

# A Tour through Core

---

ASP .NET MVC with .NET Core 2.0, Angular and Typescript

**Paul Taylor**  *June 2018*

# Pre-requisites

---

**Assumed technical knowledge** This text assumes that the coder has basic knowledge of **C#**, **HTML** and **CSS**. The book will cover more complex applications of the language, and will provide a brief explanation of each concept as we introduce them, but familiarity with variables, constructors, etc, is assumed.

Existing **.NET Programmers** yet to make the transition to **MVC**, **.NET Core**, **Typescript** or **Angular**, or all of the above, should benefit from this resource.


**Software** The exercises in the book require that the reader installs software, if he or she has not done so already. The following software packages are required.


- SQL Server 2008 or higher
  - Community edition available
- Visual Studio 2017 Community Edition
- git - command line

**External resource** the book makes use of external resources, such as **nuget**. Some exercise cannot be completed without being online.

## .Net Core

We'll be using **.NET Core 2.0** for these exercises.

 **.NET Core 2.0** is a brand new web framework which unifies Microsoft's web stack, making **.ASP .NET MVC** and something called **WebAPI** pretty much the same thing. It also runs the very latest implementation of ASP .NET MVC, is platform independent and full of some of the latest features.

 **WebApi** refers to a separate type of controller, geared toward returning **objects**, rather than views.

# Part One - Model View Controller

---

The concept of **MVC**, or **Model View Controller**, is older than the world wide web. First introduced into Smalltalk '76, a language taking part of its name from the year it was made in, the ideas behind the concept are over forty years old. The web, by contrast, began life in 1989, when Tim Berners Lee proposed the HTTP standard.

It would take almost another decade before the principles of MVC made their way into the construction of modern web sites. The release of WebObjects in 1996, finally unified web technology with MVC principles. In the years afterward, WebObjects would get ported to Java, Ruby on Rails would introduce the concept from the ground up, while Microsoft and others would begin to produce their own takes on the technology.

Today, the pattern is the **de facto** pattern for new web-facing work. The first part of this book focuses on the technology.

## MVC in the abstract.

At its heart, MVC is about the **separation of concerns**, with each letter in the TLA specifying a separate concern.

- **Model**
- **View**
- **Controller**

 **TLA** - Three letter abbreviation 😊

An MVC application is a system that has three main interconnecting components. The Wikipedia description is as good as any.

## Components

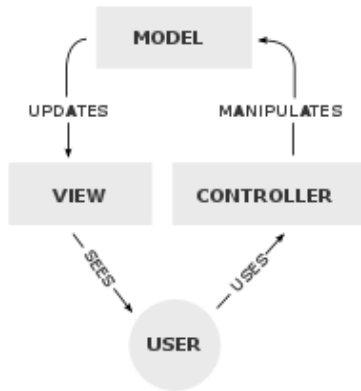
- The model is the central component of the pattern. It expresses the application's behavior in terms of the problem domain, independent of the user interface. It directly manages the data, logic and rules of the application.
  - A view can be any output representation of information, such as a chart or a diagram. Multiple views of the same information are possible, such as a bar chart for management and a tabular view for accountants.
  - The third part or section, the controller, accepts input and converts it to commands for the model or view.
- Interactions

In addition to dividing the application into three kinds of components, the model–view–controller design defines the interactions between them.[8]

## Interactions

- The model is responsible for managing the data of the application. It receives user input from the controller.
- The view means presentation of the model in a particular format.
- The controller responds to the user input and performs interactions on the data model objects. The controller receives the input, optionally validates it and then passes the input to the model.

This diagram, also lifted from Wikipedia, shows the interactions.



We'll redefine MVC in much simpler terms by the end of this lecture.

## Before MVC

### The bad old days

MVC took awhile to gain prominence in the development of modern day applications. Early web technologies, such as classic ASP or PHP, used inline code to control program flow and execution.

This tiny example of Classic ASP shows how we used to code.

```
<!DOCTYPE html>
<html>
<body>

<%
dim i
for i=1 to 6
    response.write("<h" & i & ">Heading " & i & "</h" & i & ">")
next
%>

</body>
</html>
```

Design and control flow would exist in the same place, inline. Web servers would treat each page as a separate program. The inline approach came loaded down with many disadvantages.

- **Maintainability**

- The mix of code and design meant that immediately, any maintainer needed to be aware of two quite different disciplines, development and design. Often they were not, leading to problems with reliability and software quality.
- The scope of any inline app was very difficult to know. Under such technologies, web servers were configured to treat with a certain extension as an executable program.
- Version control programs would often over-version as a result of small cosmetic changes.

- **Security**

- Sideways copying, that is to say, someone making a small change to an existing file, was common.
- Those seeking to secure the app would need to be on top of their game to ensure that no sideways copies, or other harmful scripts, were on the machine.

**Coping strategies**

The disadvantages of page based systems were apparent enough before the introduction of **.ASP .NET MVC**, to cause people to write template pages, or server side include pages, commonly used but infrequently changed, in a bid to mitigate the free for all that had sprung up around inline development.

## ASP .NET Webforms

Microsoft solved some of the problems associated with Classic ASP with ASP .NET and WebForms, but not without introducing many more, and not focused on providing a true separation of concerns. During that phase of its history, Microsoft was more concerned with helping Visual Basic developers become web developers.

The new innovation was a feature called **code behind**, in which program flow and execution were managed in a separate file to page layout and structure. This largely solved the issue of inline coding. Design and code were separate. The problem was that no designer knew what any of the design did, or what it would look like when it was finally spat out.

```
<asp:Repeater id="Repeater1" runat="server">
  <HeaderTemplate>
    <table border=1>
      <tr>
        <td><b>Company</b></td>
        <td><b>Symbol</b></td>
      </tr>
    </HeaderTemplate>
    <ItemTemplate>
      <tr>
        <td> <#% DataBinder.Eval(Container.DataItem, "Name") %> </td>
        <td> <#% DataBinder.Eval(Container.DataItem, "Ticker") %> </td>
      </tr>
    </ItemTemplate>
    <FooterTemplate>
      </table>
    </FooterTemplate>
</asp:Repeater>
<b>Repeater2:</b>
<asp:Repeater id="Repeater2" runat="server">

  <HeaderTemplate>
    Company data:
  </HeaderTemplate>

  <ItemTemplate>
    <#% DataBinder.Eval(Container.DataItem, "Name") %>
    (<#% DataBinder.Eval(Container.DataItem, "Ticker") %>)
  </ItemTemplate>
  <SeparatorTemplate>, </SeparatorTemplate>
</asp:Repeater>
```

The introduction of proprietary tags made the development of web applications and increasingly programmer led venture, made worse by early implementations of the technology, which offered wildly inconsistent performance out of the box on any other browser but Internet Explorer.

Microsoft also introduced another problem with Web Forms; **viewstate\***.



**Viewstate** was a technology that allowed information to be automatically persisted across separate web requests. Convenience came at great cost. Page weights, the file size of each page served to end users, ballooned, leaving users on slower connections (plenty of them about in the early 2000s) with a very

dissatisfying experience compared to other platforms. Perhaps most damningly, ASP .NET was slower than the product it was meant to replace.

As coders learned to work around the deficiencies of the platform, WebForms apps did become easier to maintain, if only because many developers were rolling their own versions of what Microsoft would eventually introduce itself. A Model View Controller implementation on the .NET Platform.

## ASP .NET MVC

In 2009, ASP .NET MVC 1.0 was released for the .NET Platform. Its release coincided with a number of other emerging technologies, such as dependency injection and service based architecture. It has gained widespread adoption amongst former WebForms developers, cementing itself as the de facto standard for new web applications written for the .NET Platform.



The project, along with the Razor engine, was made open source in 2012. The implementation has gone through several revisions, up to 6.0 on traditional .NET platforms, already at version 2.0 on .NET Core. .NET Core is what we'll be using in this series of lectures and exercises.

The MVC implementation solved many of the problems that bogged down Classic ASP and ASP .NET WebForms. Views were designed to be just that. The proprietary server controls of WebForms disappeared, and they were not stuffed with program control flow. Where it existed at all, it was clean and verbose, something the designers could still work with confidently.

An MVC app would no longer run any old page with the right extension. The scope of an MVC app is directly controlled by the programmer, and can be quickly and easily determined by examining the controllers in the app. If a request isn't recognised by any of the controllers, it simply won't be found. It simply won't be served.

### A simpler definition of MVC

Though the switch to MVC was something of a shock to WebForms developers, most came up with a simplified definition concept of the various components in an MVC app.

**Model** - everything your app *can* do, its functions, its data, the complete and total scope of its functionality.

**Controller** - everything we'll *let* our app do with the model.

**View** everything we'll let our users *see* of the model.

How this book will run.

Mostly, we will be doing exercises, spending a brief amount of time before each exercise to explore the concepts we're about to approach. We're also going to develop this as any professional would, using the git versioning system, developing a feature at a time, and merging our completed work back into a master repository.


We will explore, step by step, the structure and operation of an MVC application. We will demystify a lot of the magic. We will be covering a technology and a way of thinking about applications that will serve you all well in the future, as well as teaching you skills that are commercially useful and industry standard at the present time.



# Exercise 1 - Introduction to Git and our first commit

---

Each of you should have access to a repository system called git. You can find this repository by pressing the start menu button, then typing **Git Bash**. Select it from the context menu.


 **What's this?** What you're looking at is a command line interface, designed to help you manage a git repository. A git repository is your ultimate undo function. Amongst other things, it allows to return to previous copies of work, if a release is found to be defective. It allows developers to independently develop new features without compromising the integrity of the existing application or code base.

We're going to create a new git repository in an empty directory.


## A brief primer on the commands we'll be using.

git bash is a port of bash, the default command line environment on Linux systems. We'll be using the following commands in this exercise.


cd

 **change directory** - move into the specified directory.


mkdir

 **make directory** - make a directory of the specified name in the current directory.

ls

 **list directory** - get a list of files and directories in the current directory.

git

 **git** - the git command line program. Does several things depending on the sub command passed in.

## Setting your user details

git tracks changes made by authors. You can set your name and email by typing the following commands.

```
git config --global user.name "John Doe"
git config --global user.email johndoe@example.com
```

## Creating the repository

Navigate to your Projects directory, and create a new directory called git. All of our git backed projects will live here.

Type:-

```
cd /c/Projects/git
mkdir empower
```

```
cd empower
git init
```

Congratulations. You've just created a git repository on the command line.



**Depressingly true story** - There are millions of professional developers that still haven't done the same



## Initial structure

There's no rule, but most developers follow a convention whereby git repositories have well known names, **doc**, intended for documentation, and **src**, intended for source code and other assets. Let's create these now.

Type:-

```
mkdir src
mkdir doc
```

You can now type in

```
sh ls
```

To verify that the directories are there.

## Creating a vanilla MVC project.

Open up your copy of Visual Studio 2017, and click New Project.

Select **.NET Core** from the Visual C# section. Select ASP .NET MVC .NET Core Application.

Set the directory to the **src** directory we created earlier, and make sure the box to create a new directory is checked.

The name of the project should be **Empower.Mvc**.

Click **Ok** when you're ready.

## Checking the Project in.

Before we take the tour of MVC, we're going to commit our work to git at this point.

We're going to use the built-in features of Visual Studio 2017 to do this. Click on **Team Explorer**. Click the little house, then click **Changes**. You should see a list of changes and a prompt to enter a commit message. Enter "Initial check in of vanilla project".

### What's just happened?

We've created a new git repository, we've created a vanilla ASP .NET MVC Core application, and we've committed that project to git.

Whatever happens next, we can get back to that baseline.

## Exercise two - the MVC mini-tour

---

This is the vanilla MVC project for .NET core.

Let's take a look at what it contains.

### Controllers

The *Controllers* directory contain the controllers for the MVC app. They define the scope of the app.

Right now, we have one controller defined, **HomeController**. Let's open it up and take a look.

```
namespace Empower.Mvc.Controllers
{
    public class HomeController : Controller
    {
        public IActionResult Index()
        {
            return View();
        }

        public IActionResult About()
        {
            ViewData["Message"] = "Your application description page.";

            return View();
        }

        public IActionResult Contact()
        {
            ViewData["Message"] = "Your contact page.";

            return View();
        }

        public IActionResult Error()
        {
            return View(new ErrorViewModel
            {
                RequestId = Activity.Current?.Id
                ?? HttpContext.TraceIdentifier
            });
        }
    }
}
```

There are three main controller actions, **Index**, the default action. If an action isn't specified, **Index** is assumed. The vanilla Microsoft project also contains controller actions for **About**, presumably some blurb and **Contact**, which we will improve upon later.



There is also an **Error** action, which is used to report any errors that occur while the application is running.

## Views

The Views directory contains the HTML pages (*cshtml* file extension). Inside, we have two sub directories, **Home**, which contains all the views for the **HomeController** and **Shared**, which contains views that are used by multiple controller actions, such as the loading of a shared layout page which maintains design consistency.

### \_ViewStart.cshtml

This file tells the application which **Layout** to use.



**Layouts** these are shared views that define overall page structure. They are used in conjunction with the **Views** we create ourselves to create a complete web page. **Layout pages** typically have the job of referencing external resources such as **stylesheets**.

```
@{
    Layout = "_Layout";
}
```

### Shared/\_Layout.cshtml

The default layout of the site is stored in Shared/\_Layout.cshtml.

It enforces a common style across the application and ensures that common elements do not needlessly break or deviate. It contains references to stylesheets, scripts and other external resources we'll use in creating the look and feel of the application.

It is a large file, but one of the most important areas is **RenderBody()**. It is there that the content of your controller's views will appear.

```
<div class="container body-content">
    @RenderBody()
    <hr />
    <footer>
        <p>&copy; 2018 - CoreMvc</p>
    </footer>
</div>
```

### Home/Contact.cshtml

Let's open up the Home/Contact.cshtml view to see how. Note that it contains just the information relevant to contact information. Unlike our previous examples from the land of WebForms, those are standard HTML tags, easily decipherable to a designer or front end developer.

The **@** symbol is a directive to start using *Razor* syntax, which allows data to be pulled from information in the view.

```
@{
    ViewData["Title"] = "Contact";
}
```

```
<h2>@ViewData["Title"]</h2>
<h3>@ViewData["Message"]</h3>

<address>
    One Microsoft Way<br />
    Redmond, WA 98052-6399<br />
    <abbr title="Phone">P:</abbr>
    425.555.0100
</address>

<address>
    <strong>Support:</strong> <a
href="mailto:Support@example.com">Support@example.com</a><br />
    <strong>Marketing:</strong> <a
href="mailto:Marketing@example.com">Marketing@example.com</a>
</address>
```

## Models

Right now, the project only contains one model, **ErrorViewModel**, created to contain details of any errors that occur.

```
namespace Empower.Mvc.Models
{
    public class ErrorViewModel
    {
        public string RequestId { get; set; }

        public bool ShowRequestId => !string.IsNullOrEmpty(RequestId);
    }
}
```

## Startup.cs

One of the most important parts of the app, and one of the most self-explanatory. This class is automatically executed to start the application and governs exactly what the application can do.

In this vanilla state, it's making a Configuration available, allowing static files to be served, setting up MVC and establishing the routes it will respond to.

```
namespace Empower.Mvc
{
    public class Startup
    {
        public Startup(IConfiguration configuration)
        {
            Configuration = configuration;
        }

        public IConfiguration Configuration { get; }
    }
}
```

```
// This method gets called by the runtime. Use this method to add services to
the container.
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc();
}

// This method gets called by the runtime. Use this method to configure the
HTTP request pipeline.
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
        app.UseBrowserLink();
    }
    else
    {
        app.UseExceptionHandler("/Home/Error");
    }

    app.UseStaticFiles();

    app.UseMvc(routes =>
    {
        routes.MapRoute(
            name: "default",
            template: "{controller=Home}/{action=Index}/{id?}");
    });
}
}
```

## Routing

One of the important sections of **Startup.cs** is the routing setup.

It's mapping a pattern to allow HTTP requests to meet our controllers.

The format is

controller name / action name / optional id

The default controller is listed as Home, while the default action is listed as Index.

If a controller action is not specified, the app will route to /Home/Index.

## wwwroot

This is the place to put any static files that you would like to serve, such as stylesheets, scripts, images or videos.

## Exercise 2 - Housekeeping

---

One constant of Microsoft project wizards is that they almost invariably going to set up things we do not want or need. The .NET Core ASP .NET creation wizard gives us a couple of things we don't need.

First, we don't need the About page for this exercise. Secondly, we probably don't want our homepage to be filled with Microsoft links, so we'll need to change the Home view.

### Removing the About page.

Removing a page in MVC can be done by simply removing the controller action. Remember that it is the controllers in MVC that act as gatekeepers to your Model. If the controller knows nothing about an action called **About**. The page simply won't be reachable.

Use Visual Studio to:-

- Expand the Controllers directory, if it isn't already expanded.
- Open the **HomeController** class.
- Remove the entire About method.

We've removed the About action from the controller. That will stop the app from running an action called About, which we can demonstrate by running now.

The reason we can demonstrate this is because we have not removed the About page cleanly. The page is still referenced in the navigation. Run your project, and click the About link. You should see an error page.

### Removing about from the navigation

The navigation link to the now gone About page lives in the shared \_Layout.cshtml view. Let's edit that now and remove the dead link.

Once done, run your application.

The navigation link to About should be gone. New visitors to your site might never know it existed.

### One last thing

We've removed the controller method. We've removed the link to the navigation. We've still got an About.cshtml file in the Views/Home directory. It'll do no harm, but equally, we really don't need to bother ourselves with its future maintenance.

- Expand Views/Home, if it isn't already
- Delete the About.cshtml file

### Changing the home page.

The home page is too complex for our immediate needs. Let's get rid of the blurb.

- Expand Views/Home, if it isn't already
- Open Index.html
- Delete the markup in the file (leave any directives at the top alone)

Replace with the following markup



```
@{
    ViewData["Title"] = "Home Page";
}
<div class="jumbotron">
    <h1>Sightsource Empower</h1>
    <h2>ASP .NET MVC Training with .NET Core 2.0</h2>
    <h3>Implemented by <strong>Your name here</strong></h3>
</div>
```

Run your app to test. Once done, commit your work into git using Visual Studio.

### What's just happened?

We removed an entire page, learning that we needed to remove references and view files as well for a truly clean deletion.

We also changed the content of the home page to take out Microsoft's advertising hoardings.

## Exercise 3 - Our first feature

---

In the last exercise, we removed the About page but left the Contact page intact. That's because we're going to redevelop it, and make it more useful than the dumb page it already is.

It will be our first feature.

### Git - branching

So far, we've been using git in a very specific way. We've been checking all of our work into the default branch, called master. One of the big benefits of git is that we don't have to work directly with master. We can create other branches.

Think of a git branch as an alternative universe split off at a set point in time. Everything you do in that universe is independent to the master branch. Nothing you do in that universe will change the master branch, unless you ask it to. We'll talk about that later.

For right now, we're going to create one of these alternate universes for our new contact feature.

Git branches all have a name. The convention when developing a new feature is to prefix your branch name with feature/, followed by the name of the feature itself.

Our Contact work will exist in a branch called **feature/contact**.

Let's create that branch now.

- Make sure all previous work is committed into master.
- Go to the Git command line.
- Type:-

```
git branch feature/contact
```

We'll now want to switch into that alternative universe. We can do that by typing:-

```
git checkout feature/contact
```

If you switch back to Visual Studio, you'll see it knows we've changed branches. All work that we've done so far is safe in master. We can proceed with our contact work without messing with the master works.

### The plan

Now that we're in a consequence free environment, we can talk about what we want to do with the Contact feature. At present, it is Microsoft's address.

**!** **Requirement** We want to change that so that a visitor can leave his or her name, email address and a message, which will be sent to a pre-configured email address.

We will :-

- Create a new ViewModel to model this interaction.
- Change HomeController to accommodate the change
- Create a contact form in markup, using the ViewModel
- Define validation rules using DataAnnotations
- Apply validation
- Use the Controller to send the contact email

## The View Model

Right now, the **Contact** action on **HomeController** returns a default **View**, which can expose basic concerns like ViewData, and pass simple messages. We're going to create a more specific ViewModel which models our requirements.

Let's recap.

### Data requirements

- We need to capture someone's **name**.
- We need to capture someone's **email address**.
- We need to capture someone's **message**.

Right click on the **Models** directory and add a class. Call it **ContactViewModel.cs**.

### ContactViewModel

A view model is a class like any other.



**Model or View** - In the MVC part of the equation, you can think of them as a Model component, specifically one intended to be shown on a view. That's where the View part of the ViewModel comes from.

We've already determined that we will need to capture certain information.



**Data types** All the things we need to capture are text-based, whether it is someone's name, email or the message we're about to send?

We will model each required element as a **property** on the view model.

The Razor view engine will need to be able to access the properties in the view model, so let's make them public.

The convention for public properties is to title case them. Title case means the first letter of every new word is capitalised.

We're going to create a class that looks broadly like this:-

```
public class ContactViewModel
{
    public string Name { get; set; }
    public string Email { get; set; }
    public string Message { get; set; }
    public DateTime? CompletedAt { get; set; }
}
```

We're using string to capture each enterable element. There is also a public property called `CompletedAt`, a nullable `DateTime`, meaning it can either contain a `DateTime` value, or be null. We'll use that to determine whether someone successfully made contact.

## Changes to HomeController.

We've created a new class to be used on the View, but right now, the controller has no notion that we want to use that view model on the Contact page. It is presently simply returning the default View, enough to carry a simple message, but not as good as *ContactViewModel* at representing a contact request.

We're going to change the HomeController's About action to return a new *ContactViewModel* everyone someone the Contact action is requested.

The change is subtle, but makes all the difference.

```
public IActionResult Contact()
{
    ViewData["Message"] = "Get in touch with Empower";

    return View(new ContactViewModel());
}
```

The HomeController passes a parameter to the View() method. Doing this explicitly makes a new copy of *ContactViewModel* the view's model.

## Changes to Contact.cshtml

Apart from still containing Microsoft's address, there is another change we need to make to Contact.cshtml.

We need to tell it to expect a *ContactViewModel* instead of the default MVC one.

We can do that by adding the following directive to the top of Contact.cshtml.

```
@model Empower.Mvc.Models.ContactViewModel
```

Let's also clear out Microsoft's address, and start putting together a contact form.

We can start by removing their content, leaving us with the following:-

```
@model Empower.Mvc.Models.ContactViewModel
@{
    ViewData["Title"] = "Contact";
}
<h2>@ViewData["Title"]</h2>
<h3>@ViewData["Message"]</h3>
```

That's just a placeholder. We're not using anything specific on *ContactViewModel* yet. That'll change shortly.

