Data Repository Design Pattern

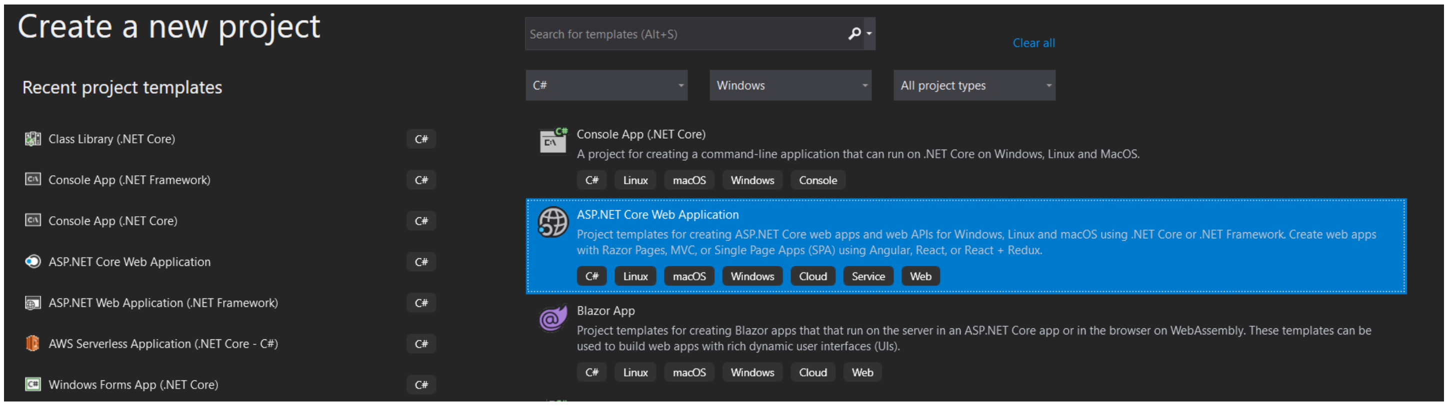
Implementing the data repository design pattern in a ASP.NET Core Web API project.

With the Repository pattern, we create an abstraction layer between the data access and the business logic layer of an application. By using it, we are promoting a more loosely coupled approach to access our data from the database. Also, the code is cleaner and easier to maintain and reuse. Data access logic is in a separate class, or sets of classes called a repository, with the responsibility of persisting the application's business model.

In this tutorial we will go through the steps to implement a data repository design pattern for an ASP.NET Core Web API from start to finish. The application we are going to create is going to be a car dealership inventory manager.

Create a new ASP.NET Core Web API

Open Visual Studio and select a new ASP.NET Core Web Application from the list of templates and select Next.



Create a new project

Recent project templates

- Class Library (.NET Core) C#
- Console App (.NET Framework) C#
- Console App (.NET Core) C#
- ASP.NET Core Web Application C#
- ASP.NET Web Application (.NET Framework) C#
- AWS Serverless Application (.NET Core - C#) C#
- Windows Forms App (.NET Core) C#

