

**In the Model file; A database, tables & relations are defined...**

Entity Framework Code First is Visual Studio using the "Model" of the database to create the database file and "scaffold" any html "pages" to perform C.R.U.D operations. (CREATE.READ.UPDATE.DELETE)

To use Entity Framework to query, insert, update, and delete data - using .NET objects - Create a Model which maps the entities and relationships that are defined in your model to tables in a database.

Once you have a model, the primary class your application interacts with is **System.Data.Entity.DbContext** (The "Context" class – a "class" in this case is both a file type and the code in a class code block).

Accessing a DbSet property on a context object represent a starting query that returns all entities of the specified type. Note that just accessing a property will not execute the query. A query is executed when:

- It is enumerated by a foreach (C#) or For Each (Visual Basic) statement.
- It is enumerated by a collection operation such as ToArray, ToDictionary, or ToList.
- LINQ operators such as First or Any are specified in the outermost part of the query.

One of the following methods are called if an entity with the specified key is not found already loaded in the context.

- The Load extension method
- DbSet.Entry.Reload
- Database.ExecuteSqlCommand
- DbSet<T>.Find,

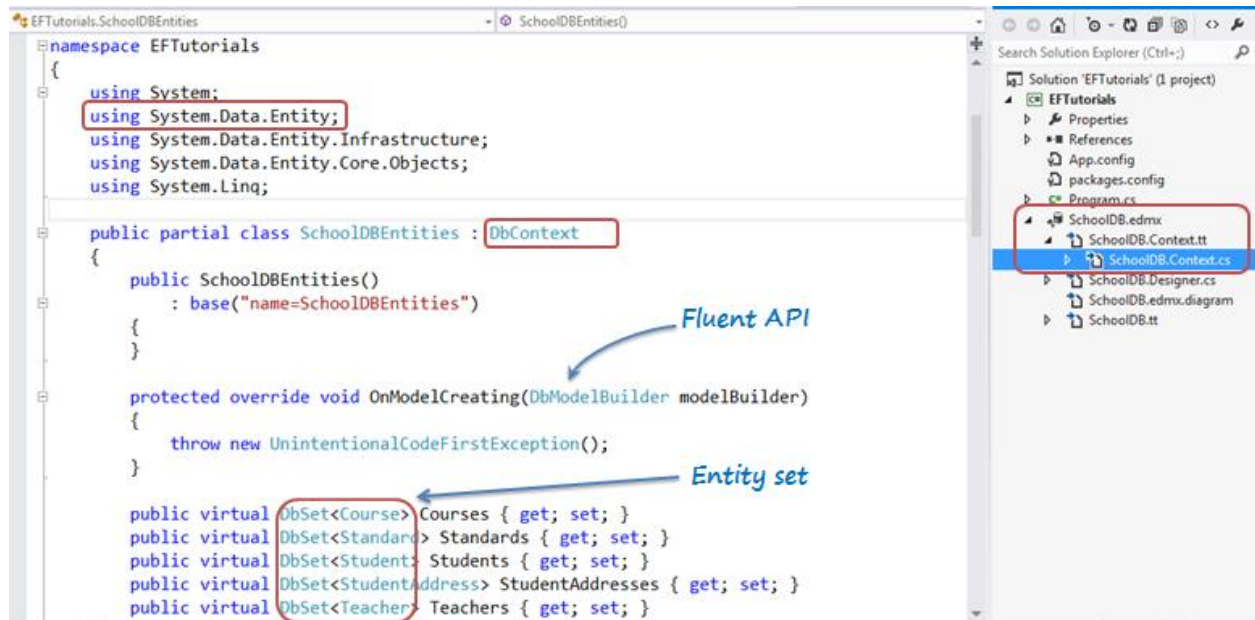
Use a DbContext associated to a model to:

1. Write and execute queries
2. Materialize query results as entity objects
3. Track changes that are made to those objects
4. Persist object changes back on the database
5. Bind objects in memory to UI controls

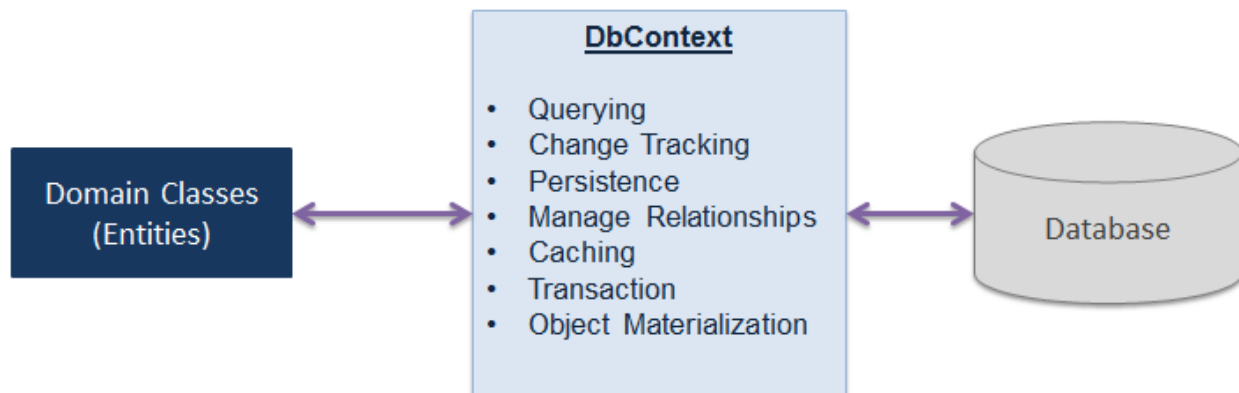
To work with "context" is to define a class that derives from DbContext and exposes DbSet properties that represent collections of the specified entities in the context.

As you have seen in the previous [Create Entity Data Model](#) section, EDM includes the `SchoolDBEntities` class, which is derived from the *System.Data.Entity.DbContext* class, as shown below. The class that derives DbContext is called context class in entity framework.

<https://www.entityframeworktutorial.net/entityframework6/dbcontext.aspx>



DbContext is an important class in Entity Framework API. It is a bridge between your domain or entity classes and the database.