We're going to define a form that'll accept the information we're looking to collect.

Right now, the form will be simple. It won't do anything. It won't submit anywhere. We're just looking to get a handle of how ContactViewModel is used.

First off, let's look at how this might look in a static html page. No MVC.

```
<form>
  <div class="form-group">
    <label for="name">Name</label>
    <input id="name" name="name" placeholder="Please enter your name" class="form-
control"/>
  </div>
  <div class="form-group">
    <label>Email</label>
    <input id="email" name="email" placeholder="Please enter your email"
class="form-control" />
  </div>
  <div class="form-group">
    <label>Message</label>
    <textarea id="message" name="message" placeholder="Please enter your message"
class="form-control"></textarea>
  </div>
  <button type="submit" class="btn btn-primary">
    Contact us
  </button>
</form>
```

Now, let's have a look at how that markup looks as written for ASP .NET Core MVC.

```
<form>
  <div class="form-group">
    <label asp-for="Name">Name</label>
    <input asp-for="Name" placeholder="Please enter your name" class="form-
control"/>
  </div>
  <div class="form-group">
    <label asp-for="Email">Email</label>
    <input asp-for="Email" placeholder="Please enter your email" class="form-
control" />
  </div>
  <div class="form-group">
    <label asp-for="Message">Message</label>
    <textarea asp-for="Message" placeholder="Please enter your message"
class="form-control"></textarea>
  </div>
  <button type="submit" class="btn btn-primary">
    Contact us
  </button>
</form>
```

Anyone notice any differences?

The key change is the introduction of *asp-for*.

This'll get picked up by the MVC view engine, and binds the form element to a property on `ContactViewModel`. When we submit the form, those properties will be filled with whatever someone entered on the form element.

Let's get that form coded up and running.

This is the resulting markup when we run the project and use **View Source**.

```
<form>
  <div class="form-group">
    <label for="Name">Name</label>
    <input placeholder="Please enter your name" class="form-control" type="text"
id="Name" name="Name" value="" />
  </div>
  <div class="form-group">
    <label for="Email">Email</label>
    <input placeholder="Please enter your email" class="form-control" type="text"
id="Email" name="Email" value="" />
  </div>
  <div class="form-group">
    <label for="Message">Message</label>
    <textarea placeholder="Please enter your message" class="form-control"
id="Message" name="Message">
  </textarea>
  </div>
  <button type="submit" class="btn btn-primary">
    Contact us
  </button>
</form>
```

Compared to some of our examples from the bad old days, that is beautiful. No special tags, no view state, just markup that a design and build expert might have produced independently.

## POSTing and validation

The form we've created doesn't do anything yet. If we click submit, all that will happen is a page reload. Worse still, any entered values will be lost. We've also got the added disadvantage of people being able to enter whatever they like into those boxes. The label on "Email" says "Email", and we'd expect an email address to be passed in, but there is presently nothing stopping our users from entering a small essay on spiders instead, difficult to read as it may be.

We'll address the most pressing issue first. The form should DO something, even if all it does is keep our values from getting lost.

The reason our form doesn't do anything is because we haven't told it *what* to do. We can rectify that fairly simply.

### Directing the form

In static HTML, we'd do this by adding the following attributes to the form tag.

```
<form action="/Home/Contact" method="post">
```

The *action* is a URL that the form will send its values to. The *method* is the HTTP verb we'll be using to get our values across.

A brief recap, the vast majority of HTTP requests, the web protocol, are GETs.

In MVC, controller actions are GET by default.

GETs transmit their values over the querystring. POSTs usually send their data through standard input, normally in key value pairs, normally as plain text - although you can change encoding to post files, etc.

In MVC, using tag helpers, the syntax is:-

```
<form asp-controller="Home" asp-action="Contact" method="post">
```

The **asp-controller** tag helper specifies which controller we want. **asp-action** specifies the action we'd like to post the form to.

This will generate the equivalent markup to the static.

### Creating a POST controller action.

Right now, our form still doesn't work. If we try to click it, it'll never find the resource it is looking for. While we've got an action that will respond to a GET request for the **/Home/Contact** route, there's nothing on our controller which will accept a POST, something we will sort immediately.

Open up HomeController, and create another action called **Contact** with the same **ActionResult** return type.

To specify that this action is for POST requests only, we're going to decorate the method with the **HttpPost** attribute. MVC will never serve this up in response to any other type of request, such as a GET or a PUT.

We're also going to ensure it receives a parameter of type **ContactViewModel**, the type we defined to carry this information, and the type that the Contact view carries information about.

When we submit the form, the values on the form will be in the corresponding properties in the **request** parameter.

```
form    // This will only respond to requests which supply the POST HTTP verb.
        // The POST verb is being provided in the 'method' attribute of the calling
        //
        [HttpPost]
        public IActionResult Contact(ContactViewModel request)
        {
            // Change the Message on the page so that we know we've POSTed.
            ViewData["Message"] = "You tried to get in touch with Empower";

            // Don't return an empty ContactViewModel.
            // Return the one we've just got.
            // This will ensure that any submitted values are available in the form
again    return View(request);
        }
```

With that change in, our form does something. Not much, but it will at least save any entered values and give an indication that something has happened.

While we're here, and this is optional, but when you've got two actions with the same name, some people prefer to explicitly decorate the Get version too. I am one of those people.

```
// The default controller verb is GET.
// Therefore, we don't need to decorate this with an [HttpGet]
//
// The upside is that we make our intention to the coders that follow.
// The downside that the coders that follow might think we don't know
// that GET is the default :D
[HttpGet]
public IActionResult Contact()
{
    ViewData["Message"] = "Get in touch with Empower";

    return View(new ContactViewModel());
}
```

## Logging the contact

Next, we're going to beef up the Contact POST action a tiny bit, and then make it *really* obvious that something has been posted.

We defined a property called **CompletedAt** on **ContactViewModel**. We're not doing anything with it right now, but we can use it to record the time of contact when someone presses the submit button. Let's make a simple addition to the action.

```
form
// This will only respond to requests which supply the POST HTTP verb.
// The POST verb is being provided in the 'method' attribute of the calling
//
// [HttpPost]
public IActionResult Contact(ContactViewModel request)
{
    // Change the Message on the page so that we know we've POSTed.
    ViewData["Message"] = "You tried to get in touch with Empower";

    // This was previously null.
    // Now, when someone submits, this will be set and returned back to the
View.
    request.CompletedAt = DateTime.UtcNow;

    // Don't return an empty ContactViewModel.
    // Return the one we've just got.
    // This will ensure that any submitted values are available in the form
again
    return View(request);
}
```



## Conditionally switching the view.

We're now going to display different things, depending on whether **CompletedAt** has a value. The GET version of **Contact** never sets it. It can only be set if the POST action has run.

We're going to use a simple conditional to check the status of **CompletedAt** on the form. If it's empty, we'll show the form. If it's not, we'll say that contact has been made. At this stage of proceedings, that'd be a slightly dishonest claim. We're only pretending that contact has happened, but such a pretense is good enough to test the UI flow of our form.

```
@if (!Model.CompletedAt.HasValue) {
<h2>@ViewData["Title"]</h2>
<h3>@ViewData["Message"]</h3>
    <form asp-controller="Home" asp-action="Contact" method="post">
        <div class="form-group">
            <label asp-for="Name">Name</label>
            <input asp-for="Name" placeholder="Please enter your name" class="form-
control"/>
        </div>
        <div class="form-group">
            <label asp-for="Email">Email</label>
            <input asp-for="Email" placeholder="Please enter your email" class="form-
control" />
        </div>
        <div class="form-group">
            <label asp-for="Message">Message</label>
            <textarea asp-for="Message" placeholder="Please enter your message"
class="form-control"></textarea>
        </div>
        <button type="submit" class="btn btn-primary">
            Contact us
        </button>
    </form>
}
else{
    <div class="alert">
        <h1>Thank you for contacting us, @Model.Name</h1>
        <div class="alert alert-success">
            <p class="text-success">
                We have logged your communication as having been sent at
<strong>@Model.CompletedAt.Value.ToLocalTime()</strong> .
            </p>
        </div>
        <div class="alert alert-warning">
            <p class="text-warning">
                Due to circumstances beyond our control, there is a <em>slight</em>
chance that no-one will read your communication. Miniscule, really. Try not to worry
about it.
            </p>
        </div>
        <div class="alert alert-danger">
            <p class="text-danger">
                I find the comments of the previous paragraph completely disingenous.
It knows full well the email send functionality hasn't been completed yet.
            </p>
        </div>
    </div>
}
```

```
        </p>
    </div>
</div>
}
```

As you can see, the **if** statement serves up different mark up based on the value of something received in **ContactViewModel**

## What just happened?

We :-

- Created a ViewModel, a type of class designed to be shown on a View.
  - Tied it to a View.
- Created a form
  - Bound elements of the ViewModel to form elements, using the MVC tag helpers.
- Created a POST controller action
  - Receives a **ContactViewModel**
  - Sets **ContactViewModel** on that object
  - Returns the altered request to the View
- Changed the Contact.cshtml view to support the two states
  - Form displaying for the first time.
  - Post-form display.

## Commit your work!

Now's a good time to commit your work. Congratulations. You've just created your first MVC form interaction (and I still know professional developers who've not done that). Don't feel too validated, though. That comes next.

## Exercise 4 - Annotation and validation

---

Our contact form talks the talk, but doesn't walk the walk. Perhaps that's just as well. As we've already opined, it'll accept absolutely anything and get to work. It'll also consider absolutely nothing to be valid too. If we click and run the form without providing any input, it won't complain. "Yeah, whatever" is this form's present attitude to user input.

We need to tighten up what the form will accept before we attempt to send any actual contact messages to anyone. Our contact form will be useless unless:-

- We guarantee that someone provides a name.
- We guarantee that someone provides a valid email address.
- We guarantee that someone provides a message.

Without doing that, our form is pretty useless. We'll receive emails without knowing who sent them, what they wanted, or how to get back to them. We need to validate.

### DataAnnotations

.NET has a set of attributes called **DataAnnotations** that can be used to decorate properties on any class we might create. We're going to annotate **ContactViewModel** to include some information about what each field should be.

Let's open up **ContactViewModel** and add some basic validation information using the attributes defined in **DataAnnotations**.

#### The **Required** attribute

By decorating a public property with **Required**, we inform MVC that the model isn't valid if a value is not provided. All three of our properties are required. Let's specify that now.

**Required** can take optional arguments, including **ErrorMessage**. We're going to use that to provide some direction to the user if he or she messes up.

```
public class ContactViewModel
{
    [Required(ErrorMessage="Please enter your name")]
    public string Name { get; set; }

    [Required(ErrorMessage = "Please enter your email")]
    public string Email { get; set; }

    [Required(ErrorMessage = "Please enter your message")]
    public string Message { get; set; }
    public DateTime? CompletedAt { get; set; }
}
```

#### The **EmailAddress** attribute

One of the things we need to collect has more specific requirements than the others. We don't just need it to be present; we need it to be a valid email address. Fortunately, there is a data annotation for that, and multiple

attributes can be applied to a single property, if needed.

Again, we can use the **ErrorMessage** to provide *specific* direction to our end users.

```
[Required(ErrorMessage = "Please enter your email")]
[EmailAddress(ErrorMessage = "Please enter a valid email")]
public string Email { get; set; }
```

If an end user enters something, but it isn't an email address, they'll be asked to enter a *valid* email address.

The **MaxLength** attribute.

We can also declare something invalid if it exceeds a certain length. This is vital in situations where we are persisting the values in fixed length database fields. We're not doing that at the moment, but we should still put some limit on the number of characters we'll allow.

EmailAddress should already cater for the 254 character limit on email addresses, but we can specify maximum lengths for Name and Message.

```
[Required(ErrorMessage="Please enter your name")]
[MaxLength(100, ErrorMessage="How long does it take you to fill in forms? Please
enter 100 characters or less.")]
public string Name { get; set; }

[Required(ErrorMessage = "Please enter your message")]
[MaxLength(4000, ErrorMessage="Too long. Didn't read.")]
public string Message { get; set; }
```

The cool thing about MaxLength is that it will emit the corresponding property in the markup, limiting normal user input to the number of characters specified.

The **DisplayName** attribute

Finally, for now, let's decorate our properties with the DisplayName attribute.

DisplayName is handy because C# property names have some pretty big English language limitations, like not being able to contain space or punctuation. We can use DisplayName to give our form friendlier labels. Whatever we set, MVC will emit in the *label* tag.

```
public class ContactViewModel
{
    [DisplayName("Your name")]
    [Required(ErrorMessage="Please enter your name")]
    [MaxLength(100, ErrorMessage="How long does it take you to fill in forms? Please
enter 100 characters or less.")]
    public string Name { get; set; }

    [DisplayName("Your email address")]
    [Required(ErrorMessage = "Please enter your email")]
    [EmailAddress(ErrorMessage = "Please enter a valid email")]
}
```

```

    public string Email { get; set; }

    [DisplayName("Your feedback")]
    [Required(ErrorMessage = "Please enter your message")]
    [MaxLength(4000, ErrorMessage="Too long. Didn't read.")]
    public string Message { get; set; }
    public DateTime? CompletedAt { get; set; }
}

```

## Checking the Model's validity.

Now that we've established the rules for each of our fields, ASP .NET MVC makes validating these fields simple. A property called `ModelState` is available in controller methods, which will only return valid if all `DataAnnotations` rules are met.

Let's open up `HomeController` again, and include a check for **`ModelState.IsValid`**.

We're going to change the behavior so that we only set the **`CompletedAt`** property *if* the `ViewModel` we receive is valid.

```

// This will only respond to requests which supply the POST HTTP verb.
// The POST verb is being provided in the 'method' attribute of the calling
form
//
[HttpPost]
public IActionResult Contact(ContactViewModel request)
{
    // Change the Message on the page so that we know we've POSTed.
    ViewData["Message"] = "You tried to get in touch with Empower";

    if (ModelState.IsValid)
    {
        // This was previously null.
        // Now, when someone submits, this will be set and returned back to
the View.
        request.CompletedAt = DateTime.UtcNow;
    }

    // Don't return an empty ContactViewModel.
    // Return the one we've just got.
    // This will ensure that any submitted values are available in the form
again
    return View(request);
}

```

Let's open up the **`Contact.cshtml`** view again and recap. If **`CompletedAt`** has no value, we show the form. If it has a value, we show the rather mendacious response page. We've just introduced a clause that says we'll only set that value, if the form is valid.

Before we try it out, what do you think will happen if we try to POST this form with invalid data?

**SPOILER**, we'll show the form, so that the user can retry.

## Reporting problems through the View.

We'll want to let our user's know what has gone wrong, if anything has gone wrong.

MVC provides support for validation errors. We're going to add a danger label to each form element, which will display if any of the property's rules have been broken.

```
<form asp-controller="Home" asp-action="Contact" method="post">
  <div class="form-group">
    <label asp-for="Name">Name</label>
    <input asp-for="Name" placeholder="Please enter your name" class="form-
control"/>
    <span asp-validation-for="Name" class="label label-danger"></span>
  </div>
  <div class="form-group">
    <label asp-for="Email">Email</label>
    <input asp-for="Email" placeholder="Please enter your email" class="form-
control" />
    <span asp-validation-for="Email" class="label label-danger"></span>
  </div>
  <div class="form-group">
    <label asp-for="Message">Message</label>
    <textarea asp-for="Message" placeholder="Please enter your message"
class="form-control"></textarea>
    <span asp-validation-for="Message" class="label label-danger"></span>
  </div>
  <button type="submit" class="btn btn-primary">
    Contact us
  </button>
</form>
```

This is all thanks to the *asp-validation-for* tag helper. By including it, along with the name of the property we care about, we're essentially saying "follow the validation rules for this property, and fill this element with an error message if you have one". If the property has valid data, it won't be shown.

Let's run our form now and try to submit it with invalid or empty data.

## What just happened?

- We decided we needed to validate **ContactViewModel** to make it useful, or at least keep it from being potentially useless.
- We used **DataAnnotations** to specify some rules for each property.
  - We also created custom error messages.
  - Some of them may have not been entirely professional.
- We changed a controller action to use **ModelState.IsValid** to determine whether the rules we set have been met.
  - We only set **CompletedAt** if the form is valid
  - This ensured that the form was redisplayed when invalid data was entered
- We added elements on the **Contact.cshtml** view to hold error messages, tied to the rules we set with the tag helper **asp-validation-for**.

## Commit your work!

We've just made sure that we're going to get some halfway decent data in from this form, and that our users will have a fair idea of what is wrong, if anything is wrong, so invalid data will be corrected. Brace yourselves. In the next exercise, we make contact.

## Exercise 5 - Contact

---

In this exercise, we're going to send the email. We're not going to do it in the optimum way, but we can refine on that later, and discuss why it isn't the optimum way. In the coming exercises, we're going to absolve most of our programming sins.

### System.Net.Mail

Microsoft .NET is now a very mature platform, but the ability to send an email has been something that it has had from the start. It still exists today.

The email functionality resides in **System.Net.Mail**.

We're going to use that ready-rolled functionality to send our contact form to an email address of our choice, namely ours. You should all have **Sightsource** email addresses at this point, so we'll make sure that the results of your contact form will come to you.

### Modifying the HomeController to send an email

We're going to use the **System.Net.Mail** to send an email, and we're going to do some bad things in the process. As usual, this isn't ideal practice, but it is sometimes good to see the bad things so that you don't do them yourself.

#### MailMessage

To send a mail through **System.Net.Mail**, we need to create a **MailMessage**. This contains basic details of sender, recipient, content and content type, as well as any attachments we may choose to add. For now, there are no files on us. We can keep things simple.

We will need to set the following details

- Sender Email ("s2empower@gmail.com")
  - **To** property on a mail message object. Takes a collection of **MailAddress**
- Sender Name ("Sightsource Bot")
  - **To** property on a mail message object. Takes a collection of **MailAddress**
- Recipient Email ("")
  - **From** property on a mail message object. Takes a **MailAddress**
- Recipient Name ("")
  - **From** property on a mail message object. Takes a **MailAddress**
- Body content
  - The name and email address of the person that sent the message
  - The message itself.
- The Subject

Let's construct the **MailMessage**

```
// This will only respond to requests which supply the POST HTTP verb.  
// The POST verb is being provided in the 'method' attribute of the calling
```

```

form
    //
    [HttpPost]
    public IActionResult Contact(ContactViewModel request)
    {
        ViewData["Message"] = "You tried to get in touch with Empower";

        // This checks validation rules to determine whether the information
        // supplied in request will meet the needs of the request.
        //
        // UPDATE: This will now validate according to the rules of the validators
        // on ContactViewModel. It'll return false if any of the rules we've set
are
        // broken.
        if (ModelState.IsValid)
        {
            // Send the mail
            var message = new MailMessage();
            var configuration = GetConfiguration();

            message.To.Add(new MailAddress("pap@elevenhitcombo.com", "Hackmeister
McHack"));
            message.From = new MailAddress("s2empower@gmail.com", "Sightsource
Bot");
            message.Body =
                $"{request.Name} ({request.Email})" +
                Environment.NewLine +
                request.Message;
            message.Subject = "New Contact Request";

            request.CompletedAt = DateTime.Now;
        }

        return View(request);
    }
}

```

The letter had been assembled. We still need to post it.

## SmtplibClient

SmtplibClient is another .NET stalwart, and has the specific job of sending **MailMessage** objects through a server. Email has become swamped by spammers to the point where many folk question its utility for actual communication. The key platform for sending email is SMTP, and as the scourge of spamming and the risk that goes with it (your entire email domain can be blacklisted if ISPs detect spam sent from your servers), most SMTP servers require credentials.

We will need to provide them, along with some basic details of how we'd like to do our business, such as using encryption for communication.

**SmtplibClient** makes all of this trivial, and is an example of why a quick search to see if something already exists is better than rolling your own.



```

// This will only respond to requests which supply the POST HTTP verb.
// The POST verb is being provided in the 'method' attribute of the calling
form
//
[HttpPost]
public IActionResult Contact(ContactViewModel request)
{
    ViewData["Message"] = "You tried to get in touch with Empower";

    // This checks validation rules to determine whether the information
    // supplied in request will meet the needs of the request.
    //
    // UPDATE: This will now validate according to the rules of the validators
    // on ContactViewModel. It'll return false if any of the rules we've set
are
    // broken.
    if (ModelState.IsValid)
    {
        // Send the mail
        var message = new MailMessage();
        var configuration = GetConfiguration();

        message.To.Add(new MailAddress("pap@elevenhitcombo.com", "Hackmeister
McHack"));
        message.From = new MailAddress("s2empower@gmail.com", "Sightsource
Bot");
        message.Body =
            $"{request.Name} ({request.Email})" +
            Environment.NewLine +
            request.Message;
        message.Subject = "New Contact Request";

        client = new SmtpClient("email-smtp.us-east-1.amazonaws.com");
        client.Port = 587;
        client.EnableSsl = true;
        client.Credentials = new System.Net.NetworkCredential(
            "<username>",
            "<password>s"
        );
        client.Send(message);

        request.CompletedAt = DateTime.Now;
    }

    return View(request);
}

```

## Small discussion of your instructor's egregious sins.

From your experience so far, what would you say I have done wrong here?

## What just happened?

- We got our Contact form functional.
  - It will now send an email to you when someone wants to get in contact.
- We could have done better.
  - Our discussion illustrated several problems with the approach.

## Commit your work

We're going to solve some of those problems in the next exercise.

## Exercise 6 - Configuration

---

Following our discussion, I've been put in my place. We know that we might have to deploy to multiple environments, and that hardcoding things like passwords, and forcing deployments for simple things like password changes is a really bad idea.

We'd like to be able to change simple details like that without changing the code that runs our site. It is time to talk about configuration.

I'm certain you've been through this before, but configuration in ASP .NET MVC 2.0 dispenses with everything else in .NET. The web.config and app.config **XML** files have been retired. **JSON** is the format du jour. appsettings.json is where application settings live now. They're very easy to manage, compared to the bloated files of the past. They are also very configurable, and very easy to get to.

### appsettings.json example

You should already have an appsettings.json file, created when your project was. Lets get our SMTP settings into a configuration file

```
{
  "Logging": {
    "IncludeScopes": false,
    "LogLevel": {
      "Default": "Warning"
    }
  },
  "Contact": {
    "SmtpHost": "localhost",
    "SmtpUsername": "<username>",
    "SmtpPassword": "<password>",
    "SmtpUseSsl": "true",
    "SmtpPort": "587",
    "DoNotReply": "s2empower@gmail.com",
    "DoNotReplyName": "Sightsource Contact Bot",
    "Destination": "pap@elevenhitcombo.com",
    "DestinationName": "Paul Taylor"
  }
}
```

### Return to Startup.cs

Those with freakishly decent memories may remember the constructor in the **Startup** class, the class that sets up our MVC application. For those that do not, it is reproduced here:-

```
public Startup(IConfiguration configuration)
{
    Configuration = configuration;
}
```

This effectively loads the configuration from **appsettings.json**.