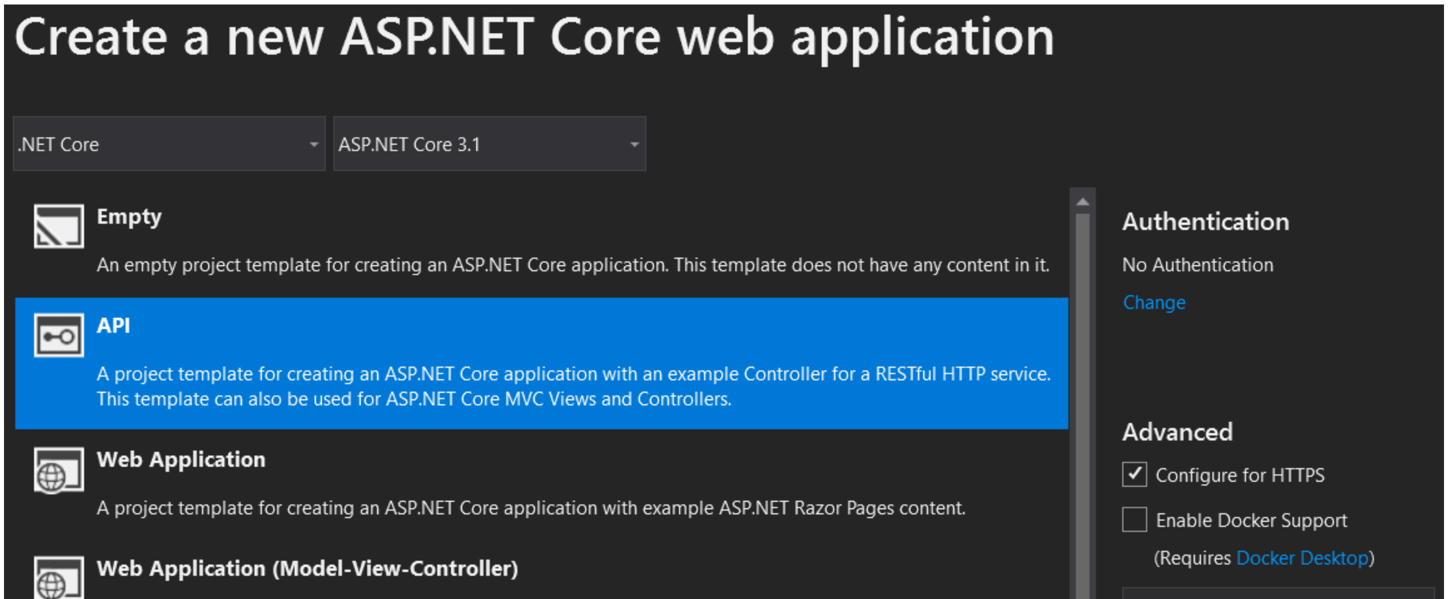
Console App (.NET Core)
A project for creating a command-line application that can run on .NET Core on Windows, Linux and MacOS.
C# Linux macOS Windows Console

ASP.NET Core Web Application
Project templates for creating ASP.NET Core web apps and web APIs for Windows, Linux and macOS using .NET Core or .NET Framework. Create web apps with Razor Pages, MVC, or Single Page Apps (SPA) using Angular, React, or React + Redux.
C# Linux macOS Windows Cloud Service Web

Blazor App
Project templates for creating Blazor apps that that run on the server in an ASP.NET Core app or in the browser on WebAssembly. These templates can be used to build web apps with rich dynamic user interfaces (UIs).
C# Linux macOS Windows Cloud Web

Name the project **DealershipInventoryManager** and hit create.

Select API for the project template type and hit create once again.



Create a new ASP.NET Core web application

.NET Core | ASP.NET Core 3.1

Empty
An empty project template for creating an ASP.NET Core application. This template does not have any content in it.

API
A project template for creating an ASP.NET Core application with an example Controller for a RESTful HTTP service. This template can also be used for ASP.NET Core MVC Views and Controllers.

Web Application
A project template for creating an ASP.NET Core application with example ASP.NET Razor Pages content.

Web Application (Model-View-Controller)

Authentication
No Authentication
Change

Advanced
- [✓] Configure for HTTPS
- [ ] Enable Docker Support
  (Requires Docker Desktop)

Initial Setup for Our Project

We first want to remove two files that come with the template.

