**.NET Core RESTFUL API – Parky Project**: <https://www.udemy.com/course/quick-introduction-to-aspnet-mvc-core-20/>

**Project Setup - .Net Core 3.1 API**

1. Create a new **Models** folder inside our solution then inside the folder create a new **class** called **NationalPark** then add the following properties:

[Key]

public int Id { get; set; }

[Required]

public string Name { get; set; }

[Required]

public string State { get; set; }

public DateTime Created { get; set; }

public byte[] Picture { get; set; }

public DateTime Established { get; set; }

1. **NOTE**: The [Key] attribute determines which property is the primary key in our database.

**Connecting to a Database**

1. Prepare your database of use (Sqlite, MySql or SQL Server) and make sure it’s running.
2. In the **appsetting**.json add the following setting as a connection string to connect your preferred database:

"ConnectionStrings": {

"DefaultConnection": "Your Database Connection String of preference"

},

1. Create a new folder called **Data** in our solution, then create a new class called **ApplicationDBContext.** This class will be responsible for interacting directly with our database.
   1. In our new class inherit from the class **DbContext**, install the **Nuget** packages if necessary.
   2. Add a constructor with a parameter that will serve as the base option provider.
   3. Connection our **NationalPark** model as a **DbSet**<>

public class ApplicationDBContext : DbContext

{

public ApplicationDBContext(DbContextOptions<ApplicationDBContext> options) : base(options)

{

}

public DbSet<NationalPark> NationalParks { get; set; }

}

1. Install the Nuget Package **Microsoft.EntityFrameworkCore.YourDatabaseNameOfPreference.**
2. In the **startup**.cs file add the following line under the ConfigureServices() method to inject our Database services into our application:

services.AddDbContext<ApplicationDBContext>(options => options.**UseYourDatabaseOfPreference**(Configuration.GetConnectionString("DefaultConnection")));

1. Now we need to add the model as a table in our database, in order to do this, we need the command line to add the migration. Install the **Nuget** package **Microsoft.EntityFrameworkCore.Tools** to get access to the migration commands.
2. Now use the following commands to add a migration:
   1. **add-migration YourMigrationName**
   2. Use the command **update-database** to create our database and table.

**National Park Repositories**

1. Create a new folder called **Repository**, then create an **Interface** called **INationalParkRepository** and a class called **NationalParkRepository**.
2. In the **INationalParkRepository** we will add the methods that will have access to perform **CRUD** operations on our national parks from our database:

public interface INationalParkRepository

{

ICollection<NationalPark> GetNationalPark();

NationalPark GetNationalPark(int nationalParkId);

bool NationalParkExists(string name);

bool NationalParkExists(int id);

bool CreateNationalPark(NationalPark nationalPark);

bool UpdateNationalPark(NationalPark nationalPark);

bool DeleteNationalPark(NationalPark nationalPark);

bool Save();

}

1. Implement the methods added in I**NationalParkRepository** to **NationalParkRepository.**
2. In our **NationalParkRepository** add our ApplicationDBContext as a private read only member variable, then create a constructor to hook up our database via **Dependency** **Injection**.

private readonly ApplicationDBContext \_db;

public NationalParkRepository(ApplicationDBContext db)

{

\_db = db;

}

1. Implement the code to perform **CRUD** operations in our national parks to the database:

public bool CreateNationalPark(NationalPark nationalPark)

{

\_db.NationalParks.Add(nationalPark);

return Save();

}

public bool DeleteNationalPark(NationalPark nationalPark)

{

\_db.NationalParks.Remove(nationalPark);

return Save();

}

public ICollection<NationalPark> GetNationalPark()

{

return \_db.NationalParks.OrderBy(a => a.Name).ToList();

}

public NationalPark GetNationalPark(int nationalParkId)

{

return \_db.NationalParks.FirstOrDefault(a => a.Id == nationalParkId);

}

public bool NationalParkExists(string name)

{

bool value = \_db.NationalParks.Any(a => a.Name.ToLower().Trim() == name.ToLower().Trim());

return value;

}

public bool NationalParkExists(int id)

{

bool value = \_db.NationalParks.Any(a => a.Id == id);

return value;

}

public bool Save()

{

return \_db.SaveChanges() >= 0 ? true : false;

}

public bool UpdateNationalPark(NationalPark nationalPark)

{

\_db.NationalParks.Update(nationalPark);

return Save();

}

1. Now we need to let our application know about our new **Repository** in order to perform **CRUD** operations by using **Dependency** **Injection**. In **startup**.cs, under the **ConfigureServices()** method, add the following service to implement dependency injection on our new Repository:
   1. services.**AddScoped**<**INationalParkRepository**, **NationalParkRepository**>();