**OR in non GEEK SPEEK..**

In a "model.cs" class file; Write code in a "class" { Block } that is "using" the System.Data.Entity.DbContext "class"

**Implement a relationship between model classes.**

1. **Photo** Table.. the One Photo has Many Comments...
2. Create "Comments" collection or "set"...

```
1. public virtual ICollection<Comment>Comments { get; set; }
```

3. **Comment** Table.. the Many Comment to One Photo
  1. public virtual Photo Photo { get; set; }

Add Controller

Model class: Photo (PhotoSharingApplication.Models)

Data context class: PhotoSharingContext (PhotoSharingApplication.Models) +

☐ Use async controller actions

Views:

☒ Generate views

☒ Reference script libraries

☒ Use a layout page:

(Leave empty if it is set in a Razor \_viewstart file)

Controller name: Photo

Add Cancel

```

7 namespace PhotoSharingApplication.Models
8 {
9     6 references
10    public class PhotoSharingContext : DbContext
11    {
12        1 reference
13        public PhotoSharingContext() : base()
14        {
15            this.Database.CommandTimeout = 180;
16        }
17        8 references
18        public DbSet<Photo> Photos { get; set; }
19        1 reference
20        public DbSet<Comment> Comments { get; set; }
21    }
22 }

```

### The virtual keyword - Loading Data

- Lazy Loading
- Eager loading
- Explicit loading

Use **virtual** keyword, when you want to load data with lazy loading.

**lazy loading** is the process whereby an entity or collection of entities is automatically loaded from the database the first time it is accessed.

For example, when using the Blog entity class defined below, *the related Posts will be loaded the first time the Posts navigation property is accessed:*

```

public class Blog
{
    public int BlogId { get; set; }
    public string Name { get; set; }

```

```

    public string Url { get; set; }
    public string Tags { get; set; }
    public virtual ICollection<Post> Posts { get; set; }
}

```

- Lazy loading can be turned off by making the Posts property non-virtual.

If lazy loading is off, Loading of the Posts collection can still be achieved using eager loading (using **Include** method) or Explicitly loading related entities (using **Load** method).

Eagerly Loading:

```

using (var context = new BloggingContext())
{
    // Load all blogs and related posts
    var blogs1 = context.Blogs
        .Include(b => b.Posts)
        .ToList();
}

```

Explicitly Loading:

```

using (var context = new BloggingContext())
{
    var blog = context.Blogs.Find(1);

    // Load the posts related to a given blog
    context.Entry(blog).Collection(p => p.Posts).Load();
}

```

Eager loading is the process whereby a query for one type of entity also loads related entities as part of the query. Eager loading is achieved by use of the Include method. For example, the queries below will load princesses and all the unicorns related to each princess.

```

using (var context = new UnicornsContext())
{
    // Load all princesses and related unicorns
    var princesses1 = context.Princesses
        .Include(p => p.Unicorns)
        .ToList();

    // Load one princess and her related unicorns
    var princess1 = context.Princesses
        .Where(p => p.Name == "Cinderella")
        .Include(p => p.Unicorns)
        .FirstOrDefault();

    // Load all princesses and related unicorns using a string
    // to specify the relationship
    var princesses2 = context.Princesses
        .Include("Unicorns")
        .ToList();
}

```

```
// Load one princess and her related unicorns using a string
// to specify the relationship
var princess2 = context.Princesses
    .Where(p => p.Name == "Cinderella")
    .Include("Unicorns")
    .FirstOrDefault();
}
```

Note that Include is an extension method in the System.Data.Entity namespace so make sure you are using that namespace.

### Eagerly loading multiple levels of related entities

It is also possible to eagerly load multiple levels of related entities. The queries below show examples of how to do this for both collection and reference navigation properties.

```
using (var context = new UnicornsContext())
{
    // Load all castles, all related ladies-in-waiting, and all related
    // princesses
    var castles1 = context.Castles
        .Include(c => c.LadiesInWaiting.Select(b => b.Princess))
        .ToList();

    // Load all unicorns, all related princesses, and all related ladies
    var unicorns1 = context.Unicorns
        .Include(u => u.Princess.LadiesInWaiting)
        .ToList();

    // Load all castles, all related ladies, and all related princesses
    // using a string to specify the relationships
    var castles2 = context.Castles
        .Include("LadiesInWaiting.Princess")
        .ToList();

    // Load all unicorns, all related princesses, and all related ladies
    // using a string to specify the relationships
    var unicorns2 = context.Unicorns
        .Include("Princess.LadiesInWaiting")
        .ToList();
}
```

Note that it is not currently possible to filter which related entities are loaded. Include will always bring in all related entities.

Use LINQ to apply filters to the query before executing it with a call to a LINQ extension method such as ToList, Load, etc.

```
using (var context = new UnicornsContext())
{
    var princess = context.Princesses.Find(1);

    // Load the unicorns starting with B related to a given princess
    context.Entry(princess)
```

```
.Collection(p => p.Unicorns)
.Query()
.Where(u => u.Name.StartsWith("B"))
.Load();

// Load the unicorns starting with B related to a given princess
// using a string to specify the relationship
context.Entry(princess)
    .Collection("Unicorns")
    .Query().Cast<Unicorn>()
    .Where(u => u.Name.StartsWith("B"))
    .Load();
}
```

**\*\***When using the Query method it is usually best to turn off lazy loading for the navigation property. This is because otherwise the entire collection may get loaded automatically by the lazy loading mechanism either before or after the filtered query has been executed.

Note that while the relationship can be specified as a string instead of a lambda expression, the returned IQueryable is not generic when a string is used and so the Cast method is usually needed before anything useful can be done with it.

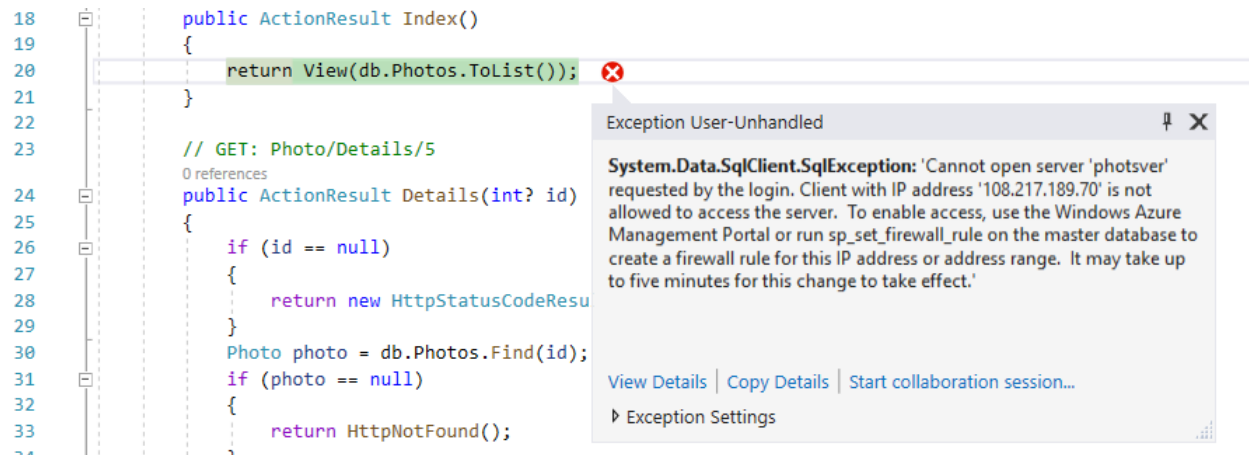
### Using Query to count related entities without loading them

