existingRegion;  }  }  } |

1. [AutoMapper](#TS17)
   1. AutoMapper is a useful utility that will map column by column the properties of one class to another.
   2. To set up AutoMapper, two things must be done:
      1. Inject AutoMapper services into the Program.cs file
      2. Create an AutoMapper class extending the AutoMapper profile.
      3. Inject AutoMapper into the RegionsController file and map the class fields
   3. Below are the detail of these two items
      1. Inject AutoMapper services into the Program.cs file
         1. Just above the var app = builder.Build(); line, enter this line:
            1. builder.Services.AddAutoMapper(cfg => { cfg.AddProfile(new AutoMapperProfiles()); });
      2. Create an AutoMapper class extending the AutoMapper profile
         1. In the API, create a folder named Mappings
         2. Inside this folder, create a class file names AutoMapperProfiles.cs
         3. Place this syntax in the AutoMapperProfiles.cs file:

The AutoMapperProfiles.cs File

|  |
| --- |
| using AutoMapper;  using NZWalks.API.Models.Domain;  using NZWalks.API.Models.DTO;  namespace NZWalks.API.Mappings  {  public class AutoMapperProfiles : Profile  {  public AutoMapperProfiles()  {  CreateMap<Region, RegionDto>().ReverseMap();  }  }  } |

* + 1. Inject AutoMapper into the RegionsController file and map the class fields
       1. Now we inject AutoMapper as an instance variable and use it to replace the fields mappings.
       2. In the two examples below, the GetAllAsync returns a list while GetById is returning just one object.
          1. AutoMapper is smart enough to know when to incorporate a list statement and when not to.
          2. For that reason we can use the exact same mapper statement for both methods.
       3. The new code is shown below:

Using AutoMapper in the RegionsController.cs File

|  |
| --- |
| [HttpGet]  public async Task<IActionResult> GetAllAsync()  {  var regionsDomain = await regionRepository.GetAllAsync();  var regionsDto = mapper.Map<List<RegionDto>>(regionsDomain);  return Ok(regionsDto);  } |

|  |
| --- |
| [HttpGet]  [Route("{id:Guid}")]  public async Task<IActionResult> GetById([FromRoute] Guid id)  {  var regionDomain = await regionRepository.GetByIdAsync(id);  if (regionDomain == null)  return NotFound();  return Ok(mapper.Map<RegionDto>(regionDomain));  } |

* + - 1. Much cleaner! We eliminated the ForEach loop when transferring each regionsDomain field to the regionsDto field.

1. Seeding the database with EF Code
   1. We will expand out our NZWalksDbContext.cs file with the data seeding commands as shown below:

The NZWalksDbContext.cs File

|  |
| --- |
| using Microsoft.EntityFrameworkCore;  using NZWalks.API.Models.Domain;  namespace NZWalks.API.Data  {  public class NZWalksDbContext : DbContext  {  public NZWalksDbContext(DbContextOptions<NZWalksDbContext> dbContextOptions) : base(dbContextOptions)  {  }  public DbSet<Difficulty> Difficulties { get; set; }  public DbSet<Region> Regions { get; set; }  public DbSet<Walk> Walks { get; set; }  protected override void OnModelCreating(ModelBuilder modelBuilder)  {  base.OnModelCreating(modelBuilder);  // Seed data for Difficulties  // Easy, Medium, Hard  var difficulties = new List<Difficulty>()  {  new Difficulty()  {  Id = Guid.Parse("54466f17-02af-48e7-8ed3-5a4a8bfacf6f"),  Name = "Easy"  },  new Difficulty()  {  Id = Guid.Parse("ea294873-7a8c-4c0f-bfa7-a2eb492cbf8c"),  Name = "Medium"  },  new Difficulty()  {  Id = Guid.Parse("f808ddcd-b5e5-4d80-b732-1ca523e48434"),  Name = "Hard"  }  };  // Seed difficulties to the database  modelBuilder.Entity<Difficulty>().HasData(difficulties);  // Seed data for Regions  var regions = new List<Region>  {  new Region  {  Id = Guid.Parse("f7248fc3-2585-4efb-8d1d-1c555f4087f6"),  Name = "Auckland",  Code = "AKL",  RegionImageUrl = "https://images.pexels.com/photos/5169056/pexels-photo-5169056.jpeg?auto=compress&cs=tinysrgb&w=1260&h=750&dpr=1"  },  new Region  {  Id = Guid.Parse("6884f7d7-ad1f-4101-8df3-7a6fa7387d81"),  Name = "Northland",  Code = "NTL",  RegionImageUrl = null  },  new Region  {  Id = Guid.Parse("14ceba71-4b51-4777-9b17-46602cf66153"),  Name = "Bay Of Plenty",  Code = "BOP",  RegionImageUrl = null  },  new Region  {  Id = Guid.Parse("cfa06ed2-bf65-4b65-93ed-c9d286ddb0de"),  Name = "Wellington",  Code = "WGN",  RegionImageUrl = "https://images.pexels.com/photos/4350631/pexels-photo-4350631.jpeg?auto=compress&cs=tinysrgb&w=1260&h=750&dpr=1"  },  new Region  {  Id = Guid.Parse("906cb139-415a-4bbb-a174-1a1faf9fb1f6"),  Name = "Nelson",  Code = "NSN",  RegionImageUrl = "https://images.pexels.com/photos/13918194/pexels-photo-13918194.jpeg?auto=compress&cs=tinysrgb&w=1260&h=750&dpr=1"  },  new Region  {  Id = Guid.Parse("f077a22e-4248-4bf6-b564-c7cf4e250263"),  Name = "Southland",  Code = "STL",  RegionImageUrl = null  },  };  modelBuilder.Entity<Region>().HasData(regions);  }  }  } |

* 1. Next, open the Package Manager Console by click on Tools, NuGet Package Manager and then Package Manager Window.
     1. You will see the Package Manager Window appear at the bottom of the screen.
  2. Next, we will issue two commands in the Package Manager Window. These two commands are:
     1. Add-Migration "Seeding Data for DB"
        1. This will creates the commands for loading the data into the Regions and Difficulties database tables.
     2. Update-Database
        1. This will execute the commands which will perform the table writes.
  3. Once we have issued these commands, the Regions and Difficulties database tables will have been loaded by the data in the NZWalksDbContext.cs file.

1. [Creating the Walks Methods](#TS18)
   1. Before getting started with the details of the code, let’s summarize what needs to be accomplished.
      1. In this first section, we will only be discussing the GetAllAsync method.
         1. We use Create as an example as the other CRUD operations will simply follow suite.
         2. After we discuss the Create CRUD method, the other CRUD operations (Read, Update and Delete) will be displayed in full.
      2. The seven steps below set the stage for displaying all Walks table entries:
         1. Adding the AddWalkRequestDto Class
            1. This class will act as a template to transmit new Walk information to the Domain Model..
         2. Adding the WalkDto Class
            1. This class will act as a template to transmit new Walk information back to the user.
         3. Creating the IWalkRepository
            1. Acts as the controlling, governing interface between the controller and the actual concrete class that will be performing the CRUD operations.
         4. Creating the SQLWalkRepository
            1. The concrete class containing the actual CRUD operational commands.
         5. Add Walks Scoped Services
            1. Binds the SQLWalkRepository file to the IWalkRepository interface within the Program.cs file.
         6. Add AutoMapper Entries
            1. Allows for easy mappings between the properties of the Walks Domain Model and DTO classes.
         7. Create the Walks Controller
            1. This class orchestrates the methods for the CRUD operations for the Walks table.
   2. Below are the details for the sevens steps listed above for the Create Method:
      1. Create the AddWalkRequestDt0 Class
         1. Right-Click on the DTO folder, select Add and Class.
         2. Name the new class AddWalkRequestDto.cs
         3. Open the Walks.cs file and copy the Name, Description, LengthInKm, WalkImageUrl, DifficultyId and RegionId lines.
         4. Paste these lines into the AddWalkRequestDto.cs file so it looks like this:

The AddWalkRequestDto.cs File

|  |
| --- |
| namespace NZWalks.API.Models.DTO  {  public class AddWalkRequestDto  {  public string Name { get; set; }  public string Description { get; set; }  public double LengthInKm { get; set; }  public string? WalkImageUrl { get; set; }  public Guid DifficultyId { get; set; }  public Guid RegionId { get; set; }  }  } |

* + 1. Create the WalkDto Class
       1. This will be used to send data back to the client.
       2. To create this file:
          1. Right-Click on the DTO folder, select add and then class.
          2. Name the class WalkDto.cs and click add.
          3. Fill out the file so it looks like this:

The WalkDto.cs File

|  |
| --- |
| namespace NZWalks.API.Models.DTO  {  public class WalkDto  {  public Guid Id { get; set; }  public string Name { get; set; }  public string Description { get; set; }  public double LengthInKm { get; set; }  public string? WalkImageUrl { get; set; }  public Guid DifficultyId { get; set; }  public Guid RegionId { get; set; }  }  } |

* + 1. Creating the IWalkRepository
       1. Just like we had an interface and a concrete method for the Regions methods, we need the same for Walks.
       2. First we will create the IWalkRepository.cs file
          1. Right-Click on the Repositories folder, click on add and then interface.
          2. Name the new interface IWalkRepository.cs
          3. While this new interface is open, let’s create the CreateAsync interface method as shown below:

The IWalkRepository.cs File

|  |
| --- |
| using NZWalks.API.Models.Domain;  namespace NZWalks.API.Repositories  {  public interface IWalkRepository  {  Task<Walk> CreateAsync(Walk walk);  }  } |