**National Park Data Transfer Objects**

1. Create a new folder called **Dtos**, then add a new class with the following properties:

public class NationalParkDto

{

public int Id { get; set; }

public string Name { get; set; }

public string State { get; set; }

public DateTime Created { get; set; }

public byte[] Picture { get; set; }

public DateTime Established { get; set; }

}

1. In order to transform out model into a **Dto**, we’ll use a third-party package called **AutoMapper**. Install the following **Nuget** packages:
   1. AutoMapper
   2. AutoMapper.Extensions.Microsoft.DependencyInjection
2. Create a new folder called **ParkyMapper**, then add a new class called **ParkyMappings**. Map the model to the dto by adding the following line of code:
   1. CreateMap<NationalPark, NationalParkDto>().ReverseMap();
3. Now we need to let our application know about the **Dto** mapping service we just added by adding the following line of code in the **startup**.cs file, under the **ConfigureServices():**
   1. services.AddAutoMapper(typeof(ParkyMappings));

**National Park Controllers**

1. Create a new empty controller in the **Controllers** folder.
2. In order to setup the client to make **API** calls we will need to bring in out **INationalParkRepository** and our **IMapper** to let the controller know what data from the model it can pass back to the client.
3. Set up a constructor to inject both of our private members **INationalParkRepository** and **IMapper:**

private readonly INationalParkRepository \_npRepo;

private readonly IMapper \_mapper;

public NationalParksController(INationalParkRepository npRepo, IMapper mapper)

{

\_npRepo = npRepo;

\_mapper = mapper;

}

1. Now we will add our first **GET** request to retrieve all the parks from our database:

[HttpGet]

public IActionResult GetNationalParks()

{

var objList = \_npRepo.GetNationalPark();

//This will ensure we send our Dto instead of the model

var objDto = new List<NationalParkDto>();

foreach (var obj in objList)

objDto.Add(\_mapper.Map<NationalParkDto>(obj));

return Ok(objDto);

}

* 1. Test in Postman using the API <https://localhost:YOUR_PORT/api/nationalparks>

1. To retrieve a single national park, create the following **GET** method:

[HttpGet("{nationalParkId:int}")]

public IActionResult GetNationalPark(int nationalParkId)

{

var obj = \_npRepo.GetNationalPark(nationalParkId);

//This will ensure we send our Dto instead of the model

var objDto = new List<NationalParkDto>();

if (obj == null)

return NotFound();

objDto.Add(\_mapper.Map<NationalParkDto>(obj));

return Ok(objDto);

}

1. Test in Postman using the API <https://localhost:YOUR_PORT/api/nationalparks/1>
2. **NOTE**: In order to have multiple **GET** requests in a controller without getting an “Ambiguous Call” error, one of the **[HttpGET] Attributes** must have a parameter, such as **[HttpGet("{nationalParkId:int}")]**
3. In order to **POST** a new National Park to our database, add the following method:

[HttpPost]

public IActionResult CreateNationalPark([FromBody] NationalParkDto nationalParkDto)

{

//ModelState contains all related encountered errors

if (nationalParkDto == null)

return BadRequest(ModelState);

if (\_npRepo.NationalParkExists(nationalParkDto.Name))

{

//Give the ModelState a message and an error code 404 to return to the client

ModelState.AddModelError("", "National Park Exists!");

return StatusCode(404, ModelState);

}

if (!ModelState.IsValid)

{

return BadRequest(ModelState);

}

//Map parameter Dto to the Model

var nationalParkObj = \_mapper.Map<NationalPark>(nationalParkDto);

//If creation was unsuccessful

if (!\_npRepo.CreateNationalPark(nationalParkObj))

{

ModelState.AddModelError("", $"Something went wrong when saving the record {nationalParkObj.Name}");

return StatusCode(500, ModelState);

}

return Ok();

}

1. Test in Postman **POST**ing to API <https://localhost:YOUR_PORT/api/nationalparks/1> and passing in the Dto

{

"name": "CA Np",

"state": "CA",

"created": "2020-01-01T00:00:00",

"established": "2000-01-01T00:00:00"

}

1. Currently we’re only returning a 200 OK when an object has been **POST**ed to the database. In order to return the object that has been created, add the following to the [**HttpGet]** attribute, above the **GetNationalPark**(**int**) method:
   1. [HttpGet("{nationalParkId:int}", Name = "GetNationalPark")]
2. Under the **CreateNationalPark** method, replace the line return Ok(); with:
   1. return CreatedAtRoute("GetNationalPark", new { nationalParkId = nationalParkObj.Id }, nationalParkObj);
3. In order to update an existing record, add the following **PATCH** method:

[HttpPatch("{nationalParkId:int}", Name = "UpdateNationalPark")]

public IActionResult UpdateNationalPark(int nationalParkId, [FromBody] NationalParkDto nationalParkDto)