REMOVE:

1. **WeatherForecastController.cs** within the controller folder.
2. **WeatherForecast.cs** file located in the project root

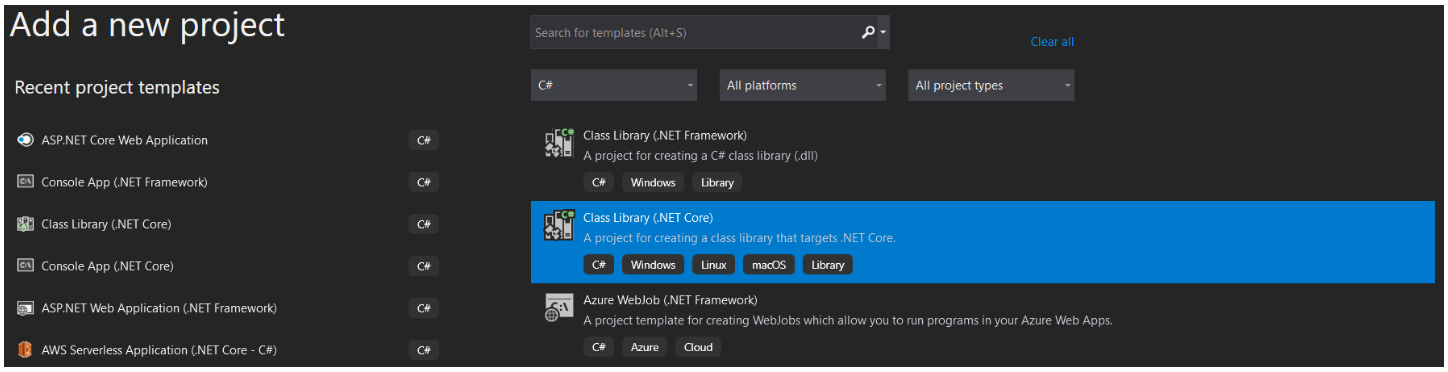
These files come with the ASP.NET Core Web API template and will serve no purpose in our application.

Adding a Class Library Project

We are going to keep all our data access logic inside a separate class library project. This way we can use the class library in other projects that need access to our database. This way we will not have to re-write any of the data access logic again.

Let's start by right clicking on our solution and selecting Add > New Project

Choose the Class Library (.NET Core) template.


Add a new project

Search for templates (Alt+S)

Clear all

Recent project templates

- ASP.NET Core Web Application — C#
- Console App (.NET Framework) — C#
- Class Library (.NET Core) — C#
- Console App (.NET Core) — C#
- ASP.NET Web Application (.NET Framework) — C#
- AWS Serverless Application (.NET Core - C#) — C#

C# | All platforms | All project types

Class Library (.NET Framework)
A project for creating a C# class library (.dll)
C# Windows Library

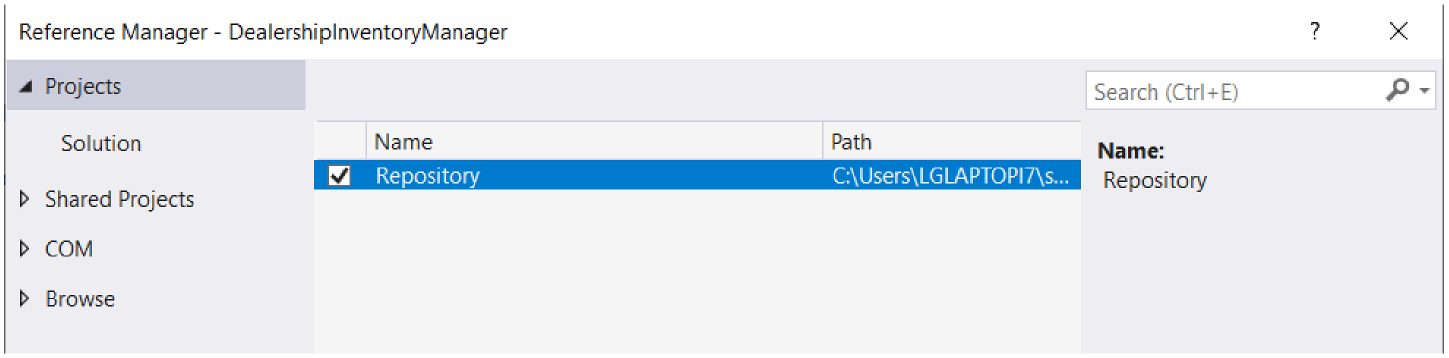
Class Library (.NET Core)
A project for creating a class library that targets .NET Core.
C# Windows Linux macOS Library

Azure WebJob (.NET Framework)
A project template for creating WebJobs which allow you to run programs in your Azure Web Apps.
C# Azure Cloud

Name the project **Repository** and hit create.