Sometimes it is useful to know how many entities are related to another entity in the database without actually incurring the cost of loading all those entities. The Query method with the LINQ Count method can be used to do this. For example:

```
using (var context = new UnicornsContext())
{
    var princess = context.Princesses.Find(1);

    // Count how many unicorns the princess owns
    var unicornHaul = context.Entry(princess)
        .Collection(p => p.Unicorns)
        .Query()
        .Count();
}
```

## Stuff



## HTML Helpers

### Space

### Areas

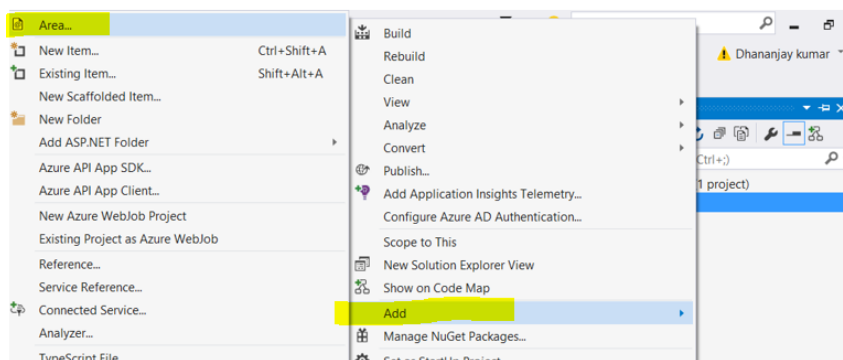
To physically partition a web project in separate, smaller functional units in an ASP.NET MVC project with their own set of controllers, views, and models.

For example; implementing an E-Commerce application with multiple business units, such as Checkout, Billing, and Search etc. use ASP.NET MVC Areas to physically partition the business components in the same project.

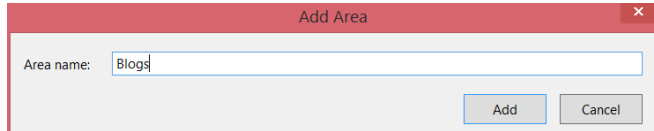
You cannot use routing or web.config files to secure your MVC application. The only supported way to secure your MVC application is to apply the [Authorize] attribute to each controller and action method (except for the login/register methods). Making security decisions based on the current area is a Very Bad Thing and will open your application to vulnerabilities.

### Creating Areas

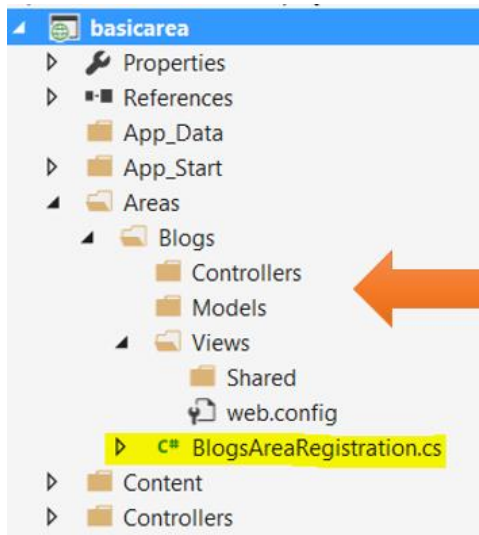
Creating Areas in an MVC project is very simple. Simply right click on the **project->Add->Area** as shown in the image below:



Here you will be asked to provide the Area name. In this example, let's name the Area "Blogs" and click on Add.



Let us stop for a moment here and explore the project. We will find a folder **Areas** has been added, and inside the Areas folder, we will find a subfolder with the name Blogs, which is the area we just created. Inside the Blogs subfolder, we will find folders for MVC components Controllers, Views, and Models.

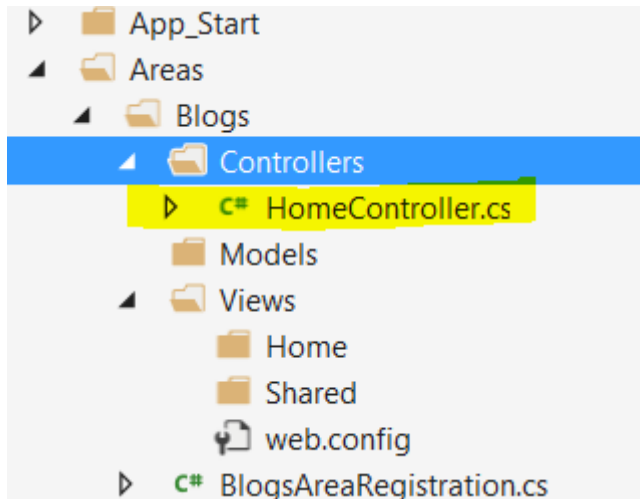


In the Area Blogs folder we will find a class **BlogAreaRegistration.cs**. In this class, routes for the Blog Area have been registered.

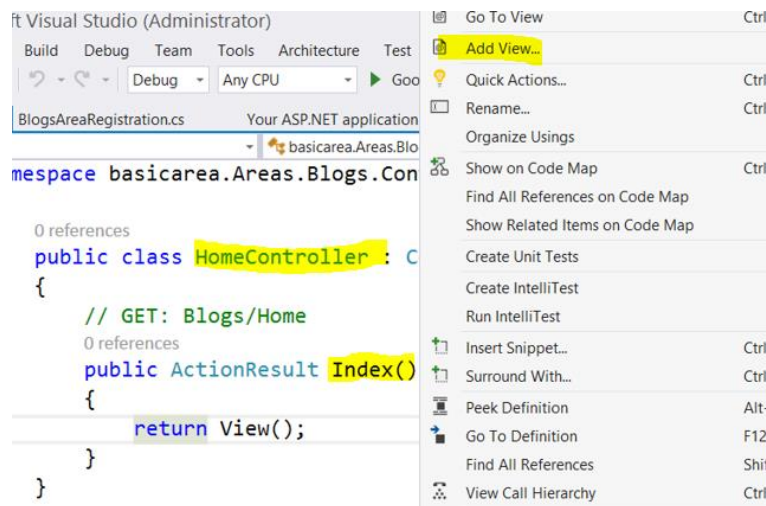
```
0 references
public override void RegisterArea(AreaRegistrationContext context)
{
    context.MapRoute(
        "Blogs_default",
        "Blogs/{controller}/{action}/{id}",
        new { action = "Index", id = UrlParameter.Optional }
    );
}
```

Now, we can add Controllers, Models, and the Views in the Area in the same way we add them normally in an MVC project. For example, to add a controller, let's right click on the controller folder in the Blogs folder and click on Add->Controller. So let us say we have added a HomeController in the Blogs controller. You will find the added controller in the project as shown in the image below:

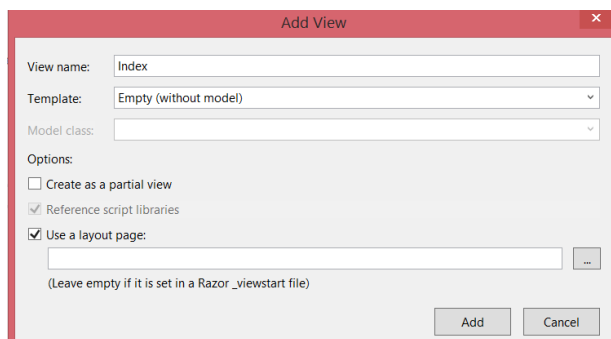




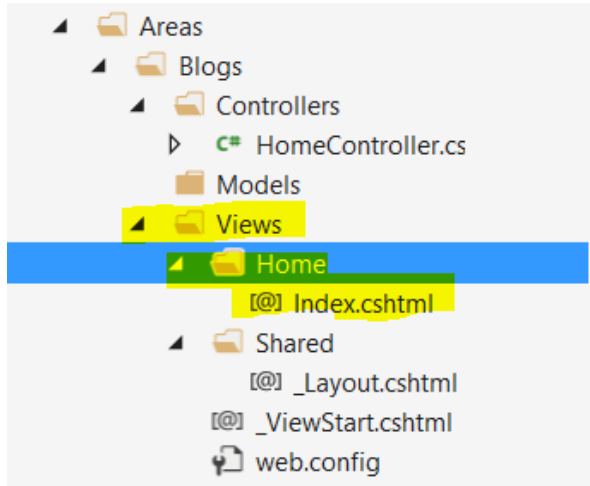
You will find that the HomeController has a method called Index. To create a view, right click on the Index action and select Add View as shown in the image below:



On the next screen, we need to select the view template, model class, and others. To keep things simpler, let us leave everything default and click on the Add button to create a View for the Index action in the Home controller of the Blogs Area.



We will see that a view was created inside the Views subfolder of the Blogs folder as shown in the image below:



To verify, let us go ahead and change the title of the view as shown in the image below:

```
@{  
    ViewBag.Title = "Blogs Area Home Index";  
}  
  
<h2>Index</h2>
```

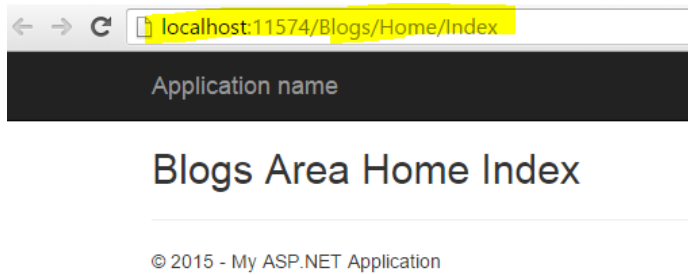
So far we have created:

1. An Area with the name Blogs
2. A controller inside that, named Home
3. A view for the Index action for the Home controller
4. Change the title of the view

As a last step to work with Areas, we need to verify whether the Areas are registered in the App\_Start of the project or not. To do this, open global.asax and add the highlighted line of code below (if it's not there already):

```
public class MvcApplication : System.Web.HttpApplication  
{  
    0 references  
    protected void Application_Start()  
    {  
        AreaRegistration.RegisterAllAreas();  
        FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);  
        RouteConfig.RegisterRoutes(RouteTable.Routes);  
        BundleConfig.RegisterBundles(BundleTable.Bundles);  
    }  
}
```

Now that we have created the Areas, let us go ahead and run the application and notice the URL.



As highlighted, to invoke the controller of the area, we need to use:

**baseurl/areaname/controllername/{actionname}**

In this case Home controller of Blog area is invoked by:

**Baseurl/Blogs/Home/Index**

Set Authorization for Areas (Discussion)

The currently accepted answer is not the most secure solution because it requires the developer to **always** remember to inherit that new base class for any new controllers or actions ("blacklisting"; allowing users access to everything unless an action is manually restricted). This especially causes problems when new developers, unacquainted with your rituals, are introduced to the project. It is easy to forget to inherit the proper controller class if done that way, especially after having taken your eyes off the project for weeks, months, or years. If a developer forgets to inherit, it isn't obvious that there is a security vulnerability in the project.

A more secure solution to this problem is to deny access to **all** requests, then decorate each action with the roles that are allowed access to the actions ("whitelisting"; preventing access to all users unless manually allowed). Now if a developer forgets to whitelist the proper authorization, the users will let you know and it's as simple as looking at other controllers for a reminder about how to give proper access. However, at least there is no major security vulnerability.

In App\_Start/FilterConfig.cs file, modify the FilterConfig class:

```
public static void RegisterGlobalFilters(GlobalFilterCollection filters)
{
    ...

    //Deny access to all controllers and actions so that only logged in
    Administrators can access them by default
    filters.Add(new System.Web.Mvc.AuthorizeAttribute() { Roles =
    "Administrator" });
}
```

This makes all actions inaccessible unless the user is logged in as an Administrator. Then for each action that you want a different authorized user to have access to, you simply decorate it with `[OverrideAuthorization]` and `[Authorize]`.

In your business logic, this allows you to use the Authorize attribute in a variety of ways without ever needing to worry about unauthorized users from accessing any functionality. Below are some examples.

**Example 1** - Only logged in Administrator and Dispatcher users will be allowed to access Index () Get and Post methods.

```
public class MarkupCalculatorController : Controller //Just continue using
the default Controller class.
{
    // GET: MarkupCalculator
    [OverrideAuthorization]
    [Authorize(Roles = "Administrator,Dispatcher")]
    public ActionResult Index()
    {
        //Business logic here.

        return View(...);
    }

    // POST: DeliveryFeeCalculator
    [HttpPost]
    [ValidateAntiForgeryToken]
    [OverrideAuthorization]
    [Authorize(Roles = "Administrator,Dispatcher")]
    public ActionResult Index([Bind(Include = "Price,MarkedupPrice")]
MarkupCalculatorVM markupCalculatorVM)
    {
        //Business logic here.

        return View(...);
    }
}
```

**Example 2** - Only authenticated users will be allowed to access the Home controller's Index () method.

```
public class HomeController : Controller
{
    [OverrideAuthorization]
    [Authorize] //Allow all authorized (logged in) users to use this action
    public ActionResult Index()
    {
        return View();
    }
}
```

**Example 3** - Unauthenticated users (i.e. anonymous users) can be allowed to access methods by using the [AllowAnonymous] attribute. This also automatically overrides the global filter without needing the [OverrideAuthorization] attribute.

```
// GET: /Account/Login
[AllowAnonymous]
public ActionResult Login(string returnUrl)
{
    ViewBag.ReturnUrl = returnUrl;
```

```
        return View();
    }

    //
    // POST: /Account/Login
    [HttpPost]
    [AllowAnonymous]
    [ValidateAntiForgeryToken]
    public async Task<ActionResult> Login(LoginViewModel model, string returnUrl)
    {
        ...
    }
```

**Example 4** - Only admins will be allowed access to methods that lack the [Authorize] attribute.

```
public class LocationsController : Controller
{
    // GET: Locations
    public ActionResult Index()
    {
        //Business logic here.
        return View(...);
    }
}
```

#### Some notes.

You must use the [OverrideAuthorization] attribute if you want to limit the access to a particular action to specific roles. Otherwise, the [Authorize] attribute properties will be ignored and only the default role (Administrator in my example) will be allowed, even if you specify other roles (e.g. Dispatcher, etc.) because of the global filter. Any unauthorized users will be redirected to the login screen.