* + 1. Creating the SQLWalkRepository.
       1. Now we need to create the concrete class the IWalkRepository interface will use to execute the SQL commands.
          1. Right-Click on the Repositories folder, click on add and then class.
          2. Name the new class SQLWalkRepository.cs
          3. Below will be the contents of this class:

The SQLWalkRepository.cs File

|  |
| --- |
| using NZWalks.API.Data;  using NZWalks.API.Models.Domain;  namespace NZWalks.API.Repositories  {  public class SQLWalkRepository : IWalkRepository  {  private readonly NZWalksDbContext dbContext;  public SQLWalkRepository(NZWalksDbContext dbContext)  {  this.dbContext = dbContext;  }  public async Task<Walk> CreateAsync(Walk walk)  {  await dbContext.Walks.AddAsync(walk);  await dbContext.SaveChangesAsync();  return walk;  }  }  } |

* + 1. Add Walks Scoped Services.
       1. Add IWalkRepository and SQLWalkRepository to the Program.cs file as scoped services
       2. builder.Services.AddScoped<IWalkRepository, SQLWalkRepository>();
       3. This is telling the API that when IWalkRepository requests a method to be executed, it will call SQLWalkRepository to do the job.
    2. Add AutoMapper Entries
       1. Add the mapping for the create method to the AutoMapperProfiles.cs File
          1. CreateMap<AddWalkRequestDto, Walk>();
          2. CreateMap<Walk,WalkDto>().ReverseMap();
    3. Create the Walks Controller.
       1. Right-Click on the Controllers folder, select add and then Controller.
       2. In the Add New Scaffolded Item window, select API in the left window and API Controller – Empty on the right. Then click the Add button.
       3. Name the controller WalksController.cs and click the Add button.
       4. This will bring up the new controller file
       5. Now fill out the WalksController.cs file so it looks like this:

The WalksController.cs File

|  |
| --- |
| using AutoMapper;  using Microsoft.AspNetCore.Mvc;  using NZWalks.API.Models.Domain;  using NZWalks.API.Models.DTO;  using NZWalks.API.Repositories;  namespace NZWalks.API.Controllers  {  [Route("api/**[controller]**")]  [ApiController]  public class WalksController : ControllerBase  {  private readonly IMapper mapper;  private readonly IWalkRepository walkRepository;  public WalksController(IMapper mapper, IWalkRepository walkRepository)  {  this.mapper = mapper;  }  [HttpPost]  public async Task<IActionResult> Create([FromBody] AddWalkRequestDto addWalkRequestDto)  {  var walkDomainModel = mapper.Map<Walk>(addWalkRequestDto);  await walkRepository.CreateAsync(walkDomainModel);  return Ok(mapper.Map<WalkDto>(walkDomainModel));  }  }  } |

1. [Validations in ASP.NET](#TS19)
   1. Often we may need to check if the user input for adding to updating records is valid
      1. For instance, when adding or updating the Code field, we may want to ensure the code entered into the field is exactly 3 digits.
      2. Also we might want to ensure the name enter does not exceed 100 characters.
      3. We will add annotations to the AddRegionRequestDto class to enforce these policies as shown below

The AddRegionRequestDto.cs File

|  |
| --- |
| using System.ComponentModel.DataAnnotations;  namespace NZWalks.API.Models.DTO  {  public class AddRegionRequestDto  {  [Required]  [MinLength(3, ErrorMessage = "Code is minimum of 3 characters")]  [MaxLength(3, ErrorMessage = "Code is maximum of 3 characters")]  public string Code { get; set; }  [Required]  [MaxLength(100, ErrorMessage = "Name has a maximum length of 100 characters")]  public string Name { get; set; }  public string? RegionImageUrl { get; set; }  }  } |

* 1. Now we will add custom action filters into our API to check for these filter annotations.
     1. Directly under the API, create a folder called CustomActionFilters
     2. Inside this filter, create a new class called ValidateModelAttribute.cs
     3. This class will look like this:

The ValidateModelAttribute.cs Class

|  |
| --- |
| using Microsoft.AspNetCore.Mvc;  using Microsoft.AspNetCore.Mvc.Filters;  namespace NZWalks.API.CustomActionFilters  {  public class ValidateModelAttribute : ActionFilterAttribute  {  public override void OnActionExecuting(ActionExecutingContext context)  {  if (context.ModelState.IsValid == false)  {  context.Result = new BadRequestResult();  }  }  }  } |

* 1. Finally we add the **[ValidateModel]** annotation to the Create method and add the CustomActionFilters namespace to the RegionsController.cs file.
     1. What will happen is the controller will see the **[ValidateModel]** tag and look for a file called ValidateModelAttribute.cs.
     2. Below is the code for the Create method with the **[ValidateModel]** tag added:

Adding [ValidateModel] to the Create Method

|  |
| --- |
| [HttpPost]  [ValidateModel]  public async Task<IActionResult> Create([FromBody] AddRegionRequestDto addRegionRequestDto)  {  var regionDomainModel = mapper.Map<Region>(addRegionRequestDto);  regionDomainModel = await regionRepository.CreateAsync(regionDomainModel);  var regionDto = mapper.Map<RegionDto>(regionDomainModel);  return CreatedAtAction(nameof(GetById), new { id = regionDto.Id }, regionDto);  } |

* 1. Now the user syntax is checked even before the Create method is checked.
  2. We would do this for all our class in all our controllers that need user input checked.

1. [Filtering, Sorting and Pagination](#TS20)
   1. Filtering
      1. When querying our data, we probably do not want to see all the rows, but rather a specific listing based on a filter value.
      2. If this were SQL, we could use the where clause to narrow down our returned listing.
      3. With Entity Framework, we can mimic the SQL where clause by using the search keywords.
         1. We have seen this in use in a URL. For example:
            1. <https://www.espn.com/football/scores?Team=Lions>
            2. This will return the scores for the Detroit Lions football team.
      4. In our example, we will be using filtering on the WalksController and specifically on the GetAll method.
         1. We will need to make changes to three files:
            1. WalksController.cs
            2. IWalkRepository.cs
            3. SQLWalkRepository.cs
         2. Modifying the WalksController.cs file
            1. Below is a sniped of the GetAll() method

The WalksController.cs File GetAll Method

|  |
| --- |
| [HttpGet]  public async Task<IActionResult> GetAll([FromQuery] string? filterOn, [FromQuery] string? filterQuery)  {  return Ok(mapper.Map<List<WalkDto>>(await walkRepository.GetAllAsync(filterOn, filterQuery)));  } |

* + - * 1. We have added the syntax [FromQuery] string? filterOn, [FromQuery] string? filterQuery inside the GetAll() parentheses.

The [FromQuery] annotation means the parameters will be coming straight from the URL line.

The variable filterOn will be the column name and filterQuery will contains the actual string we are querying.

Note that filterOn and filterQuery are not keywords, but rather just names we have selected because they make sense.

We could have just as easily named they mickeyMouse and donandDuck.

In our ESPN example, filterOn = Team and filterQuery = Lions.

* + - * 1. When calling the walkRepository (IWalkRepository.cs), we pass these two parameters to the GetAllAsync interface method as shown below:

await walkRepository.GetAllAsync(filterOn, filterQuery)

* + - 1. Modifying the IWalkRepository.cs file
         1. When the GetAll method passes its request to the GetAllAsync abstract method, it will look like this:

Task<List<Walk>> GetAllAsync(string? filterOn = null, string? filterQuery = null);

* + - * 1. A couple of things to notice here.

string? means these two values can be null.

filterOn = null, filterQuery = null sets the default values to null.

* + - 1. Modifying the SQLWalkRepository.cs file
         1. The GetAllAsync method here analyzes to see if the values of filterOn and filterQuery both have values in them.

If they do, it starts its analysis of the field names using the switch statement as shown below:

The GetAllAsync Method in the SQLWalkRepository.cs File

|  |
| --- |
| public async Task<List<Walk>> GetAllAsync(string? filterOn = null, string? filterQuery = null)  {  IQueryable<Walk> walks = dbContext.Walks.Include("Difficulty").Include("Region").AsQueryable();  if (string.IsNullOrWhiteSpace(filterOn) == false && string.IsNullOrWhiteSpace(filterQuery) == false)  {  switch (filterOn.ToUpper())  {  case "NAME":  walks = walks.Where(x => x.Name.Contains(filterQuery));  break;  case "DESCRIPTION":  walks = walks.Where(x => x.Description.Contains(filterQuery));  break;  }  }  return await walks.ToListAsync();  } |

The walks.Where(x => x.Name.Contains(filterQuery)) statement is the SQL equivalent to:

SELECT \* FROM Walks WHERE Name = @FilterQuery;

* 1. Sorting
     1. When we receive several rows of data, we may want a particular column sorted by either ascending or descending order.
     2. In the filtering section above, we added parameters to the GetAll method.
     3. We will add additional parameters to the same line to including sort on a particular column and whether we want ascending or descending order.
     4. Like the filtering section, we will have to make changes to the same three files.
        1. These three files are:
           1. WalksController.cs
           2. IWalkRepository.cs
           3. SQLWalkRepository.cs
        2. Modifying the WalksController.cs file.
           1. Below is a sniped of the GetAll() method with the sorting fields added:

The WalksController.cs File GetAll Method

|  |
| --- |
| [HttpGet]  public async Task<IActionResult> GetAll([FromQuery] string? filterOn, [FromQuery] string? filterQuery,  [FromQuery] string? sortBy, [FromQuery] bool? isAscending)  {  return Ok(mapper.Map<List<WalkDto>>(await walkRepository.GetAllAsync(filterOn, filterQuery,sortBy,isAscending ?? true)));  } |

