INF 354 notes

API

Contents

1. Create API .NET 6 blank solution:

Create new project ▸ Select ASP.NET Core Web API ▸ Framework should be .NET 6.0 (Long term support)

Configure API solution:

appsettings.json:

“DefaultConnection”: “Server= insert the machine server here; Database= name of DB here; Trusted_Connection=True; MultipleActiveResultSets=True”

Program.cs:

```
using APIII.Models;

using Microsoft.EntityFrameworkCore;


var builder = WebApplication.CreateBuilder(args);


builder.Services.AddDbConext<AppDbCotext>(options =>
options.UseSqlServer(builder.Configuration.GetConnectionString(“DefaultConnection”)));
```

This code sets up a connection to a SQL Server database for an API application. The database connection string is stored in the "DefaultConnection" field of the appsettings.json file. The connection string includes the server name, database name, and specifies a trusted connection and support for multiple active result sets.

The Program.cs file includes a reference to the AppDbContext class, which is used to manage the database context for the application. The WebApplication.CreateBuilder() method is called to create a new web application builder object.

The builder object's Services property is used to add a DbContext object to the service collection. The AddDbContext<AppDbContext>() method is used to add a new DbContext object to the service collection with options to use SQL Server and to retrieve the connection string from the appsettings.json file using the "DefaultConnection" key.

Overall, this code sets up the database connection for an API application using the Entity Framework Core and SQL Server.

2. Create the Entities/Models

Models folder ▸Customer.cs:

```
using System.ComponentModel.DataAnnotations;

namespace APIII.Models
    {
        public class Customer
        {
            [Key]
            public int CustId { get; set; }

            [MaxLength(50)]
            public string LastName { get; set; } = string.Empty


            [MaxLength(50)]
            public string FirstName { get; set; } = string.Empty

            public string? Address { get; set; }

            public string? City { get; set; }

            [StringLength(2)]
            public string? State { get; set; }

            [StringLength(5)]
            public string? PostalCode { get; set; }

            [StringLength(10)]
            public string PhoneNumber { get; set; } = string.Empty
```

This code defines the "Customer" entity class for an API application in the Models folder. The Customer class includes several properties with various attributes that specify validation rules and data types.

The [Key] attribute is applied to the "CustId" property to indicate that it is the primary key for the Customer entity.

The [MaxLength(50)] attribute is used to limit the maximum length of the "LastName" and "FirstName" properties to 50 characters, and default value for each property is set to an empty string.

The "Address", "City", "State", "PostalCode", and "PhoneNumber" properties are all nullable string types that represent various pieces of contact information for the customer. The [StringLength] attribute is used to limit the maximum length of the "State", "PostalCode", and "PhoneNumber" properties to 2, 5, and 10 characters, respectively.

Overall, this code defines the Customer entity with properties that match the expected data for a customer, and specifies validation rules for those properties.

How do you write that the attribute is a foreign key?

To indicate that a property is a foreign key in Entity Framework Core, you can use the [ForeignKey] attribute. You would apply this attribute to the property that references the related entity. For example:

```
public class Order
{
    [Key]
    public int OrderId { get; set; }

    // foreign key to Customer entity
    public int CustomerId { get; set; }

    // navigation property to Customer entity
    [ForeignKey("CustomerId")]
    public Customer Customer { get; set; }

    // other properties for the Order entity
}
```

In this example, the "CustomerId" property is the foreign key to the related Customer entity, and the "Customer" property is a navigation property that allows you to access the related

Customer entity. The [ForeignKey] attribute is used to specify the name of the foreign key property ("CustomerId") that is being referenced by the "Customer" navigation property.