Using the [OverrideAuthorization] attribute causes the action to ignore the global filter you set. Therefore, you **must** reapply the [Authorize] attribute whenever you use the override so that the action remains secure.

#### Regarding whole areas and controllers

To restrict by areas, as you are asking, put the [OverrideAuthorization] and [Authorize] attributes on the controller instead of the individual actions

## \$ - string interpolation (C# reference)

```
$"{DateTime.UtcNow}: {logData}"
```

- <https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/tokens/interpolated>
- Example: <https://stackoverflow.com/questions/32878549/whats-with-the-dollar-sign-string>

It's the new feature in C# 6 called **Interpolated Strings**.

The easiest way to understand it is: an interpolated string expression creates a string by **replacing the contained expressions with the ToString** representations of the expressions' results.

For example, you have class **Point**:

```
public class Point
{
    public int X { get; set; }

    public int Y { get; set; }
}
```

Create 2 instances:

```
var p1 = new Point { X = 5, Y = 10 };
var p2 = new Point { X = 7, Y = 3 };
```

Now, you want to **output it to the screen**. The 2 ways that you usually use:

```
Console.WriteLine("The area of interest is bounded by (" + p1.X + "," + p1.Y + ") and (" + p2.X + "," + p2.Y + ")");
```

As you can see, concatenating string like this makes the code hard to read and error-prone. You may use `string.Format()` to make it nicer:

```
Console.WriteLine(string.Format("The area of interest is bounded by({0},{1}) and ({2},{3})", p1.X, p1.Y, p2.X, p2.Y));
```

This creates a new problem:

1. You have to maintain the number of arguments and index yourself. If the number of arguments and index are not the same, it will generate a runtime error.

For those reasons, we should use new feature:

```
Console.WriteLine($"The area of interest is bounded by ({p1.X},{p1.Y}) and ({p2.X},{p2.Y})");
```

The compiler now maintains the placeholders for you so you don't have to worry about indexing the right argument because you simply place it right there in the string.

## Static Files

How to serve static files such as html, JavaScript, CSS, or image files on HTTP request without any server-side processing.

ASP.NET Core application cannot serve static files by default. We must include `Microsoft.AspNetCore.StaticFiles` middleware in the request pipeline.

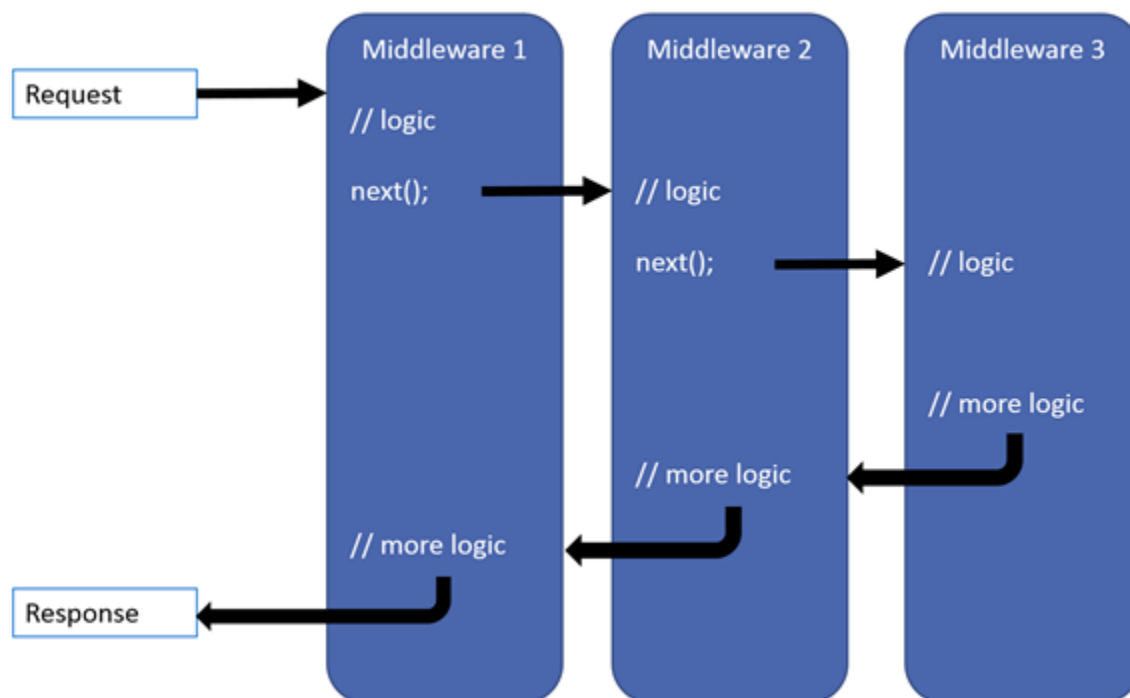
<https://www.tutorialsteacher.com/core/aspnet-core-static-file>

Static files in ASP.NET Core: <https://docs.microsoft.com/en-us/aspnet/core/fundamentals/static-files?view=aspnetcore-3.1>

ASP.NET Core Middleware:

<https://docs.microsoft.com/en-us/aspnet/core/fundamentals/middleware/?view=aspnetcore-3.1>

The ASP.NET Core request pipeline consists of a sequence of request delegates, called one after the other. The following diagram demonstrates the concept. The thread of execution follows the black arrows.



Each delegate can perform operations before and after the next delegate. Exception-handling delegates should be called early in the pipeline, so they can catch exceptions that occur in later stages of the pipeline.