* + - * 1. The variables sortBy and isAscending are not key words and can be replaced by any variable name you wish.
        2. In the return line, the statement isAscending ?? true means if the bool value is null, make it true.
      1. Modifying the IWalkRepository.cs file
         1. Task<List<Walk>> GetAllAsync(string? filterOn = null, string? filterQuery = null, string? sortBy = null, bool isAscending = true);
      2. Modifying the SQLWalkRepository.cs file
         1. Below id the GetAllAsync method with sorting added

The GetAllAsync Method in the SQLWalkRepository.cs File

|  |
| --- |
| public async Task<List<Walk>> GetAllAsync(string? filterOn = null, string? filterQuery = null, string? sortBy = null, bool isAscending = true)  {  IQueryable<Walk> walks = dbContext.Walks.Include("Difficulty").Include("Region").AsQueryable();  // Filtering  if (string.IsNullOrWhiteSpace(filterOn) == false && string.IsNullOrWhiteSpace(filterQuery) == false)  {  switch (filterOn.ToUpper())  {  case "NAME":  walks = walks.Where(x => x.Name.Contains(filterQuery));  break;  case "DESCRIPTION":  walks = walks.Where(x => x.Description.Contains(filterQuery));  break;  }  }  // Sorting  if (string.IsNullOrWhiteSpace(sortBy) == false)  {  switch (sortBy.ToUpper())  {  case "NAME":  walks = isAscending ? walks.OrderBy(x => x.Name) : walks.OrderByDescending(x => x.Name);  break;  case "LENGTH":  walks = isAscending ? walks.OrderBy(x => x.LengthInKm) : walks.OrderByDescending(x => x.LengthInKm);  break;  }  }  return await walks.ToListAsync();  } |

* 1. Pagination
     1. This is the ability to load only so many rows in a display at one time.
     2. If we have 10,000 rows in the Walks table, we probably don’t want to display all 10,000 at once.
     3. We can break the page size down and request x amount of rows per page as well as requesting a particular page number.
     4. In following suite of displaying the code with Filtering and sorting, we will update the same three files, adding the pagination variables pageNumber and pageSize.
        1. These three files are:
           1. WalksController.cs
           2. IWalkRepository.cs
           3. SQLWalkRepository.cs.
        2. Modifying the WalksController.cs file
           1. Below is a sniped of the GetAll() method with the pagination fields added

The WalksController.cs File GetAll Method

|  |
| --- |
| [HttpGet]  public async Task<IActionResult> GetAll([FromQuery] string? filterOn, [FromQuery] string? filterQuery,  [FromQuery] string? sortBy, [FromQuery] bool? isAscending, [FromQuery] int pageNumber = 1, [FromQuery] int pageSize = 1000)  {  return Ok(mapper.Map<List<WalkDto>>(await walkRepository.GetAllAsync(filterOn, filterQuery,sortBy,isAscending ?? true, pageNumber, pageSize)));  } |

* + - 1. Modifying the IWalkRepository.cs file

The GetAllAsync Abstract Method in the IWalkRepository.cs file

|  |
| --- |
| Task<List<Walk>> GetAllAsync(string? filterOn = null, string? filterQuery = null,  string? sortBy = null, bool isAscending = true, int pageNumber = 1, int pageSize = 1000); |

* + - 1. Modifying the SQLWalkRepository.cs file

The GetAllAsync Method in the SQLWalkRepository.cs File

|  |
| --- |
| public async Task<List<Walk>> GetAllAsync(string? filterOn = null, string? filterQuery = null,  string? sortBy = null, bool isAscending = true, int pageNumber = 1, int pageSize = 1000)  {  IQueryable<Walk> walks = dbContext.Walks.Include("Difficulty").Include("Region").AsQueryable();  // Filtering  if (string.IsNullOrWhiteSpace(filterOn) == false && string.IsNullOrWhiteSpace(filterQuery) == false)  {  switch (filterOn.ToUpper())  {  case "NAME":  walks = walks.Where(x => x.Name.Contains(filterQuery));  break;  case "DESCRIPTION":  walks = walks.Where(x => x.Description.Contains(filterQuery));  break;  }  }  // Sorting  if (string.IsNullOrWhiteSpace(sortBy) == false)  {  switch (sortBy.ToUpper())  {  case "NAME":  walks = isAscending ? walks.OrderBy(x => x.Name) : walks.OrderByDescending(x => x.Name);  break;  case "LENGTH":  walks = isAscending ? walks.OrderBy(x => x.LengthInKm) : walks.OrderByDescending(x => x.LengthInKm);  break;  }  }  // Pagination  var skipResults = (pageNumber - 1) \* pageSize;  return await walks.Skip(skipResults).Take(pageSize).ToListAsync();  } |

1. [Securing the ASP.NET API](#TS21)
   1. Setting up Authentication.
      1. Modifying the appsettings.json file
         1. Open the appsettings.json file
         2. Add the Jwt section until you get this far. Note that for the long string next to “key”, I just typed a bunch of garbage letters and numbers.

|  |
| --- |
| {  "Logging": {  "LogLevel": {  "Default": "Information",  "Microsoft.AspNetCore": "Warning"  }  },  "AllowedHosts": "\*",  "ConnectionStrings": {  "NZWalksConnectionString": "Server=DAVES\_PC;Database=NZWalksDb;TrustServerCertificate=true;Trusted\_Connection=true;"  },  "Jwt": {  "Key": "HJIHOIugiGGuhuhuhuohuGGIUGuhhiGUgiygpjj88786uggi9897898HHUHUHihiohih",  "Issuer": "",  “Audience”: “”  }  } |

* + - 1. Now we need the value for the issuer. To get this, follow these steps:
         1. Right-Click on the Project (NZWalks.API) and go to Properties
         2. In the Properties menu, expand Debug and click on General
         3. In the main screen, click on Open debug launch profiles UI to open it.
         4. In the Launch Profiles dialog box, click on HTTPS on the left and scroll down the center box until you see App URL.
         5. This URL will display both the https and http URLs. **Copy only the https portion**.
         6. Close the Properties box, go back to the appsettings.json file and paste the https in the Issuer quotes.

The Issuer and Audience lines should now look like this. (Note: Your port number may be different):

"Issuer": “<https://localhost:7257>/”

"Audience": “<https://localhost:7257/>”

* + - * 1. Save and close the appsettings.json file.
    1. Modifying the Program.cs file
       1. We will be adding two items to the Program.cs file.
          1. builder.Services.AddAuthentication
          2. app.UseAuthentication
       2. Below are the details of the two items above:
          1. builder.Services.AddAuthentication

Just a precursor, this line is going to be a mouthful!

Don’t worry about understanding all of it.

Put this line just above the var app = builder.Build() line:

|  |
| --- |
| builder.Services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)  .AddJwtBearer(options =>  options.TokenValidationParameters = new TokenValidationParameters  {  ValidateIssuer = true,  ValidateAudience = true,  ValidateLifetime = true,  ValidateIssuerSigningKey = true,  ValidIssuer = builder.Configuration["Jwt:Issuer"],  ValidAudience = builder.Configuration["Jwt:Audience"],  IssuerSigningKey = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(builder.Configuration["Jwt:Key"]))  }); |

Note that for the syntax ["Jwt:Issuer"],["Jwt:Audience"] and ["Jwt:Key"], these are in brackets because they are pulling the values from the appsettings.json file.

Make sure the spelling of these three items is exactly the same as it is in appsettings.json.

* + - * 1. app.UseAuthentication

Just above the app.UseAuthorization() line in Program.cs, add the app.UseAuthentication(); line.

Now the Authentication and Authorization lines should be together line this:

|  |
| --- |
| app.UseAuthentication();  app.UseAuthorization(); |

Save the Program.cs file and close it.

* 1. Now we have set up our Authentication environment.
  2. In the next section we will apply this authentication to our controllers.

1. [Defining Controller Authorization](#TS22)
   1. It is time to apply authorization to one of our controllers.
      1. For this we will pick the Regions controller as a test platform.
      2. Currently our Regions controllers class RegionsController.cs class has instance Route and ApiController fields set as this:

|  |
| --- |
| [Route("api/**[controller]**")]  [ApiController] |

* + 1. We will add the [Authorize] to this definition so it looks like this:

|  |
| --- |
| [Route("api/**[controller]**")]  [ApiController]  [Authorize] |

* + 1. Now when we call any method in the Regions Controller via Swagger, we will get a 401 error because there is no Jwt token presented to authenticate the user:  
         
       A screen shot of a computer error

       AI-generated content may be incorrect.
    2. Now the user will need authorization to access the methods
    3. In the next section we will learn about creating Jwt tokens so the user can be authenticated properly.