## Accessing the configuration with ConfigurationBuilder programmatically.

You can load configurations independently should you wish.

```
private IConfiguration GetConfiguration()
{
    var builder = new ConfigurationBuilder()
        .SetBasePath(Directory.GetCurrentDirectory())
        .AddJsonFile("appsettings.json");

    return builder.Build();
}
```

**appsettings.json** is a convention; as the example above shows, we can specify different configurations and add them at will.

Let's put the above method in **HomeController**. It'll help us out with our hardcoding problem.

Once that method is in, we can get a configuration anytime we need it inside **HomeController**.

We can access the configuration with :-

```
var configuration = GetConfiguration();
```

### Accessing configuration values

The **IConfiguration** object allows for easy retrieval of configuration items. **JSON** is essentially a tree of data, objects within objects. The `:` character is used to separate levels of the tree.

Using the example above, if we wanted to get the value of Contact -> SmtpHost, can use the configuration object as follows:-

```
var smtpHost = configuration["Contact:SmtpHost"]
```

The **IConfiguration** implementation will dive into the JSON file and pull it out.

### Casting configuration values

**IConfiguration** values are returned as objects. That is because .NET doesn't want to make any assumptions about the kind of data that is held there. It could be a string. It could be a number. It could be a tree of objects in itself.

### Text

Turning configuration values into strings is trivial with the *string interpolation* syntax.

```
var smtpHost = $"{configuration["Contact:SmtpHost"]}";
```

## Other types

Because **IConfiguration** returns **object**, which can be anything, cast other types as you need to; just be aware that your app will break if something cannot be cast to the type you say it is.

```
var useSsl = (bool)_settingsService.GetSetting("Contact:SmtpUseSsl");
```

A config aware controller.

```
// This will only respond to requests which supply the POST HTTP verb.
// The POST verb is being provided in the 'method' attribute of the calling
form
//
[HttpPost]
public IActionResult Contact(ContactViewModel request)
{
    ViewData["Message"] = "You tried to get in touch with Empower";

    // This checks validation rules to determine whether the information
    // supplied in request will meet the needs of the request.
    //
    // UPDATE: This will now validate according to the rules of the validators
    // on ContactViewModel. It'll return false if any of the rules we've set
are
    // broken.
    if (ModelState.IsValid)
    {
        // Send the mail
        var message = new MailMessage();
        var configuration = GetConfiguration();

        message.To.Add(new MailAddress($"
{configuration["Contact:Destination"]}", $"
{configuration["Contact:DestinationName"]}"));
        message.From = new MailAddress($"
{configuration["Contact:DoNotReply"]}", $"{configuration["Contact:SenderName"]}");
        message.Body = request.Message;
        message.Subject = "New Contact Request";

        var client = new SmtpClient($"{configuration["Contact:SmtpHost"]}");
        client.Port = (int)configuration["Contact:SmtpPort"];
        client.EnableSsl = (int)configuration["Contact:SmtpUseSsl"];
        client.Credentials = new System.Net.NetworkCredential(
            $"{configuration["Contact:SmtpUsername"]}",
            $"{configuration["Contact:SmtpPassword"]}");
        client.Send(message);

        request.CompletedAt = DateTime.Now;
    }
}
```

```
        return View(request);  
    }
```

## What just happened?

- We added configuration elements to our existing appsettings.json file
- Doing this allowed us to take hard-coded and likely to break elements of the app outside the realm of code.
  - We can now change settings just by changing a JSON file
  - We don't have to redeploy an app, we can just change one file, appsettings.json, and most of the upstream changes can be catered for.

## Commit your work!

Let's get this bad boy into **git**. For now, we are feature complete.

## Merge your work.

Our first feature is complete. As long as we're all happy with it, let's merge that work into master.

Once you've committed your work, you can drop into the git bash shell.

Type

```
git checkout master  
git merge feature/contact
```

What does this do?

This puts all the work you've done for contact back into the master branch.

### Conflicts?

VS is great place to resolve merge issues. Speak to me if you have any.

## Commit your work again!

Once any conflicts have been resolved, let's get our work into the master branch.

You'll need to type:-

```
git commit -m "Merging contact into master"
```

## What's just happened?

- We've just merged our first feature into the master branch
  - Until we merged, it was completely independent of the master branch
  - After we merged, our new feature became a part of the master branch
  - Any new branches we create from master will contain these changes.

## What did we do today?

- We:-
  - learned the history of MVC
  - got the technical definition.
  - got a workable definition.
  - took a tour around Microsoft's ASP .NET Core 2.0 MVC.
  - saw the horrors of what went on before the MVC pattern was introduced to the web.
  - created **ContactViewModel**, changing the Controller method to emit it.
  - adapted the view, creating a form, bound to the properties of **ContactViewModel**
  - created a POST action to accept input from the form on out view, in **ContactViewModel** format.
  - created a switch, using Razor, to conditionally display the form or an acknowledgement that something had been posted.
  - used the **DataAnnotations** functionality to specify rules for the things in **ContactViewModel**
  - added a check on **ModelState.IsValid**, hooked into **DataAnnotations**, to determine the validity of the submitted input.
  - modified our switch to only show acknowledgement when the data was valid.
  - *finally* actually sent an email, if imperfectly.
  - introduced configuration, allowing us to change settings without hardcoding or re-deploying the application.
  - merged our completed branch back into master.

## Discussion

What did we do right? What did we do wrong? What do we still have to do.

## End of day one wrap up

---

## Day Two - Dependency Injection and Service Based Architecture.

---

### Recap of Day One.

We were able to use the Model View Controller pattern to send an email with the controller. The problem is, we played fast and loose with the definitions we laid out earlier. Our controller is actually *doing* stuff. It really shouldn't be. If we remember our amended definition of Model, View and Controller, it's really the Model that should be doing stuff. The controller should just control the stuff the Model gets to do.

In other words, the controller should just be giving orders once it knows those orders are valid. It shouldn't be carrying out those orders itself.



## Services

We're going to move the behaviour we put in our controller's Contact POST action into a service. A service in computing is much like a service in real life. It's a thing that does a thing, normally something that you don't want to do yourself. Pizza delivery is great example of a real life service we use every day, and something we can put in an MVC context. We are the controllers. We decide whether we want pizza or not. We call a pizza service using an interface; a phone number, a website, or an app.

We are not particularly concerned about how the service is implemented, as long as it doesn't take too long, it actually works, we don't get food poisoning the next morning or subsequently discover that the place our Tuesday night food comes from is infested with rats.

The boys and girls at the pizza store are relying on services too. They did not build their restaurant from scratch. They were not up in the mountains, mining stone, or in the quarries, getting sand, making windows. They do not have cows in the back yard, there's no dairy or cheese rendering facilities on the premises. You might not even have a dairy in your town. For all of these things, they rely on services too.

They need people to mill, people to farm, people to move product so that it can be refined, raw animal and plant products, the process of photosynthesis for all of that to happen, and almost no human involved in that chain does all of these things themselves. Each of them depends on other services to provide their services.

It's an example of a *dependency chain*, happening in real life. We need food, so we call a pizza service. They need cheese, flour and ingredients. They use services for that, such as hauliers, farmers, miners, plants, animals and for photosynthesis, celestial bodies that also happen to be giant nuclear fusion reactors.

A great definition of a service is that "I need you to do something for me. Don't much care how you do it". Each and every part of the pizza service dependency chain can be described that way.

We call a telephone line, or use an online service to order pizza. We want pizza. As long as the quality is good, and the delivery is timely, we don't much care how it's done. The guys at the pizza store are going to need mozzarella, tomatoes and various processed meats to meet their needs, and like us, they have an interface - again, usually telephone or online interfaces that allow them to order their products.

Some of the things they need will have *multiple dependencies*. It's all very well ordering mozzarella, but it's no good if you can't get it to the pizza service. Mozzarella needs multiple services, production and transport. As long as mozzarella is delivered, we don't much care how they're delivered.

## A recap on Interfaces

A recap on interfaces is needed because it's probably one of the most overused terms in computing. More than anything else, an interface declares a contractual obligation. Returning to our pizza store example, we might use the following interface to define its operations.

```
public interface IPizzaService
{
    FoodOrderResponse FulfilWalkupOrder(FoodOrderRequest request);
    DeliveredFoodOrderResponse FulfilTelephoneOrder(DeliveredFoodOrderRequest request);
    DeliveredFoodOrderResponse FulfilOnlineOrder(DeliveredFoodOrderRequest request);
}
```

Even though we don't know precisely what *FoodOrderResponse*, *DeliveredFoodOrderResponse*, or the request objects contain, if we assume they contain all the things you need to fulfil a walk up or delivered order, that is a pretty good description of *our* interaction with the pizza service is.

Our needs are simple. Can you give me a box full of pizza if I walk up for it? Can you deliver boxes full of pizza if I order them, either by phone or online?

The C# interface above declares that we'd like something done. Anyone that claims to be an *IPizzaService* should be able to do these things. The interface has absolutely no concept of how these things will be done. It's perhaps the perfect expression of I don't care how it's done, just do it.

Everything claiming it can run an *IPizzaService* will be expected to do it.

## Exercise 1 - Creating a Services project

---

Create a new git branch based off master. Let's call it feature/services. Checkout feature/services.

A services, or contracts project, is the home for your contracts, and most of your interfaces. We are going to create a new project that will come to define the entire scope of our model, to anyone that can read a bit of English and cares to look. With a services based approach, and sensible naming conventions, we can make life much easier for future coders. They will simply be able to look at the services we've defined, and be able to deduce the exact scope of our application.

In Visual Studio, create a new C# class library called **Empower.Services**. Ensure it is sited in the **src** directory.

### Adding our first contract, or service

We want to get our controller out of the "running code" business and back into the "controlling code" business. It really shouldn't be concerned with "how to send an email". We are going to delegate that responsibility to an email service.

Let's add a new item in Visual Studio project to our newly created Services project. Scroll down and find the interface type. Call it **IEmailService**.

Our new interface should look like this to begin with:-

```
namespace Empower.Services.Interfaces
{
    /// <summary>
    /// This interface defines and describes all operations we'd want to
    /// perform for email operations
    /// </summary>
    public interface IEmailService
    {
        DateTime? SendEmail(string toName, string toEmail, string bodyText, string
subject, string fromName, string fromEmail);
    }
}
```

Anything that says it can run an *IEmailService* must provide that method. Given the following parameters:-

- toName
- toEmail
- bodyText
- fromName
- fromEmail
- subject

The *IEmailService* promises to do something called SendEmail, returning a Nullable date time.

### Creating a concrete implementation

We've declared something called an *IService*. This, as we've said, is like a contract. Any class setting itself up to perform, or implement that service, has to fulfil that contract. We're going to create a class that does that. Such classes are known as concrete implementations; they are where the work is done.

## New project - Empower.Network.Services

We're going to start creating new projects quite frequently with our service based approach. While we could create an implementation of *IService* that lives in the same project as our MVC project, we might want to use our email service in other things we can do. By creating a separate project to house this functionality, we get to re-use *IService* with other work we may wish to do in the future.

We're going to create a project designed to handle all our Network-type services. Email is a network protocol for sending messages electronically, so this new project will be the perfect home.

Let's create a new **c# class library** project called **Empower.Network.Services**.

### Project references

Our new project will need to know about **Empower.Services**. Our MVC project will need to know about **Empower.Services** and **Empower.Network.Services**. Let's create those references now.

### New class - EmailService

Now our projects are linked up, we can create an **EmailService** in **Empower.Network.Services** which references *IService* in *Empower.Services*.

Let's add a new class called *EmailService* to **Empower.Network.Services**.

### Implementing *IService*

Our newly created class doesn't have any ties to *IService*. They're named similarly, but as far as the compiler is concerned, that's just coincidence.

As programmers, we need to explicitly *implement* the interface. The implementation syntax is identical to inheritance syntax, although if you are implementing an interface *and* inheriting at the same time, the base class must come before any interface declarations.

Here, we're just implementing *IService*, and initially, not very well.

Our class meets the demands of the *IService*. It will compile. It won't actually run.

```
namespace Empower.Network.Services
{
    // This is EmailService explicitly declaring "I will do the work of
    *IService*"
    public class EmailService : IService
    {
        public DateTime? SendEmail(string toName, string toEmail, string bodyText,
string fromName, string fromEmail)
        {
            // Which is something of a fib at this point.
            throw new NotImplementedException();
        }
    }
}
```

```
}
}
```

### Moving the controller implementation into EmailService.

We've already got something that sends an email defined in our **HomeController**. We're going to move that code into the EmailService, putting it into the **SendEmail**.

For now, we're also going to take a very naughty coding decision which you shouldn't do in real life, and we'll be fixing afterward.

We've copied and pasted a block of code to allow access to our configuration. That'll give us a working method right now, but it's something that we won't leave in.

If we did that, there's a good chance that we could end up taking divergent paths on getting configurations, something we want to remain the same.

```
namespace Empower.Network.Services
{
    // This is EmailService explicitly declaring "I will do the work of
    *IEmailService*"
    public class EmailService : IEmailService
    {
        public DateTime? SendEmail(string toName, string toEmail, string bodyText,
            string fromName, string fromEmail)
        {
            var dateSent = (DateTime?)null;

            // Send the mail
            var message = new MailMessage();
            var configuration = GetConfiguration();

            message.To.Add(new MailAddress(toEmail, toName));
            message.From = new MailAddress(fromEmail, fromName);
            message.Body = bodyText;
            message.Subject = subject;

            try
            {
                var client = new SmtpClient($"{configuration["Contact:SmtpHost"]}");
                client.Port = (int)configuration["Contact:SmtpPort"];
                client.EnableSsl = (int)configuration["Contact:SmtpUseSsl"];
                client.Credentials = new System.Net.NetworkCredential(
                    $"{configuration["Contact:SmtpUsername"]}",
                    $"{configuration["Contact:SmtpPassword"]}"
                );
                client.Send(message);
                dateSent = DateTime.UtcNow;
            }
            catch (Exception ex)
            {
                // TODO: Add logging at some point
            }
        }
    }
}
```

```

        return dateSent;
    }
}

// This is a very naughty temporary inclusion.
// We should never violate the DRY principle in production code.
// We will refactor this with a saner construction.
private IConfiguration GetConfiguration()
{
    var builder = new ConfigurationBuilder()
        .SetBasePath(Directory.GetCurrentDirectory())
        .AddJsonFile("appsettings.json");

    return builder.Build();
}
}

```

### Refactoring HomeController to use EmailService.

Now that we've moved the SendEmail functionality into EmailService, the Contact action in HomeController can be modified to take advantage of this functionality.

Again, we're going to go about things the wrong way before settling on correct practice. For right now, we're going to ignore the fact that we created an interface called **IEmailService** and just create a new instance of EmailService directly.

Our new action should look like this:-

```

// This will only respond to requests which supply the POST HTTP verb.
// The POST verb is being provided in the 'method' attribute of the calling
form
//
[HttpPost]
public IActionResult Contact(ContactViewModel request)
{
    ViewData["Message"] = "You tried to get in touch with Empower";

    // This checks validation rules to determine whether the information
    // supplied in request will meet the needs of the request.
    //
    // UPDATE: This will now validate according to the rules of the validators
    // on ContactViewModel. It'll return false if any of the rules we've set
are
    // broken.
    if (ModelState.IsValid)
    {
        // This is bad practice. While we've gotten away with creating a
concrete
        // instance of EmailService on the fly, the controller will be much
better
        // served through an injected instance of IEmailService
        var emailService = new EmailService();
    }
}

```

```

        var configuration = GetConfiguration();

        emailService.SendEmail(
            $"{configuration["Contact:DestinationName"]}",
            $"{configuration["Contact:Destination"]}",
            "Sightsource Empower Feedback",
            $"Sent by: {request.Name} <{request.Email}>{Environment.NewLine}
{request.Message}" ,
            $"{configuration["Contact:DoNotReplyName"]}",
            $"{configuration["Contact:DoNotReply"]}"
        );
    }

    return View(request);
}

```

We've moved on a little. Our controller method is more or less just controlling now. If what it receives is valid, it will use a method defined on a service to send the email.

Our implementation still has problems. Even though the controller method is no longer concerning itself with the business of sending emails, delegating that to an EmailService instance, it's still plucking values from the configuration to make its calls. Worse, we've violated the Do Not Repeat Yourself principle by having identical definitions of GetConfiguration in HomeController, which we cannot presently avoid without doing something else.

The design of our method is part of the problem. It's too general. It is designed to send any email, to any address, from any address, meaning that our controller method needs to supply all of those values itself. In reality, our contact form is at present, only concerned with sending email to one address, from one address, including the user's detail in the request.

Crucially, we're calling the service from a concrete instance we created ourselves. We've gotten away with it now because our EmailService implementation has no other dependencies. If it needed something else, we would need to provide it ourselves. If its dependency needed something else, we'd need to provide that too. There is a whole framework that will do that for us if we set it up correctly.

## Wiring in dependency injection

### A recap

The dependency injection pattern allows a programmer to make use of injected dependencies. Dependencies are established using a one-time setup routine, which establishes how each service is going to be delivered. In .NET Core 2.0 apps, these dependencies are, by convention, established in Startup.cs.

Let's open that up, and add the following line to **ConfigureServices**.

```

// This method gets called by the runtime. Use this method to add services to the
// container.
public void ConfigureServices(IServiceCollection services)
{
    services.AddTransient<IEmailService, EmailService>();
    services.AddMvc();
}

```

.NET Core 2.0 knows all about dependency injection. We can use the vanilla implementation for now.

The new line is:-

```
services.AddTransient<IEmailService, EmailService>();
```

This line tells MVC that :-

- We're declaring a service
- It's called *IEmailService*
- When someone asks for an *IEmailService*, we're going to give them an *EmailService* from **Empower.Network.Services**.
- The **AddTransient** command means that MVC will spin up a new temporary *EmailService* instance every time an *IEmailService* service is required. There are other strategies for service creation which will discuss in due course.

### Constructor injection

Now that we've declared **IEmailService** as a service which can be resolved by creating a new **EmailService**, we're in a position to use **constructor injection** to **inject** the service into **HomeController**. We can remove the direct reference to *EmailService*.

First though, we're going to create a new variable that will house the service we're looking to inject.

```
private readonly IEmailService _emailService;
```

A *readonly* variable can only be set in a constructor. Accordingly, it is commonly used in **constructor injection**.

Constructor injection isn't particularly fancy. It's simply a pattern whereby you give a class the things it needs at start-up. It is the job of the dependency injection engine to make sure those needs are satisfied.

We're going to change our constructor method so that it now takes in an **IEmailService** object, and uses it to set our new *readonly* variable.

```
public HomeController(IEmailService emailService)
{
    _emailService = emailService;
}
```

From now, if we want to access *IEmailService*'s functionality in the class, all we have to do is reference *\_emailService*. Let's do that in the controller method now.

```
[HttpPost]
public IActionResult Contact(ContactViewModel request)
{
    ViewData["Message"] = "You tried to get in touch with Empower";
}
```



```

        // We're now longer newing up an EmailService.
        // We're just using _emailService.
        if (ModelState.IsValid)
        {
            var configuration = GetConfiguration();

            request.CompletedAt = _emailService.SendEmail(
                $"{configuration["Contact:DestinationName"]}",
                $"{configuration["Contact:Destination"]}",
                "Sightsource Empower Feedback",
                $"Sent by: {request.Name} <{request.Email}>{Environment.NewLine}
{request.Message}" ,
                $"{configuration["Contact:DoNotReplyName"]}",
                $"{configuration["Contact:DoNotReply"]}",
                $"{configuration["Contact:SmtpHost"]}"
            );
        }

        return View(request);
    }

```

We're in much better shape inside our controller. It is now in the business of controlling, using a service which it doesn't know any details about, save the fact that it says it can *SendEmail*. It's much like our experience with the pizza guys. We know they can take a phone order, or an online order, and that either will result in pizza, but we're not concerned with how it is done. HomeController is in that state now.

## What just happened?

- We created our first contract, or service, embodied as **IService**
  - Any class taking up this contract, or implementing it, needs to provide everything IService says it does.
- We created our first concrete service class, which explicitly claimed it could do the work of **IService**
  - We moved our MailMessage and SmtpClient logic there.
  - We naughtily copied GetConfiguration(), leaving ourselves with some technical debt.
- We configured a dependency rule in Startup.cs
  - Whenever a class requires an IService, the system is configured to provide an instance of **Service**.
- We changed HomeController to have a dependency on **IService**
  - We specified this dependency by using **constructor injection**, a means of giving a class the services it needs through dependency injection.
- HomeController now knows very little about the specifics of sending an email.
  - It relies on a service for this work.
  - It does not know how this service works, nor does it care.

## Commit your work

This feature isn't perfect, but we are at a stage where we can commit our work, and start looking to our next service.

We're staying on the feature/services branch for now.

## Exercise 2 - Settings

---

One of the biggest problems introduced during the last exercise was the fact we ended up copying and pasting a means to get a configuration. While it *works*, it's far from ideal practice. We shouldn't repeat ourselves, especially when we have other options available.

We can define a settings service, with the unique job of granting us access to every setting we care about in our application, or at least in the parts that exist so far.

### SettingsService

For right now, let's keep the behaviour we want from our service relatively simple. An *ISettingsService* should be able to get me a setting value, in string or object format, if I provide it with a setting key.

Let's create that contract in **Empower.Services**

```
namespace Empower.Services.Interfaces
{
    public interface ISettingsService
    {
        object GetSetting(string settingKey);
        string GetStringSetting(string settingKey);
    }
}
```

### Another new .NET Class Library project

We've just declared what the service *should* do. We don't have anything that will fulfil that service as of yet, and it isn't a perfect fit for many of our existing class libraries.

Let's create a new class library called **Empower.Settings.Services** to house our concrete implementation of this behaviour.

Ensure it is sited in the **src** directory. This project should reference **Empower.Services**. **Empower.Mvc** should reference this project.

### SettingsService

Let's create a new class called **SettingsService**. Like our **EmailService**, it will *implement* the **ISettingsService** contract, promising to do the stuff it says it will do.

The class is pretty simple. Let's put it together now.

The first thing we need to do is state that **SettingsService** will do **ISettingsService** work.

```
public class SettingsService : ISettingsService
```

If we try to compile our work now, VS will inform us of a couple of errors, because we haven't yet provided implementations for the mandatory **ISettingsService** methods.

We should get to work on them now, but we need something else first.