The simplest possible ASP.NET Core app sets up a single request delegate that handles all requests. This case doesn't include an actual request pipeline. Instead, a single anonymous function is called in response to every HTTP request.

```
public class Startup
{
    public void Configure(IApplicationBuilder app)
    {
        app.Run(async context =>
        {
            await context.Response.WriteAsync("Hello, World!");
        });
    }
}
```

Chain multiple request delegates together with [Use](#). The next parameter represents the next delegate in the pipeline. You can short-circuit the pipeline by *not* calling the *next* parameter. You can typically perform actions both before and after the next delegate, as the following example demonstrates:

```
public class Startup
{
    public void Configure(IApplicationBuilder app)
    {
        app.Use(async (context, next) =>
        {
            // Do work that doesn't write to the Response.
            await next.Invoke();
            // Do logging or other work that doesn't write to the Response.
        });

        app.Run(async context =>
        {
            await context.Response.WriteAsync("Hello from 2nd delegate.");
        });
    }
}
```

When a delegate doesn't pass a request to the next delegate, it's called *short-circuiting the request pipeline*. Short-circuiting is often desirable because it avoids unnecessary work.

## References

1. C# - Interface:
  - a. <https://www.tutorialsteacher.com/csharp/csharp-interface>
2. ASP.NET Web API
  - a. Works more or less the same way as ASP.NET MVC web application except that it sends data as a response instead of html view.



- b. It is like a webservice or WCF service but the exception is that it only supports HTTP protocol.
    - c. <https://www.tutorialsteacher.com/webapi/what-is-web-api>
  3. Working with DbContext:
    - a. <https://docs.microsoft.com/en-us/ef/ef6/fundamentals/working-with-dbcontext>
  4. Fundamentals of garbage collection:
    - a. In the common language runtime (CLR), the garbage collector serves as an automatic memory manager. It provides the following benefits:
      - i. Enables you to develop your application without having to free memory.
      - ii. Allocates objects on the managed heap efficiently.
      - iii. Reclaims objects that are no longer being used, clears their memory, and keeps the memory available for future allocations. Managed objects automatically get clean content to start with, so their constructors do not have to initialize every data field.
      - iv. Provides memory safety by making sure that an object cannot use the content of another object.
    - b. <https://docs.microsoft.com/en-us/dotnet/standard/garbage-collection/fundamentals>
  5. Relationships, navigation properties and foreign keys:
    - a. <https://docs.microsoft.com/en-us/ef/ef6/fundamentals/relationships>
  6. Code First Data Annotations:
    - a. <https://docs.microsoft.com/en-us/ef/ef6/modeling/code-first/data-annotations>
  7. The "virtual" keyword:
    - a. it can control [lazy loading](#) -- if you use the virtual keyword on an ICollection/one-to-many relationship property, it will be lazy-loaded by default, whereas if you leave the virtual keyword out, it will be eager-loaded.
    - b. [More efficient change tracking](#). If you meet all the following requirements then your change tracking can use a more efficient method by hooking your virtual properties. From the link:
  8. Loading Related Entities:
    - a. <https://blogs.msdn.microsoft.com/adonet/2011/01/31/using-dbcontext-in-ef-4-1-part-6-loading-related-entities/>
  9. Role-based authorization in ASP.NET Core
    - a. <https://docs.microsoft.com/en-us/aspnet/core/security/authorization/roles?view=aspnetcore-2.2>
  10. Areas in ASP.NET MVC
    - a. <https://dzone.com/articles/areas-in-aspnet-mvc>
  11. Areas in ASP.NET Core
    - a. <https://docs.microsoft.com/en-us/aspnet/core/mvc/controllers/areas?view=aspnetcore-2.2>
  12. Areas in ASP.NET MVC 4
    - a. <https://www.codeproject.com/Articles/714356/Areas-in-ASP-NET-MVC>
  13. How can we set authorization for a whole area in ASP.NET MVC?
    - a. <https://stackoverflow.com/questions/2319157/how-can-we-set-authorization-for-a-whole-area-in-asp-net-mvc>
  14. Securing your ASP.NET MVC 3 Application
  15. <https://blogs.msdn.microsoft.com/rickandy/2011/05/02/securing-your-asp-net-mvc-3-application/>
  16. To be continued...

**ASP.NET Web API vs WCF**

Web API	WCF
Open source and ships with .NET framework.	Ships with .NET framework
Supports only HTTP protocol.	Supports HTTP, TCP, UDP and custom transport protocol.
Maps http verbs to methods	Uses attributes based programming model.
Uses routing and controller concept similar to ASP.NET MVC.	Uses Service, Operation and Data contracts.
Does not support Reliable Messaging and transaction.	Supports Reliable Messaging and Transactions.
Web API can be configured using HttpConfiguration class but not in web.config.	Uses web.config and attributes to configure a service.
Ideal for building RESTful services.	Supports RESTful services but with limitations.

**When to choose WCF?**

- Choose WCF if you use .NET Framework 3.5. Web API does not support .NET 3.5 or below.
- Choose WCF if your service needs to support multiple protocols such as HTTP, TCP, Named pipe.
- Choose WCF if you want to build service with WS-\* standards like Reliable Messaging, Transactions, Message Security.
- Choose WCF if you want to use Request-Reply, One Way, and Duplex message exchange patterns.

**When to choose ASP.NET Web API?**

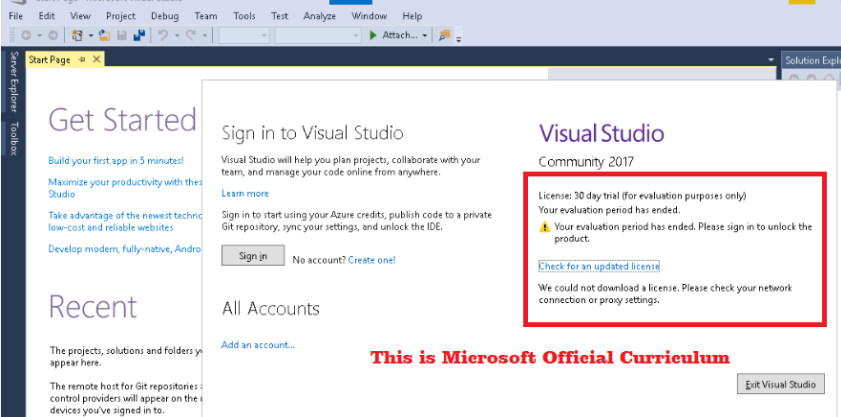
- Choose Web API if you are using .NET framework 4.0 or above.
- Choose Web API if you want to build a service that supports only HTTP protocol.
- Choose Web API to build RESTful HTTP based services.
- Choose Web API if you are familiar with ASP.NET MVC.

**MOC Issues 2020 – Still VS 2017 for Version ‘D’**

**Task 3: Add simple functionality**

1. In the **CakeStoreApi - Microsoft Visual Studio** in **Solution Explorer**, right-click **CakeStoreApi**, and then click **New Folder**.
2. In the **NewFolder** box, type **Models**, and then click **Enter**.
3. In the **CakeStoreApi - Microsoft Visual Studio** in **Solution Explorer**, right-click **Models**, and then click **Class**.
4. In the **Add New Item - CakeStoreApi** dialog **Name** box, type **CakeStore**, and then click **Add**.
5. In the **CakeStore.cs** code block, place the cursor at the second { (opening braces) sign, press Enter, and then type the following code:

```
public int Id { get; set; }
public string CakeType { get; set; }
public int Quantity { get; set; }
```



a deep dive to dependency injection in ASP.NET Core and MVC Core. We will walk through almost every conceivable option for injecting dependencies into your components.