1. [Setting up Identity in the API](#TS23)
   1. User will be able to login to the application with a user name and password.
   2. Here we will set up our Auth database This will:
      1. Create New Connection String
      2. Create New DbContext with Roles which we will seed
      3. Inject DbContext and Identity
      4. Run EF Core Migrations
   3. Setting up Roles
      1. There will two types of roles that we set up:
         1. Reader
            1. This role will have GET rights
         2. Writer
            1. This role will have POST, PUT and DELETE rights.
   4. Auth Table Connection String
      1. Now we will add the Connection String in appsettings.json for the Auth database.
         1. Open appsettings.json.
         2. Currently we have only one connection string in appsettings.json:

|  |
| --- |
| "ConnectionStrings": {  "NZWalksConnectionString": "Server=DAVES\_PC;Database=NZWalksDb;TrustServerCertificate=true;Trusted\_Connection=true;"  }, |

* + - 1. We will add the second connection string in appsettings.json for the Auth database as show below:

|  |
| --- |
| "ConnectionStrings": {  "NZWalksConnectionString": "Server=DAVES\_PC;Database=NZWalksDb;TrustServerCertificate=true;Trusted\_Connection=true;",  "NZWalksAuthConnectionString": "Server=DAVES\_PC;Database=NZWalksAuthDb;TrustServerCertificate=true;Trusted\_Connection=true;"  }, |

* + - 1. Save and close the appsettings.json file.
  1. Injecting the Auth DbContext
     1. Like with the data DbContext, we need to inject the Auth DbContext into Program.cs
     2. Add the Auth DbContext below the data DbContext in Program.cs as shown below:

|  |
| --- |
| builder.Services.AddDbContext<NZWalksDbContext>(options =>  options.UseSqlServer(builder.Configuration.GetConnectionString("NZWalksConnectionString")));  **builder.Services.AddDbContext<NZWalksAuthDbContext>(options =>**  **options.UseSqlServer(builder.Configuration.GetConnectionString("NZWalksAuthConnectionString")));** |

* 1. Adding the Auth DbContext Class
     1. We need to create and seed the NZWalksAuthDb database so it knows what roles exist.
     2. To do so, we will create the NZWalksAuthDbContext.cs file in the Data folder.
     3. To create this file, perform these steps:
        1. Right-Click on the Data folder, select Add and select Class.
        2. Name this class NZWalksAuthDbContext.cs
        3. Populate this file with the following:

The NZWalksAuthDbContext.cs File

|  |
| --- |
| using Microsoft.AspNetCore.Identity;  using Microsoft.AspNetCore.Identity.EntityFrameworkCore;  using Microsoft.EntityFrameworkCore;  namespace NZWalks.API.Data  {  public class NZWalksAuthDbContext : IdentityDbContext  {  public NZWalksAuthDbContext(DbContextOptions<NZWalksAuthDbContext> options) : base(options)  {  }  protected override void OnModelCreating(ModelBuilder builder)  {  base.OnModelCreating(builder);  var readerRoleId = "60253706-a336-419e-98a1-6f2f4bba6f51";  var writerRoleId = "8180367e-8b1e-49d5-afa8-fa1a84a4f176";  var roles = new List<IdentityRole>  {  new IdentityRole  {  Id = readerRoleId,  ConcurrencyStamp = readerRoleId,  Name = "Reader",  NormalizedName = "Reader".ToUpper()  },  new IdentityRole  {  Id = writerRoleId,  ConcurrencyStamp = writerRoleId,  Name = "Writer",  NormalizedName = "Writer".ToUpper()  }  };  builder.Entity<IdentityRole>().HasData(roles);  }  }  } |

* 1. Run the Auth database migration
     1. Click on Tools, NuGet Pack Manager and Package Manager Console
     2. In the Package Manager Console window, enter these two commands:
        1. Add-Migration “Create Auth DB” -Context “NZWalksAuthDbContext”
        2. Update-Database -Context “NZWalksAuthDbContext”
     3. Now the NZWalksAuthDb database is created as we can see in SMS

|  |
| --- |
|  |

* 1. Setting up Identity
     1. We have to inject the Identity packages into our solution.
     2. We will be adding two entries into the Program.cs file.
     3. Just above the builder.Services.AddAuthentication line in Program.cs, add these two statements:

|  |
| --- |
| builder.Services.AddIdentityCore<IdentityUser>()  .AddRoles<IdentityRole>()  .AddTokenProvider<DataProtectorTokenProvider<IdentityUser>>(“NZWalks”)  .AddEntityFrameworkStores<NZWalksAuthDbContext>()  .AddDefaultTokenProviders();  builder.Services.Configure<IdentityOptions>(options =>  {  options.Password.RequireDigit = false;  options.Password.RequireLowercase = false;  options.Password.RequireNonAlphanumeric = false;  options.Password.RequireUppercase = false;  options.Password.RequiredLength = 6;  options.Password.RequiredUniqueChars = 1;  }); |

* 1. Creating the Auth Controller
     1. Now that we have all our Authentication and Identity entries into Program.cs, we need to create the Auth controller.
     2. To accomplish this, follow these steps:
        1. Adding the RegisterRequestDto.cs class
           1. Right-Click on the DTO folder, click add and class.
           2. Name the new Class RegisterRequestDto.cs.
           3. Fill it out as it is shown below

The RegisterRequestDto.cs File

|  |
| --- |
| using System.ComponentModel.DataAnnotations;  namespace NZWalks.API.Models.DTO  {  public class RegisterRequestDto  {  [Required]  [DataType(DataType.EmailAddress)]  public string Username { get; set; }  [Required]  [DataType(DataType.Password)]  public string Password { get; set; }  public string[] Roles { get; set; }  }  } |

* + - 1. Adding the AuthController.cs class
         1. Right-Click on the Controller folder, select Add and Controller.
         2. In the Add New Scaffolded Item menu, click on API in the left menu and MVC Controller – Empty in the center window.
         3. Name this new controller AuthController.cs and click on the Add button.
      2. Fill out the AuthController.cs file as shown below:

The AuthController.cs File

|  |
| --- |
| using Microsoft.AspNetCore.Identity;  using Microsoft.AspNetCore.Mvc;  using NZWalks.API.Models.DTO;  namespace NZWalks.API.Controllers  {  [Route("api/**[controller]**")]  [ApiController]  public class AuthController : ControllerBase  {  private readonly UserManager<IdentityUser> userManager;  public AuthController(UserManager<IdentityUser> userManager)  {  this.userManager = userManager;  }  [HttpPost]  [Route("Register")]    public async Task<IActionResult> Register([FromBody] RegisterRequestDto registerRequestDto)  {  var identityUser = new IdentityUser  {  UserName = registerRequestDto.Username,  Email = registerRequestDto.Username  };  var identityResult = await userManager.CreateAsync(identityUser, registerRequestDto.Password);  if (identityResult.Succeeded)  {  // Add roles to user  if (registerRequestDto.Roles != null & registerRequestDto.Roles.Any())  {  identityResult = await userManager.AddToRolesAsync(identityUser, registerRequestDto.Roles);  if (identityResult.Succeeded)  {  return Ok("User was registered");  }  }  }  return BadRequest("Something went wrong");  }  }  } |

* 1. Testing out our Identity solution
     1. Run the application and go to Swagger. Choose the Auth controller and POST register method.
     2. Enter this information for the register method:

|  |
| --- |
|  |

* + 1. Click the execute button. You should get this result:

|  |
| --- |
|  |

* + 1. In SMS, we can select from the AspNetRoles, AspNetUserRoles and AspNetUsers tables and see the entries for each table.

1. [Login Functionality using Password](#TS24)
   1. Now that we have our JWT Identity set up, we need to create a Login method on our Auth Controller.
   2. To set up this method, use the code below when creating the login method inside the AuthController.cs file:
      1. Adding the LoginRequestDto.cs file
         1. Create a new class inside the DTO folder and name it LoginRequestDto.cs
         2. Add this text to the file:

The LoginRequestDto.cs File

|  |
| --- |
| using System.ComponentModel.DataAnnotations;  namespace NZWalks.API.Models.DTO  {  public class LoginRequestDto  {  [Required]  [DataType(DataType.EmailAddress)]  public string Username { get; set; }  [Required]  [DataType(DataType.Password)]  public string Password { get; set; }  }  } |

* + 1. Adding the Login method to the AuthController.com file

The Login Method in AuthController.cs

|  |
| --- |
| [HttpPost]  [Route("Login")]  public async Task<IActionResult> Login([FromBody] LoginRequestDto loginRequestDto)  {  var user = await userManager.FindByEmailAsync(loginRequestDto.Username);  if (user != null)  {  var checkPasswordResult = await userManager.CheckPasswordAsync(user, loginRequestDto.Password);  if (checkPasswordResult)  {  return Ok();  }  }  return BadRequest("Username or password is incorrect");  } |

* 1. Now when you run the application, you can test your Username and Password against the NZWalksAuthDb database!

1. [Creating JWT Tokens](#TS25)
   1. When creating tokens, we will be leveraging interfaces since we should not be doing this work in the controller.
   2. Here are the steps for the token creation process:
      1. Create Repository Interface for Token Creation Methods
         1. Right-Click on the Repository folder, choose and class
         2. Select interface and name it ITokenRepository.cs
         3. Place the follow text I this file

The ITokenRepository.cs File

|  |
| --- |
| using Microsoft.AspNetCore.Identity;  namespace NZWalks.API.Repositories  {  public interface ITokenRepository  {  string CreateJWTToken(IdentityUser user, List<string> roles);  }  } |

* + 1. Create Implementation class for the interface file
       1. Right-Click on the Repositories folder, select add and class.
       2. Name this class TokenRepository.cs
       3. Place the following text in this file:

The TokenRepository.cs File

|  |
| --- |
| using System.IdentityModel.Tokens.Jwt;  using System.Security.Claims;  using System.Text;  using Microsoft.AspNetCore.Identity;  using Microsoft.IdentityModel.Tokens;  namespace NZWalks.API.Repositories  {  public class TokenRepository : ITokenRepository  {  private readonly IConfiguration configuration;  public TokenRepository(IConfiguration configuration)  {  this.configuration = configuration;  }  public string CreateJWTToken(IdentityUser user, List<string> roles)  {  // Create Claims  var claims = new List<Claim>();  claims.Add(new Claim(ClaimTypes.Email, user.Email));  foreach (var role in roles)  {  claims.Add(new Claim(ClaimTypes.Role, role));  }  var key = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(configuration[“Jwt:Key”]));  var credentials = new SigningCredentials(key, SecurityAlgorithms.HmacSha256);  var token = new JwtSecurityToken(  configuration[“Jwt:Issuer”],  configuration[“Jwt:Audience”],  claims,  expires: DateTime.Now.AddMinutes(15),  signingCredentials: credentials);  return new JwtSecurityTokenHandler().WriteToken(token);  }  }  } |