## Injecting a configuration

Previously, we were using a `ConfigurationBuilder` to read the `appsettings.json` file. That is the file that is read by default, by the MVC engine, during `Startup.cs`.

Instead of building it to order everytime, we can **inject** an `IConfiguration` instance into our settings service, so that we have the configuration file available, in programmatic form, inside our `SettingsService`.

Let's create a readonly variable to store the incoming configuration.

```
private readonly IConfiguration _configuration;
```

And then create a constructor that will accept an `IConfiguration` object as a dependency.

```
public SettingsService(IConfiguration configuration)
{
    _configuration = configuration;
}
```

We can now access a populated configuration anywhere in the `SettingsService` class through the `_configuration` variable.

Creating the methods that are required by **`ISettingsService`** is now trivial. We're just passing through to methods available on `_configuration`.

```
// These methods are simply pass throughs to methods on
// IConfiguration, available through _configuration
public object GetSetting(string settingKey)
{
    return _configuration[settingKey];
}

// This method is using string interpolation to coerce the
// result into a string.
public string GetStringSetting(string settingKey)
{
    var rawSetting = GetSetting(settingKey);
    return $"{rawSetting}";
}
```

Our completed `SettingsService` should look as follows. It's presently a little dumb, but it meets its contractual requirements.

```
public class SettingsService : ISettingsService
{
    private readonly IConfiguration _configuration;
```

```
public SettingsService(IConfiguration configuration)
{
    _configuration = configuration;
}

public object GetSetting(string settingKey)
{
    return _configuration[settingKey];
}

public string GetStringSetting(string settingKey)
{
    var rawSetting = GetSetting(settingKey);
    return $"{rawSetting}";
}
}
```

## Adding ISettingsService to the list of injectable services

Let's head back into **Startup.cs** and make a bold declaration. Any time that any piece of code needs an **ISettingsService**, we will instruct MVC to give it the **SettingsService** defined in **Empower.Settings.Services**.

Last time, we changed the **ConfigureServices** method, creating an association between **IService** and **EmailService** using a method called **AddTransient**. That choice has the effect of a new **EmailService** being created every time an **IService** was asked for.

When declaring these associations, we can also control the lifetime of the objects that are created to service them. We don't have to create a brand new **SettingsService** every time someone asks for one. These kind of settings do not change from user to user. It might be far more efficient to just have one copy of the **SettingsService** object in memory that can be served up any time something needs a setting.

We can use **AddSingleton** in our configuration of this service instead of **AddTransient**. This will ensure that the dependency injection engine creates just one instance of **SettingsService**, passing it around to anything that requires it.

Let's define that relationship now.

```
// This method gets called by the runtime. Use this method to add services to
the container.
public void ConfigureServices(IServiceCollection services)
{
    // Only one copy of the settings service will exist in memory at any time.
    services.AddSingleton<ISettingsService, SettingsService>();

    // On demand. Multiple instances will be created as needed.
    services.AddTransient<IService, EmailService>();

    services.AddMvc();
}
```

When we run our application, anything requiring a setting can make use of **ISettingsService**. Presently, nothing is expressing a requirement, something that we'll change in the next part of the exercise.

## Injecting ISettingsService into other classes.

We have two classes that could really use access to Settings. The **HomeController** is one, while **EmailService** is the other. Both are presently reliant on the copied and pasted `GetConfiguration()` private method, our technical debt from the last exercise. We're going to settle that debt now.

### EmailService

Right now, **EmailService** is using the copied `GetConfiguration()` private method to build up a configuration each time one is needed. The first thing we're going to do is change where it gets its information from. We're going to use **constructor injection** to specify that from now on, **EmailService** is going to need access to **ISettingsService** to work.

As usual, we'll create a readonly variable to hold a reference to **ISettingsService**.

```
private readonly ISettingsService _settingsService;
```

And we'll create a constructor which takes a **ISettingsService** parameter, setting our local reference to the injected service. The **AddSingleton** declaration we made earlier on will ensure that **EmailService** gets an **ISettingsService** to work with.

```
public EmailService(ISettingsService settingsService)
{
    _settingsService = settingsService;
}
```

### Changing SendEmail to use the injected \_settingsService

Now that we have access to settings, we no longer need to rely on the rolled-every-time access to the configuration file accessed through our redundant method `GetConfiguration`. We can ask for these values from our copy of **ISettingsService**.

```
// This has now been marked private.
// It is no longer directly callable from an IEmailService, but then it
doesn't need to be.
// The stuff that is callable can still use it inside this class.
private DateTime? SendEmail(string toName, string toEmail, string subject,
string bodyText, string fromName, string fromEmail)
{
    var dateSent = (DateTime?)null;

    // Send the mail
    var message = new MailMessage();
    message.To.Add(new MailAddress(toEmail, toName));
    message.From = new MailAddress(fromEmail, fromName);
    message.Body = bodyText;
```

```

        message.Subject = subject;

        try
        {
            // SMTP Host practically won't change.
            // It no longer needs to be passed as a variable.
            var client = new
SmtpClient(_settingsService.GetStringSetting("Contact:SmtpHost"));
            client.Port = (int)_settingsService.GetSetting("Contact:SmtpPort");
            client.EnableSsl =
(bool)_settingsService.GetSetting("Contact:SmtpUseSsl");
            client.Credentials = new System.Net.NetworkCredential(
                _settingsService.GetStringSetting("Contact:SmtpUsername"),
                _settingsService.GetStringSetting("Contact:SmtpPassword")
            );
            client.Send(message);
            dateSent = DateTime.UtcNow;
        }
        catch (Exception ex)
        {
            // TODO: Add logging at some point
        }

        return dateSent;
    }

```

We can now delete `GetConfiguration()` from **EmailService**, thanks to our injected alternative. Technical debt paid.

## HomeController briefly revisited

HomeController only really needs access to settings because it needs to know where to send contact emails to. It is a demand on the method we're using to `SendEmail`, made by the contractual requirements of the **IEmailService** interface. Before we modify HomeController, perhaps we're better off asking whether it is getting good service from **IEmailService**.

Irrespective of the sender of the message, the resulting email will always be sent from a Sightsource Bot to a listed Sightsource email address. Those addresses and names all come from the configuration, do not change between requests and are available to EmailService. If EmailService were able to read those values from the configuration, **HomeController** would not need direct access to the configuration at all.

## Refactoring IEmailService

Let's add another declaration to IEmailService. From now on, it is going to support a method called `SendContactEmail`. The signature is as follows.

```

// In this particular case, that'll be sending a contact email.
DateTime? SendContactEmail(string fromEmail, string fromName, string message);

```

This is a much more focused method. Unlike our previous method, which was designed to send an email from anyone to anyone, this method only wants the bare essentials collected on the form.

- Who is the email from?

- What is the person's email address?
- What is the person's message?

All of the other things we need can be pulled from the configuration via `ISettingsService`. We've defined a method that needs much less from any calling client to do its work.

## Creating the concrete implementation

Our **EmailService** class is now out of date, and no longer meeting its contractual requirement to do everything that **IEmailService** says it will do. We'll need to ensure that it honours the **SendContactEmail** method if we want our project to continue building.

We don't have to reinvent the wheel. We can use our existing **SendEmail** in aid of this effort. We also get to use the injected **ISettingsService** to get at the values we need.

```
public DateTime? SendContactEmail(string fromEmail, string fromName, string
message)
{
    return SendEmail(
        _settingsService.GetStringSetting("Contact:DestinationName"),
        _settingsService.GetStringSetting("Contact:Destination"),
        "Sightsource Empower - Feedback",
        $"Sent by: {fromName} <{fromEmail}>{Environment.NewLine}{message}",
        _settingsService.GetStringSetting("Contact:DoNotReplyName"),
        _settingsService.GetStringSetting("Contact:DoNotReply")
    );
}
```

## A tiny bit of privatisation

At this point, we have to ask, do we really need **SendEmail** to be a **public** method on **EmailService**, or a contractual requirement of **IEmailService**. Now that we have a simpler method to call, our app doesn't actually need to call **SendEmail** directly at all. **SendContactEmail** will do it on our app's behalf.

I'd argue that this is a bit of functionality that we might like to use, but the ability to let someone send an email from any address to any address is probably not something we want to publicly advertised.

## Making public private

We can change the **accessibility** of **SendEmail** by changing the public accessibility to private.

```
// This has now been marked private.
// It is no longer directly callable from an IEmailService, but then it doesn't need
to be.
// The stuff that is callable can still use it inside this class.
private DateTime? SendEmail(string toName, string toEmail, string subject, string
bodyText, string fromName, string fromEmail)
```

## Relaxing IEmailService

Our **EmailService** concrete implementation doesn't fulfil the contractual requirements of **IEmailService** anymore, because the **SendEmail** method is no longer visible. We're going to relax the requirements on **IEmailService**. The *only* thing we want it to do now is **SendContactEmail**.

Our reworked interface should look as follows:-

```
namespace Empower.Services.Interfaces
{
    /// <summary>
    /// This interface defines and describes all operations we'd want to
    /// perform for email operations
    /// </summary>
    public interface IEmailService
    {
        // The big generic SendEmail is no longer in this interface. Instead, this
        // contains operations that more closely resemble the actual operations that
        // the site will perform.
        //
        // In this particular case, that'll be sending a contact email.
        DateTime? SendContactEmail(string fromEmail, string fromName, string message);
    }
}
```

Changing HomeController to call SendContactEmail.

Now that we've added this new functionality, we can greatly simplify what goes on in our **HomeController**'s action method. The new method will do most of the work for us, using other services to fulfil those needs if it cannot directly do so itself.

Our new POST contact method looks like this:-

```
// This will only respond to requests which supply the POST HTTP verb.
// The POST verb is being provided in the 'method' attribute of the calling form
//
[HttpPost]
public IActionResult Contact(ContactViewModel request)
{
    ViewData["Message"] = "You tried to get in touch with Empower";

    if (ModelState.IsValid)
    {
        request.CompletedAt =
            _emailService.SendContactEmail(request.Email, request.Name,
            request.Message);
    }

    return View(request);
}
```

It's *tiny*!



All the details it used to handle are now being handled by `_emailService`. It's only job is to check the validity of the data it receives, handing everything else over to **IEmailService**.

We can also delete the last vestiges of **GetConfiguration()** from our app. We can remove our copy in **HomeController**, as that class is no longer directly concerned with configuration.

The controller is finally, really, truly, just controlling 😊

## What just happened?

- We decided to create a service that would pluck settings from the application's configuration for us
  - With just two methods, **GetSetting** and **GetStringSetting**, **ISettingsService** was defined in **Empower.Services**, our library which defines our apps' major operations.
- We created a new C# class library entitled **Empower.Settings.Services**.
  - We created a **concrete implementation** of **ISettingsService** called **SettingsService**.
    - We specified that **SettingsService** depended on getting an **IConfiguration**.
    - We completed implementations for **GetSetting** and **GetStringSetting**
- We injected **ISettingsService** into **Empower.Network.Services** **EmailService** class.
- We refactored **EmailService** to use the injected settings
  - We *deleted* our naughty copy and pasted code.
- We broadened the behaviour of **IEmailService** to include an option to **SendContactEmail**, a specialised version of **SendEmail** able to deduce most parameters.
  - We implemented that method in **EmailService** reusing existing code and services.
- We changed the Contact POST action on **HomeController** to call the new simplified method.
  - We deleted **GetConfiguration()** there too!
- We changed **SendEmail** to be a private implementation method for **EmailService**.
  - It is no longer a contractual requirement of **IEmailService**.
  - It is still used in the fulfilment of **SendContactEmail**
- We built a dependency chain.
- We truly separated concerns.
- We did a lot of MVC

## Commit your work

Merge back into master. It's time to cover the Domain. Don't worry. You'll adapt.

## Exercise 3 - The Domain

---

So far, we've only really dealt with three kinds of projects. We've got our MVC application, **Empower.Services**, a collection of service interfaces defining what our service classes, and by extension what our app can do. The final type of project making up the number is the service implementation project. We created **Empower.Network.Services** to handle the work of **IEmailService** and **Empower.Settings.Services** to handle the work of **ISettingsService**.

One of the key reasons that our app is structured this way, and that we haven't just coded up all of our services in the MVC project, is future reusability. Someone wanting to produce a mobile app version of this site would be able to use those libraries without having to reference anything in the MVC project. By placing service implementation classes in their own libraries, future developers can simply reference and reuse the work we've done here. There is value in keeping services independent.

If that's true, then there is also a great deal of value in keeping data structures, classes and other common requirements defined outside of the MVC project. One of the most common ways developers achieve this separation is through the creation of a Domain project.

In this exercise, we're going to learn what a domain project is, what to put in it, and how it helps us create apps that are not only reusable, but easy to maintain.

### Another of the most misused words in computing

**Domain** is used everywhere in computing. We've got Domain names, Windows Domain Servers. For a purposes, a domain can be treated as all the things a system needs to effectively do its job. Those guys at the pizza store have ovens in their domain. They need them to do their jobs. Deliveries are part of their domain, incoming and outgoing. If one were minded, one could come up with a reasonable approximation of all the objects those pizza guys need to complete their service.

That's how we're treating **domain**. It's all the figurative pots and pans of our pizza shop. The things our services will work with. The things we might want to allow other things, not just our MVC app, to use.

### Create a new git branch from master

We're going to put our new work into a branch called feature/domain.

Create that branch and check it out.

### Create the Empower.Domain project

Within Visual Studio, create a new C# Class Library project called **Empower.Domain**.

Delete the usual Class1.cs nonsense.

### Commands

Until now, we've not made any real use of **commands**. Our **SendEmail** method in **EmailService** is a good example of this not occurring. Its inputs are a stream of primitive values. It returns the date and time of any successful contact, but provides no more information than that when something goes wrong.

```
// This has now been marked private.  
// It is no longer directly callable from an IEmailService, but then it doesn't need  
// to be.  
// The stuff that is callable can still use it inside this class.  
private DateTime? SendEmail(string toName, string toEmail, string subject, string  
bodyText, string fromName, string fromEmail)
```

This style of method is a pain from a maintenance perspective too. If we need to add extra fields, we need to alter every call into `SendEmail`, providing the new information. Entering information by ordinal position is fault prone, not caught by the compiler when the types are the same, as they all are in this method.

**Commands** offer a better alternative. Request and response commands are a common convention.

## Request and Response

**Request** and **Response** classes are designed to convey the inputs and outputs of a command. The requirements of a command are lodged in the **Request** object. The **Response** object is designed to carry any gleaned result. We're going to use that convention to rework our **SendEmail** mechanism.

## Client

Another convention that Domain libraries follow is the Client subdirectory. This directory typically contains information that is likely to be passed around and/or populated by client applications. Small, lightweight, dumb classes designed to shuttle information back and forth. Some programmers refer to this type of class as "data transport objects", or DTOs.

Create a subdirectory in the Domain project called Client.

## Enums

The Domain.Client library is a decent place to store **enumerations**, structures that allow us to define our applications without resorting to *magic numbers*

Let's create an Enums subdirectory.

Once complete, let's add a new file called EmailEnums - this just needs to be a code file, as opposed to class.

```
namespace Empower.Domain.Enums  
{  
    public enum EmailDestination  
    {  
        Unknown = 0,  
        Contact = 1  
    }  
}
```

The **EmailDestination** enum is a very short list of where our email might go. Either to a Contact address, or we don't know.

## Commands

Next, let's create a subdirectory inside Client called Commands. We'll house our command classes here, and we'll split them according to function.

Inside Commands, create two new subdirectories, Requests and Responses.

### SendEmailRequest & SendEmailResponse

We're going to create two new classes that encapsulate a request to send an email, and a response containing the result of the request.

In the Requests directory, create a class called **SendEmailRequest**. Let's deck it out in the following way. Note that we've used the **EmailDestination** enum as a parameter too.

```
public class SendEmailRequest
{
    // These will come directly from the form
    public string FromEmail { get; set; }
    public string FromName { get; set; }
    public string Message { get; set; }

    // This will be provided inside the controller
    public EmailDestination Destination { get; set; }
}
```

Let's head into the Response directory and craft something to hold the result of a SendEmailRequest.

Create a new class called **SendEmailResponse**.

```
public class SendEmailResponse
{
    // The ? means that the type can be NULL
    public DateTime? CompletedAt { get; set; }

    // A very simple place to store an error
    public string ErrorMessage { get; set; }

    // A calculated property which tells us if there were any problems
    public bool HasErrored => (!string.IsNullOrEmpty(ErrorMessage));
}
```

This is a very simple class, not entirely dissimilar to the way we do things now. We've included a nullable **DateTime** called **CompletedAt**. We'll only set this if the send was successful. We've defined a new property called **ErrorMessage**, along with a convenience property called **HasErrored**, which returns true if the **ErrorMessage** is non-empty.

### References

In the first instance, we'll need to ensure that **Empower.Services**, **Empower.Network.Services** and **Empower.Mvc** all reference the new **Empower.Domain** project. It's time to start hooking Domain in.

## Refactoring IEmailService to use Request and Response objects.

Now we've got command objects defined in Domain, we can refactor `IEmailService` to use those in preference to the currently established primitives.

Let's open up **Empower.Services** and change the needs of **IEmailService**.

We're now going to pass a single parameter, a **SendEmailRequest**, and we're going to return an **SendEmailResponse**.

The changed code should look as follows:-

```
/// <summary>
/// This interface defines and describes all operations we'd want to
/// perform for email operations
/// </summary>
public interface IEmailService
{
    // The big generic SendEmail is no longer in this interface. Instead, this
    // contains operations that more closely resemble the actual operations that
    // the site will perform.
    //
    // In this particular case, that'll be sending a contact email.
    SendEmailResponse SendContactEmail(SendEmailRequest request);
}
```

## Making good on our change in EmailService

Of course, the above change is going to come as a complete shock to our concrete implementation, **EmailService**, once again out of step with the ever changing demands of **IEmailService**.

The refactoring isn't that hard. We simply need to pull information from the properties of the request object, rather than individual parameters.

We also need to ensure we return a **SendEmailResponse**.

```
public SendEmailResponse SendContactEmail(SendEmailRequest request)
{
    // NEW: Create a response.
    var response = new SendEmailResponse();

    response.CompletedAt = SendEmail(
        _settingsService.GetStringSetting("Contact:DestinationName"),
        _settingsService.GetStringSetting("Contact:Destination"),
        "Sightsource Empower - Feedback",
        // CHANGE: pull from request properties, instead of individual
variables
        $"Sent by: {request.FromName} <{request.FromEmail}>
{Environment.NewLine}{request.Message}",
        _settingsService.GetStringSetting("Contact:DoNotReplyName"),
        _settingsService.GetStringSetting("Contact:DoNotReply")
    );

    // Sent at will only be NULL if something has gone wrong
    if (!response.CompletedAt.HasValue)
```

```
        {
            response.ErrorMessage = "Could not send your email";
        }

        return response;
    }
}
```

## Fixing the controller method.

Both **IEmailService** and **EmailService** have now been refactored, which means our controller call to **SendContactEmail** no longer works. We'll need to refactor the controller method to use the request and response objects.

```
[HttpPost]
public IActionResult Contact(ContactViewModel request)
{
    ViewData["Message"] = "You tried to get in touch with Empower";

    if (ModelState.IsValid)
    {
        // We're now constructing a SendEmailRequest
        // instead of simply passing parameters.
        //
        // The controller method is a bit more verbose, but a lot more
        // readable.
        var sendResult =
            _emailService.SendContactEmail(
                new SendEmailRequest
                {
                    FromEmail = request.Email,
                    FromName = request.Name,
                    Message = request.Message
                });

        if (sendResult.HasErrored)
        {
            ViewData["Message"] = sendResult.ErrorMessage;
        }
    }

    return View(request);
}
```

## Compile and run your work

Congratulations. You're now running a version of **IEmailService** which accepts commands. You may ask, reasonably, why this was important. First, encapsulating everything into a request is quite useful. The command is complete. You do not have to guess about which parameter goes where, if it is needed at all. More importantly, you can provide a detailed response that you can act on, including error messages, details of a transaction, etc.

## What just happened?

- We created a new project, **Empower.Domain**, which is designed to hold all of the structures and classes we need to do our job.
- We created a Client subdirectory.
  - By convention, the Domain.Client namespace has been used to contain simple classes aimed at transporting data between layers of the application.
  - Another convention is to call such classes DTOs, or data transport objects.
- We introduced the notion of **Commands**
  - Specifically **Request** and **Response** commands
- We created two new command objects to represent the sending of an email
  - **SendEmailRequest**
  - **SendEmailResponse**
- We refactored **IService**
  - Our stream of parameters is gone.
  - We pass in one object of type **SendEmailRequest**, encapsulating everything we need to send an email request.
  - We return a **SendEmailResponse** object, which is designed to report on the **SendContactEmail** operation.
- We refactored **HomeController** to send a **SendEmailRequest** through `_emailService`.
  - We also worked with a **SendEmailResponse** to show the final state in the POST action.

## Commit your work.

Let's commit all work to date, and merge our **feature/domain** work back into the master branch.

## Exercise 4 - Adapters

---

We all know what an adapter is. We use them every day, whether it is getting a big USB hole to talk to a small one, a UK plug to work with an American socket or at perhaps its crudest and purest form, duct tape. In real life, we use adapters when we need to hook a couple of things together and they don't quite fit. The adapter bridges the gap.

The Adapter pattern is used in computing too. Just as in real life, its job is to bridge the gap between two things that don't quite fit. We've already and perhaps unknowingly created an adapter in our HomeController Contact POST action.

```
// We've got a ContactViewModel.
// _emailService needs a SendEmailRequest
//
// We are adapting on the fly, in the controller,
// when we create a new SendEmailRequest
var sendResult =
    _emailService.SendContactEmail(
        new SendEmailRequest
        {
            FromEmail = request.Email,
            FromName = request.Name,
            Message = request.Message
        });
```

So adapting isn't hard. We've already done it. We can just do it in a more disciplined, predictable and more productive way. We can use the Adapter pattern.

Before we do that, let's have a small sort out with respect to our ViewModels.

Create a new branch

Create a new git branch off master called feature/adapters.

## What goes in a Domain?

During exercise 3, we created a **Domain** library, to house structures that allow us to do our work. A common conundrum that coders face is where the dividing line is. What stays in the MVC project? What moves to the **Domain** project. A simple rule of thumb is to ask the following questions:-

- Is my class or data structure entirely concerned with MVC?
  - e.g. the Startup.cs class, Controller classes.
  - If so, the class stays in MVC.
- Does my class or data structure have any value outside of MVC?
  - This could be yes because
    - You want to use it in other libraries
    - It has information useful to other libraries
  - If it's yes, the class should probably be in the Domain

## Moving ViewModels into Domain

At first glance, ViewModels might be seen as a certainty for residence in the MVC project. While that is certainly true of the **cshtml** views that dictate how these are shown, it is not true for ViewModels, especially when we apply the two questions above. ViewModels are really dumb transport objects. They do not contain any web-specific information, certainly nothing specific to MVC.