{

//ModelState contains all related encountered errors

if (nationalParkDto == null || nationalParkId != nationalParkDto.Id)

return BadRequest(ModelState);

var nationalParkObj = \_mapper.Map<NationalPark>(nationalParkDto);

//If update was unsuccessful

if (!\_npRepo.UpdateNationalPark(nationalParkObj))

{

ModelState.AddModelError("", $"Something went wrong when updating the record {nationalParkObj.Name}");

return StatusCode(500, ModelState);

}

return NoContent();

}

1. Test in Postman **PATCH**ing to API <https://localhost:44385/api/nationalparks/4> and using the **Dto** of an existing object, example:

{

"id": 4,

"name": "CA Np NEW",

"state": "CA",

"created": "2020-01-01T00:00:00",

"picture": "",

"established": "2000-01-01T00:00:00"

}

1. In order to **DELETE** an object from the database by providing an id, implement the following method:

[HttpDelete("{nationalParkId:int}", Name = "DeleteNationalPark")]

public IActionResult DeleteNationalPark(int nationalParkId)

{

//ModelState contains all related encountered errors

if (!\_npRepo.NationalParkExists(nationalParkId))

return NotFound();

var nationalParkObj = \_npRepo.GetNationalPark(nationalParkId);

//If update was unsuccessful

if (!\_npRepo.DeleteNationalPark(nationalParkObj))

{

ModelState.AddModelError("", $"Something went wrong when deleting the record {nationalParkObj.Name}");

return StatusCode(500, ModelState);

}

return NoContent();

}

1. Test in Postman using a **DELETE** on API<https://localhost:44385/api/nationalparks/3>

**API Documentation with Swashbuckle Nuget Package**

1. Install **Nuget** package **Swashbuckle.AspNetCore.**
2. To use the new **service**, add it as a service under the **CofigureServices()** method in **startup**.cs

services.AddSwaggerGen(options => {

options.SwaggerDoc("ParkyOpenAPISpec", new Microsoft.OpenApi.Models.OpenApiInfo()

{

Title = "Parky API",

Version = "1"

});

});

* 1. In the object above you can also add other properties, such as **Description**, **Contact**, **License**, among others.
  2. Go to URI <https://localhost:44385/swagger/ParkyOpenAPISpec/swagger.json> to test.

1. Now we need to deploy the service by adding it to the request pipeline. Under the **Configure()** method, after the conditional that checks if the app is running in development or production add the app.**UseSwagger**(); method call.
2. Now to add a UI to our API documentation, add the following code to the request pipeline:

app.UseSwaggerUI(options => {

options.SwaggerEndpoint("/swagger/ParkyOpenAPISpec/swagger.json", "Parky API");

});

* 1. Go to URI <https://localhost:44385/swagger/index.html> to test.

1. To make the UI run on **startup**, expand **Properties** in the **solution** **explorer**, open the **launchSettings**.json and remove "weatherforecast" from the **launchUrl** setting, the setting should look like "launchUrl":"", then change the app.**UseSwaggerUI()** method under the **Configure**() method to look like the following:

app.**UseSwaggerUI**(options => {

options.SwaggerEndpoint("/swagger/ParkyOpenAPISpec/swagger.json", "Parky API");

options.RoutePrefix = "";

});

1. Test by launching the application.
2. Now that we have a UI for our APIs, we’ll need to add descriptions for each call. To do this first:
   1. Right click in the ParkyAPI project, then click on properties. In the Build section, find the XML documentation file check box and enable it, you can shorten the name of the file if preferred by changing it in the text box.
   2. In the **NationalParksController**, on top of each call, type in 3 slashes (///), this will bring up XML documentation tags to populate each API’s description.

/// <summary>

/// Get a list of all national parks.

/// </summary>

/// <returns></returns>

[HttpGet]

public IActionResult GetNationalParks()

{

var objList = \_npRepo.GetNationalPark();

//This will ensure we send our Dto instead of the model

var objDto = new List<NationalParkDto>();

foreach (var obj in objList)

objDto.Add(\_mapper.Map<NationalParkDto>(obj));

return Ok(objDto);

}

* 1. In the **startup**.cs file, under the **ConfigureServices()** method, change the **services**.**AddSwaggerGen**() method to look like the following:

services.AddSwaggerGen(options => {

options.SwaggerDoc("ParkyOpenAPISpec", new Microsoft.OpenApi.Models.OpenApiInfo()

{

Title = "Parky API",

Version = "1"

});

var xmlCommentFile = $"{Assembly.GetExecutingAssembly().GetName().Name}.xml";

var cmlCommentsFullPath = Path.Combine(AppContext.BaseDirectory, xmlCommentFile);

options.IncludeXmlComments(cmlCommentsFullPath);

});