* + 1. Inject the ITokenRepository into the application.
       1. In the Program.cs class, place this line directly after the other two AddScoped lines:
          1. builder.Services.AddScoped<ItokenRepository, TokenRepository>();
    2. Create the LoginResponseDto class
       1. Right-Click on the DTO folder, select add and class.
       2. Name this class LoginResponseDto.cs
       3. Place the following text in this file

The LoginResponseDto.cs file

|  |
| --- |
| namespace NZWalks.API.Models.DTO  {  public class LoginResponseDto  {  public string JwtToken { get; set; }  }  } |

* + 1. The AuthController.cs File
       1. Here is the fully populated AuthController.cs file

The AuthController.cs File

|  |
| --- |
| using Microsoft.AspNetCore.Identity;  using Microsoft.AspNetCore.Mvc;  using NZWalks.API.Models.DTO;  using NZWalks.API.Repositories;  namespace NZWalks.API.Controllers  {  [Route("api/**[controller]**")]  [ApiController]  public class AuthController : ControllerBase  {  private readonly UserManager<IdentityUser> userManager;  private readonly ITokenRepository tokenRepository;  public AuthController(UserManager<IdentityUser> userManager, ITokenRepository tokenRepository)  {  this.userManager = userManager;  this.tokenRepository = tokenRepository;  }  [HttpPost]  [Route("Register")]    public async Task<IActionResult> Register([FromBody] RegisterRequestDto registerRequestDto)  {  var identityUser = new IdentityUser  {  UserName = registerRequestDto.Username,  Email = registerRequestDto.Username  };  var identityResult = await userManager.CreateAsync(identityUser, registerRequestDto.Password);  if (identityResult.Succeeded)  {  // Add roles to user  if (registerRequestDto.Roles != null & registerRequestDto.Roles.Any())  {  identityResult = await userManager.AddToRolesAsync(identityUser, registerRequestDto.Roles);  if (identityResult.Succeeded)  {  return Ok("User was registered");  }  }  }  return BadRequest("Something went wrong");  }  // Login method  // POST: /api/Auth/Login  [HttpPost]  [Route("Login")]  public async Task<IActionResult> Login([FromBody] LoginRequestDto loginRequestDto)  {  var user = await userManager.FindByEmailAsync(loginRequestDto.Username);  if (user != null)  {  var checkPasswordResult = await userManager.CheckPasswordAsync(user, loginRequestDto.Password);  if (checkPasswordResult)  {  // Get Roles for this user  var roles = await userManager.GetRolesAsync(user);  if (roles != null)  {  // Create Token  var jwtToken = tokenRepository.CreateJWTToken(user, roles.ToList());  var response = new LoginResponseDto  {  JwtToken = jwtToken  };  return Ok(response);  }  }  }  return BadRequest("Username or password incorrect");  }  }  } |

1. [Role Based Authorization](#TS26)
   1. Previously in our RegionsController.cs file, we applied the [Authorize] tag at the controller level as shown below:

|  |
| --- |
| 1. // https://localhost:xxxx/api/Regions 2. [Route("api/**[controller]**")] 3. [ApiController] 4. [Authorize] |

* 1. Now we will remove this [Authorize] tag from the controller level and apply it to each method individually.
  2. In this way, we can control whether a method requires a Reader role or a Writer role to execute the method.
     1. Typically the R in CRUD is a reader roles while the CUD’s in CRUD are writers roles.
  3. Below are two examples of how we will apply the [Authorize] tag.
  4. The first is for the GetAllAsync method which will be a Reader role
     1. Applying the [Authorize] tag to the GetAllAsync method:

|  |
| --- |
| [HttpGet]  **[Authorize(Roles = "Reader")]**  public async Task<IActionResult> GetAllAsync()  {  var regionsDomain = await regionRepository.GetAllAsync();  var regionsDto = mapper.Map<List<RegionDto>>(regionsDomain);  return Ok(regionsDto);  } |

* + 1. As you can see, we modified the Authorize tag to add the role type: [Authorize(Roles = "Reader")]
  1. The second is for the Create method which will be a Writer role:
     1. Applying the [Authorize] tag to the Create method:

|  |
| --- |
| [HttpPost]  [ValidateModel]  **[Authorize(Roles = "Writer")]**  public async Task<IActionResult> Create([FromBody] AddRegionRequestDto addRegionRequestDto)  {  var regionDomainModel = mapper.Map<Region>(addRegionRequestDto);  regionDomainModel = await regionRepository.CreateAsync(regionDomainModel);  var regionDto = mapper.Map<RegionDto>(regionDomainModel);  return CreatedAtAction(nameof(GetById), new { id = regionDto.Id }, regionDto);  } |

* + 1. As you can see, we modified the Authorize tag to add the role type: [Authorize(Roles = "Writer")]
  1. If we want a method to be available to both Reader and Writer, we can make the Authorize like this: [Authorize(Roles = "Reader,Writer")]
  2. Now we simply apply this philosophy to all our methods, giving Create, Update and Delete Writer roles and Get method the Reader role.

1. [Adding Authorization to Swagger](#TS27)
   1. Open the Program.cs file and make these changes
      1. Add using Microsoft.OpenApi.Models; as a using statement near the top of the file.
      2. Find the line builder.Services.AddSwaggerGen(); add replace it with this statement:

|  |
| --- |
| builder.Services.AddSwaggerGen(options =>  {  options.SwaggerDoc("v1", new OpenApiInfo { Title = "NZ Walks API", Version = "v1" });  options.AddSecurityDefinition(JwtBearerDefaults.AuthenticationScheme, new OpenApiSecurityScheme  {  Name = "Authorization",  In = ParameterLocation.Header,  Type = SecuritySchemeType.ApiKey,  Scheme = JwtBearerDefaults.AuthenticationScheme  });  options.AddSecurityRequirement(new OpenApiSecurityRequirement  {  {  new OpenApiSecurityScheme  {  Reference = new OpenApiReference  {  Type = ReferenceType.SecurityScheme,  Id = JwtBearerDefaults.AuthenticationScheme  },  Scheme = "Oauth2",  Name = JwtBearerDefaults.AuthenticationScheme,  In = ParameterLocation.Header  },  new List<string>()  }  });  }); |

1. [Testing the JWT Token Environment](#TS28)
   1. Adding the Reader and Writer user roles.
      1. Now that we have completed the implementation of the JWT token environment, it is time for testing.
      2. To start with fresh, I would manually clear out any existing users from the NZWalksAuthDb tables.
      3. To do this, issue these commands in SMS
         1. delete from AspNetUserRoles
         2. delete from AspNetUsers
      4. Now go in and register two new users, one for the reader role and the other for the writer role:
         1. Note: Before doing so, I would modify the TokenRepository.cs file and change the expiration time:
            1. From: expires: DateTime.Now.AddMinutes(15)
            2. To: expires: DateTime.Now.AddMinutes(120)
            3. This will give you 2 hours of testing before the tokens expire.
         2. Register Two New Users
            1. Start the application and use the Browser window to go into Swagger.
            2. In the Auth Controller section, register 2 new users with these attributes:

User 1

Username: [reader@nzwalks.com](mailto:reader@nzwalks.com)

Password: Reader123

Role: Reader

User 2:

Username: [writer@nzwalks.com](mailto:writer@nzwalks.com)

Password: Writer123

Role: Writer

* + - 1. Login with the two users and capture the JWT Token
         1. In Swagger, select the Login option

User 1

Username: [reader@nzwalks.com](mailto:reader@nzwalks.com)

Password: Reader123

Copy and paste the jwttoken below:

|  |
| --- |
| eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJodHRwOi8vc2NoZW1hcy54bWxzb2FwLm9yZy93cy8yMDA1LzA1L2lkZW50aXR5L2NsYWltcy9lbWFpbGFkZHJlc3MiOiJyZWFkZXJAbnp3YWxrcy5jb20iLCJodHRwOi8vc2NoZW1hcy5taWNyb3NvZnQuY29tL3dzLzIwMDgvMDYvaWRlbnRpdHkvY2xhaW1zL3JvbGUiOiJSZWFkZXIiLCJleHAiOjE3NTU5NDUzNDMsImlzcyI6Imh0dHBzOi8vbG9jYWxob3N0OjcyNTcvIiwiYXVkIjoiaHR0cHM6Ly9sb2NhbGhvc3Q6NzI1Ny8ifQ.lAvZ4mlFnMzBCoqZJaJ4EePgHxDI9Zd2KJuRajGVNKE |

User 2

Username: [writer@nzwalks.com](mailto:writer@nzwalks.com)

Password: Writer123

Copy and paste the jwttoken below:

|  |
| --- |
| eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJodHRwOi8vc2NoZW1hcy54bWxzb2FwLm9yZy93cy8yMDA1LzA1L2lkZW50aXR5L2NsYWltcy9lbWFpbGFkZHJlc3MiOiJ3cml0ZXJAbnp3YWxrcy5jb20iLCJodHRwOi8vc2NoZW1hcy5taWNyb3NvZnQuY29tL3dzLzIwMDgvMDYvaWRlbnRpdHkvY2xhaW1zL3JvbGUiOiJXcml0ZXIiLCJleHAiOjE3NTU5NDU3OTAsImlzcyI6Imh0dHBzOi8vbG9jYWxob3N0OjcyNTcvIiwiYXVkIjoiaHR0cHM6Ly9sb2NhbGhvc3Q6NzI1Ny8ifQ.mBgxHcqUgetIHtq\_ejhmpZ3TCLokjImee27Ojdt7Nj0 |

* + - 1. Use Authorize button to register user
         1. In the Swagger browser window, there is a green Authorize button in the upper right hand side of the screen.
         2. We will be using this button to register each of the two users to test their authorizations.
         3. Testing the Reader functionality

Registering the Reader JWT Token

Copy the jwttoken captured when logging in with the [reader@nzwalks.com](mailto:reader@nzwalks.com) account.