Secondly, ViewModels are annotated with information about how big fields should be, or their type and composition. That is information that could be useful to other classes. According to our rule of thumb questions, ViewModels should really live in the Domain.

## Expanding Domain.Client

ViewModels more than fit the definition of being a Client class. Let's navigate to the **Empower.Domain** project, open the **Client** directory and create a new sub-directory called **ViewModels**. Add two new classes, ContactViewModel and ErrorViewModel.

Copy the class definitions (inside the namespace tag) from their soon to be obsolete equivalents in **Empower.Mvc/Models**. Your completed classes should look as follows.

### ContactViewModel.cs

```
namespace Empower.Domain.Client.ViewModels
{
    public class ContactViewModel
    {
        [DisplayName("Your name")]
        [Required(ErrorMessage="Please enter your name")]
    }
}
```



```

        [MaxLength(100, ErrorMessage="How long does it take you to fill in forms?
Please enter 100 characters or less.")]
        public string Name { get; set; }

        [DisplayName("Your email address")]
        [Required(ErrorMessage = "Please enter your email")]
        [EmailAddress(ErrorMessage = "Please enter a valid email")]
        public string Email { get; set; }

        [DisplayName("Your feedback")]
        [Required(ErrorMessage = "Please enter your message")]
        [MaxLength(4000, ErrorMessage="Too long. Didn't read.")]
        public string Message { get; set; }
        public DateTime? CompletedAt { get; set; }
    }
}

```

### ErrorViewModel.cs

```

namespace Empower.Domain.Client.ViewModels
{
    public class ErrorViewModel
    {
        public string RequestId { get; set; }

        public bool ShowRequestId => !string.IsNullOrEmpty(RequestId);
    }
}

```

We can now delete the **ContactViewModel** and **ErrorViewModel** classes from the **Models** subdirectory of the MVC project. We'll need to make some minor alterations.

### Changing \_ViewImports.cshtml

We need to tell MVC that it should no longer look in Empower.Mvc.Models for its ViewModels. Let's open up **Views/\_ViewImports.cshtml** and tell it all about our new namespace.

```

@using Empower.Mvc
@using Empower.Domain.Client.ViewModels
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
@*
    This is the view imports file. We made one important change here which told MVC
to look for ViewModels in the
    Empower.Domain.Client.ViewModels space, instead of Empower.Mvc.

    Any class in a namespace we reference with the using keyword can be referenced
without fully
    qualifying it.
*@

```

Save the changes. Build and run your project. You should now be using ViewModels housed in **Empower.Domain**, where they can also be used for other purposes.

## New project - Empower.Adapters

The groundwork we did in moving ViewModels has prepared us well for our next task. Let's create a new project called **Empower.Adapters**

It should reference

- Empower.Domain

It should be referenced by

- Empower.Mvc
- Empower.Services

### A basic adapter

We're going to create an adapter designed to adapt **ContactViewModel** instances into **SendEmailRequest**, as well as the opposite operation, turning **SendEmailResponse** objects into **ContactViewModel** objects.

For right now, we're going to spin it up as a concrete instance. We'll have to improve on that later, but it does mean we can focus on the fundamentals of what an adapter does. They are quite simple. This is one of the simplest imaginable.

In your **Empower.Adapters** project, create a subdirectory called **ViewModels**. This will house any adapter that deals with ViewModels.

Add a new class in that subdirectory called **ContactAdapter**.

We're going to add two new methods, **AdaptRequest**, which will turn a **ContactViewModel** into a **SendEmailRequest** and an **AdaptResponse**, which will turn a **SendEmailResponse** into a **ContactViewModel** object.

```
namespace Empower.Adapters.ViewModels
{
    public class ContactAdapter
    {
        public SendEmailRequest AdaptRequest(ContactViewModel viewModel,
        SendEmailRequest request)
        {
            // Very simple mapping.
            // Just copying one property into another
            request.FromEmail = viewModel.Email;
            request.FromName = viewModel.Name;
            request.Message = viewModel.Message;

            return request;
        }

        public ContactViewModel AdaptResponse(ContactViewModel viewModel,
        SendEmailResponse response)
        {
```

```

        // Conditional mapping based on the state of the response.
        // If we've errored, we set the status message to reflect that.
        if (response.HasErrored)
        {
            viewModel.HasErrored = true;
            viewModel.StatusMessage = response.ErrorMessage;
        }

        return viewModel;
    }
}

```

## Using the Adapter in a controller

It's nowhere near optimal, but we can use that adapter, as is, in our controller method.

```

[HttpPost]
public IActionResult Contact(ContactViewModel request)
{
    ViewData["Message"] = "You tried to get in touch with Empower";

    if (ModelState.IsValid)
    {
        var adapter = new ContactAdapter();

        // We're now adapting a send request
        var sendResult =
            _emailService.SendContactEmail(
                adapter.AdaptRequest(request, new SendEmailRequest()));

        // Bit long winded, but adapt the response.
        // HasErrored will be set by the adapter if anything has gone wrong.
        request = adapter.AdaptResponse(request, sendResult);
    }

    return View(request);
}

```

## Changing the view to alert the user of errors.

Now let's open up Contact.cshtml and add the following conditional display.

```

@if (Model.HasErrored){
    <div class="alert alert-danger">
        <h3>@Model.StatusMessage</h3>
    </div>
}

```

This will just inform our user if the operation has errored in any way.

## A smarter adapter

Once again, we've gotten away with firing up a concrete **ContactAdapter** instead of using some form of dependency injection. It'll work for now, because it has no other dependencies, but other adaptation processes may be more involved. Some adapters may require external information to successfully adapt their information. Some adapters will have dependencies. By new'ing up a concrete class, we lose the chaining that dependency injection gives us.

Furthermore, this sort of thing, turning a ViewModel into a command, and vice versa, is something that we may end up doing with many of our adapters.

## A request / response adapter

In general, the demands of a request response object are quite easy to establish. If we give it a ViewModel, we'd like it to give us a command we can run on a service. If we give it the result of a command, we'd like a ViewModel in return.

While the specifics of the ViewModels and command objects are going to alter, the process being performed is the same on a general level. We cannot predict every property of every ViewModel or command object, but we can enforce a mechanism whereby adaptations at least work on the same level.

We're going to create an *IRequestResponseAdapter*.

In VS, navigate to your Adapters project and create a new directory called **Interfaces**. This does constitute a small break with prior convention. So far, we've been putting all of our interface definitions into **Empower.Services**. However, the whole app doesn't need to know about the composition of this interface. It is only concerned with the **Empower.Adapters** part of the solution. Sometimes, there is a good case for keeping an interface close to the implementation.

Create a new interface called **IRequestResponseAdapter**.

We're going to use *generics* to define the types in this interface, to allow greater reusability.

Our interface declaration is going to introduce them.

```
public interface IRequestResponseAdapter<TRequest, TResponse, TViewModel>
```

This interface is essentially declaring "I am a IRequestResponseAdapter. I will work with a type called TRequest. I will work with a type called TResponse. I will work with a type called TViewModel. Not really sure what any of them are, if I'm honest".

The three identifiers beginning with T are generic type placeholders, established elsewhere through declaration. Right now, IRequestResponseAdapter doesn't actually know much about these types. They are all type **object**.

```
namespace Empower.Adapters.Interfaces
{
    /// <summary>
    /// This is an interface which describes what a request response adapter should
    do.
    ///
    /// It is a generic interface. That means the coder must include the types being
```

```

used
    /// in any declaration.
    ///
    /// There are good reasons to have application level service controllers that
exist
    /// in the Services project, but not all interfaces need to live there.
    ///
    /// There are often good cases for keeping the Interface close to the problem it
solves.
    ///
    /// </summary>
    /// <typeparam name="TRequest"></typeparam>
    /// <typeparam name="TResponse"></typeparam>
    /// <typeparam name="TViewModel"></typeparam>
    public interface IRequestResponseAdapter<TRequest, TResponse, TViewModel>
    {
        TRequest AdaptRequest(TViewModel viewModel, TRequest request);
        TViewModel AdaptResponse(TViewModel viewModel, TResponse response);
    }
}

```

## Getting ContactAdapter onboard.

We can amend ContactAdapter so that it subscribes to this new contractual behaviour pretty easily. In fact, ContactViewModelAdapter already does what IRequestResponseAdapter demands.

We just need to specify that it can do it.

Open up **ContactAdapter**. We're doing to declare that right now.

```

public class ContactAdapter :
    IRequestResponseAdapter<SendEmailRequest, SendEmailResponse, ContactViewModel>

```

We've tagged on a declaration that ContactAdapter is a **IRequestResponseAdapter**, using **SendEmailRequest** as its **TRequest** type, **SendEmailResponse** as its **TResponse** type and **ContactViewModel** as the **TViewModel** type.

This class *already fulfils* the needs of **IRequestResponseAdapter**. It can adapt a request. It can adapt a response.

## Injecting ContactAdapter as an IRequestResponseAdapter.

Our **HomeController** keeps playing house to bad practice. Right now, we're using a concrete implementation of **ContactAdapter** to perform our adaptation. Now that we've defined the general behaviour of a request response adapter, we can use dependency injection to gain all of the benefits.

## Wiring the adapter up

Before we can use an injected dependency, we have to tell MVC how to satisfy a request for an IRequestResponseAdapter.

Let's open Startup.cs in the MVC project, and add some more configuration to our services.

```

// This method gets called by the runtime. Use this method to add services to the
container.
public void ConfigureServices(IServiceCollection services)
{
    // Only one copy of the settings service will exist in memory at any time.
    services.AddSingleton<ISettingsService, SettingsService>();

    // On demand. Multiple instances will be created as needed.
    services.AddTransient<IEmailService, EmailService>();

    // We've told the dependency injection engine that whenever a class asks
for
    // a IRequestResponseAdapter, with those three types specified
    // that the ContactAdapter should be provided as the implementation
    //
    // Note, this can get a little cumbersome with lots of adapters, but there
    // are ways to simplify this with Factory patterns.
    //
    // For right now, this will do.
    //
    services.AddSingleton<IRequestResponseAdapter<SendEmailRequest,
SendEmailResponse, ContactViewModel>, ContactAdapter>();

    services.AddMvc();
}

```

We've told the DI framework about our RequestResponseAdapter, and that when one is asked for using those types, MVC should supply it with a **ContactAdapter**

```

private readonly
IRequestResponseAdapter<SendEmailRequest, SendEmailResponse, ContactViewModel>
_contactAdapter;

/// <summary>
/// Note : We are now injecting a contactAdapter without directly specifying
its type.
/// </summary>
/// <param name="emailService"></param>
/// <param name="contactAdapter"></param>
public HomeController(
    IEmailService emailService,
    IRequestResponseAdapter<SendEmailRequest, SendEmailResponse,
ContactViewModel> contactAdapter)
{
    _emailService = emailService;

    // Our Adapter will now be injected. All we need to do now is assign the
incoming
    // variable to our private variable
    _contactAdapter = contactAdapter;
}

```

We can now rid the **HomeController** of yet another problem, the concrete creation of **ContactAdapter**. If we want to use its functionality, we can reference it using the **\_contactAdapter** variable, injected through **constructor injection**.

Our POST action method becomes a little simpler.

```

// This will only respond to requests which supply the POST HTTP verb.
// The POST verb is being provided in the 'method' attribute of the calling
form
//
[HttpPost]
public IActionResult Contact(ContactViewModel request)
{
    // Now just setting the property on the model.
    request.StatusMessage = "You tried to get in touch with Empower";

    // This checks validation rules to determine whether the information
    // supplied in request will meet the needs of the request.
    //
    // UPDATE: This will now validate according to the rules of the validators
    // on ContactViewModel. It'll return false if any of the rules we've set
are
    // broken.
    if (ModelState.IsValid)
    {
        // The adapter is now handling all the mapping of fields from
        // our view model into our request object.
        //
        // We are also using the static methods created on the
        SendEmailRequest object
        // to create a new contact request without too much fuss.
        //
        var sendResult =
            _emailService.SendContactEmail(
                _contactAdapter.AdaptRequest(request,
        SendEmailRequest.NewContactRequest)
            );

        // We can also use the adapter to update our view model from the
        response we've received.
        request = _contactAdapter.AdaptResponse(request, sendResult);

    }

    return View(request);
}

```

We don't have to *new up* a **ContactAdapter** anymore. We're still using one - we're just using a set of rules defined in **Startup.cs** to ensure it gets used.

## What just happened?

- We rediscovered the ubiquity of Adapters in real life.

- We discovered that Adapters are quite common in computing too.
  - Adapters are a programming pattern that help one thing fit with another
- In preparation, we moved our ViewModels into the Domain.Client namespace
  - We asked questions about whether they really belong in Mvc or Domain
  - We decided that they have a use outside Mvc, particularly if we want to adapt them
- We created a new class library, **Empower.Adapters** designed to hold our adapters.
- We created a **ContactAdapter**, designed to turn ViewModels into command requests and command responses back into ViewModels.
- We created a broad generic service definition called **IRequestResponseAdapter**, which generalised and genericised the work we did in ContactAdapter.
- We amended **ContactAdapter** to say that it was capable of fulfilling the requirements of **IRequestResponseAdapter**.
- We changed **Startup.cs**, specifying that if it saw anything claiming to be an **IRequestResponseAdapter**, dealing with the **SendEmailRequest**, **SendEmailResponse** and **ContactViewModel** objects, MVC should serve up a **ContactAdapter**.
- We got **HomeController** to ask for an **IRequestResponseAdapter** with *just* those characteristics through *constructor injection*.
- We modified our Contact POST action to use the injected adapter, available in **\_contactAdapter**.

## Commit your work

We're almost done with **feature/adapters**. We've a couple more improvements to make in Exercise 5.

## Exercise 5 - The final day two refactors

---

We're in good shape. We've created a Domain library to hold all of our business concepts. We filled that with some Commands, encapsulating the inputs and outputs of sending an email. We've created an Adapter that'll translate our ContactViewModel into a SendEmailRequest, and also adapt the response into something useful.

There are a couple of things we should not ignore and a couple of questions we need to ask.

### The problem with IRequestResponseAdapter

We created a generic interface which defined some broad functionality for adapters. Luckily enough, our implementation of **ContactAdapter** does actually do something sensible with the values.

A future coder could use that interface with classes that are not request commands, command responses or ViewModels. That's not what the class is designed for. It would be good if we could somehow constrain the type.

The less used interpretation of an interface.

Whenever we've used interfaces so far, they've always been a facade over something that *does* something. We have an **IEmailService**, which is something that services our app's email requests. We have an **IRequestResponseAdapter**, which promises to faithfully adapt things. The "does something but doesn't let on how it does it" has been the de facto way we've treated interface declarations in the past.



Just as usefully, interfaces can just be used to say *i am something*.

Let's labour the point.



Let's open up our **Empower.Domain** project, open the *Client* subdirectory and create a new directory called **Interfaces**. This is another time we're going to keep our interfaces close to our implementations.

We are going to add the following, highly contrived, likely to annoy Ali, interface names. We'll refactor them later.

Add the following interfaces to the **Interfaces** directory:-

- IAmARequest
- IAmAResponse
- IAmAViewModel

Make them all public. We should have three very simple code files that look like this:-

```
public interface IAmARequest
{
}
```

```
public interface IAmAResponse
{
}
```

```
public interface IAmAViewModel
{
}
```

Note that we've defined no behaviour. We do not need to. We are just interested in the *I am* qualities of an interface.

## Constraining IRequestResponseAdapter

We don't need to take any old object in IRequestResponseAdapter. We can place some constraints, and now that we've defined a few interfaces, we can do just that.

```
public interface IRequestResponseAdapter<TRequest, TResponse, TViewModel>
    // We don't want people to implement this interface with any old type.
    //
    // The "where" parts of this declaration specify that TRequest has to say
    IAmARequest.
    //
    where TRequest : IAmARequest
    where TResponse : IAmAResponse
    where TViewModel : IAmAViewModel
{
    TRequest AdaptRequest(TViewModel viewModel, TRequest request);
}
```

```

        TViewModel AdaptResponse(TViewModel viewModel, TResponse response);
    }

```

We can introduce constraints using the **where** **TRequest** needs to be able to say **IAmARequest**, **TResponse** needs to be able to say **IAmAResponse** and **TViewModel** needs to be able to say **IAMAViewModel**.

Our **ContactAdapter** class, compliant from the start on **IRequestResponseAdapter**, is now broken. None of the types it deals with can claim those things.

Modifying our Domain.Client objects to comply.

We can fix **ContactAdapter** elsewhere. All we need to do is make sure that its participating objects can claim they are requests, responses or viewmodels.

Let's open up **SendEmailRequest** and get it to explicitly say **IAmARequest**.

```

public class SendEmailRequest : IAmARequest
{
    // ....
}

```

Same deal with **SendEmailResponse**, except it needs to claim **IAmAResponse**.

```

public class SendEmailResponse : IAmAResponse
{
}

```

And finally, let's do **ContactViewModel**. It needs to claim **IAMAViewModel**.

```

public class ContactViewModel : IAmAViewModel
{
}

```

With those three declarations, we've unbroken **ContactAdapter**. All of its objects now meet the constraints we've laid down. Anything wanting to use the adapter has to make similar claims.

## Emergency renaming

Ali is probably quivering in a corner right now. *IAmARequest*, *IAmAResponse* and *IAMAViewModel* are very verbose. Let's rename them to **IRequest**, **IResponse** and **IViewModel** respectively. Choose the option to update all references.

This should turn your Domain.Client objects as so:-

```

public class SendEmailRequest : IRequest
{
    // ....
}

```

```
public class SendEmailResponse : IResponse
{
```

```
public class ContactViewModel : IViewModel
{
```

Finally, let's make a *few* demands of **IResponse**. Anything claiming to be an **IResponse** should be able to tell us if there is an **ErrorMessage**, whether the command **HasErrored** and which time it was **CompletedAt**.

```
public interface IResponse
{
    bool HasErrored { get; }
    string ErrorMessage { get; set; }
    DateTime? CompletedAt { get; set; }
}
```

What have we achieved with this refactor?

More than anything else, we've made sure that any programmer wanting to use that adapter pattern has to at least be intentional about it. The classes should at least claim to be the types of classes they should be. No-one is going to be able to fire up our Adapter with a copy of **System.Text.StringBuilder**.

Our classes all pass the constraint test, because we've got them to make the claim.

We've also laid some of the foundations for tomorrow, and discovered that an interface's sole use isn't just *I do something*. We can use the *I am something* character of interfaces to ensure that only appropriate types are used in concrete implementations.

## Is the controller still controlling?

The second refactor concerns **HomeController**. It is in much better shape than it was before, but does it really need to be the thing that adapts **ContactViewModel** back and forth from the Command objects we've defined in **Domain.Client**?

The controller should be **controlling**, not **adapting**.

What does the Contact POST action actually need?

This is an interesting question, but the answer is obvious when you look at the inputs and outputs. It receives a **ContactViewModel**. It returns a **ContactViewModel**. If there was a service that handled all of the adaptation, and just took and returned **ContactViewModel**, we'd simplify **HomeController** and could move the details about the adapter into the intermediary class.

## New Empower.Services interface

Open up the **Empower.Services** project. Create a new **interface** file and call it **IViewModelService**. Nothing else but our MVC app is going to use this, so there *is* an argument that we could define this in the MVC project.

However, one of the benefits of putting most of your major operations in a **Services** library is for future coders. They'll be able to peruse this library and quickly see that we've created a library for MVC operations.

The interface should look like this:-

```
public interface IViewModelService
{
    ContactViewModel Contact(ContactViewModel model);
}
```

Eventually, the interface will be an index of pretty much everything our MVC app will allow the controllers to do. This will soon be the only service available to any controller. Right now, there's only one proper MVC operation we need our app to do; send a contact email.

### New project?

For once, no. This service is *solely concerned* with servicing the needs of our MVC app. There's no good reason not to make it obvious.

In your MVC app, create a new subdirectory called Services.

We can put our MVC specific service in the project it relates to. Let's create a new class called **ViewModelService** in that Services directory, *implementing* the *IViewModelService*

```
public class ViewModelService : IViewModelService
{
    public ContactViewModel Contact(ContactViewModel model)
    {
        throw new NotImplementedException();
    }
}
```

Our class meets the contractual requirements, but is still throwing a **NotImplementedException**. We need to implement the behaviour. Thanks to other services we have available, this task should be trivial.

### Injecting the required services

Our **ViewModelService** can make use of other services we've created. We just need to inject them, as we have before. In reality, it only needs two services **IEmailService**, which will control email operations, and an instance of **IRequestResponseAdapter**, able to translate **ContactViewModel** objects into **SendEmailRequest** objects, and also being able to adapt the command response back into a ViewModel.

It needs two services.

- IEmailService
- IRequestResponseAdapter

Let's put those on the required list with **constructor injection** and create variables so that we can reference them.

```

public class ViewModelService : IViewModelService
{
    private readonly
    IRequestResponseAdapter<SendEmailRequest, SendEmailResponse, ContactViewModel>
    _contactAdapter;
    private readonly IEmailService _emailService;

    public ViewModelService
    (
        IRequestResponseAdapter<SendEmailRequest, SendEmailResponse, ContactViewModel>
        contactAdapter,
        IEmailService emailService
    )
    {
        // Assign our injected services, allowing them to access them.
        _emailService = emailService;
        _contactAdapter = contactAdapter;
    }

    public ContactViewModel Contact(ContactViewModel model)
    {
        throw new NotImplementedException();
    }
}

```

Now we've done that, implementing the Controller method should be trivial. All we need to do is *adapt* the **ContactViewModwl** into command objects, adapt the response and send it back.

```

public ContactViewModel Contact(ContactViewModel model)
{
    var commandResponse = _emailService.SendContactEmail(
        _contactAdapter.AdaptRequest(model, new SendEmailRequest());
    );

    model = _contactAdapter.AdaptResponse(model, commandResponse);

    return model;
}

```

We've completed the implementation of IViewModelServices. Time to wire it up.

## Changes to Startup.cs

We need to tell our MVC how app what to do when asked for an implementation of **IViewModelService**. It's time to open up **Startup.cs**, and set **ViewModelService**.

