If we’re dealing with views like the full mvc package we’re gonna have to derive from the Controller Interface class

If we’re only building an api with controllers and models then we’re gonna derive from the ControllerBase interface

We should add [ApiController] Atrribute above the class of the controller, adding this causes automatic http attribute routing, and automatic 400 responses.

Below that we add the [Route(“[controller]”)], (for url route)

Controller CRUD:

IActionResult 🡪 Allows us to return a message along with an object.

These message may be Ok, BadRequest ..

IActionResult does not show schema for the method

Where as if we use ActionResult<T> it shows the schema of T

using System.Text.Json.Serialization;

namespace dotnet\_rpg.Models

{

    [JsonConverter(typeof(JsonStringEnumConverter))]

    public enum RpgClass

    {

        Knight = 1,

        Mage = 2,

        Cleric = 3

    }

}

JsonConverter converts from numbers to strings

[HttpGet("GetAll")]

        //or

        //[Route("GetAll")]

        public ActionResult<List<Character>> Get()

        {

            return Ok(characters);

        }

Route could be combinied with HttpGet or set all alone under it

[HttpGet("{id}")]

        //or

        //[Route("{id}")]

        public ActionResult<Character> GetSingle(int id)

        {

            return Ok(characters.FirstOrDefault(c => c.Id == id));

        }

If we want to separate business logic from actual normal coding (controller) and not end up with a fat controller

We should implement the repository pattern (Services, IServcies) and use dependency injection later on in the controller

Now to do that we’d have to create a services folder then folder for each entity including the interface and its implementation

public interface ICharacterService

    {

        List<Character> GetAllCharacters();

        Character GetCharacterById(int id);

        List<Character> AddCharacter(Character newCharacter);

    }

public class CharacterService : ICharacterService

    {

        private static List<Character> characters = new List<Character>

        {

            new Character(),

            new Character { Id = 1, Name = "Sam"}

        };

        public List<Character> AddCharacter(Character newCharacter)

        {

            characters.Add(newCharacter);

            return characters;

        }

        public List<Character> GetAllCharacters()

        {

            return characters;

        }

        public Character GetCharacterById(int id)

        {

            var character = characters.FirstOrDefault(c => c.Id == id);

            if (character is not null)

                return character;

            throw new Exception("Character not found!");

        }

    }

Now there is a thing called asynchronous and synchronous calls

For normal and small apps it wouldn’t matter whether we use it or not but its always better to do so so that when your app scales up u’d always have a solid foundation   
  
So to start using asynchronous calls all we have to do to the interface is add Task Type behind each method

public interface ICharacterService

    {

        Task<List<Character>> GetAllCharacters();

        Task<Character> GetCharacterById(int id);

        Task<List<Character>> AddCharacter(Character newCharacter);

    }

Now we’d have to add the async word to each method in the children classes along with the Task Data type

public class CharacterService : ICharacterService

    {

        private static List<Character> characters = new List<Character>

        {

            new Character(),

            new Character { Id = 1, Name = "Sam"}

        };

        public async Task<List<Character>> AddCharacter(Character newCharacter)

        {

            characters.Add(newCharacter);

            return characters;

        }

        public async Task<List<Character>> GetAllCharacters()

        {

            return characters;

        }

        public async Task<Character> GetCharacterById(int id)

        {

            var character = characters.FirstOrDefault(c => c.Id == id);

            if (character is not null)

                return character;

            throw new Exception("Character not found!");

        }

    }

Now we’d get a warning there under each method saying this async method is not doing anything asynchronously that’s totally normal now especially that we’re not dealing with a database right now instead we are dealing with a list instead so we’ll fix that later

Now last but not least we’d have to also add the async and Task data type to each method in the controller class and additionally we’ll add the await to each Service or repository call   
look at the following code:

[ApiController]

    [Route("api/[controller]")]

    public class CharacterController : ControllerBase

    {

        private readonly ICharacterService \_characterService;

        public CharacterController(ICharacterService characterService)

        {

            \_characterService = characterService;

        }

        [HttpGet("GetAll")]

        //or

        //[Route("GetAll")]

        public async Task<ActionResult<List<Character>>> Get()

        {

            return Ok(await \_characterService.GetAllCharacters());