Before we move on we need to create a connection between our main DealershipInventoryManager project and out Repository class library project.

Right click on Dependencies in the main project and select Add Reference…



Click the check box next to Repository and hit ok. This will make the Repository project accessible for our main DealershipInventoryManager project.

Adding Entity Framework

Next, we need to install Entity Framework into our project. Entity Framework Core is a library that allows us to access the database from our applications. It is designed as an object-relational mapper (ORM) and it works by mapping the relation database to the applications database model.

Go up to the Tools > NuGet Package Manager > Manage NuGet Packages for Solution…

Now select the browse tab and search and install the packages listed below. Make sure you install them to your Repository project.

INSTALL:

1. Microsoft.EntityFrameworkCore
2. Microsoft.EntityFrameworkCore.Tools
3. Microsoft.EntityFrameworkCore.SqlServer

We need to install the Micorsosft.EntityFrameworkCore package so we can get access to the DbContext class which our ApplicationDbContext class will be inheriting from. The Microsoft.EntityFrameworkCore.Tools will allow us to perform migrations via the Package Manager Console.

For us to have a successful migration later, we also need to install a package to our main project. Right click on the DealershipInventoryManager project and select Manage NuGet Packages…

INSTALL:

1. Microsoft.EntityFrameworkCore.Design

We are now setup to move onto creating our ApplicationDbClass.

Creating the ApplicationDbContext Class

Entity Framework Core looks for our ApplicationDbContext class and uses all the public DbSet properties to map their names to the names of tables in our database. Then it goes inside a class which is provided in the DbSet<T> property and maps all the public properties to the table columns in the table with the same name and types. The ApplicationDbContext must inherit from the DbContext class through the ApplicationContext constructor by using DbContextOptions parameter.

Inside of our Repository class library, lets add a new folder called: **Data** Inside this folder create a new class and name it: **ApplicationDbContext**

We need to now make this ApplicationDbContext class inherit from the DbContext class that is brought in from the Microsoft.EntityFrameworkCore package.

We then will write out the ApplicationDbContext class to utilize the DbContext class that we are inheriting from.

```
public class ApplicationDbContext : DbContext
{
    public ApplicationDbContext(DbContextOptions options)
        :base(options)
    {

    }
}
```

In the next steps we will add a connection string to our database and register our ApplicationDbContext into the IOC container.

Configure ApplicaitonDbContext as Service

Before we configure our ApplicationDbContext into our IOC container, we will have to add a connections string to provide Entity Framework Core the information needed to connect to our database.

In the root directory of your project, open the **appsettings.json** file and add the following lines.

```json
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  },
  "ConnectionsStrings": {
    "sqlConnection": "server={your server name here};
database=DealershipInventory; Integrated Security=true"
  },
  "AllowedHosts": "*"
}
```

We will be using this connection string in the next step when we register our ApplicationDbContext class with the applications services.

Open the startup.cs file located in your project root. You will notice the **ConfigureServices()** method. Modify the method to include:

```csharp
     public void ConfigureServices(IServiceCollection services)
     {
        services.AddDbContext<ApplicationDbContext>(opts =>
           opts.UseSqlServer(Configuration.GetConnectionString
           ("sqlConnection")));

        services.AddControllers();
     }
```

The AddDbContext method is used to register our ApplicationDbContext class into the IOC contatiner. We then specifiy the sql server connection inside the UseSqlServer extension method. You can see we are accessing the "sqlConnection" string that we just set in the appsettings.json file, by using the GetConnectionString method on Configuration. Configuration references this appsettings.json file.

We have now successfully readied our ApplicationDbContext class to be used with Dependency Injection inside our app. We could now inject the ApplicationDbContext into the constructor of our controllers to obtain a data access point to our database.