Let's create the association in **ConfigureServices**. We'll add it as a singleton. While we're about it, we may as well change the **lifecycle** of **IEmailService** to a singleton too. We're not reliant on any persistent internal state to run our methods.

```

public void ConfigureServices(IServiceCollection services)
{
    // Only one copy of the settings service will exist in memory at any time.
    services.AddSingleton<ISettingsService, SettingsService>();

    // On demand. Multiple instances will be created as needed.
    services.AddSingleton<IEmailService, EmailService>();
    services.AddSingleton<IRequestResponseAdapter<SendEmailRequest,
SendEmailResponse, ContactViewModel>, ContactAdapter>();
    services.AddSingleton<IViewModelService, ViewModelService>();
    services.AddMvc();
}

```

## Changes to HomeController.

We no longer need an **IEmailService** or **IRequestResponseAdapter** for **HomeController**. We can now give it the exact thing it needs to do its job. We can simplify the implementation again by passing in an **IViewModelService** instance, which already knows how to do all that stuff in a format that **HomeController** understands and doesn't have to dig for.

Our new readonly variables in **HomeController** should be as such, just one, the injected **IViewModelService**.

```

private readonly IViewModelService _mvcViewModelService;

/// <summary>
/// Note : We are now injecting a contactAdapter without directly specifying
its type.
/// </summary>
/// <param name="mvcViewModelService"></param>

public HomeController(
    IViewModelService mvcViewModelService
)
{
    _mvcViewModelService = mvcViewModelService;
}

```

## Changes to the Contact POST action

Thanks to our abstraction, our new Contact POST action is simpler than ever.

Everything is still happening. We've just delegated responsibility to the appropriate services.

```

[HttpPost]
public IActionResult Contact(ContactViewModel request)
{
    // Now just setting the property on the model.
    request.StatusMessage = "You tried to get in touch with Empower";
}

```

```

        if (ModelState.IsValid)
        {
            request = _mvcViewModelService.SendContactEmail(request);
        }

        return View(request);
    }

```

## What's just happened?

- We constrained our **IRequestResponseAdapter** using the **where** keyword to insist they had some sort of claim to be involved in adaptation.
- We created three "I am" interfaces, that do nothing else but constrain **IRequestResponseAdapter** to the sort of types it says it will handle.
- We amended our Request, Response and ViewModel objects so that they all declared what they were.
- This satisfied the need of our new **IRequestResponseAdapter** constraints, and fixed other code in the process.
- We questioned whether controllers should be involved in adaptation, even if they are just calling a service to adapt.
- We created a new service, called **IViewModelService**, designed to accept and return MVC ViewModels.
  - This uses a combination of **Adapters** and other services to get its result.
- We wired up the new dependency.
- We removed old dependencies from **HomeController**, now delegated to **IViewModelService**.
- We injected an instance of **IViewModelService**.
- We refactored our Contact POST model so that it uses **IViewModelService**, something purpose built to deal with ViewModels.

## Commit your work

We are complete on **feature/adapters**. Let's commit our work and merge back into master.

## What happened today?

- We learned the true nature of services
- We learned that services have dependencies, other services that they can use to complete their services.
- Services are key strategies for abstracting or hiding implementation details.
- Dependency injection allows us to model a chain of dependencies, allowing each service to find the services it needs to complete its work.
- We moved our email functionality, once defined in a controller method, into a service.
  - We created an interface, called **IEmailService**
  - We introduced **IEmailService** as a **constructor injection** dependency in **HomeController**.
  - We told MVC which class to use when something else was asking for an **IEmailService**
- We found we were violating DRY principles.
  - We'd copied and pasted code.
  - We created a single place for configuration items, **ISettingsService**, to cater for services that needed access to configuration.
  - Our concrete implementation of **ISettingsService**, **Empower.Settings.Services**, had a dependency itself, **IConfiguration**, which allows it easy access to the configuration.
- We created a more suitable method on **IEmailService**, requiring less input from the client to use.
- We were able to simplify **HomeController**, now completely freed from having to find configuration values.

- We created **Empower.Domain**, a class library designed to hold the things we need to conduct our business.
  - Created a **Client** subdirectory, intended for objects that would be accessible to other levels of the application.
    - In client, created a **Commands** subdirectory.
      - **Requests** subdirectory contains things we're asking for.
      - **Responses** subdirectory contains things we asked for.
- Refactored **IEmailServices** to take a **SendEmailRequest** as its request, and return a **SendEmailResponse** as its response.
  - Did the follow up work inside **EmailService**
  - Injected **ISettingsService** into **EmailService**, removing any need for it to have direct access to the configuration.
- Refactored **HomeController** to work with command objects.
- Created **Empower.Adapters**, designed to house classes that adapt one kind of thing into another kind of thing.
- Wrote **ContactAdapter**, which did all the transformation between the view model, **SendEmailRequest** and **SendEmailResponse**.
  - Used the adapter directly in **HomeController**
- Defined a broad interface, **IRequestResponseAdapter**, designed to perform similar work to **ContactAdapter**.
- Amended **ContactAdapter** to implement the demands of **IRequestResponseAdapter**,
- Used an injected adapter in **HomeController** to perform adaptations.
- Placed **constraints** on the sort of objects that **IRequestResponseAdapter** will accept.
  - Defined three *I am* interfaces:-
    - **IRequest**
    - **IResponse**
    - **IViewModel**
  - The **IRequestResponseAdapter** will only work with objects that claim to be those things.
  - Amended our Domain.Client classes to subscribe to these attributes.
- Created a new service, **IViewModelService**, specifically designed to deal with and return ViewModels.
  - The concrete implementation has **IEmailService** and **IRequestResponseAdapter** available to fulfil the request.
  - It is purely a mapping and wrapping class, but it means that we have managed to separate concerns, have each layer perform a specific function.
- Created one big dependency chain.
  - Here's how it goes.
    - **HomeController** *depends on*
      - **IViewModelService**
      - *implemented by* **ViewModelService** *depends on*
        - **IEmailService**
        - *implemented by* **EmailService** *depends on*
          - **ISettingsService**
          - *implenented by* **SettingsService** *depends on*
            - **IConfiguration**
            - *provided by* **Startup.cs**
    - **IRequestResponseAdapter**
    - *implemented by* **ContactAdapter**

## End of day two discussion

---



# Day Three - Persistence and CRUD

---

Today's exercises are going to be geared toward using persisted data in our application. Persistence is merely the technique whereby your computer doesn't forget everything that it has done. It doesn't come for free in any application, least of all web applications, which are famously forgetful. HTTP is a stateless protocol, meaning that it does not store any state between requests. Unless we provide a means to identify users on subsequent requests, nothing is remembered between requests. The same kind of principle exists with your computer, or phone. Every time either machine boots up, it is a blank slate. It's the process of restoring data from some persisted store, such as hard drive or memory card, that turns your machine from a factory setting piece of hardware into the device you know.

Loading and saving information is important.

## Persistence in web applications

If we want to remember anything about our users in our application, we need to be able to save details about them as they are entered, and be in a position to automatically reload them given the right inputs. The usual backing store for such information is a database. As users enter information, web apps usually save the important stuff to a database, providing mechanisms to reload it at a later point.

## SQL and NoSQL

Traditionally, most database backing stores have been relational databases, usually accessible on a programmatic level using a variant of SQL, standing for Standard Query Language. Some folks refer to SQL as "sequel", so listen out for that, but don't be tempted to adopt the term yourself. Anyone that really knows what they're talking about will know you've not really looked into the tech. Plus, it can lead to some fairly embarrassing job advertisements when combined with poor spelling.

Sequal dev wanted!

## SQL and relational databases

Relational databases store information in tables which relate to each other. When retrieving information from a relational database, it's pretty typical to be gathering that information from a number of different tables.

This is an example of a query pulling some order information from three related tables.

```
SELECT
    o.OrderNumber,
    o.OrderDate,
    ol.Quantity,
    ol.Price,
    ol.SalesTax,
    pr.Name
FROM
    Orders o
INNER JOIN
    OrderLines ol
ON ol.OrderId = o.Id
INNER JOIN
    Products p
ON ol.ProductId = p.Id
```

## NoSQL and document databases

NoSQL is an example of a document database. Instead of having multiple related tables, all information relating to a concept is stored in a document. The common convention of the day is to use **json** as the document format. An order in a NoSQL database might look as follows:-

```
{
  "Order": {
    "OrderNumber": 12345,
    "OrderDate": "2018/05/25",
    "OrderLines": [
      {
        "LineNumber": 1,
        "Quantity": 1,
        "Price": 10.00,
        "SalesTax": 1.25,
        "Product": {
          "Name": "Biffo Bleach"
        }
      }
    ]
  }
}
```

## Relational vs non-relational

Each database type has implementation advantages over the other. A well designed relational database will be likely be smaller, feature less duplication and will be able to enforce schema structure, including the types that should be used to store each piece of information. The drawbacks of a relational database are that they tend to be harder to query for newcomers, and as they scale up, the computing cost of maintaining and organising those relationships goes up.

Non-relational databases tend to be larger, as each document contains not only the information, but also markers to denote what that information is, in every single document. Even so, non-relational can be much faster for certain use cases. The key retrieval task is finding the document in the database. Once that document has been found, the system has everything it needs. A major drawback of non-relational databases is that by and large, they do not care about what is in the document. The developer needs to ensure that data going in is of the right type, and that any systems that rely on a non-relational can cope with changing documents.

Typically, this involves document validation which a relational database would give us for free.

Both have excellent use cases. Non-relational databases really lend themselves to the document-based web. Relational databases show their true worth in systems where many related concepts interact. Even so, relational databases have long been used to run web sites which do little more than show documents, just as non-relational databases have been employed for use cases where there is a lot of related data.

For the exercises today, we'll be using a relational database, and we will learn how to save details of users in an SQL database via our Model and NHibernate, an object relational mapper which is very good at turning .NET objects into saved database records.

Before we start, we've got some housekeeping to do.

## Exercise 1 - Housekeeping

---

In all previous exercises, we've been using the out-of-the-box dependency injection engine that ships with .NET Core 2.0's MVC engine. It has served us well so far, but we'll get far better value from Ninject, a DI framework which offers finer control over how we resolve dependencies.

### Create a new branch

Let's create a new git branch from master, called **feature/ninject**. We'll use this branch to modify the dependency injection, with a view to merging it back in when complete.

### Installing Ninject with nuget

In Visual Studio, click the **Tools** top menu option. From the dropdown list that appears, select **Nuget Package Manager**. When that menu expands, choose **Manage nuget packages for solution**. You should be presented with a new window. Inside the **Browse** tab on that, search for **Ninject**.

Select Ninject, and look at the list of projects in the right hand pane. Check the box next to **Empower.Mvc**, and click install.

The package manager knows that you're looking for .NET Core 2.0 packages. It'll automatically install a version of Ninject that is suitable for the version and type of .NET Framework we are using. The output window that emerges when you click **Install** shows the package manager system *targeting* libraries for our application.

**nuget** automatically resolves dependencies, so if a library you need requires something else to work, the package manager will download the dependency too.

### Plumbing

There's not going to be a whole lot of discussion on this topic, and quite a bit of copying and pasting. We're **plumbing**. It's something that programmers occasionally have to do, usually with the assistance of an article from the Internet, and we usually don't care too much about the implementation details, as long as it works. We need to replace the default dependency resolver with **Ninject**. We've just downloaded a package; we've yet to set it up.

### New directories and classes

We're going to create two new directories in the **Empower.Mvc** project, off the root. Let's do that now:-

- Create a directory called **Filters**
- Create a directory called **Infrastructure**

#### Infrastructure

We're going to need to add three classes here:-

##### Activators.cs

Add a new class called **Activators.cs**. Copy and paste the following code into the class, replacing all other text.

```

using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Controllers;
using Microsoft.AspNetCore.Mvc.ViewComponents;
using System;

namespace Empower.Mvc.Infrastructure
{
    public sealed class DelegatingControllerActivator : IControllerActivator
    {
        private readonly Func<ControllerContext, object> controllerCreator;
        private readonly Action<ControllerContext, object> controllerReleaser;

        public DelegatingControllerActivator(Func<ControllerContext, object>
controllerCreator,
        Action<ControllerContext, object> controllerReleaser = null)
        {
            this.controllerCreator = controllerCreator ??
                throw new ArgumentNullException(nameof(controllerCreator));
            this.controllerReleaser = controllerReleaser ?? ((_ , __) => { });
        }

        public object Create(ControllerContext context) =>
this.controllerCreator(context);
        public void Release(ControllerContext context, object controller) =>
            this.controllerReleaser(context, controller);

        public sealed class DelegatingViewComponentActivator : IViewComponentActivator
        {
            private readonly Func<Type, object> viewComponentCreator;
            private readonly Action<object> viewComponentReleaser;

            public DelegatingViewComponentActivator(Func<Type, object>
viewComponentCreator,
            Action<object> viewComponentReleaser = null)
            {
                this.viewComponentCreator = viewComponentCreator ??
                    throw new ArgumentNullException(nameof(viewComponentCreator));
                this.viewComponentReleaser = viewComponentReleaser ?? (_ => { });
            }

            public object Create(ViewComponentContext context) =>
this.viewComponentCreator(context.ViewComponentDescriptor.TypeInfo.AsType());

            public void Release(ViewComponentContext context, object viewComponent) =>
                this.viewComponentReleaser(viewComponent);
        }
    }
}

```

#### ApplicationBuilderExtensions.cs

Create a new class in **Infrastructure** called ApplicationBuilderExtensions.cs. Set the entire contents to this.

```
using System;
using System.Linq;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc.ApplicationParts;
using Microsoft.AspNetCore.Mvc.Controllers;
using Microsoft.Extensions.DependencyInjection;
using Ninject;

namespace Empower.Mvc.Infrastructure
{

    public static class ApplicationBuilderExtensions
    {
        public static void BindToMethod<T>(this IKernel config, Func<T> method)
        => config.Bind<T>().ToMethod(c => method());

        public static Type[] GetControllerTypes(this IApplicationBuilder builder)
        {
            var manager =
builder.ApplicationServices.GetRequiredService<ApplicationPartManager>();

            var feature = new ControllerFeature();
            manager.PopulateFeature(feature);

            return feature.Controllers.Select(t => t.AsType()).ToArray();
        }

        public static T GetRequestService<T>(this IApplicationBuilder builder) where T
: class
        {
            if (builder == null) throw new ArgumentNullException(nameof(builder));

            return GetRequestServiceProvider(builder).GetService<T>();
        }

        private static IServiceProvider GetRequestServiceProvider(IApplicationBuilder
builder)
        {
            var accessor =
builder.ApplicationServices.GetService<IHttpContextAccessor>();

            if (accessor == null)
            {
                throw new InvalidOperationException(
typeof(IHttpContextAccessor).FullName);
            }

            var context = accessor.HttpContext;

            if (context == null)
            {
                throw new InvalidOperationException("No HttpContext.");
            }
        }
    }
}
```

```

        return context.RequestServices;
    }
}

```

Finally, for **Infrastructure**, let's add **AspNetCoreExtensions.cs**, setting the content as follows.

```

using System;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.DependencyInjection;
using Empower.Mvc.Filters;
using Microsoft.AspNetCore.Mvc.Controllers;
using Microsoft.AspNetCore.Mvc.ViewComponents;
using static Empower.Mvc.Infrastructure.DelegatingControllerActivator;

namespace Empower.Mvc.Infrastructure
{
    public static class AspNetCoreExtensions
    {
        public static void AddRequestScopingMiddleware(this IServiceCollection
services,
            Func<IDisposable> requestScopeProvider)
        {
            if (services == null)
            {
                throw new ArgumentNullException(nameof(services));
            }

            if (requestScopeProvider == null)
            {
                throw new ArgumentNullException(nameof(requestScopeProvider));
            }

            services
                .AddSingleton<IStartupFilter>(new
                    RequestScopingStartupFilter(requestScopeProvider));
        }

        public static void AddCustomControllerActivation(this IServiceCollection
services,
            Func<Type, object> activator)
        {
            if (services == null) throw new ArgumentNullException(nameof(services));
            if (activator == null) throw new ArgumentNullException(nameof(activator));

            services.AddSingleton<IControllerActivator>(new
                DelegatingControllerActivator(
                    context =>
                    activator(context.ActionDescriptor.ControllerTypeInfo.AsType())));
        }

        public static void AddCustomViewComponentActivation(this IServiceCollection

```

```

services,
    Func<Type, object> activator)
{
    if (services == null) throw new ArgumentNullException(nameof(services));
    if (activator == null) throw new ArgumentNullException(nameof(activator));

    services.AddSingleton<IViewComponentActivator>(
        new DelegatingViewComponentActivator(activator));
}
}
}

```

## Filters

We've just one entry to add in **Filters**, and we'll have a wider discussion of the technology later, but for now, add a new class in the **Filters** subdirectory called **RequestScopingStartupFilter**

```

using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using System;

namespace Empower.Mvc.Filters
{
    public sealed class RequestScopingStartupFilter : IStartupFilter
    {
        private readonly Func<IDisposable> requestScopeProvider;

        public RequestScopingStartupFilter(Func<IDisposable> requestScopeProvider)
        {
            if (requestScopeProvider == null)
            {
                throw new ArgumentNullException(nameof(requestScopeProvider));
            }

            this.requestScopeProvider = requestScopeProvider;
        }

        public Action<IApplicationBuilder> Configure(Action<IApplicationBuilder>
nextFilter)
        {
            return builder =>
            {
                ConfigureRequestScoping(builder);

                nextFilter(builder);
            };
        }

        private void ConfigureRequestScoping(IApplicationBuilder builder)
        {
            builder.Use(async (context, next) =>
            {

```

```

        using (var scope = this.requestScopeProvider())
        {
            await next();
        }
    });
}
}
}

```

## Changes to Startup.cs

Our new version of Startup.cs is going to use all of the building blocks we've just introduced to switch out the default dependency injection implementation with **Ninject**

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using Empower.Services.Interfaces;
using Empower.Network.Services;
using Empower.Settings.Services;
using Empower.Adapters.Interfaces;
using Empower.Domain.Client.Command.Request;
using Empower.Domain.Client.Command.Response;
using Empower.Domain.Client.ViewModels;
using Empower.Adapters.ViewModels;
using NHibernate;
using Ninject;
using System.Threading;
using Ninject.Activation;
using Ninject.Infrastructure.Disposal;
using Microsoft.AspNetCore.Http;
using Empower.Mvc.Infrastructure;
using Microsoft.AspNetCore.Mvc.ViewFeatures.Internal;

namespace Empower.Mvc
{
    public class Startup
    {
        // New variables to assist with request scoping
        private readonly AsyncLocal<Scope> scopeProvider = new AsyncLocal<Scope>();
        private IKernel Kernel { get; set; }

        private object Resolve(Type type) => Kernel.Get(type);
        private object RequestScope(IContext context) => scopeProvider.Value;

        private sealed class Scope : DisposableObject { }

        public Startup(IConfiguration configuration)
        {

```



```

        Configuration = configuration;
    }

    public IConfiguration Configuration { get; }

    // This method gets called by the runtime. Use this method to add services to
    the container.
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddSingleton<IHttpContextAccessor, HttpContextAccessor>();
        services.AddRequestScopingMiddleware(() => scopeProvider.Value = new
Scope());
        services.AddCustomControllerActivation(Resolve);
        services.AddCustomViewComponentActivation(Resolve);

        services.AddMvc();
    }

    // This method gets called by the runtime. Use this method to configure the
    HTTP request pipeline.
    public void Configure(IApplicationBuilder app, IHostingEnvironment env)
    {
        // All DI is now setup in this link
        this.Kernel = this.RegisterApplicationComponents(app);

        if (env.IsDevelopment())
        {
            app.UseDeveloperExceptionPage();
            app.UseBrowserLink();
        }
        else
        {
            app.UseExceptionHandler("/Home/Error");
        }

        app.UseStaticFiles();

        app.UseMvc(routes =>
        {
            routes.MapRoute(
                name: "default",
                template: "{controller=Home}/{action=Index}/{id?}");
        });
    }

    private IKernel RegisterApplicationComponents(IApplicationBuilder app)
    {
        var kernel = new StandardKernel();

        // Register all controllers.
        foreach (var ctrlType in app.GetControllerTypes())
        {
            kernel.Bind(ctrlType).ToSelf().InScope(RequestScope);
        }

        ;

        kernel.Bind<IConfiguration>().ToMethod(m => this.Configuration)

```

```

        .InSingletonScope();
        kernel.Bind<ISettingsService>().To<SettingsService>()
            .InSingletonScope();
        kernel.Bind<IDatabaseSettingsService>().To<IDatabaseSettingsService>
            ().InSingletonScope();
        kernel.Bind<IEmailService>().To<EmailService>().InSingletonScope();
        kernel.Bind<IRequestResponseAdapter<SendEmailRequest, SendEmailResponse,
            ContactViewModel>>()
            .To<ContactAdapter>().InSingletonScope();

        // Cross-wire required framework services
        kernel.BindToMethod(app.GetRequestService<IViewBufferScope>);

        return kernel;
    }
}

```

## What have we just introduced?

In a nutshell, Ninject. We've now got a more performant and more configurable DI engine for use in our persistence work.

We've not talked about this exercise much, not least what each of the components are doing. Ordinarily, we probably wouldn't have taken this step at all, and would have found a nuget package capable of doing the same. Ninject has been slower than most to move over to .NET Core 2.0. While we do have compatible binaries thanks to .NET Standard, we've had to include a community solution to get ourselves up and running.

The only thing we'll be changing in the future is this method, in **Startup.cs**

```

private IKernel RegisterApplicationComponents(IApplicationBuilder app)
{
    var kernel = new StandardKernel();

    // Register all controllers.
    foreach (var ctrlType in app.GetControllerTypes())
    {
        kernel.Bind(ctrlType).ToSelf().InScope(RequestScope);
    }
;

    kernel.Bind<IConfiguration>().ToMethod(m => this.Configuration)
        .InSingletonScope();
    kernel.Bind<ISettingsService>().To<SettingsService>()
        .InSingletonScope();
    kernel.Bind<IDatabaseSettingsService>().To<IDatabaseSettingsService>
        ().InSingletonScope();
    kernel.Bind<IEmailService>().To<EmailService>().InSingletonScope();
    kernel.Bind<IRequestResponseAdapter<SendEmailRequest, SendEmailResponse,
        ContactViewModel>>()
        .To<ContactAdapter>().InSingletonScope();

    // Cross-wire required framework services
    kernel.BindToMethod(app.GetRequestService<IViewBufferScope>);
}

```