1. You will notice that when building the project there are dozens of **warnings**, this is because XML documentation now requires you to add a description for all the classes in the project. In order to suppress these warning, right click in the **ParkyAPI** project, then click on properties. In the Build section, find the **Suppress** **warning** option, then add the warning **Code** you would like to suppress, in our case is code **1591**.
2. If you peek at some of the API documentation, you may have noticed that some of the response codes are incorrect. Example, the **NationalParks/{nationalParkId}** should have a return code of **201** instead of **200**. To fix this, we can add some attributes on top of the **Controllers’** methods to specify the objects and codes that the method will return if the call is successful or not. Add the following attribute to the corresponding methods:

[HttpGet]

[ProducesResponseType(200, Type = typeof(List<NationalParkDto>))]

[ProducesResponseType(400)]

public IActionResult GetNationalParks()

[HttpGet("{nationalParkId:int}", Name = "GetNationalPark")]

[ProducesResponseType(200, Type = typeof(NationalParkDto))]

[ProducesResponseType(400)]

[ProducesResponseType(404)]

[ProducesDefaultResponseType]

public IActionResult GetNationalPark(int nationalParkId)

[HttpPost]

[ProducesResponseType(201, Type = typeof(NationalParkDto))]

[ProducesResponseType(400)]

[ProducesResponseType(404)]

[ProducesResponseType(500)]

public IActionResult CreateNationalPark([FromBody] NationalParkDto nationalParkDto)

[HttpPatch("{nationalParkId:int}", Name = "UpdateNationalPark")]

[ProducesResponseType(204)]

[ProducesResponseType(404)]

[ProducesResponseType(500)]

public IActionResult UpdateNationalPark(int nationalParkId, [FromBody] NationalParkDto nationalParkDto)

1. **NOTE**: When you have a **ProcdureReponseType** that is the same across all APIs (Example, [ProducesResponseType(400)]), you can add it at the top of the class name, & it will apply to all methods:

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

[ApiController]

[ProducesResponseType(400)]

public class NationalParksController : ControllerBase

[HttpDelete("{nationalParkId:int}", Name = "DeleteNationalPark")]

[ProducesResponseType(204)]

[ProducesResponseType(404)]

[ProducesResponseType(409)]

[ProducesResponseType(500)]

public IActionResult DeleteNationalPark(int nationalParkId)

**Adding a new Trails model**

1. Create a new **Trail** model in the **Model** folder and add the following properties:

public class Trail

{

[Key]

public int Id { get; set; }

[Required]

public string Name { get; set; }

[Required]

public double Distance { get; set; }

public enum DifficultyType { Easy, Moderate, Difficult, Expert }

public DifficultyType Difficulty { get; set; }

public int NationalParkId { get; set; }

[ForeignKey("NationalParkId")]

public NationalPark NationalPark { get; set; }

public DateTime DateCreated { get; set; }

}

1. Create a new **TrailDto** in the **Dto** folder and add the following properties:

public class TrailDto

{

public int Id { get; set; }

[Required]

public string Name { get; set; }

[Required]

public double Distance { get; set; }

public DifficultyType Difficulty { get; set; }

[Required]

public int NationalParkId { get; set; }

public NationalParkDto NationalPark { get; set; }

}

1. Map the new **Trail** Model to the new **TrailDto** in the **ParkyMappins** file:
   1. CreateMap<Trail, TrailDto>().ReverseMap();

**Creating the Trail Repository**

1. In the **ApplicationDBContext** file under the **Data** folder class add the following **DbSet**<> :
   1. public DbSet<Trail> Trails { get; set; }
2. Create an **Interface** called **ITrailRepository** and a class called **TrailRepository**.
3. In the **ITrailRepository** add the following code :

public interface ITrailRepository

{

ICollection<Trail> GetTrails();

ICollection<Trail> GetTrailsInNationalPark(int npId);

Trail GetTrail(int trailId);

bool TrailExists(string name);

bool TrailExists(int id);

bool CreateTrail(Trail trail);

bool UpdateTrail(Trail trail);

bool DeleteTrail(Trail trail);

bool Save();

}

1. In the **TrailRepository** add the following code :

public class TrailRepository : ITrailRepository

{

private readonly ApplicationDBContext \_db;

public TrailRepository(ApplicationDBContext db)

{

\_db = db;

}

public bool CreateTrail(Trail trail)

{

\_db.Trails.Add(trail);

return Save();

}

public bool DeleteTrail(Trail trail)

{

\_db.Trails.Remove(trail);

return Save();

}

public ICollection<Trail> GetTrails()

{

return \_db.Trails.Include(c => c.NationalPark).OrderBy(a => a.Name).ToList();

}

public Trail GetTrail(int trailId)

{

return \_db.Trails.Include(c => c.NationalPark).FirstOrDefault(a => a.Id == trailId);

}

public bool TrailExists(string name)

{

bool value = \_db.Trails.Any(a => a.Name.ToLower().Trim() == name.ToLower().Trim());

return value;

}

public bool TrailExists(int id)