Creating and Adding Models to EF Core

Models are classes that Entity Framework uses for mapping to the database table. Inside our Repository project, create a new folder named: **Models** Inside this folder we will create two new classes. These classes will be **Car** and **Customer**.

```
    public class Car
    {
        [Key]
        public int CardId { get; set; }
        public string Make { get; set; }
        public string Model { get; set; }
        public string Year { get; set; }
        public string Color { get; set; }
        public string Condition { get; set; }
        public decimal Price { get; set; }
        public DateTime ArrivalDate { get; set; }
    }
```

```
    public class Customer
    {
        [Key]
        public int CustomerId { get; set; }
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public bool LoanApproved { get; set; }
        public DateTime CustomerSince { get; set; }
    }
```

Now that we have created these models we want to add them to a DbSet<> property on our ApplicationDbContext. Entity Framework Core will use the DbSet<> properties to map the models to tables in our database.

Edit the ApplicationDbContext class to contain the following properties.

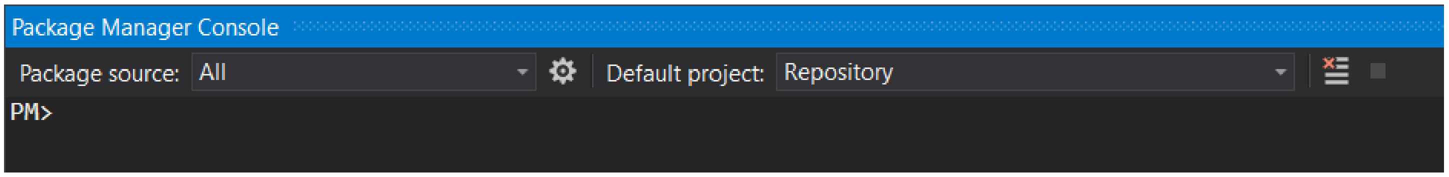
```
    public class ApplicationDbContext : DbContext
    {
        public ApplicationDbContext(DbContextOptions options)
            :base(options)
        {

        }
        public DbSet<Car> Cars { get; set; }
        public DbSet<Customer> Customers { get; set; }
    }
```

Now that we have added the models to the ApplicationDbContext class, we should do a migration to initialize our database. Because we installed the Microsoft.EntityFrameworkCore.Tools package we can perform our first migration via the Package Manager Console.

Go to Tools > NuGet Package Manager > Package Manager Console

We need to make sure we set the Package Manager Console Default project set to our Repository project.



Package source: All

Default project: Repository

```
PM>
```

Inside the Package Manager Console enter the command:

Add-Migration Intital

And then

Update-Database

Once those commands are complete, we can confirm our database has been created by selecting View > SQL Server Object Explorer and navigating to SQL Server > {server name} > Databases. Within the Databases folder you should see the DealershipInventory database. You can further expand that to view the tables.

This completes the basic MS SQL database setup for ASP.NET Core Web API. Next, we are going to implement the Repository Design Pattern around our ApplicaitonDbContext to abstract the data access layer.

Creating Generic Repository Base

Lets being by creating a generic template that will handle all CRUD operations for our repository classes.

Create a new folder in the Repository project named: Contracts. Inside the folder create a new interface named IRepositoryBase. Add the following code to the interface:

```
    public interface IRepositoryBase<T>
    {
        IQueryable<T> FindAll();
        IQueryable<T> FindByCondition(Expression<Func<T, bool>>
         expression);
        void Create(T entity);
        void Update(T entity);
        void Delete(T entity);
    }
```

We add the <T> after the name of the interface allows us to specify an arbitrary type T to methods at compile-time.

IQueryable<T> is used to represent the response of a query. IQueryable is enumerable and is a smart choice because it saves resources.

By using "Expression<Func…" as a parameter inside of our FindByCondition variable, we can pass in LINQ statements to filter down our queries.