```
        return kernel;  
    }
```

Should we feel uncomfortable about introducing this code?

Inconvenienced, certainly. However, we reference external code that **we don't see** all the time. It would have been better to have gotten a pre-rolled solution that we did not have to plumb in ourselves.

## Re-run the app.

If all is good, let's commit our work and merge **feature/ninject** back into **master** using git. We're more than ready to start putting in some persistence code.

## Exercise 2 - SQL

---

We're going to spend today working on the sign up and retrieval of users. To do this, we're going to need a database to work.

### Creating a new database

Open up SQL Management Studio, and connect to **localhost**. Right click the databases icon in the left hand pane and select **New Database** from the pop-up menu.

Type **empowermvc** into the **Database Name** prompt and press return.

If you expand the **Databases** tab, your new **empowermvc** database should be present.

### Creating a service login

We're going to need a means of connecting to our database via code. The most usual way to do that is with an **SQL Server Service** account. This is an account with a username and password that we can use to access the database from our .NET code. Every access will be performed under the aegis of this account.

To create a new service account for our application, let's do the following:-

- Expand the **Security** tab.
- Right click **Logins**
- Choose **New Login**

A dialog should appear.

- Enter the login name as **empowermvc\_usr**
- Change the Authentication from **Windows** to **SQL Server Authentication**
- Uncheck
  - Enforce Password Policy
  - Enforce Password Expiration
  - User must change password at next login
- Enter !3mp0w3rMvc! as the password and the confirmation
  - Don't worry, we'll not have to remember it.
- Change the **Default Database** to **empowermvc**
- Click the **User Mapping** pane on the left hand side
  - Select **empowermvc** from the list of databases in the right hand pane
  - Check **db\_owner** in the bottom pane.

Login - New

Select a page

General

Server Roles

User Mapping

Securables

Status

Connection

Server:  
(local)  
Connection:  
DESKTOP-498ASKG\User  
[View connection properties](#)

Progress

Ready

Script Help

Login name:  

empowemvc\_usr

Search...

Windows authentication

SQL Server authentication

Password:  

.....

Confirm password:  

.....

Specify old password

Old password:

Enforce password policy

Enforce password expiration

User must change password at next login

Mapped to certificate

Mapped to asymmetric key

Map to Credential

Mapped Credentials

Credential	Provider
------------	----------

Remove

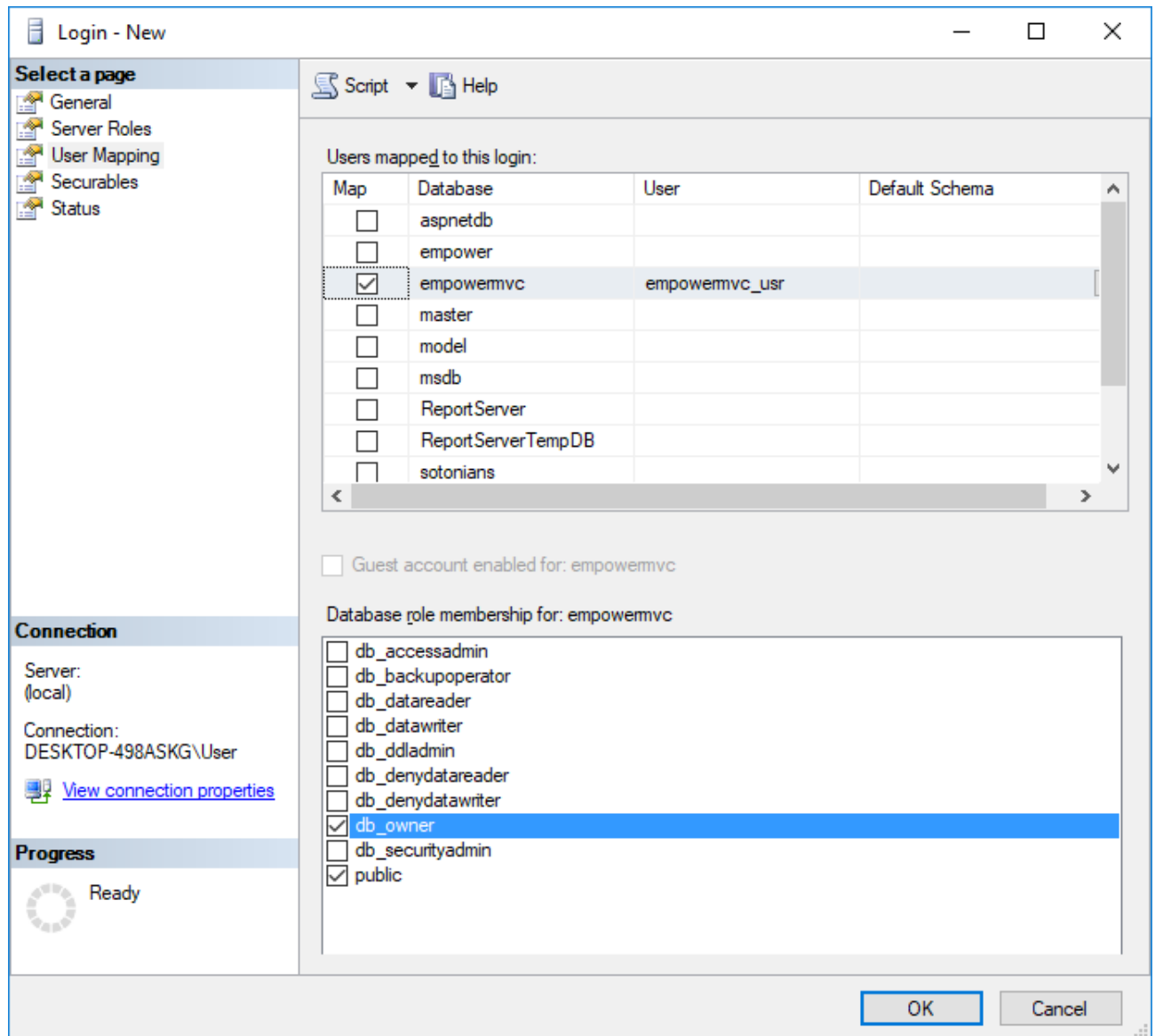
Default database:  
empowemvc

Default language:  
<default>

OK

Cancel

93 / 153



## Creating our tables

We've got two tables to create, Roles and Users.

### Roles

The Roles table will contain the types of role a user can have in our app. The SQL is as follows. Hit **New Query**, copy and paste the table details in and hit F5.

```
USE [empowemvc]
GO

/***** Object: Table [dbo].[Roles]    Script Date: 28/06/2018 10:31:10 *****/
SET ANSI_NULLS ON
GO

SET QUOTED_IDENTIFIER ON
GO

SET ANSI_PADDING ON
GO
```

```

CREATE TABLE [dbo].[Roles](
    [Id] [int] IDENTITY(1,1) NOT NULL,
    [Name] [varchar](50) NOT NULL,
    [Alias] [varchar](50) NOT NULL,
    [SortOrder] [int] NOT NULL,
    CONSTRAINT [PK_Roles] PRIMARY KEY CLUSTERED
(
    [Id] ASC
)WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY = OFF,
ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS = ON) ON [PRIMARY]
) ON [PRIMARY]

GO

SET ANSI_PADDING OFF
GO

CREATE UNIQUE NONCLUSTERED INDEX [IX_Unique_Alias] ON [dbo].[Roles]
(
    [Alias] ASC
)WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, SORT_IN_TEMPDB = OFF,
IGNORE_DUP_KEY = OFF, DROP_EXISTING = OFF, ONLINE = OFF, ALLOW_ROW_LOCKS = ON,
ALLOW_PAGE_LOCKS = ON)
GO

```

## Users

Users are going to contain details of our signed up users. Click **New Query** again, copying the following SQL in. Press F5 when done.

```

USE [empowermvc]
GO

/***** Object: Table [dbo].[Users]    Script Date: 28/06/2018 10:31:27 *****/
SET ANSI_NULLS ON
GO

SET QUOTED_IDENTIFIER ON
GO

SET ANSI_PADDING ON
GO

CREATE TABLE [dbo].[Users](
    [Id] [int] IDENTITY(1,1) NOT NULL,
    [RoleId] [int] NOT NULL,
    [Username] [varchar](50) NOT NULL,
    [Email] [varchar](300) NOT NULL,
    [JoinDate] [datetime] NOT NULL,
    [Password] [varchar](300) NOT NULL,
    [PasswordSalt] [varchar](300) NOT NULL,
    CONSTRAINT [PK_Users] PRIMARY KEY CLUSTERED
(

```

```
        [Id] ASC
    )WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY = OFF,
    ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS = ON) ON [PRIMARY]
    ) ON [PRIMARY]

GO

SET ANSI_PADDING OFF
GO

ALTER TABLE [dbo].[Users] WITH CHECK ADD CONSTRAINT [FK_Users_Roles] FOREIGN
KEY([RoleId])
REFERENCES [dbo].[Roles] ([Id])
GO

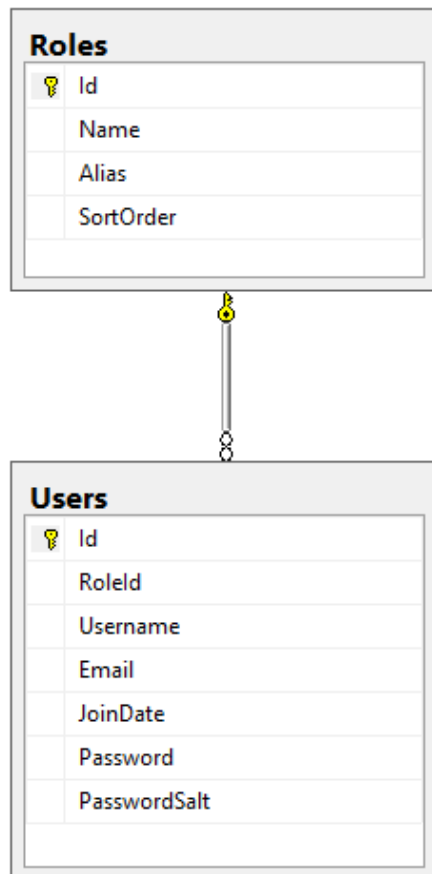
ALTER TABLE [dbo].[Users] CHECK CONSTRAINT [FK_Users_Roles]
GO
```

## Showing the relationship

Right click **Database diagrams** (click yes if you're asked to install database diagrams) and click **Add new diagram**. The list of tables in your newly created database are shown. Select both and press **Ok**

You should see the following diagram.





We've got a just about relational database. The **Users** table had a **RoleId**, which ties it to the **Roles** table. This is a *one to many* relationship. One *Role* can be used by many *Users*. A *User* can only ever have one *Role* in the present design. It's one of the simplest relational structures you are likely to encounter, but for our purposes, it is perfect.

## Creating some roles

Let's use the following SQL to establish four groups of users.

```
INSERT INTO
    [dbo].[Roles]
(
    [Name]
    ,[Alias]
    ,[SortOrder]
)
VALUES
    ('Admin',
     'admin',
     1),
    ('Sign up',
     'signup',
     2),
    ('Poster',
     'poster',
```

```
3),  
( 'Suspended',  
  'suspended',  
4);
```

- Admin
  - Someone that can administer the site.
- SignUp
  - Someone that has signed up on the site.
- Poster
  - Someone that is approved to post on the site.
- Suspended
  - Someone that was once a poster, and for whatever reason, is no longer able to post.

### A word on the Alias field.

Both of our tables have a primary key defined in **Id**. In the database, that is an **identity** column, automatically populating with an autoincrementing number. The **Id** column will uniquely identify any row in the database, and its autoincrementing behaviour means that the database can keep creating new, uniquely identifiable keys without human intervention.

While they are very useful in the operation of our database, it is considered very bad practice to enshrine these numbers into code, such as stored procedures or T-SQL. First of all, outside of the database, they have no meaning whatsoever. You cannot deduce the likely content of a database row with 17,373. Worse, those numbers can change, especially if the database is ever rebuilt from scratch, in a slightly different, world breaking order, as often happens when new environments are spun up.

The **Alias** field is employed here as an alternate primary key with some semantic meaning to the programmer, and a guarantee that even if the auto-generated *Id* of a row changes (say, someone deletes it and re-inserts using a new ID), the row can still be queried with a human-friendly, unique identifier.

Normally bereft of punctuation or capitalisation, **Alias** are short hand names that you can rely on, especially across environments, and use in code without inviting the ire of the coding community.

### A note on the use of **Id**

Conventions have differed on the naming of primary key columns (I have previously used different conventions, such as **TableNameId**; Sightsource have theirs), but going for **Id** as primary key name makes a lot of sense for our implementation. It is the default primary key name for NHibernate too, so by observing the convention in the database, we save ourselves from code further on. It's also the de facto standard in community articles; if you can't beat them, join them.

## What's just happened?

- We created a new SQL Server database called **empowermvc**
- We created a login for that database called **empowermvc\_usr**
  - We granted it access to **empowermvc**
- We used scripts to generate two tables
- We used Management Studio's Diagramming Capability to show the relationship between **Users** and **Roles**
  - A user has one role.
  - A Roles can define many users.

- We put some default values in the **Roles** table.
  - We talked about why having an alternate primary key might be a good idea.
- We are ready to start integrating our application with a relational database.

## Exercise 3 - Persistence with nHibernate.

---

Now that we've got our dependency injection and database houses sorted, we can begin to think about integrating our application with nHibernate, an Object Relational Mapper, a class of product that turns the content of programmatic content, such as classes, into database rows, and vice versa.

### A brief recap on nHibernate

nHibernate is a .NET port of a Java product, Hibernate. Its maturity, borrowed from the Java platform, is one of the reasons it is still such a popular ORM, despite pre-packaged alternatives such as Entity Framework being available. If there's something you need to do in nHibernate, there's normally a means to do it, and a vast resource .NET and Java resources to call on.

### New branch - feature/persistence

Let's create a new branch for the persistence work. Let's name it **feature/persistence** and check it out.

### New project - Empower.NHibernate

Within Visual Studio, create a new C# project and call it Empower.NHibernate. This class will hold the fundamentals of our nHibernate implementation.

### New project - Empower.NHibernate.Entities

We're immediately going to create another project called **Empower.NHibernate.Entities**.

This needs to reference **Empower.NHibernate**.

#### Entities

An **Entity** is a representation of a single database row, in code, represented as a class. They are the basic transport unit of an object relational mapper project. They literally are the objects that get mapped into database rows. In most respects, **Entities** look like any other C# class. Here is the **User** entity, which will represent one row in the **Users** database table.

```
public class User
{
    public virtual int Id {get; set;}
    public virtual Role Role { get; set; }
    public virtual string Username { get; set; }
    public virtual string Email { get; set; }
    public virtual DateTime JoinDate { get; set; }
    public virtual string Password { get; set; }
    public virtual string PasswordSalt { get; set; }
}
```

There's virtually no difference in that class to a standard C# class. You might expect it to be full of complex database code or helpers. The only difference is the **virtual** keyword, which allows other classes to change its definition and

behaviour. nHibernate turns these entities into Proxy Classes, which can speak to services beyond this simplistic definition. For us coders, the difficulty is all abstracted away.

## Mappings

nHibernate cannot handle **Entities** without a bit of help. In many implementations, an XML mapping file is employed, which does the job of specifying which code column should map to which database column. This is required because although nHibernate has its own conventions, it might be called on to map a system where conventions are different. The job of mapping is to wire up code properties to database fields.

### Fluent Mapping

In recent years, Fluent mapping has become a feature of many ORMs. Instead of specifying mappings in an XML file, which might get corrupted, we specify mappings in code. There is a project called FluentNHibernate that implements this pattern, which we can make use of.

We'll get to that, and mapping in general later, but for now, here is an example of a mapping class.

```
``// Inherits from ClassMap<T>
// We want to map a User here.
// Hence we inherit from ClassMap<User>
public class UserMap : ClassMap<User>
{
    public UserMap()
    {
        // Specify the table
        Table("Users");

        // By default, nHibernate assumes
        // property names are the same as
        // their database columns.
        //
        // If the column name is different,
        // you can use .Column("YourFunkyColumnName")
        Id(x => x.Id);
        Map(x => x.Username);
        Map(x => x.Email);
        Map(x => x.JoinDate);
        Map(x => x.Password);
        Map(x => x.PasswordSalt);

        // We're referencing another table here.
        // No need to dig further at this point,
        // just keep an eye on it.
        References(x => x.Role)
            .Column("RoleId");
    }
}
```

### Return to nuget

We're going to install **nuget** and reinforce something about it that we've spoken of before. Open the **Manage nuget packages for solution** from the Tools menu again.

Search for **FluentNHibernate** in the **Browse** tab.

Install it to **Empower.NHibernate**, **Empower.NHibernate.Entities** and **Empower.Mvc**.

Now **FluentNHibernate** depends on **NHibernate**. We haven't installed any reference to nHibernate yet, but **nuget** will grab it for us anyway, due to the way it is able to resolve dependent packages.

Why not put Entities in Empower.NHibernate?

We're going to be creating some base level stuff in our NHibernate that we may wish to use in other stuff. By separating our **Entities** into a different project, we can continue to reuse **Empower.NHibernate** as a core library, even if the database tables we use change over time.

Defining our entities

We're going to create our **Entity** classes in **Empower.NHibernate.Entities**. Right now, we need to model two different database tables, **Users** and **Roles**. To keep things easy, let's keep the C# property names the same as the Database column names.

**An Entity, in the abstract.**

nHibernate requires that every **Entity** have a **primary key** defined.

The strategy we've chosen in our database design is to use an auto-incrementing integer called **Id**. That'll serve us for about two billion rows, decent enough for most applications (for the record, DBAs tend to get twitchy at the one million mark).

We know that every one of our entity objects will have an integer property called **Id**. We can enshrine that in code.

We're going to create an **abstract class**. An abstract class never gets newed up directly, but other classes can inherit from them, and they can be newed up while benefiting from the abstract classes functionality.

Go to your **Empower.NHibernate** project and create a new subdirectory, **Base**. Once that's done, create a new class called **Entity**. Mark it with the **abstract** keyword.

Give it a **public** property called **Id**.

Your class should look like this:-

```
namespace Empower.NHibernate.Base
{
    public abstract class Entity
    {
        public virtual int Id { get; set; }
    }
}
```

That's good enough, but we could probably do better with a generic implementation.

```
namespace Empower.NHibernate.Base
{
    public abstract class Entity<T>
    {
        public virtual T Id { get; set; }
    }
}
```

This would cover us if we ever needed to change our **primary key** type, say from an **int** ( 2 billion ) to a **long**.

For now, we'll go with a simpler implementation, and if we need something more complex later, we can revisit the generic option, or in more complex cases, such as when multiple columns are used to uniquely identify a record (known as a **compound key**), we can develop something else.

We'll go with the original abstract class.

```
namespace Empower.NHibernate.Base
{
    public abstract class Entity
    {
        public virtual int Id { get; set; }
    }
}
```

### Defining Entities with the abstract.

Let's shift into our **Empower.NHibernate.Entities** project and add our **Entities**. We need to add two new classes, one to represent each of the database tables.

Add the following classes in **Empower.NHibernate.Entities** :-

- User
- Role

We'll fill out the details later, but for right now, that *User* represents a row in the *Users* table, while *Role* represents a row in the *Roles* table.

### Why the singular and plural?

It's a convention, which you don't have to follow, and many don't. Many would also call their database table **User** too, and nHibernate would look for that by default.

The rationale for employing this particular convention is that the **User** C# class represents *one user*, while the database tables will contain lots of users and or roles, hence the pluralization in the database.

It's a personal preference and as long as people can follow your work, few will complain.

### User

Our user entity is relatively straightforward to create. We just need to ensure that we have a placeholder for each database element we want to play around with in C#.

We need to define the names and types of everything we want going back and forth. We *could* give our **User** entity completely different **property names** to **database field names**, but let's keep things simple. By default, nHibernate's mapping component will look for **database field names** that match our **property names**. We have to tell it different for it to do otherwise. Let's keep things simple.

Here's our **User** entity class. With one omission, we've created a class full of property names that broadly match the types of their database equivalents.

```
public class User : Entity
{
    // Id is inherited from Entity.
    public virtual Role Role { get; set; }
    public virtual string Username { get; set; }
    public virtual string Email { get; set; }
    public virtual DateTime JoinDate { get; set; }
    public virtual string Password { get; set; }
    public virtual string PasswordSalt { get; set; }
}
```

Our **User** class *inherits* from the *abstract* **Entity** class we defined earlier in the exercise. We haven't defined a property called **Id** in this particular class. We get it for free by declaring that this **User** class is an entity.

#### The Role property

The **Role** property, of type **Role**, is interesting. It's a link into a child object which contains the details of that role. We can access the properties of the role by referencing the property on a user instance.

It'll make more sense once we implement the **Role** entity.

#### Role

The Role entity is the other side of the **User** relationship. Where the user had one role, roles have several **User** objects associated with them. Here is our role definition.

```
using Empower.NHibernate.Base;
using System.Collections.Generic;

namespace Empower.NHibernate.Entities
{
    public class Role : Entity
    {
        public virtual string Name { get; set; }
        public virtual string Alias { get; set; }
        public virtual int SortOrder { get; set; }
        public virtual IList<User> Users { get; set; }
    }
}
```



We're modelling the basic properties of **Role**. Again, because we're inheriting from **Entity**, we're getting the **Id** property for free. We don't have to redefine it here.

The interesting thing about **Role** is its **Users** property. That represents all of the users that are in the selected role. When we've properly mapped that, we'll be able to find all Users in a role, just by invoking **.Users** on an active **Role** instance.

## Adding mapping definitions.

Now that we've defined our entities, we need to tell NHibernate how to populate, and use, the things we declare as entities.

### New directory in Empower.NHibernate.Entities - Mappings

Create a new directory in **Empower.NHibernate.Entities** called **Mappings**. We're going to add two new classes to the **Mappings** subdirectory:-

- UserMap
  - Describes how a row in the **Users** table row becomes a **User** entity object.
- RoleMap
  - Describes how a row in the **Roles** table row becomes a **User** entity object.

### ClassMap

When we installed **FluentNHibernate** we gained access to the **ClassMap** abstract class. This is a helper class, provided by **FluentHibernate**, which knows a great deal about mapping .NET classes to database rows, but expects you to define them.

With nHibernate defaults, and our DB design, it's very trivial.

By default, the mapper assumes our property names are named after our database columns. That is precisely the way they've named them.

### UserMap.cs

First, we're going to define our code relationship with the **Users** table.

In the **Mappers** subdirectory, add a new class called **UserMap**. This class will inherit from **ClassMap**, where **T** will be **Entities.User**

The process of mapping takes place in the mapping class' constructor.

```
public class UserMap : ClassMap<User>
{
    public UserMap()
    {
        // The NHibernate will look for
        // a table matching the name of
        // the entity class it is mapping
        // by default;
        //
        // We've gone for something different
        // so will need to specify it with the Table()
    }
}
```

```

        // method
        //
        Table("Users");

        // The Id field must always be mapped.
        Id(x => x.Id);

        // If our database column names
        // were different from our C#
        // name, we'd need to specify what
        // they were. By default, Fluent will
        // assume that we've called them the same
        // thing.