{

bool value = \_db.Trails.Any(a => a.Id == id);

return value;

}

public bool Save()

{

return \_db.SaveChanges() >= 0 ? true : false;

}

public bool UpdateTrail(Trail trail)

{

\_db.Trails.Update(trail);

return Save();

}

public ICollection<Trail> GetTrailsInNationalPark(int npId)

{

return \_db.Trails.Include(c => c.NationalPark).Where(c => c.NationalParkId == npId).ToList();

}

}

**Trails Controller**

1. In the **startup**.cs file add a new service under the **ConfigureServices()** method to perform **dependency injection** on the **Trails Repository:**
   1. services.AddScoped<ITrailRepository, TrailRepository>();
2. Create a new empty controller under the Controllers folder called **TrailsController** and add the following code:

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

[ApiController]

[ProducesResponseType(400)]

public class TrailsController : ControllerBase

{

private readonly ITrailRepository \_trailRepo;

private readonly IMapper \_mapper;

public TrailsController(ITrailRepository trailRepo, IMapper mapper)

{

\_trailRepo = trailRepo;

\_mapper = mapper;

}

/// <summary>

/// Get a list of all trails.

/// </summary>

/// <returns></returns>

[HttpGet]

[ProducesResponseType(200, Type = typeof(List<TrailDto>))]

[ProducesResponseType(400)]

public IActionResult GetTrails()

{

var objList = \_trailRepo.GetTrails();

//This will ensure we send our Dto instead of the model

var objDto = new List<TrailDto>();

foreach (var obj in objList)

objDto.Add(\_mapper.Map<TrailDto>(obj));

return Ok(objDto);

}

/// <summary>

/// Get individual trail

/// </summary>

/// <param name="trailId">The trail Id</param>

/// <returns></returns>

[HttpGet("{trailId:int}", Name = "GetTrail")]

[ProducesResponseType(200, Type = typeof(TrailDto))]

[ProducesResponseType(404)]

[ProducesDefaultResponseType]

public IActionResult GetTrail(int trailId)

{

var obj = \_trailRepo.GetTrail(trailId);

//This will ensure we send our Dto instead of the model

var objDto = new List<TrailDto>();

if (obj == null)

return NotFound();

objDto.Add(\_mapper.Map<TrailDto>(obj));

return Ok(objDto);

}

[HttpPost]

[ProducesResponseType(201, Type = typeof(TrailDto))]

[ProducesResponseType(404)]

[ProducesResponseType(500)]

public IActionResult CreateTrail([FromBody] TrailDto trailDto)

{

//ModelState contains all related encountered errors

if (trailDto == null)

return BadRequest(ModelState);

if (\_trailRepo.TrailExists(trailDto.Name))

{

//Give the ModelState a message and an error code 404 to return to the client

ModelState.AddModelError("", "Trail Exists!");

return StatusCode(404, ModelState);

}

if (!ModelState.IsValid)

{

return BadRequest(ModelState);

}

//Map parameter Dto to the Model

var trailObj = \_mapper.Map<Trail>(trailDto);

//If creation was unsuccessful

if (!\_trailRepo.CreateTrail(trailObj))

{

ModelState.AddModelError("", $"Something went wrong when saving the record {trailObj.Name}");

return StatusCode(500, ModelState);

}

return CreatedAtRoute("GetTrail", new { trailId = trailObj.Id }, trailObj);

}

[HttpPatch("{trailId:int}", Name = "UpdateTrail")]

[ProducesResponseType(204)]

[ProducesResponseType(404)]

[ProducesResponseType(500)]

public IActionResult UpdateTrail(int trailId, [FromBody] TrailDto trailDto)

{

//ModelState contains all related encountered errors

if (trailDto == null || trailId != trailDto.Id)

return BadRequest(ModelState);

var trailObj = \_mapper.Map<Trail>(trailDto);

//If update was unsuccessful

if (!\_trailRepo.UpdateTrail(trailObj))

{

ModelState.AddModelError("", $"Something went wrong when updating the record {trailObj.Name}");

return StatusCode(500, ModelState);

}

return NoContent();

}

[HttpDelete("{trailId:int}", Name = "DeleteTrail")]

[ProducesResponseType(204)]

[ProducesResponseType(404)]

[ProducesResponseType(409)]

[ProducesResponseType(500)]

public IActionResult DeleteTrail(int trailId)

{

if (!\_trailRepo.TrailExists(trailId))

return NotFound();

var trailObj = \_trailRepo.GetTrail(trailId);

//If update was unsuccessful

if (!\_trailRepo.DeleteTrail(trailObj))

{

ModelState.AddModelError("", $"Something went wrong when deleting the record {trailObj.Name}");

return StatusCode(500, ModelState);

}

return NoContent();

}

}

1. If you peek at the Trails documentation, you’ll notice that The **POST** and **PATCH** APIs have an incorrect definition of the object that will be passed in, the **Trail** object parameter should include the National Park ID instead of the entire National Park object. To fix this we will need to **create a new Dto**.
2. Create a new **Dto** named **TrailCreateDto** to use on the **POST** method, then add the following properties:

public class TrailCreateDto

{

[Required]

public string Name { get; set; }

[Required]

public double Distance { get; set; }

public DifficultyType Difficulty { get; set; }

[Required]

public int NationalParkId { get; set; }

}

a. **NOTE**: We do not need the ID on this Dto because the database auto generates the ID when a new entry is created.