Click on the Authorize green button.

Enter the word Bearer followed by a space and then the jwttoken

Click on the green Authorize button.

The reader account is now registered. Let’s test it.

Testing Read methods

In the Regions controller area of the Swagger browser screen, expand the GET /api/Regions window and click on the Execute bar

You will see that we can now see all the Region details displayed.

Testing Write methods

First, copy one of the Id Guids from the Get listing. We will be using this Id to try to delete it.

Id: 906cb139-415a-4bbb-a174-1a1faf9fb1f6

In the Regions controller area of the Swagger browser screen, expand the DELETE /api/Regions/(id} and click the Try it out button.

In the id box, paste the id Guid your just copied and click the Execute bar.

Upon this operation, we will receive a 403 error since we only have Reader rights and this operation required Writer writes.

* + - * 1. Testing the Writer functionality

First, click the Green Authorize button and click Logout.

This now invalidate the reader token.

Register the Writer JWT Token

Copy the jwttoken captured when logging in with the [writer@nzwalks.com](mailto:writer@nzwalks.com) account.

Click on the Authorize green button.

Enter the word Bearer followed by a space and then the jwttoken

Click on the green Authorize button

Testing Write Methods

Using the same Guid we tried to delete previously, expand the DELETE option again and click the Try it out button.

In the id box, paste the id Guid your just copied and click the Execute bar

This time we get a 200 response which signifies the operation was successful.

* 1. We have now successfully implemented JWT Token security into our application!
  2. Finally, go back into the TokenRepository.cs file and reset the expires time window to what is was before testing:
     1. expires: DateTime.Now.AddMinutes(15)
     2. You will now need to stop and restart the application to have this new time implemented.

1. [Image Uploading in ASP.NET](#TS29)
   1. The ability to upload images and files is essential in any API solution.
   2. In this section we will learn how to do this.
   3. There are 10 parts to this section:
      1. Creating the Image Domain Model
      2. Adding the Image to NZWalksDbContext
      3. Rerun EF Core Migrations
      4. Add Http Context Accessor to Program.cs
      5. Creating the Images folder
      6. Creating the Image Upload Request DTO
      7. Creating the Image Repository Interface.
      8. Creating the Local Image Repository
      9. Inject IImageRepository Scoped Services into Program.cs
      10. Creating the Image Controller
   4. The follow section describes the topics above in detail
      1. Creating the Image Domain Model
         1. Right-Click on the Domain folder, select add and class.
         2. Name this new class Image.cs
         3. Fill in this new class with the following text:

The Image.cs Class

|  |
| --- |
| using System.ComponentModel.DataAnnotations.Schema;  namespace NZWalks.API.Models.Domain  {  public class Image  {  public Guid Id { get; set; }  [NotMapped] // Denotes ‘IformFile File’ is excluded from database mapping since this file will be stored locally on the laptop.  Public IformFile File { get; set; }  public string fileName { get; set; }  public string? FileDescription { get; set; }  public string FileExtension { get; set; }  public long FileSizeInBytes { get; set; }  public string FilePath { get; set; }  }  } |

* + 1. Adding the Image to NZWalksDbContext
       1. Open the NZWalksDbContext.cs fie
       2. Under the three DbSets, add the Images DbSet:

|  |
| --- |
| public DbSet<Difficulty> Difficulties { get; set; }  public DbSet<Region> Regions { get; set; }  public DbSet<Walk> Walks { get; set; }  **public DbSet<Image> Images { get; set; }** |

* + 1. Rerun EF Core Migrations
       1. Open the Package Manager Console window and run these commands:
          1. Add-Migration “Adding Images Table” -Context “NZWalksDbContext”
          2. Update-Database -Context “NZWalksDbContext”
       2. If you run into issues, you can delete the NZWalksDb database and rerun this.
          1. This will reload everything from scratch.
          2. Just remember to reload all the seed records when you are done.
    2. Add Http Context Accessor to Program.cs
       1. This is required so the application dynamically pull the URL it is being hosted at.
       2. Within the Program.cs file, directly below builder.Services.AddControllers(), add this line:
          1. builder.Services.AddHttpContextAccessor();
    3. Creating the Images folder
       1. This folder will contain our images we wish to display.
       2. Right-Click on the API title (NZWalks.API), select Add and Folder
       3. Name the folder Images
    4. Creating the Image Upload Request DTO
       1. We will need a DTO class to feed the Upload method the necessary field data
       2. This DTO will be named ImageUploadRequestDto.cs
       3. To create this DTO, follow these steps:
          1. Right-Click on the DTO folder, select Add and then Class
          2. Name this DTO ImageUploadRequestDto.cs.
          3. Fill in this file with the following text:

The ImageUploadRequestDto.cs file

|  |
| --- |
| using System.ComponentModel.DataAnnotations;  namespace NZWalks.API.Models.DTO  {  public class ImageUploadRequestDto  {  [Required]  public IformFile File { get; set; }  [Required]  public string FileName { get; set; }  public string? FileDescription { get; set; }  }  } |

* + 1. Creating the Image Repository Interface
       1. This interface will call the implementation class using its abstract methods
       2. To create this interface, follow these steps:
          1. Right-Click on the Repositories folder, select Add and select Class.
          2. Select Interface from the Add New Item dialog box, name it IimageRepository.cs and click the Add button.
          3. File in this file with the following text:

The IimageRepository.cs Interface

|  |
| --- |
| using NZWalks.API.Models.Domain;  namespace NZWalks.API.Repositories  {  public interface IimageRepository  {  Task<Image> Upload(Image image);  }  } |

* + 1. Creating the Local Image Repository
       1. This will be the implementation class for the IimageRepository.cs file
       2. To create this class, follow these steps
          1. Right-Click on the Repositories folder, select Add and select Class.
          2. Select class from the Add New Item dialog box, name it LocalImageRepository.cs and click the Add button.
          3. Fill in this file with the following text

The LocalImageRepository.cs File

|  |
| --- |
| using NZWalks.API.Data;  using NZWalks.API.Models.Domain;  namespace NZWalks.API.Repositories  {  public class LocalImageRepository : IimageRepository  {  private readonly IwebHostEnvironment webHostEnvironment;  private readonly IhttpContextAccessor httpContextAccessor;  private readonly NZWalksDbContext dbContext;  public LocalImageRepository(IwebHostEnvironment webHostEnvironment, IhttpContextAccessor httpContextAccessor, NZWalksDbContext dbContext)  {  this.webHostEnvironment = webHostEnvironment;  this.httpContextAccessor = httpContextAccessor;  this.dbContext = dbContext;  }  public async Task<Image> Upload(Image image)  {  var localFilePath = Path.Combine(webHostEnvironment.ContentRootPath, “Images”,$”{image.FileName}{image.FileExtension}”);  // Upload Image to Local Path  using var stream = new FileStream(localFilePath, FileMode.Create);  await image.File.CopyToAsync(stream);  // <https://localhost:1234/images/image.jpg>  var urlFilePath = $”{httpContextAccessor.HttpContext.Request.Scheme}://{httpContextAccessor.HttpContext.Request.Host}{httpContextAccessor.HttpContext.Request.PathBase}/Images/{image.FileName}{image.FileExtension}”;  image.FilePath = urlFilePath;  // Add Image to the Images table  await dbContext.Images.AddAsync(image) ;  await dbContext.SaveChangesAsync();  return image;  }  }  } |

* + 1. Inject IimageRepository Scoped Services into Program.cs
       1. Below the other three builder.Services.AddScoped lines, add this line:
          1. builder.Services.AddScoped<IimageRepository, LocalImageRepository>();
    2. Creating the Image Controller
       1. To add the Image controller, follow these steps:
          1. Right-Click on the Controller folder, select add and select Controller.
          2. In the Add New Scaffolded Item window, select API in the left pane and Add Controller – Empty in the center pane, then click the Add button.
          3. Name this controller ImagesController.cs
          4. Fill in this file with the following test:

The ImagesController.cs Class

|  |
| --- |
| using Microsoft.AspNetCore.Http;  using Microsoft.AspNetCore.Mvc;  using NZWalks.API.Models.Domain;  using NZWalks.API.Models.DTO;  using NZWalks.API.Repositories;  namespace NZWalks.API.Controllers  {  [Route("api/**[controller]**")]  [ApiController]  public class ImagesController : ControllerBase  {  private readonly IImageRepository imageRepository;  public ImagesController(IImageRepository imageRepository)  {  this.imageRepository = imageRepository;  }  // POST: /api/Images/Upload  [HttpPost]  [Route("Upload")]  public async Task<IActionResult> Upload([FromForm] ImageUploadRequestDto request)  {  ValidateFileUpload(request);  if (ModelState.IsValid)  {  // convert DTO to Domain model  var imageDomainModel = new Image  {  File = request.File,  FileExtension = Path.GetExtension(request.File.FileName),  FileSizeInBytes = request.File.Length,  FileName = request.FileName,  FileDescription = request.FileDescription,  };  // User repository to upload image  await imageRepository.Upload(imageDomainModel);  return Ok(imageDomainModel);  }  return BadRequest(ModelState);  }  private void ValidateFileUpload(ImageUploadRequestDto request)  {  var allowedExtensions = new string[] { ".jpg", ".jpeg", ".png" };  if (!allowedExtensions.Contains(Path.GetExtension(request.File.FileName)))  {  ModelState.AddModelError("file", "Unsupported file extension");  }  if (request.File.Length > 10485760)  {  ModelState.AddModelError("file", "File size more than 10MB, please upload a smaller size file.");  }  }  }  } |

* 1. Now the code is in place to upload a file.
  2. Run the application and in Swagger in the Images controller, open the Upload POST method and upload a file.
  3. If the file meets the upload requirements, it will appear in the Images folder.