Next, we want to create the class that will inherit from IRepositoryBase. In the root of the Repository project create a new abstract class named: RepositoryBase

```
    public abstract class RepositoryBase<T> : IRepositoryBase<T> where T : class
    {
        protected ApplicationDbContext ApplicationDbContext { get; set; }
        public RepositoryBase(ApplicationDbContext applicationDbContext)
        {
            ApplicationDbContext = applicationDbContext;
        }
        public IQueryable<T> FindAll()
        {
            return ApplicationDbContext.Set<T>().AsNoTracking();
        }
        public IQueryable<T> FindByCondition(Expression<Func<T, bool>> expression)
        {
            return ApplicationDbContext.Set<T>().Where(expression).AsNoTracking();
        }
        public void Create(T entity)
        {
            ApplicationDbContext.Set<T>().Add(entity);
        }
        public void Update(T entity)
        {
            ApplicationDbContext.Set<T>().Update(entity);
        }
        public void Delete(T entity)
        {
            ApplicationDbContext.Set<T>().Remove(entity);
        }
    }
```

Let's start by explaining what is going on inside of this class.

First, we are using the "where T : class". Here we are putting a constraint on what T can represent. This puts the constraint that the parameter of T must be of type class.

Next, we can see we are using our "Expression<Func…" parameter and plugging it into a Where LINQ method.

Lastly, we are using the "Set<T>". We use this when we don't know the entity type that we want to use. By using the Set<T> it takes the class parameter of T and injects in into the statements.

By using the generic type T, it gives our RepositoryBase class the optimum amount of reusability because we don't have to specify the exact model for the RepositoryBase class to work with. Notice, because the class is abstract, we will never be instantiating it, rather it will be inherited by other classes.

Creating User Classes per Model

We are going to implement user classes for each model. Because each of these classes will inherit from our RepositoryBase, they will have access to all the RepositoryBase class methods.

Let's start by creating interfaces for each of our models.

Inside of the Contracts folder in the Repository project create two interfaces: **ICarRepository** and **ICustomerRepository**

```
    public interface ICarRepository : IRepositoryBase<Car>
    {
    }
```

```
    public interface ICustomerRepository : IRepositoryBase<Customer>
    {
    }
```

The reason why we are creating a interface for each model is for potential additional model-specific methods.

Next let's create the user classes that will inherit these interfaces. Inside the Data folder in the Repository project, create two more class: **CarRepository** and **CustomerRepository**

```
    public class CarRepository : RepositoryBase<Car>, ICarRepository
    {
        public CarRepository(ApplicationDbContext applicationDbContext)
            :base(applicationDbContext)
        {
        }
    }
    public class CustomerRepository : RepositoryBase<Customer>,
    ICustomerRepository
    {
        public CustomerRepository(ApplicationDbContext
    applicationDbContext)
            :base(applicationDbContext)
        {
        }
    }
```

Here we are creating user class repositories for each of our models. When we inherit the RepositoryBase class we pass in the class in which the generic T on RepositoryBase will be representing. In the constructor of these classes we expect an ApplicationDbContext to be passed in. We then pass it to the parent class (RepositoryBase) to be utilized within RepositoryBase.

The final step is creating the full repository wrapper which will be the class we will utilize for data access inside of our controllers and classes.

Creating the Repository Wrapper

Let's bring all the things we have created together. Inside of the Contracts folder create another interface. Name this interface: **IRepositoryWrapper**

```
    public interface IRepositoryWrapper
    {
        ICarRepository Car { get; }
        ICustomerRepository Customer { get; }
        void Save();
    }
```

This is a contract for the inheriting class to implement these interfaces as well as the Save() method.