1. Now in the **TrailsController** change the return types for the following methods to **TrailUpsertDto**
   1. public IActionResult CreateTrail([FromBody] **TrailCreateDto** trailDto)
2. Create another **Dto** named **TrailUpdateDto** to use on the **PATCH** method, then add the following properties:

public class TrailUpdateDto

{

[Key]

public int Id { get; set; }

[Required]

public string Name { get; set; }

[Required]

public double Distance { get; set; }

public DifficultyType Difficulty { get; set; }

[Required]

public int NationalParkId { get; set; }

}

1. Now in the **TrailsController** change the return types for the following methods to **TrailUpsertDto**
   1. public IActionResult UpdateTrail(int trailId, [FromBody] **TrailUpdateDto** trailDto)
2. Map the **Trail** Model to the **Dtos** in the **ParkyMappins** file:
   1. CreateMap<Trail, TrailCreateDto>().ReverseMap();
   2. CreateMap<Trail, TrailUpdateDto>().ReverseMap();
3. Now that we have added a completely new **Trail** entity to our application, we must migrate by using the commands **add-migration [Migration name]** followed by the command **update-database.**

**Documentation API Versioning**

1. With **Swagger**, **we can split multiple APIs into separate documentation web pages**. To do so, add a new set of options in the services.**AddSwaggerGen**() method, under the **ConfigureServices**() method, in **startup**.cs in the following way:

services.AddSwaggerGen(options => {

options.SwaggerDoc("ParkyOpenAPISpecNP", new Microsoft.OpenApi.Models.OpenApiInfo()

{

Title = "Parky API National Parks",

Version = "1",

Description= "Parky API National Parks"

});

options.SwaggerDoc("ParkyOpenAPISpecTrails", new Microsoft.OpenApi.Models.OpenApiInfo()

{

Title = "Parky API Trails",

Version = "1",

Description = "Parky API Trails"

});

var xmlCommentFile = $"{Assembly.GetExecutingAssembly().GetName().Name}.xml";

var cmlCommentsFullPath = Path.Combine(AppContext.BaseDirectory, xmlCommentFile);

options.IncludeXmlComments(cmlCommentsFullPath);

});

1. Now, in the app.**UseSwaggerUI**() method, under the **Configure**() method, add another set of options in the following way:

app.UseSwaggerUI(options => {

options.SwaggerEndpoint("/swagger/ParkyOpenAPISpecNP/swagger.json", "Parky API NP");

options.SwaggerEndpoint("/swagger/ParkyOpenAPISpecTrails/swagger.json", "Parky API Trails");

options.RoutePrefix = "";

});

1. Finally, go to each controller, and add the following corresponding attribute at the top of each class to let out Swagger service know how to group each API in the documentation:

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

[ApiController]

[ApiExplorerSettings(GroupName = "ParkyOpenAPISpecNP")]

[ProducesResponseType(400)]

public class NationalParksController : ControllerBase

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

[ApiController]

[ApiExplorerSettings(GroupName = "ParkyOpenAPISpecTrails")]

[ProducesResponseType(400)]

public class TrailsController : ControllerBase

1. Test by running the application, then click on the **Select Definition drop down** in the top right, to select your API of choice.

**Attribute Routing**

1. Sometimes you will have **multiple** **GET** **methods**, or multiple methods with the same action in your controller and your application won’t be able to recognize the difference between the 2 because they’re on the same route, to fix this you need **Attribute** **Routing**. Add the following method in the **TrailsController** with a new attribute route:

[HttpGet("GetTrailsInNationalPark/{nationalParkId}")]

[ProducesResponseType(200, Type = typeof(TrailDto))]

[ProducesResponseType(404)]

[ProducesDefaultResponseType]

public IActionResult GetTrailInNationalPark(int nationalParkId)

{

var objList = \_trailRepo.GetTrailsInNationalPark(nationalParkId);

if (objList == null)

return NotFound();

var objDto = new List<TrailDto>();

foreach (var obj in objList)

objDto.Add(\_mapper.Map<TrailDto>(obj));

return Ok(objDto);

}

1. **NOTE:** You can replace GetTrailsInNationalPark with the route keyword [action] and it will do the same.