Dependency injection is at the core of ASP.NET Core. It allows the components in your app to have improved testability. It also makes your components only dependent on *some* component that can provide the needed services.

As an example, here we have an interface and its implementing class:

```
public interface IDataService
{
    IList<DataClass> GetAll();
}

public class DataService : IDataService
{
    public IList<DataClass> GetAll()
    {
        //Get data...
        return data;
    }
}
```

If another service depends on `DataService`, they are dependent on this particular implementation. Testing a service such as that can be quite a lot more difficult. If instead the service depends on `IDataService`, they only care about the *contract* provided by the interface. It doesn't matter what implementation is given. It makes it possible for us to pass in a mock implementation for testing the service's behaviour.

## Service lifetime

Before we can talk about how injection is done in practice, it is *critical* to understand what is **service lifetime**. When a component requests another component through dependency injection, whether the instance it receives is unique to that instance of the component or not depends on the lifetime. Setting the lifetime thus decides how many times a component is instantiated, and if a component is shared.

There are 3 options for this with the built-in DI container in ASP.NET Core:

1. Singleton
2. Scoped
3. Transient

**Singleton** means only a single instance will ever be created. That instance is shared between all components that require it. The same instance is thus used always.

**Scoped** means an instance is created once per *scope*. A scope is created on every request to the application, thus any components registered as Scoped will be created once per request.

**Transient** components are created every time they are requested and are never shared.

It is important to understand that if you register component A as a singleton, it cannot depend on components registered with Scoped or Transient lifetime. More generally speaking:

A component cannot depend on components with a lifetime smaller than their own.

The consequences of going against this rule should be obvious, the component being depended on might be disposed before the dependent.

Typically you want to register components such as application-wide configuration containers as Singleton. Database access classes like Entity Framework contexts are recommended to be Scoped, so the connection can be re-used. Though if you want to run anything in parallel, keep in mind Entity Framework contexts cannot be shared by two threads. If you need that, it is better to register the context as Transient. Then each component gets their own context instance and can run in parallel.

## Service registration

Registering services is done in the `ConfigureServices(IServiceCollection)` method in your `Startup` class.

Here is an example of a service registration:

```
services.Add(new ServiceDescriptor(typeof(IDataService), typeof(DataService), ServiceLifetime.Transient));
```

That line of code adds `DataService` to the service collection. The service type is set to `IDataService` so if an instance of that type is requested, they get an instance of `DataService`. The lifetime is also set to `Transient`, so a new instance is created every time.

ASP.NET Core provides various extension methods to make registering services with various lifetimes and other settings easier.

Here is the earlier example using an extension method:

```
services.AddTransient<IDataService, DataService>();
```

Little bit easier right? Under the covers it calls the earlier of course, but this is just easier. There are similar extension methods for the different lifetimes with names you can probably guess.

If you want, you can also register on a single type (implementation type = service type):

```
services.AddTransient<DataService>();
```

But then of course the components must depend on the concrete type, which may be unwanted.

## Implementation factories

In some special cases, you may want to take over the instantiation of some service. In this case, you can register an *implementation factory* on the service descriptor. Here is an example:

```
services.AddTransient<IDataService, DataService>((ctx) =>
{
    IOtherService svc = ctx.GetService<IOtherService>();
```

```
//IOtherService svc = ctx.GetRequiredService<IOtherService>();  
return new DataService(svc);  
});
```

It instantiates `DataService` using another component `IOtherService`. You can get dependencies registered in the service collection with `GetService<T>()` or `GetRequiredService<T>()`.

The difference is that `GetService<T>()` returns `null` if it can't find the service. `GetRequiredService<T>()` throws an `InvalidOperationException` if it can't find it.

### Singletons registered as constant

If you want to instantiate a singleton yourself, you can do this:

```
services.AddSingleton<IDataService>(new DataService());
```

It allows for one very interesting scenario. Say `DataService` implements two interfaces. If we do this:

```
services.AddSingleton<IDataService, DataService>();  
services.AddSingleton<ISomeInterface, DataService>();
```

We get two instances. One for both interfaces. If we want to share an instance, this is one way to do it:

```
var dataService = new DataService();  
services.AddSingleton<IDataService>(dataService);  
services.AddSingleton<ISomeInterface>(dataService);
```

If the component has dependencies, you can build the service provider from the service collection and get the necessary dependencies from it:

```
IServiceProvider provider = services.BuildServiceProvider();  
  
IOtherService otherService = provider.GetRequiredService<IOtherService>();  
  
var dataService = new DataService(otherService);  
services.AddSingleton<IDataService>(dataService);  
services.AddSingleton<ISomeInterface>(dataService);
```

Note you should do this at the end of `ConfigureServices` so you have surely registered all dependencies before this.

### Injection

Now that we have registered our components, we can move to actually using them.

The typical way in which components are injected in ASP.NET Core is **constructor injection**. Other options do exist for different scenarios, but constructor injection allows you to define that this component will not work without these other components.

As an example, let's make a basic logging middleware component:

```
public class LoggingMiddleware
{
    private readonly RequestDelegate _next;

    public LoggingMiddleware(RequestDelegate next)
    {
        _next = next;
    }

    public async Task Invoke(HttpContext ctx)
    {
        Debug.WriteLine("Request starting");
        await _next(ctx);
        Debug.WriteLine("Request complete");
    }
}
```

There are **three** different ways of injecting components in middleware:

1. Constructor
2. Invoke parameter
3. HttpContext.RequestServices

Let's inject our component using all three:

```
public class LoggingMiddleware
{
    private readonly RequestDelegate _next;
    private readonly IDataService _svc;

    public LoggingMiddleware(RequestDelegate next, IDataService svc)
    {
        _next = next;
```

```
        _svc = svc;  
    }  
  
    public async Task Invoke(HttpContext ctx, IDataService svc2)  
    {  
        IDataService svc3 = ctx.RequestServices.GetService<IDataService>();  
        Debug.WriteLine("Request starting");  
        await _next(ctx);  
        Debug.WriteLine("Request complete");  
    }  
}
```

The middleware is *instantiated only once* during the app lifecycle, so **the component injected through the constructor is the same for all requests that pass through.**

The component injected as a parameter for `Invoke` is absolutely required by the middleware, and it will throw an `InvalidOperationException` if it can't find an `IDataService` to inject.

The third one uses the `RequestServices` property on the `HttpContext` to get an optional dependency using `GetService<T>()`. The property is of type `IServiceProvider`, so it works exactly the same as the provider in implementation factories. If you want to require the component, you can use `GetRequiredService<T>()`.