Finally, lets create the actual wrapper that we will utilize inside of our controllers and classes. Inside of the root of the Repository project, create a class named: **RepositoryWrapper**

```
    public class RepositoryWrapper : IRepositoryWrapper
    {
        private ApplicationDbContext _context;
        private ICarRepository _car;
        private ICustomerRepository _customer;
        public ICarRepository Car
        {
            get
            {
                if(_car == null)
                {
                    _car = new CarRepository(_context);
                }
                return _car;
            }
        }
        public ICustomerRepository Customer
        {
            get
            {
                if(_customer == null)
                {
                    _customer = new CustomerRepository(_context);
                }
                return _customer;
            }
        }
        public RepositoryWrapper(ApplicationDbContext context)
        {
            _context = context;
        }
        public void Save()
        {
            _context.SaveChanges();
        }
    }
```

Perfect! We have exposed the repository properties to be accessible when using this RepositoryWrapper!

In each property we check if the private instance is set to an instantiation. If it has not been set yet (is null) then we create a new instantiation. We inject the ApplicationDbContext class using dependency injection into the constructor so we can pass it down to the respective repositorys.

The last thing we need to do is add the RepositoryWrapper to our services so we can access it via Dependency Injection.

Registering the RepositoryWrapper for DI

The final step before we can inject out repository wrapper inside of our controllers and classes is to register it as a scoped service. Open the Startup.cs class in our main project and add the following lines of code:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<ApplicationDbContext>(opts =>
        opts.UseSqlServer(
             Configuration.GetConnectionString("sqlConnection")));

    services.AddScoped<IRepositoryWrapper, RepositoryWrapper>();

    services.AddControllers();
}
```

That's it! We have now successfully implemented the Repository Design Pattern inside an ASP.NET Core Web API project. Each time we add a new model to our project we need to follow the steps from the "Creating User Classes per Model" section down.

Using Example

Here is an example of it being used inside of a controller.

```
[Route("api/[controller]")]
[ApiController]
public class ExampleController : ControllerBase
{
    private IRepositoryWrapper _repo;
    public ExampleController(IRepositoryWrapper repo)
    {
        _repo = repo;
    }
    [HttpPost]
    public IActionResult Post([FromBody] Car car)
    {
        _repo.Car.Create(car);
        var cars = _repo.Car.FindByCondition(c => c.Color == "Blue");
        _repo.Car.Update(car);
        _repo.Car.Delete(car);
        _repo.Save();

        return Ok(cars);
    }
}
```

We are using dependency injection to inject our RepositoryWrapper class into the controller via the constructor. We then access one of the repository properties and perform a series of actions.

Adding Methods to Specific Repositories

It best to take this even one more step further. Currently, our CustomerRepository does not contain any logic. The best practice is to create model repository specific methods that are explicit as to what model and what actions is being performed. This again is creating even further abstraction from the database connection. Now, the business logic will only be aware of model specific repository actions.

Let's add a new method signature to our ICustomerRepository interface:

```
public interface ICustomerRepository : IRepositoryBase<Customer>
{
    Customer GetCustomer(int customerId);
    void CreateCustomer(Customer customer);
}
```

Let's implement these method signatures inside of the CustomerRepository class:

```
public class CustomerRepository : RepositoryBase<Customer>,
ICustomerRepository
{

    public CustomerRepository(ApplicationDbContext applicationDbContext)
        :base(applicationDbContext)
    {
    }

    public Customer GetCustomer(int customerId) =>
        FindByCondition(c =>
c.CustomerId.Equals(customerId)).SingleOrDefault();

    public void CreateCustomer(Customer customer) => Create(customer);
}
```

Now, if we were to use this in our controller:

```
            [HttpPost]
            public IActionResult Post([FromBody] Customer customer)
            {
                var customerFromDb =
     _repo.Customer.GetCustomer(customer.CustomerId);
                _repo.Customer.CreateCustomer(customer);
                _repo.Save();

                return Ok(customerFromDb);
            }
```

This is how all model specific repositories should be accessed.

Conclusion

The Repository Pattern allows us to reduce the amount of redundant code. We can also take it a step further and implement access restriction by manipulating the "get" of each repository property on our RepositoryWrapper class. The Repository Pattern is an industry standard for any professional ASP.NET Web API or MVC application that implements data access.