**User Authorization**

1. In order to restrict access to desired **ACTION** methods in a controller, we must add the attribute **[Authorize]** to prevent our application from providing information to unauthorized clients.
   1. Add this attribute to the **GetNationalPark**(int nationalParkId) method in the **NationalParksController**.
2. Create a new **User** model in the **Models** folder and add the following properties:

public class User

{

public int Id { get; set; }

public string Username { get; set; }

public string Password { get; set; }

public string Role { get; set; }

[NotMapped]

public string Token { get; set; }

}

1. **NOTE**: The **[NotMapped]** attribute will prevent the Token property from being created as a column in our database.
2. Now connect our new **User** model to the database by creating a new **DbSet**<> in the ApplicationDBContext.
   1. public DbSet<User> Users { get; set; }
3. Now use the following commands to add the new migration:
   1. **add-migration YourMigrationName**
   2. **update-database**

**User Repository**

1. Create a new **Interface** called **IUserRepository** with the following methods:

public interface IUserRepository

{

bool IsUniqueUser(string username);

User Authenticate(string username, string password);

User Register(string username, string password);

}

1. Create a new class called **UserRepository**, then inherit from the newly created interface **IUserRepository** and implement the required methods.
2. Inject our new data context **DbSet<User> User**, and prepare it to be injected into the constructor:

private readonly ApplicationDBContext \_userDb;

public UserRepository(ApplicationDBContext userDb)

{

\_userDb = userDb;

}

1. In **startup**.cs, add the newly created service under the **ConfigureServices**() method:
   1. services.AddScoped<IUserRepository, UserRepository>();

**Implementing our API Secret Key, Bearer Tokens, Cross-Origin Resource Sharing & Authentication**

1. In the **appsettings**.json file, add the following setting property:

"AllowedHosts": "\*",

"AppSetting": {

"Secret": "My Super Secret API Key"

}

1. In order to use our secret key to generate a **Web** **Token**, we need to retrieve it with **dependency injection**. Add the following service under the **ConfigureServices**() method in **startup**.cs:

//Generates a Token based on out secret key in appsettings.json

services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme).AddJwtBearer(options => {

options.TokenValidationParameters = new TokenValidationParameters

{

ValidateIssuerSigningKey = true,

//This need the secret's exact location from the appsettings.json, exactly how the object properties were writen

IssuerSigningKey = new SymmetricSecurityKey(Encoding.ASCII.GetBytes(Configuration.GetSection("AppSettings:Token").Value)),

ValidateIssuer = false,

ValidateAudience = false

};

});

1. Add a new method to the request pipeline to integrate our new service by adding the method app.UseAuthentication();
   1. **NOTE**: You must add app.UseAuthentication() first, before app.UseAuthorization(), otherwise the app will break.
2. We’ll also need to integrate **CORS** so that the client allows calls to different servers other than the origin server. Add the following code in the **Configure**() method:
   1. app.UseCors(x => x.AllowAnyOrigin().AllowAnyMethod().AllowAnyHeader());
   2. **NOTE**: There could be some security issues with the above implementation.
3. In the **ConfigureServices**() method add **service services.AddCors();**

**Token Creation and Setup**

1. In the **UserRepository**, add the following member variable:
   1. private readonly IConfiguration \_config;
2. Then in the **UserRepository** **constructor**, inject the new IConfiguration \_**config** to retrieve the secret.
3. In the **UserRepository**, populate the Authenticate() method with the following code:

public User Authenticate(string username, string password)

{

//Retrieve the user that matches the given user name and password

var user = \_userDb.Users.SingleOrDefault(x => x.Username == username && x.Password == password);

//If this user does not exist, return NULL

if (user == null)

return null;

//If the user was found, generate and set up an Authentication Token that user

var tokenHandler = new JwtSecurityTokenHandler();

var key = Encoding.ASCII.GetBytes(\_config.GetSection("AppSettings:Token").Value);

//Set token description

var tokenDescriptor = new SecurityTokenDescriptor

{

Subject = new ClaimsIdentity(new Claim[] {

new Claim(ClaimTypes.Name, user.Id.ToString())

}),

//Token expiration

Expires = DateTime.UtcNow.AddDays(7),

SigningCredentials = new SigningCredentials(new SymmetricSecurityKey(key)

,SecurityAlgorithms.HmacSha256Signature)

};

//Create the token

var token = tokenHandler.CreateToken(tokenDescriptor);

user.Token = tokenHandler.WriteToken(token);

user.Password = "";

return user;

}

**Users Controller**

1. In the Controllers folder, create a new empty controller called UsersController and add the following attributes:

[Authorize]

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

[ApiController]

[ApiExplorerSettings(GroupName = "ParkyOpenAPISpecUsers")]

1. Setup the documentation by adding a new option in the services.AddSwaggerGen() under the **ConfigureServices**() method in **startup**.cs:

options.SwaggerDoc("ParkyOpenAPISpecUsers", new Microsoft.OpenApi.Models.OpenApiInfo()