If `IDataService` was registered as *singleton*, we get the same instance in all of them.

If it was registered as *scoped*, `svc2` and `svc3` will be the same instance, but different requests get different instances.

In the case of *transient*, all of them are always different instances.

Use cases for each approach:

1. **Constructor:** Singleton components that are needed for all requests
2. **Invoke parameter:** Scoped and transient components that are always necessary on requests
3. **RequestServices:** Components that may or may not be needed based on runtime information

I would try to avoid using `RequestServices` if possible, and only use it when the middleware must be able to function without some component as well.

## Startup class

In the constructor of the Startup class, you can at least inject `IHostingEnvironment` and `ILoggerFactory`. They are the only two interfaces mentioned in the [official documentation](#). There may be others, but I am not aware of them.

```
public Startup(IHostingEnvironment env, ILoggerFactory loggerFactory)  
{
```

```
}
```

The `IHostingEnvironment` is used typically to setup configuration for the application. With the `ILoggerFactory` you can setup logging.

The `Configure` method allows you to inject any components that have been registered.

```
public void Configure(  
    IApplicationBuilder app,  
    IHostingEnvironment env,  
    ILoggerFactory loggerFactory,  
    IDataService dataSvc)  
{  
}
```

So if there are components that you need during the pipeline configuration, you can simply require them there.

If you use `app.Run()/app.Use()/app.UseWhen()/app.Map()` to register simple middleware on the pipeline, you cannot use constructor injection. Actually the only way to get the components you need is through `ApplicationServices/RequestServices`.

Here are some examples:

```
IDataService dataSvc2 = app.ApplicationServices.GetService<IDataService>();  
app.Use((ctx, next) =>  
{  
    IDataService svc = ctx.RequestServices.GetService<IDataService>();  
    return next();  
});  
app.Map("/test", subApp =>  
{  
    IDataService svc1 = subApp.ApplicationServices.GetService<IDataService>();  
    subApp.Run((context =>  
    {  
        IDataService svc2 = context.RequestServices.GetService<IDataService>();  
        return context.Response.WriteAsync("Hello!");  
    }));  
});  
app.MapWhen(ctx => ctx.Request.Path.StartsWithSegments("/test2"), subApp =>
```



```
{  
    IDataService svc1 = subApp.ApplicationServices.GetService<IDataService>();  
    subApp.Run(ctx =>  
    {  
        IDataService svc2 = ctx.RequestServices.GetService<IDataService>();  
        return ctx.Response.WriteAsync("Hello!");  
    });  
});
```

So you can request components at configuration time through `ApplicationServices` on the `IApplicationBuilder`, and at request time through `RequestServices` on the `HttpContext`.

### Injection in MVC Core

The most common way for doing dependency injection in MVC is *constructor injection*.

You can do that pretty much anywhere. In controllers you have a few options:

```
public class HomeController : Controller  
{  
    private readonly IDataService _dataService;  
  
    public HomeController(IDataService dataService)  
    {  
        _dataService = dataService;  
    }  
  
    [HttpGet]  
    public IActionResult Index([FromServices] IDataService dataService2)  
    {  
        IDataService dataService3 = HttpContext.RequestServices.GetService<IDataService>();  
        return View();  
    }  
}
```

If you wish to get dependencies later based on runtime decisions, you can once again use `RequestServices` available on the `HttpContext` property of the `Controller` base class (well, `ControllerBase` technically).

You can also inject services required by specific actions by adding them as parameters and decorating them with the `FromServicesAttribute`. This instructs MVC Core to get it from the service collection instead of trying to do model binding on it.

### **Razor views**

You can also inject components in Razor views with the new `@inject` keyword:

```
@using Microsoft.AspNetCore.Mvc.Localization
@inject IViewLocalizer Localizer
```

Here we inject a view localizer in `_ViewImports.cshtml` so we have it available in all views as `Localizer`.

You should not abuse this mechanism to bring data to views that should come from the controller.

### **Tag helpers**

Constructor injection also works in **tag helpers**:

```
[HtmlTargetElement("test")]
public class TestTagHelper : TagHelper
{
    private readonly IDataService _dataService;

    public TestTagHelper(IDataService dataService)
    {
        _dataService = dataService;
    }
}
```

### **View components**

Same with **view components**:

```
public class TestViewComponent : ViewComponent
{
    private readonly IDataService _dataService;

    public TestViewComponent(IDataService dataService)
    {
        _dataService = dataService;
    }
}
```

```
public async Task<IViewComponentResult> InvokeAsync()
{
    return View();
}
}
```

View components also have the `HttpContext` available, and thus have access to `RequestServices`.

### **Filters**

**MVC filters** also support constructor injection, as well as having access to `RequestServices`:

```
public class TestActionFilter : ActionFilterAttribute
{
    private readonly IDataService _dataService;

    public TestActionFilter(IDataService dataService)
    {
        _dataService = dataService;
    }

    public override void OnActionExecuting(ActionExecutingContext context)
    {
        Debug.WriteLine("OnActionExecuting");
    }

    public override void OnActionExecuted(ActionExecutedContext context)
    {
        Debug.WriteLine("OnActionExecuted");
    }
}
```

**However**, we can't add the attribute as usual on a controller since it has to get dependencies at runtime.

We have these two options for adding it on controller- or action level:

```
[TypeFilter(typeof(TestActionFilter))]
```

```
public class HomeController : Controller
{
}

// or

[ServiceFilter(typeof(TestActionFilter))]
public class HomeController : Controller
{
}
```

The key difference is that `TypeFilterAttribute` will figure out what are the filters dependencies, fetches them through DI, and creates the filter. `ServiceFilterAttribute` on the other hand attempts to find the filter from the service collection!

To make `[ServiceFilter(typeof(TestActionFilter))]` work, we need a bit more configuration:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddTransient<TestActionFilter>();
}
```

Now `ServiceFilterAttribute` can find the filter.

If you wanted to add the filter globally:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc(mvc =>
    {
        mvc.Filters.Add(typeof(TestActionFilter));
    });
}
```

There is no need to add the filter to the service collection this time, it works as if you had added a `TypeFilterAttribute` on every controller.

### HttpContext

I've mentioned `HttpContext` multiple times now. What about if you want to access `HttpContext` outside of a controller/view/view component? To access the currently signed-in user's claims for example?

You can simply inject `IHttpContextAccessor`, like here:

```
public class DataService : IDataService
```

```
{  
    private readonly HttpContext _httpContext;  
  
    public DataService(IOtherService svc, IHttpContextAccessor contextAccessor)  
    {  
        _httpContext = contextAccessor.HttpContext;  
    }  
    // ...  
}
```

This allows your service layer access to **HttpContext** without requiring you to pass it through every method call.

<https://joonasw.net/view/aspnet-core-di-deep-dive>