1. [Serving Static Files through ASP.NET](#TS30)
   1. We can now upload images to ASP.NET and store them.
   2. Unfortunately ASP.NET is unable to serve static files by default like .CSS, Image files and HTML.
      1. If we look at the FilePath field in the Images table we see this entry:
         1. [https://localhost:7257/Images/Internet Icon.jpg](https://localhost:7257/Images/Internet%20Icon.jpg)
      2. This URL will not display the image
   3. To fix this issue, we will modify the Program.cs files using these steps.
      1. Open the Program.cs file
      2. Put this line just above the app.MapControllers() line:

|  |
| --- |
| app.UseStaticFiles(new StaticFileOptions  {  FileProvider = new PhysicalFileProvider(Path.Combine(Directory.GetCurrentDirectory(),"Images")),RequestPath = "/Images"  }); |

* 1. Now if we upload the same image and display the URL it works:
     1. <https://localhost:7257/Images/Internet%20Icon.jpg>

1. [Logging in ASP.NET](#TS31)
   1. To enable logging in ASP.NET, we will use Serilog.
   2. Ensure these four packages are loaded via NuGet in your application
      1. Serilog
      2. Serilog.AspNetCore
      3. Serilog.Sinks.Console
      4. Serilog.Sinks.File
   3. Here are the steps for setting up logging:
      1. Inject Serilog in Application
      2. Inject Logger in Regions Controller
   4. The follow section describes the topics above in Detail:
      1. Inject Serilog in Application
         1. Open the Program.cs file
         2. Place this statement directly below the var builder = WebApplication.CreateBuilder(args) statement:

|  |
| --- |
| var logger = new LoggerConfiguration()  .WriteTo.Console()  .MinimumLevel.Information()  .CreateLogger();  builder.Logging.ClearProviders();  builder.Logging.AddSerilog(logger); |

* + 1. Inject Logger in Regions Controller
       1. Add Logger into the constructor:

|  |
| --- |
| private readonly NZWalksDbContext dbContext;  private readonly IRegionRepository regionRepository;  private readonly IMapper mapper;  **private readonly ILogger<RegionsController> logger;**  public RegionsController(NZWalksDbContext dbContext, IRegionRepository regionRepository,IMapper mapper,**ILogger<RegionsController> logger**)  {  this.dbContext = dbContext;  this.regionRepository = regionRepository;  this.mapper = mapper;  **this.logger = logger;**  } |

* + 1. Add logging to the GetAll method
       1. Below in the GetAllAsync method with logging applied:

|  |
| --- |
| [HttpGet]  [Authorize(Roles = "Reader")]  public async Task<IActionResult> GetAllAsync()  {  logger.LogInformation("GetAllAsync Regions method was invoked");  var regionsDomain = await regionRepository.GetAllAsync();  var regionsDto = mapper.Map<List<RegionDto>>(regionsDomain);  logger.LogInformation($"Logged this information: {JsonSerializer.Serialize(regionsDomain)}");  return Ok(regionsDto);  } |

* + 1. Run the application and view console window
       1. Now we will execute the GetAll() method in the Regions controller:

|  |
| --- |
| [13:28:57 INF] Route matched with {action = "GetAll", controller = "Regions"}. Executing controller action with signature System.Threading.Tasks.Task`1[Microsoft.AspNetCore.Mvc.IActionResult] GetAll() on controller NZWalks.API.Controllers.RegionsController (NZWalks.API).  [13:28:57 WRN] You do not have a valid license key for the Lucky Penny software AutoMapper. This is allowed for development and testing scenarios. If you are running in production you are required to have a licensed version. Please visit https://luckypennysoftware.com to obtain a valid license.  [13:28:57 INF] Executing action method NZWalks.API.Controllers.RegionsController.GetAll (NZWalks.API) - Validation state: Valid  [13:28:57 INF] GetAllAsync Regions method was invoked  [13:28:58 INF] Executed DbCommand (47ms) [Parameters=[], CommandType='Text', CommandTimeout='30']  SELECT [r].[Id], [r].[Code], [r].[Name], [r].[RegionImageUrl]  FROM [Regions] AS [r]  [13:28:59 INF] Logged this information: [{"Id":"906cb139-415a-4bbb-a174-1a1faf9fb1f6","Code":"NSN","Name":"Nelson","RegionImageUrl":"https://images.pexels.com/photos/13918194/pexels-photo-13918194.jpeg?auto=compress\u0026cs=tinysrgb\u0026w=1260\u0026h=750\u0026dpr=1"},{"Id":"f7248fc3-2585-4efb-8d1d-1c555f4087f6","Code":"AKL","Name":"Auckland","RegionImageUrl":"https://images.pexels.com/photos/5169056/pexels-photo-5169056.jpeg?auto=compress\u0026cs=tinysrgb\u0026w=1260\u0026h=750\u0026dpr=1"},{"Id":"14ceba71-4b51-4777-9b17-46602cf66153","Code":"BOP","Name":"Bay Of Plenty","RegionImageUrl":null},{"Id":"6884f7d7-ad1f-4101-8df3-7a6fa7387d81","Code":"NTL","Name":"Northland","RegionImageUrl":null},{"Id":"f077a22e-4248-4bf6-b564-c7cf4e250263","Code":"STL","Name":"Southland","RegionImageUrl":null},{"Id":"cfa06ed2-bf65-4b65-93ed-c9d286ddb0de","Code":"WGN","Name":"Wellington","RegionImageUrl":"https://images.pexels.com/photos/4350631/pexels-photo-4350631.jpeg?auto=compress\u0026cs=tinysrgb\u0026w=1260\u0026h=750\u0026dpr=1"}]  [13:28:59 INF] Executed action method NZWalks.API.Controllers.RegionsController.GetAll (NZWalks.API), returned result Microsoft.AspNetCore.Mvc.OkObjectResult in 1071.4475ms.  [13:28:59 INF] Executing OkObjectResult, writing value of type 'System.Collections.Generic.List`1[[NZWalks.API.Models.DTO.RegionDto, NZWalks.API, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null]]'.  [13:28:59 INF] Executed action NZWalks.API.Controllers.RegionsController.GetAll (NZWalks.API) in 1274.3415ms  [13:28:59 INF] Executed endpoint 'NZWalks.API.Controllers.RegionsController.GetAll (NZWalks.API)'  [13:28:59 INF] Request finished HTTP/2 GET https://localhost:7257/api/Regions - 200 null application/json; charset=utf-8 1378.7248ms |

* + - 1. As you can see, the items in red are the logging information we injected in the Regions GetAll() method.
  1. Changing the Minimum logging level in Program.cs
     1. We can change the logging level from information to Warning if we want less logging:

|  |
| --- |
| var logger = new LoggerConfiguration()  .WriteTo.Console()  .**MinimumLevel.Warning()**  .CreateLogger();  builder.Logging.ClearProviders();  builder.Logging.AddSerilog(logger); |

* + 1. Logging warning in the GetAll() Method.
       1. We can also log warning messages like this:
          1. logger.LogWarning(“This is a warning message”);
          2. logger.LogError(“This is an error message”);
    2. Using logging in a try/catch block
       1. Logging is especially in Try/Catch as we can customize our error messages:

|  |
| --- |
| try  {  Some Code Here  }  catch(Exception ex)  {  Logger.LogError(ex, ex.Message) ;  } |

* 1. Logging information to a text file
     1. First we need to create the Logs folder directly under the API.
     2. Next we need to update our logger in the Program.cs file to add writing to a file:

|  |
| --- |
| var logger = new LoggerConfiguration()  .WriteTo.Console()  **.WriteTo.File("Logs/NZWalks\_Log.txt",rollingInterval: RollingInterval.Day)**  .MinimumLevel.Warning()  .CreateLogger();  builder.Logging.ClearProviders();  builder.Logging.AddSerilog(logger); |

* + 1. Now logging will create a new folder every day at midnight to log to.

1. [Implementing Global Exception Handling](#TS32)
   1. We could set up logging for each method, but that would be a lot of repetitive code and it would make our code look chunky.
   2. There is a better way to log events that doesn’t involve individual method logging.
      1. There are 2 steps in setting up Global Logging.
         1. Creating the ExceptionHandlerMiddleware class
         2. Inject Middle into Middle Pipeline
      2. Below are the detailed steps to set up Globel Logging
         1. Creating the ExceptionHandlerMiddleware class
            1. Create a new folder under the API called Middlewares.
            2. Inside this folder create a new class called ExceptionHandlerMiddleware.cs
            3. Place the following text in the class file:

The ExceptionHandlerMiddleware.cs File

|  |
| --- |
| using System.Net;  namespace NZWalks.API.Middlewares  {  public class ExceptionHandlerMiddleware  {  private readonly ILogger<ExceptionHandlerMiddleware> logger;  private readonly RequestDelegate next;  public ExceptionHandlerMiddleware(ILogger<ExceptionHandlerMiddleware> logger, RequestDelegate next)  {  this.logger = logger;  this.next = next;  }  public async Task InvokeAsync(HttpContext httpContext)  {  try  {  await next(httpContext);  }  catch (Exception ex)  {  var errorId = Guid.NewGuid();  // Log this exception  logger.LogError(ex,$"{errorId} : {ex.Message}");  //Return a custon error response  httpContext.Response.StatusCode = (int)HttpStatusCode.InternalServerError;  httpContext.Response.ContentType = "application/json";  var error = new  {  Id = errorId,  ErrorMessage = "Something went wrong. IT staff is investigating."  };  await httpContext.Response.WriteAsJsonAsync(error);  }  }  }  } |