3. Configure the DbContext:

```
using Microsoft.EntityFrameworkCore;

namespace APIII.Models

{

        public class AppDbConext: DbContext

        {

                Public AppDbContextOptions(DbContextOptions<AppDbContext> options) :
                base (options)

                {

                }


                public DbSet<Guide> Guides { get; set; }


                public DbSet<Trip> Trips { get; set; }



                public DbSet<Customer> Customers { get; set; }


                protected override void OnModelCreating(ModelBuilder modelBuilder)

                {

                        Base.OnModelCreating(modelBuilder);

                }
```

4. Setup and Configure Repository

IRepository.cs:

```
namespace APIII.Models
{
    public interface IRepository
    {
        void Add<T>(T entity) where T : class;

        void Delete<T>(T entity) where T : class;

        Task<bool> SaveChangesAsync();

        //Customer
        Task<Customer[]> GetAllCustomerAsync();

        Task<Customer> GetCustomerAsync(int custId);

        //Trip
        Task<Trip> GetTripAsync(int tripId);

        //Guide
        Task<Guide> GetGuideAsync(string guideNum);
    }
}
```

This code defines an interface called "IRepository" in the Models namespace for an API application. The "IRepository" interface declares several methods for accessing and modifying data in the application's database.

The Add<T>() and Delete<T>() methods are generic methods that accept an entity of type T, which must be a class. These methods are used to add or remove an entity from the database.

The SaveChangesAsync() method is used to save changes made to the database asynchronously. This method returns a boolean value indicating whether the changes were successfully saved.

The remaining methods in the interface are specific to the Customer, Trip, and Guide entities in the application. These methods are used to retrieve one or more entities of each type from the database. The GetAllCustomerAsync() method returns an array of all Customer entities in the database, while the GetCustomerAsync() method retrieves a single Customer entity based on its "custId" property value. Similarly, the GetTripAsync() and GetGuideAsync() methods retrieve a single Trip entity based on its "tripId" property value or a single Guide entity based on its "guideNum" property value, respectively.

Overall, this code defines a contract for an interface that provides methods for interacting with the application's database. The interface includes methods for adding, removing, and retrieving entities from the database, as well as a method for saving changes made to the database.

Repository.cs:

```
using Microsoft.EntityFrameworkCore;
namespace APIII.Models
{
	public class Repository : IRepository
	{
		private readonly AppDbContext _appDbContext;

		public Repository(AppDbContext appDbContext)
		{
			_appDbContext = appDbContext;
		}
		public void Add<T>(T entity) where T : class
		{
			_appDbContext.Add(entity)
		}
		public void Delete<T>(T entity) where T : class
			{
			  _appDbContext.Remove(entity);
			}
		public async Task<Customer[]> GetAllCustomersAsync()
			{
			  IQueryable<Customer> query = _appDbContext.Customers;
			  return await query.ToArrayAsync();
			}
		public async Task<Customer> GetCustomerAsync(int custId)
			{
			  IQueryable<Customer> query = _appDbContext.Customers.Where(c
			  => c.CustId == custId);
			  return await query.FirstOrDefaultAsync();
			}
```

```
public async Task<Guide> GetGuideAsync(string guideNum)
        {
            IQueryable<Guide> query = _appDbContext.Guides.Where(c =>
            c.GuideNum == guideNum);
            return await query.FirstOrDefaultAsync();
        }
public async Task<Trip> GetTripAsync(int tripId)
        {
            IQueryable<Trip> query = _appDbContext.Trips.Include(g =>
            g.Guides).Where(c => c.TripId == tripId);
            return await query.FirstOrDefaultAsync();
        }
public async Task<bool> SaveChangesAsync()
        {
            return await _appDbContext.SaveChangesAsync() > 0;
        }
    }
}
```

This code defines a class called "Repository" in the Models namespace for an API application that implements the "IRepository" interface. The "Repository" class provides the implementation for the methods declared in the "IRepository" interface.

The constructor of the "Repository" class takes an instance of "AppDbContext" as a parameter and assigns it to a private readonly field called "_appDbContext". This instance of "AppDbContext" is used to interact with the database.

The "Add<T>()" method adds an entity of type T to the database. The "Remove<T>()" method removes an entity of type T from the database. These methods use the "_appDbContext" instance to add or remove an entity.

The "GetAllCustomersAsync()" method retrieves all Customer entities from the database and returns them as an array. The "GetCustomerAsync(int custId)" method retrieves a single Customer entity from the database based on its "custId" property value.

The "GetGuideAsync(string guideNum)" method retrieves a single Guide entity from the database based on its "guideNum" property value. The "GetTripAsync(int tripId)" method retrieves a single Trip entity from the database based on its "tripId" property value.