        }

        [HttpGet("{id}")]

        //or

        //[Route("{id}")]

        public async Task<ActionResult<Character>> GetSingle(int id)

        {

            return Ok(await \_characterService.GetCharacterById(id));

        }

        [HttpPost]

        public async Task<ActionResult<List<Character>>> AddCharacter(Character newCharacter)

        {

            return Ok(await \_characterService.AddCharacter(newCharacter));

        }

    }

Now we’re gonna add a **ServiceReponse Wrapper** Class

That has the following properties:

public class ServiceResponse<T>

    {

        public T? Data { get; set; }

        public string Message { get; set; } = string.Empty;

        public bool Success { get; set; } = false;

    }

This class is basically a better way to move data throughout your application and you’d also include a message to each response along with Success of bool type to help frontends handle response in a better and easier way

Now to use this we’re gonna change the IServices, Services, ServicesController again..  
  
  
Service:  
public class CharacterService : ICharacterService

    {

        private static List<Character> characters = new List<Character>

        {

            new Character(),

            new Character { Id = 1, Name = "Sam"}

        };

        public async Task<ServiceResponse<List<Character>>> AddCharacter(Character newCharacter)

        {

            characters.Add(newCharacter);

            return new ServiceResponse<List<Character>>

            {

                Data = characters,

                Message = "Added Character Successfully",

                Success = true

            };

        }

        public async Task<ServiceResponse<List<Character>>> GetAllCharacters()

        {

            return new ServiceResponse<List<Character>>

            {

                Data = characters,

                Message = "Added Character Successfully",

                Success = true

            };

        }

        public async Task<ServiceResponse<Character>> GetCharacterById(int id)

        {

            var character = characters.FirstOrDefault(c => c.Id == id);

            return new ServiceResponse<Character>

            {

                Data = characters.FirstOrDefault(c => c.Id == id),

                Message = "Added Character Successfully",

                Success = true

            };

        }

    }

IService:  
  
public interface ICharacterService

    {

        Task<ServiceResponse<List<Character>>> GetAllCharacters();

        Task<ServiceResponse<Character>> GetCharacterById(int id);

        Task<ServiceResponse<List<Character>>> AddCharacter(Character newCharacter);

    }

Controller:

[ApiController]

    [Route("api/[controller]")]

    public class CharacterController : ControllerBase

    {

        private readonly ICharacterService \_characterService;

        public CharacterController(ICharacterService characterService)

        {

            \_characterService = characterService;

        }

        [HttpGet("GetAll")]

        //or

        //[Route("GetAll")]

        public async Task<ActionResult<ServiceResponse<List<Character>>>> Get()

        {

            return Ok(await \_characterService.GetAllCharacters());

        }

        [HttpGet("{id}")]

        //or

        //[Route("{id}")]

        public async Task<ActionResult<ServiceResponse<Character>>> GetSingle(int id)

        {

            return Ok(await \_characterService.GetCharacterById(id));

        }

        [HttpPost]

        public async Task<ActionResult<ServiceResponse<List<Character>>>> AddCharacter(Character newCharacter)

        {

            return Ok(await \_characterService.AddCharacter(newCharacter));

        }

    }

Now we’re gonna be implementing the dtos are used to control how much data you want to show for each entity

We basically create a folder containing the dtos then we go to the ICharacterService and change character to GetCharacterDto type cuz in the 3 cases in the end we’re returning a character even in the add method we’re adding then returning which is why we go to the controller and also replace everything with character to GetCharacterDto but in the post method we change the parameter from characters to AddCharacterDto.

Then to map each dto to its entity we use AutoMapper

dotnet add package AutoMapper.Extensions.Microsoft.DependencyInjection

after injecting the automapper we do the following \_mapper.Map<Wanted Destination>(Current);

then we need to create a class (Configure) and lets call it AutoMapperProfile and let it inherit or derive from Profile class then inside it create a constructor using ctor then tab

and we’re gonna make a map inside it:

CreateMap<Character, GetCharacterDto>();

This is to convert character to get character dto

public class AutoMapperProfile : Profile

    {

        public AutoMapperProfile()

        {

            CreateMap<Character, GetCharacterDto>();

        }

    }