* + - 1. Inject Middle into Middle Pipeline
         1. Open the Program.cs file
         2. Place this line just above the app.UseHttpsRedirection() line:

app.UseMiddleware<ExceptionHandlerMiddleware>();

* 1. Now, no matter where the error occurs, it will be logged in accordance to the Logger Configuration in Program.cs

1. Overview of the UI
   1. In this section we will be creating a User Interface (UI) to consume the API we have just created.
   2. Visit <https://learn.microsoft.com/en-us/dotnet/api/system.net.http.httpclient> to learn more about the HttpClient Class.
2. Creating the UI Regions Controller.
   1. In this section we will be creating the Regions Controller and the index.cshtml file
   2. Creating the RegionsController.cs file:
      1. In the UI section, right-click on the Controllers folder, select new and then Controller.
      2. In the Add New Scaffolded Item box, select MVC => Controller on the left and MVC Controller – Empty in the center box.
      3. Name the controller RegionsController.cs and click the Add button.
      4. The RegionsController.cs file will now appear on the screen.
   3. Creating the index.cshtml view
      1. Note: This index.cshtml file will be used to pull information from the Regions Controller in the API.
      2. In the text of the RegionsController.cs file, there is a line that reads return View()
      3. Right-Click on the word View() and select Add View from the selection box.
      4. In the Add New Scaffolded Item box, select Razor View -Empty and click the Add button.
      5. Make the name of this View index.cshtml and click the Add button.
      6. The index.cshtml file is now created.
3. Injecting the HTTP Client Factory
   1. This will inject the HTTP client into the Program.cs file
   2. To accomplish this, follow these steps:
      1. Open the Program.cs file within the UI project. (Not the Program.cs file in the API)
      2. Directly above the var app = builder.Build() line, enter this line:
         1. builder.Services.AddHttpClient();
      3. Save the Program.cs file
4. Creating the GetAll Client Request
   1. Now we will populate the RegionsController.cs and index.cshtml files to show the raw Json data from the Regions table.:
      1. I want to note here that we will be using ViewBag.
      2. This is just a utility to show the contents of a variable. It’s a bit like print variable in PHP just to see what is in it.
      3. We will be replacing ViewBag when we put together our formal display, but for now we just want to see the contents of the Region Json data.
      4. One thing to note concerning the RegionsController.cs file, which being the IHttpClientFactory.
         1. In the Program.cs file we added the HttpClient services:
         2. In the RegionsController.cs file, we inject the Http Client Factory into the constructor as shown below:

|  |
| --- |
| private readonly IHttpClientFactory httpClientFactory;  public RegionsController(IHttpClientFactory httpClientFactory)  {  this.httpClientFactory = httpClientFactory;  } |

* + - 1. We then create the client object using the httpClientFactory object:
         1. var client = httpClientFactory.CreateClient();
    1. Below are the two files and the output from the program run:

The RegionsController.cs File

|  |
| --- |
| namespace NZWalks\_UI.Controllers  {  public class RegionsController : Controller  {  private readonly IHttpClientFactory httpClientFactory;  public RegionsController(IHttpClientFactory httpClientFactory)  {  this.httpClientFactory = httpClientFactory;  }  public async Task<IActionResult> Index()  {  try  {  // Get all Regions from Web API  var client = httpClientFactory.CreateClient();  // Call the Regions controller in the API. Get this URL (https://localhost:7257) from appsettings.json in the API.  var httpResponseMessage = await client.GetAsync("https://localhost:7257/api/regions");  // If the request to the API is not successful, this line will throw an exception  httpResponseMessage.EnsureSuccessStatusCode();  var stringResponseBody = await httpResponseMessage.Content.ReadAsStringAsync();  // ViewBag is just a utility to show the raw data. It will only show the Json data.  // We will need to build additional code to nicely format the data.  ViewBag.Response = stringResponseBody;  // return Ok(httpResponseMessage);  }  catch (Exception ex)  {  // Log the exception  }  return View();  }  }  } |

The index.cshtml File

|  |
| --- |
| @\*  For more information on enabling MVC for empty projects, visit https://go.microsoft.com/fwlink/?LinkID=397860  \*@  @{  }  <h1 class="mt-3">Regions</h1>  @if(ViewBag.Response is not null)  {  <p>@ViewBag.Response</p>  } |

Output From the index.cshtml File

|  |
| --- |
|  |

* 1. Creating an object from the data.
     1. The issue with this setup is that the data is coming back as a string.
     2. We need to deserialized the data and place it into an object so the individual fields can be pulled and displayed how we want them.
     3. The following steps accomplish this task:
        1. Within the UI project, create a DTO folder inside the Models folder.
        2. Right-Click on the DTO folder and create a new class named RegionDto.cs
        3. Copy the four properties from RegionDto.cs file in the API and paste them into the RegionDto.cs file in the UI project.

The UI RegionDto.cs File

|  |
| --- |
| namespace NZWalks\_UI.Models.DTO  {  public class RegionDto  {  public Guid Id { get; set; }  public string Code { get; set; }  public string Name { get; set; }  public string? RegionImageUrl { get; set; }  }  } |

* 1. Now we will see the formalized versions of the RegionsController.cs and index.cshtml files showing how to list the regions rows

The RegionsController.cs File

|  |
| --- |
| using Microsoft.AspNetCore.Mvc;  using NZWalks\_UI.Models.DTO;  namespace NZWalks\_UI.Controllers  {  public class RegionsController : Controller  {  private readonly IHttpClientFactory httpClientFactory;  public RegionsController(IHttpClientFactory httpClientFactory)  {  this.httpClientFactory = httpClientFactory;  }  [HttpGet]  public async Task<IActionResult> Index()  {  List<RegionDto> response = new List<RegionDto>();  try  {  // Get all Regions from Web API  var client = httpClientFactory.CreateClient();  // Call the Regions controller in the API. Get this URL (https://localhost:7257) from appsettings.json in the API.  var httpResponseMessage = await client.GetAsync("https://localhost:7257/api/regions?sortBy=Name");  // If the request to the API is not successful, this line will throw an exception  httpResponseMessage.EnsureSuccessStatusCode();  response.AddRange(await httpResponseMessage.Content.ReadFromJsonAsync<IEnumerable<RegionDto>>());  }  catch (Exception ex)  {  // Log the exception  }  return View(response);  }  }  } |

The index.cshtml File

|  |
| --- |
| @using NZWalks\_UI.Models.DTO  @model IEnumerable<NZWalks\_UI.Models.DTO.RegionDto>  @{  }  <h1 class="mt-3">Regions</h1>  <table width="'100%'">  <thead>  <tr>  <th width='30%'>Id</th>  <th width='10%'>&nbsp</th>  <th width='10%'>Code</th>  <th width='10%'>&nbsp</th>  <th width='20%'>Name</th>  <th width='20%'>&nbsp</th>  </tr>  </thead>  <tbody>    @foreach (var region in Model)  {  <tr>  <td width='30%'>@region.Id</td>  <th width='10%'>&nbsp</th>  <td width='10%'>@region.Code</td>  <th width='10%'>&nbsp</th>  <td width='20%'>@region.Name</td>  <th width='20%'>&nbsp</th>  </tr>  }  </tbody>  </table> |

Output From the index.cshtml File

|  |
| --- |
|  |

1. Adding a Region using POST
   1. This process can be a bit more complicated than expected to let me break it down for you.
      1. There are three files that come into play with this process:
         1. RegionsController.cs
         2. Add.cshtml
         3. AddRegionViewModel.cs
      2. These files are discussed in detail below:
         1. RegionsController.cs
            1. Regarding the process of inserting a new Regions record, there are two methods involved; both of them named Add().
            2. The first Add() method uses HttpGet
            3. The Second Add() method uses HttpPost.

This sends the AddRegionViewModel.cs object (Carrying the data) to the API for processing.

* + - 1. Add.cshtml
         1. This HTML file draws the labels and input boxes on the screen for entering data as well as the Save button.
         2. Each input statement contains an asp-for=”*Field Name*” line.
         3. At the top of the file is the line: @model NZWalks.UI.Models.AddRegionViewModel

This command tells the asp-for statement to look into AddRegionViewModel for the corresponding fields so they can be bound together.

If there is a mismatch between the spelling for the Field Names in the asp-for=”Field Name” and those in AddRegionViewModel, you will get an error.

* + - * 1. Upon clicking the Save button, the AddRegionViewModel is instantiated with the inputted values and is hanging out in memory, ready to get processed by the POST Add() method.
      1. AddRegionViewModel.cs
         1. This class is the linkage between the HttpGet Add() method and the HttpPost Add() method.
    1. How the process works
       1. When the Add button is click in the main screen, the HttpGet Add() method is called.
       2. It spawns the Add.cshtml file to display the three input lines along with the submit button on the screen.
       3. Once the data is entered for the new Regions record and the Save button clicked, the **form** method="post"> line calls the HttpPost Add() method.
       4. The HttpPost Add() does the following:
          1. It creates a HTTP Client Factory object named client.
          2. The httpRequestMessage then binds all the necessary input to send to the API.

This is much line binding the Context Type, Method and Body inside the @Param variable in PowerShell.

The RequestUri = new Uri("https://localhost:7257/api/regions"), is key as this targets the API Regions Post method for processing.

* + - 1. Next, the method takes the AddRegionViewModel object, sends it to the API for processing.
      2. The response variable receives HTTP response data from the RegionDto class which now houses the processed data.
         1. When this response is received, the record has already been written by the API Region Create() method.
      3. Finally, the line return RedirectToAction("Index", "Regions") activated the Index method which displays all the current region records, including the one just added.
    1. Below are the three files listed above in their entirety