The "GetTripAsync()" method includes a call to the "Include()" method to load the associated "Guides" entities for the retrieved "Trip" entity.

Finally, the "SaveChangesAsync()" method saves any changes made to the database asynchronously and returns a boolean value indicating whether any changes were successfully saved.

Overall, this code implements the methods declared in the "IRepository" interface to provide an implementation for interacting with the application's database. The class provides methods for adding, removing, and retrieving entities from the database, as well as a method for saving changes made to the database.

Program.cs:

builder.Services.AddScoped<IRepository, Repository>();

This code adds a scoped service to the application's dependency injection container using the "AddScoped()" method provided by the "builder.Services" object. The service being added is the "IRepository" interface, which is mapped to the "Repository" class.

The "AddScoped()" method takes two arguments: the type of the service to be added, and the implementation of that service. In this case, the "IRepository" interface is the type of the service being added, and the "Repository" class is the implementation of that service.

By adding the "IRepository" interface and its implementation to the dependency injection container, the application can resolve instances of "IRepository" wherever it is needed throughout the application. This allows for better separation of concerns and makes the application more modular and testable.

The "Scoped" lifetime means that a new instance of "Repository" will be created for each HTTP request, and that instance will be used throughout the duration of that request. Once the request has been completed, the instance of "Repository" will be disposed of by the framework.

5. Create a CRUD endpoints

Controllers ▸ CustomerController.cs

GET Request
ALL customers:

```
[HttpGet]

[Route(“GetAllCustomers”)]


public async Task<IActionResult> GetAllCustomers()

{

        try

        {

                var results = await _repository.GetAllCustomersAsync();

        }

        catch (Exception)

        {

                return StatusCode(500, “Internal Server Error.”);

        }

}
```

One customer by ID:

```
[HttpGet]
[Route(“GetCustomer/{custId}”)]

Public async Task<IActionResult> GetCustomerAsync(int custId)
{
        Try
        {
                var results = await _repository.GetCustomerAsync(custId);
                if (result == null) return NotFound(“Customer does not exist”);
                return Ok(results);
        }
        Catch (Exception)
        {
                return StatusCode(500, “Internal Server Error.”);
        }
}
```

POST Request
[HttpPost]

[Route(“AddCustomer”)]

```
public async Task<IActionResult> AddCustomer(CustomerViewModel cvm)
{
        var customer = new Customer
        {
                LastName = cvm.LastName,
                FirstName = cvm.FirstName,
                Etc.
        }

        try
        {
                _repository.Add(customer);
                await _repository.SaveChangesAsync();
        }
        catch (Exception)
        {
                return BadRequest(“Invalid transaction”);
        }
```

PUT Request
[HttpPut]

[Route(“EditCustomer/{custId})]

```
public async Task<ActionResult<CustomerViewModel>> EditCustomer(int custId,
CustomerViewModel customerModel)
{
    try
    {
        var existingCustomer = await _repository.GetCustomerAsync(custId);
        if (existingCustomer == null) return NotFound($”The customer does not
        exist”);

        existingCustomer.LastName = customerModel.LastName;
        existingCustomer.FirstName = customerModel.FirstName;
        existingCustomer.Address = customerModel.Address;
        etc.
        if (await _repository.SaveChangesAsync())
        {
            return Ok(existingCustomer);
        }
    }
    catch (Exception)
    {
        return StatusCode(500, “Internal Server error”);
    }
    return BadRequest(“Your request is invalid”);
}
```

DELETE Request
[HtppDelete]

[Route(“DeleteCustomer/{custId}”)]

```
public async Task<IActionResult> DeleteCustomer(int custId)
{
    try
    {
        var existingCustomer = await _reposity.GtCustomerAsync(custId);
        if (existingCustomer == null) return NotFound($”The customer does not exist”);
        _repository.Delete(existingCustomer);
        if (await _repository.SaveChangesAsync()) return Ok(existingCustomer);
    }
    catch (Exception)
    {
        return StatusCode(500, “internal server error”);
    }
    return BadRequest(“request invalid”);
}
```