{

Title = "Parky API Users",

Version = "1",

Description = "Parky API Users"

});

1. Then add the option to the request pipeline method app.UseSwaggerUI(), under the **Configure**() method:
   1. options.SwaggerEndpoint("/swagger/ParkyOpenAPISpecUsers/swagger.json", "Parky API Users");
2. Back in the **UserController**, populate the **Authentiocate**() method with the following code:

[AllowAnonymous]

[HttpPost("authenticate")]

public User Authenticate(string username, string password)

{

//Retrieve the user that matches the given user name and password

var user = \_userDb.Users.SingleOrDefault(x => x.Username == username && x.Password == password);

//If this user does not exist, return NULL

if (user == null)

return null;

//If the user was found, generate and set up an Authentication Token that user

var tokenHandler = new JwtSecurityTokenHandler();

var key = Encoding.ASCII.GetBytes(\_config.GetSection("AppSettings:Token").Value);

//Set token description

var tokenDescriptor = new SecurityTokenDescriptor

{

Subject = new ClaimsIdentity(new Claim[] {

new Claim(ClaimTypes.Name, user.Id.ToString())

}),

//Token expiration

Expires = DateTime.UtcNow.AddDays(7),

SigningCredentials = new SigningCredentials(new SymmetricSecurityKey(key)

,SecurityAlgorithms.HmacSha256Signature)

};

//Create the token

var token = tokenHandler.CreateToken(tokenDescriptor);

user.Token = tokenHandler.WriteToken(token);

return user;

}

1. Test the new feature by trying out the new API creating a test user in the database and sending the following object:

{

"username": "test",

"password": "test"

}

* 1. **NOTE**: An authentication token should be returned in the response.

**Documenting and Setting up Swagger to Test Login Auth API**

1. In **startup**.cs, in the **services.AddSwaggerGen()** method, under the **ConfigureServices**() method, add the following options:

//Swagger Auth Bearer features

options.AddSecurityDefinition("Bearer", new OpenApiSecurityScheme

{

Description = "JWT Authorization header using Bearer scheme. \r\n\r Enter 'Bearer' [space] " +

"and then your token in the test input below.\r\n\r\n" +

"Example: \"Bearer 12345abc\"",

Name = "Authorization",

In = ParameterLocation.Header,

Type = SecuritySchemeType.ApiKey,

Scheme = "Bearer"

});

options.AddSecurityRequirement(new OpenApiSecurityRequirement()

{

{

new OpenApiSecurityScheme

{

Reference = new OpenApiReference

{

Type = ReferenceType.SecurityScheme,

Id = "Bearer"

},

Scheme = "oauth2",

Name = "Bearer",

In = ParameterLocation.Header,

},

new List<string>()

}

});

**User Register**

1. In the **UserRepository** populate the following 2 methods:

public bool IsUniqueUser(string username)

{

//Retrieve the user that matches the given user name and password

var user = \_userDb.Users.SingleOrDefault(x => x.Username == username);

//If this user does not exist, return NULL

if (user == null)

return true;

return false;

}

public User Register(string username, string password)

{

User userObj = new User()

{

Username = username,

Password = password

};

\_userDb.Users.Add(userObj);

\_userDb.SaveChanges();

userObj.Password = "";

return userObj;

}

1. In the **UsersController** add a new register method:

[AllowAnonymous]

[HttpPost]

public IActionResult Register([FromBody] User model)

{

if (!\_userRepo.IsUniqueUser(model.Username))

return BadRequest(new { message = "Username already exists" });

var user = \_userRepo.Register(model.Username, model.Password);

if (user == null)

return BadRequest(new { message = "Error while registering..." });

return Ok();

}

1. Test API by passing in object:

{

"username": "test",

"password": "test"

}

**User Roles**

1. In order to give users, the ability to restrict access to a certain APIs, we must provide the Authorize attribute with the parameter Role. For example **[Authorize(Roles = "Admin")].**
2. In the **UserRepository**, under the **Authenticate**() method, in the **tokenDescriptor** object definition, add the following line of code under the Subject property:
   1. new Claim(ClaimTypes.Role, user.Role)
3. Under the **Register**() method, give the user the admin role by adding **Role = "Admin"** in the userObj.
4. Above the **GetTrails(trail id)** method, add the attribute **[Authorize(Roles = "Admin")]** and test your API changes.

**Fixing the model documentation**

1. Create a new **AuthUser** model in the **Models** folder with the following properties:

public class AuthUser

{

[Required]

public string Username { get; set; }

[Required]

public string Password { get; set; }

}

1. In the **UsersController**, change the parameter type for both, the **Register([FromBody] AuthUser model)** and **Authenticate([FromBody] AuthUser model)** methods to the new **AuthUser** model.