        // If we have (and we have), we don't
        // need to specify the column name
        Map(x => x.Username);
        Map(x => x.Email);
        Map(x => x.JoinDate);
        Map(x => x.Password);
        Map(x => x.PasswordSalt);

        // This is our reference into
        // the user's Role. We've told the fluent
        // mapper that the user's Role can be looked
        // up by using the RoleId column on the
        // Users table to look it up.
        References(x => x.Role)
            .Column("RoleId");
    }
}

```

## RoleMap.cs

Next, create a new class called RoleMap.cs. This needs to inherit from **ClassMap**.

Once again, we'll be using the mapper's constructor to define mappings and relationships.

```

public class RoleMap : ClassMap<Role>
{
    public RoleMap()
    {
        // Set the table name again.
        Table("Roles");
        Id(x => x.Id);
        Map(x => x.Name);
        Map(x => x.Alias);
        Map(x => x.SortOrder);

        // Specifies that this role can
        // have many Users associated with
        // it.
        HasMany(x => x.Users)
            // Specifies that the Users can

```

```

        // be looked up using the
        // RoleId column in the Users
        // table.
        .KeyColumn("RoleId")
        // By default, NHibernate entities
        // are responsible for saving their child objects.
        //
        // We don't want that behaviour here.
        // We want Users to be responsible for saving
        // their own role associations.
        //
        // We use the Inverse() method to specify this.
        .Inverse();
    }
}

```

## Relationship linked

With our new mapping classes in place, we've described:-

- The means by which our **Entity** objects should be populated.
- The means by which our **Entity** objects are related
  - From a **User** entity, we can access a property called **Role**, which will give us a **Role** entity.
  - From the **Role** entity, we can access a property called **Users**, which will return a list of **User** entities associated with the role.

## ISession

Now that we've defined our relationships, it's time to begin hooking up the application to the database through NHibernate. The library provides an abstraction called **ISession** which grants access to an active NHibernate session. As coders, we can use **ISession** as a gateway to rich persistence functionality. We need to ensure our app can get access to it first.

## ISessionFactory

Deep down, **ISession** objects are generated with an **ISessionFactory**. This is a service that we'll configure with knowledge about our database, such as how to connect to it. Thereafter, we can ask our **ISessionFactory** to provide **ISession** objects, which will all be aware of the **empowermvc** database.

## Database configuration

For **ISessionFactory** to work, we need to provide it with details of our database and how we want to connect.

The key piece of information is our **connection string**. We've not included that in the application yet. Let's include it now.

Open up **appsettings.json**. Add the following entry to the file:-

```

"Database": {
  "ConnectionString": "Data Source=localhost;Initial Catalog=empowermvc;User
ID=empowermvc_usr;Password=^Emp0w3r^",

```

```
"SessionContext": "web"
}
```

The **json** appsettings format is new, and much more configurable than the configuration files before it. Programmers can pretty much define their own configuration structure.

Now that we've established our database settings in **appsettings.json**, we *could* just use an instance of **ISettingsService** to get at this information. We're going to be little more intentional and specific about it.

### IDatabaseSettingsService

We're going to create a small service in **Empower.Services** called **IDatabaseSettingsService**. It'll have two jobs, providing a **connection string** and providing a **thread context**. Both will be returned in **string** format. Both are readonly.

```
public interface IDatabaseSettingsService
{
    string ConnectionString { get; }
    string SessionContext { get; }
}
```

### DatabaseSettingsService concrete implementation

Add a new class in **Empower.Settings.Services** called **DatabaseSettingsService**, and *implement* **IDatabaseSettingsService**. Thanks to things we've already written, the implementation is going to be a *cinch*.

We've already got something that does most of the job. **ISettingsService** will pull values from appsettings.json, given a string key. We can inject that into our concrete implementation.

```
public class DatabaseSettingsService : IDatabaseSettingsService
{
    private readonly ISettingsService _settingsService;

    public DatabaseSettingsService(ISettingsService _settingsService)
    {
        _settingsService = settingsService;
    }
}
```

We've now got access to `_settingsService` within **DatabaseSettingsService** instances.

Completing our implementation is simple. We're simply passing through our request to `_settingsService`, which resolves our configuration query.

```
public class DatabaseSettingsService : IDatabaseSettingsService
{
    // Included because we know that
    // our db config section is inside a "Database"
    // tag.
```

```
//
// If we ever change that convention,
// all we need to do is change this to whatever
// the new key is.
private const string SettingPrefix = "Database";

private readonly ISettingsService _settingsService;

public DatabaseSettingsService(ISettingsService settingsService)
{
    _settingsService = settingsService;
}

// These properties, satisfying the demands
// of IDatabaseSettingsService, simply use
// another service we've written to provide
// their service.
public string ConnectionString => _settingsService.GetStringSetting($"{SettingPrefix}:ConnectionString");
public string SessionContext => _settingsService.GetStringSetting($"{SettingPrefix}:SessionContext");
}
```

## Wiring up the service in Ninject

We need to return to **Startup.cs** in **Empower.Mvc** to establish how requests for **IDatabaseSettingsServices** requests will be satisfied.

```
kernel.Bind<IDatabaseSettingsService>()
    .To<DatabaseSettingsService>()
    .InSingletonScope();
```

We only need one copy of the concrete instance for the entire app, so we've given the service singleton scope.

## Empower.NHibernate.Transactions

Now we've established a way for our database settings to be easily provided to an **ISessionFactory** instance, we need a place to house it.

## New project - Empower.NHibernate.Transactions

Create a new C# Class library called **Empower.NHibernate.Transactions**. It will need to reference:-

- **Empower.NHibernate**
- **Empower.NHibernate.Entities**
- **Empower.Services**

Return to the **nuget** package manager and install **FluentNHibernate** into **Empower.NHibernate.Transactions**

This class library is going to be responsible for spinning up our NHibernate implementation and providing a transaction framework.

## Transactions

SQL Server, like many other relational databases, has transactions at its heart. The main idea driving transactions is atomicity, another way of saying "it all happens, or none of it happens". Defined transactions are very useful for maintaining database integrity.

e.g. Not having an Order record with no Order lines, orphaned records, etc.

A transaction is only **committed** if it succeeds. It is **rolled back** if it does not, leaving no change to the database.

By default, SQL server runs in what is known as **auto commit** mode. Every statement is committed immediately if it is valid. We're going to introduce a class called **UnitOfWork** to control this process and grant us access to the database, but first, we need to describe what it does in C# with an **interface**

### New Interface in Empower.NHibernate

Let's create a new directory in **Empower.NHibernate** called **Interfaces**. Add a new interface in this directory called **IUnitOfWork**.

```
public interface IUnitOfWork
{
    // Start a transaction
    void BeginTransaction();
    // Commit a transaction
    void Commit();

    // Rollback a transaction
    void Rollback();

    // Provide access to the
    // NHibernate session
    ISession Session { get; }
}
```

Our concrete implementation will need to provide those methods.

### New class in Empower.NHibernate.Transactions

Create a new class called **UnitOfWork** in **Empower.NHibernate.Transactions**. It should implement **IUnitOfWork**.

```
public class UnitOfWork : IUnitOfWork
{
}
}
```

We need to know connection string details, so let's inject **IDatabaseSettingsService**.

```
private readonly IDatabaseSettingsService _databaseSettingService;
```

```
public UnitOfWork(IDatabaseSettingsService databaseSettingsService)
{
    _databaseSettingsService = databaseSettingsService;
}
```

We've now got access to the database settings.

We need to add some member variables for the class to do its job.

First, we need a **ISessionFactory** to create **ISession** objects, in order to satisfy the **Session** contractual demand of **IUnitOfWork**.

Next, we need an NHibernate **ITransaction** instance, to help us out with our **Commit**, **Rollback** and **BeginTransaction** demands.

Finally, we should probably provide the **property** demanded by **IUnitOfWork, Session**.

```
private static ISessionFactory _sessionFactory;
private ITransaction _transaction;
private readonly IDatabaseSettingsService _databaseSettingsService;
public ISession Session { get; private set; }
```

Note that the **ISessionFactory** is marked **static**. There will only ever be one copy of that in memory, no matter how many **UnitOfWork** instances we create. This is because the creation of an **ISessionFactory** is expensive. We don't have to do it every time, so we'll just do it once.

Let's add a new method to do that now. This constitutes the bridge from our app into the database. We are using Fluent Configuration.

```
private void InitialiseSessionFactory()
{
    // We only want one copy of this
    // Only create the session factory if we haven't already set it up.
    if (_sessionFactory == null)
    {
        var firstConfig =
            Fluently.Configure()
                .Database
                (
                    // Database type
                    MsSqlConfiguration.MsSql2008
                        .UseOuterJoin()
                        // Connection string coming from
                        // _databaseSettingsService
                        .ConnectionString(c =>
                            c.Is(_databaseSettingsService.ConnectionString)))
                .Mappings(m =>
                {
                    m.FluentMappings
                        .Conventions
                        .Setup(x =>
                        {
```

```

        x.Add(AutoImport.Never());
    })
    // This is where we tell
    // NHibernate to find our mappings
    // Fluent will scan and process all
    // of them.
    .AddFromAssemblyOf<UserMap>());

    }

};

// Now we've set the _sessionFactory,
// it won't happen again.
_sessionFactory =
    firstConfig
        .ExposeConfiguration(
            // Finally, we need to tell NHibernate how we're running.
            // Our SessionContext is "web"
            c => c.SetProperty(
                "current_session_context_class",
                _databaseSettingsService.SessionContext))
        .BuildSessionFactory();
}
}

```

With our configuration established, we should add a call to the routine in the constructor.

```

public UnitOfWork(IDatabaseSettingsService databaseSettingsService)
{
    _databaseSettingsService = databaseSettingsService;

    InitialiseSessionFactory();

    // _sessionFactory now able to create ISession objects
}

```

We know that after that call, we'll have a `_sessionFactory` variable we can use to create **ISession** objects. Let's make use of that by making good on our **Session** promise now.

```

public UnitOfWork(IDatabaseSettingsService databaseSettingsService)
{
    _databaseSettingsService = databaseSettingsService;

    InitialiseSessionFactory();

    // Connected session now available through the Session project.
    Session = _sessionFactory.OpenSession();
}

```

Now we're satisfying the condition for a connected session, our other methods become pretty easy to implement.



**BeginTransaction** will populate `_transaction` with an **ITransaction**, using the **Session** property to do it.

```
public void BeginTransaction()
{
    _transaction = Session.BeginTransaction();
}
```

**Commit** merely calls the **Commit** method on **ITransaction**, if a transaction exists and is ongoing.

```
public void Commit()
{
    try
    {
        // commit transaction if there is one active
        if (_transaction != null && _transaction.IsActive)
            _transaction.Commit();
    }
    catch
    {
        // rollback if there was an exception
        if (_transaction != null && _transaction.IsActive)
            _transaction.Rollback();

        throw;
    }
    finally
    {
        Session.Dispose();
    }
}
```

**Rollback** follows a similar path, passing through to **ITransaction**'s method of the same name.

```
public void Rollback()
{
    try
    {
        if (_transaction != null && _transaction.IsActive)
            _transaction.Rollback();
    }
    finally
    {
        Session.Dispose();
    }
}
```

We've completed all of the demands of the `IUnitOfWork` interface. It's now time to tell the dependency injection framework how to sort it all out.

## Dependency injection

We need to tell MVC that it can resolve requests for **IUnitOfWork** requests with the concrete instance of **UnitOfWork** we created.

```
kernel.Bind<IUnitOfWork>().To<UnitOfWork>().InScope(RequestScope);
```

Note that we've bound it in **RequestScope**. We should see one **UnitOfWork** request created per HTTP request. Anything that needs an **IUnitOfWork** instance can now receive one via constructor injection.

## NHibernateBase

Now that we can spin up **IUnitOfWork** instances, let's create an object that'll receive one. In your **Empower.NHibernate** project, create a new directory called **Base**. Add a class in that directory called **NHibernateBase.cs**.

Base classes are useful tools for code reuse. This NHibernateBase class will provide a framework other classes can build on. Most of our NHibernate services will use NHibernateBase in some way, either by directly inheriting or using a class that does.

```
public abstract class NHibernateBase
{
    // Stores a reference to the UnitOfWork
    // The protected keyword means that derived
    // classes can make use of this property too.
    protected readonly IUnitOfWork UnitOfWork;

    // A reference to the UnitOfWork's session
    protected virtual ISession Session => UnitOfWork.Session;

    // Inject the UnitOfWork object into this class
    protected NHibernateBase(IUnitOfWork unitOfWork)
    {
        // Set the UnitOfWork property so other things
        // can use it.
        UnitOfWork = unitOfWork;
    }

    #region Auto Transaction Wrapping

    protected virtual TResult Transact<TResult>(Func<TResult> func)
    {
        if (!Session.Transaction.IsActive)
        {
            // Wrap in transaction
            TResult result;
            using (var tx = Session.BeginTransaction())
            {
                result = func.Invoke();
                tx.Commit();
            }
            return result;
        }
    }
}
```

```

        // Don't wrap;
        return func.Invoke();
    }

    protected virtual void Transact(Action action)
    {
        Transact(() =>
        {
            action.Invoke();
            return false;
        });
    }

    #endregion
}

```

It's another example of an abstract class. We can never directly create an **NHibernateBase** object, only objects that inherit from it.

## Commit your work

That was a lot of plumbing, and we've not actually done anything with NHibernate yet.

Commit your work, and merge **feature/persistence** into **master**.

## What just happened?

- We created three new projects
  - Empower.NHibernate
    - Containing our most basic NHibernate plumbing, such as **IUnitOfWork**, **NHibernateBase** and base classes like **Entity**.
  - Empower.NHibernate.Entities
    - Contains our project specific entities and mapping.
    - Deliberately kept separate to allow us to reuse **Empower.NHibernate** on different databases.
  - Empower.NHibernate.Transactions
    - Contains details of how to spin up NHibernate transactions and sessions.
    - This functionality is contained within **IUnitOfWork**
- We created a new service, **IDatabaseSettings**
  - In the concrete implementation, we injected the functionality of **ISettingsService**, another service we created, to simplify our implementation.
- We wired up our new services with Ninject dependency injection.
- We created a base class for future NHibernate work, which receives an **IUnitOfWork** instance, making it available for derived classes.
- We have connected NHibernate to our app in a way that satisfies our service based approach. NHibernate is just a service that we can call on through the **IUnitOfWork** interface, now injectable into any class.

## Exercise 4 - Rather CRUD repositories.

---

When it comes down to it, there are four main operations you can perform with database rows.

- You can **create** them.
- You can **retrieve** them
- You can **update** them
- You can **delete** them

In this exercise, we are going to build a generic **repository** that will allow us to perform those operations on all of our entities.

Let's start by defining what we want it to do.

### New git branch

Let's create a new git branch called **feature/repository** and check it out.

### IRepository

Open your **Empower.NHibernate** project, and add a new interface called **IRepository**.

```
// T is a generic type that will be used
// with an entity.
public interface IRepository<T>
{

}
```

It's a generic interface, meaning we should be able to use it with any object. However, we want some guarantees of what the IRepository service will receive as a type parameter. Specifically, we want to know that it is *entity like* and that it has an integer **Id** property.

We can tighten that up with a new interface. In the **Interfaces** directory of **Empower.NHibernate** create a new interface called **IEntity**. It'll have the most basic outline, but it will help enormously in our repository implementation.

```
public interface IEntity
{
    int Id;
}
```

We'll need to revisit our **Entity** base class. It already fulfils **IEntity** behaviour, but we need it to declare that.

```
public abstract class Entity : IEntity
{
    public virtual int Id { get; set; }
}
```

We can now constrain the type passed to **IRepository**.

```
// T is a generic type that will be used
// with an entity.
public interface IRepository<T>
    where T : IEntity
{
}
```

Thanks to the constraint, we know that whatever gets passed into **IRepository** will always be an **IEntity**, and will always be able to furnish us with an **Id** value in **int** format.

Next, we need to define the things our repository will to.

While **Create** is the database convention for making a new row, the convention in C# when adding new things is **Add**. Let's respect the convention of the language we're in.

Similarly, **Get** is preferred in C# over **Retrieve**. With those conventional caveats established, here's what we want **IRepository** to be able to do.

### Add (Create)

Given an entity of type T, add it to the database, and return it to the client.

```
T Add(T entity);
```

### Count

Get a row count of the table the entity relates to.

```
// Get a count of all rows
int Count();
```

### Get (Retrieve single)

Given an id, load the entity associated with it and return it.

```
T Get(int id);
```

### Delete

Delete the entity passed in, returning a record count of how many rows are left.

```
int Delete(T entity);
```

## Update

Update the row passed in, returning the newly updated copy of the entity that represents it.

```
T Update(T entity);
```

## GetQueryOver

Return an `IQueryOver<T,T>` object that we can use to filter the table on.

```
IQueryOver<T,T> GetQueryOver();
```

## IRepository in full

Our completed interface should be decked out as follows.

```
public interface IRepository<T> where T : IEntity
{
    // Create a new row
    T Add(T entity);

    // Get a count of all rows
    int Count();

    // Get the specific DB record denoted by id
    T Get(int id);

    // Delete the entity passed in, returning a record count of how many rows are
    // left.
    int Delete(T entity);

    // Update the row passed in, returning the newly updated copy of the entity that
    // represents it
    T Update(T entity);

    IQueryOver<T,T> GetQueryOver();
}
```

## Implementing IRepository

Thanks to work in the previous exercise, there is a lot we can lean on when creating the concrete implementation of **IRepository**. Let's head to **Empower.NHibernate** and create a new directory, **Repository**. Once that is complete, create a new class and call it **NHibernateRepository**.

Inheritance, generics and implementation

We need to describe what `NHibernateRepository` is. Like the interface, it is generic. It takes a type parameter. We also want to use the functionality we defined in **NHibernateBase**. Finally, we'll need to declare that it can do the work of an `IRepository`.

Our initial definition will look something like the following, broken over several lines for readability and annotation

```
// generic class
public class NHibernateRepository<TEntity>
    // inherits from base class
    : NHibernateBase
    // says it can perform IRepository<T> functionality.
    , IRepository<TEntity>
    // constrains the repository to dealing with
    // types that derive from Entity.
    where TEntity : Entity
{
}
```

Next, we'll need to write a constructor which satisfies the needs of the **base class**, which needs an **IUnitOfWork** to work.

We'll use **constructor** injection to receive it here, and pass it up to **NHibernateBase**.

```
// Provide NHibernateBase with its requirement.
public NHibernateRepository(IUnitOfWork unitOfWork) : base(unitOfWork)
{
}
```

We've now got to satisfy the requirements of **IRepository**. With the stuff we have for free from the base class, it isn't that hard.

### Add (create)

We're using the **Session** property handed to us from the base class to save a new entity to the database and return it.

```
public TEntity Add(TEntity entity)
{
    Session.Save(entity);
    return entity;
}
```

### Count

The **Session** property is used to access a **QueryOver**, which gets a row count.

```
public int Count()
{
    return Session.QueryOver<TEntity>().RowCount();
}
```

### Get (Retrieve)

The **Session** property is used to access a **QueryOver**, which uses a where clause to load the row with the specified **id**.

```
public TEntity Get(int id)
{
    return Session.QueryOver<TEntity>().Where(c => c.Id == id).SingleOrDefault();
}
```

### Update

The **Session** property's SaveOrUpdate method is invoked.

```
public TEntity Update(TEntity entity)
{
    Session.SaveOrUpdate(entity);
    return entity;
}
```

### Delete

The **Session** property's Delete method is invoked.

```
public int Delete(TEntity entity)
{
    Session.Delete(entity);
    return Count();
}
```

### GetQueryOver (filter)

The **Session** property's GetQueryOver method is invoked and returned.

```
public IQueryOver<TEntity, TEntity> GetQueryOver()
{
    return Session.QueryOver<TEntity>();
}
```



## Complete repository

The completed generic repository will look like this, and without too much extra effort, will now perform repository operations for any entity.

```
public class NHibernateRepository<TEntity> : NHibernateBase, IRepository<TEntity>
where TEntity : Entity
{
    public NHibernateRepository(IUnitOfWork unitOfWork) : base(unitOfWork)
    {
    }

    public TEntity Add(TEntity entity)
    {
        Session.Save(entity);
        return entity;
    }

    public int Count()
    {
        return Session.QueryOver<TEntity>().RowCount();
    }

    public int Delete(TEntity entity)
    {
        Session.Delete(entity);
        return Count();
    }

    public TEntity Get(int id)
    {
        return Session.QueryOver<TEntity>().Where(c => c.Id == id).SingleOrDefault();
    }

    public TEntity Update(TEntity entity)
    {
        Session.SaveOrUpdate(entity);
        return entity;
    }

    public IQueryOver<TEntity, TEntity> GetQueryOver()
    {
        return Session.QueryOver<TEntity>();
    }
}
```

## Informing dependency injection of the change

We need to tell MVC how to resolve requests for an `IRepository<>`, something Ninject handles more gracefully than the baked in implementation. The syntax is slightly different because we're declaring a generic repository.

Let's return to **Startup.cs** in **Empower.Mvc** and add a new binding to `ConfigureApplicationComponents`.

```
kernel.Bind(typeof IRepository<>))
    .To(typeof NHibernateRepository<>))
    .InScope(RequestScope)
```

Thanks to this line, we can inject an **IRepository** into any service that needs it.

## Commit your work

Commit your work and merge **feature/repository** back into **master**.

## What just happened?

- We defined the major operations that can be performed at database row level.
  - Create
  - Retrieve
  - Update
  - Delete
- We defined what we wanted our repository to do, by creating **IRepository**, which will work with any type claiming to be an **IEntity**.
- We created a concrete implementation which inherits from **NHibernateBase**, the class we created at the end of the last exercise to make life easier. It did. We were able to create simple passthroughs which used the **IUnitOfWork** interface to gain access to the NHibernate **ISession** interface.
- We wired up the dependency in Ninject
  - Other classes can demand a copy of **IRepository** through **constructor injection** and expect to be served.
- The low level building NHibernate blocks are for now, complete.

## Exercise 5 - Empower.NHibernate.Services

---

With our app connected and able to service demands for repositories, we're now in a position where we can use that plumbing and those tools to provide complex services specific to our application.

### New git branch

Create a new git branch called **feature/nhservices** and check it out.

### New service in Empower.Services

The definition for our first NHibernate service will live in the **Empower.Services** project, outside the **NHibernate** space. This is by design. We may switch to a different ORM in the future, and may want to provide a different implementation of **IUserService**. As long as the new implementation satisfies the requirements, our app will keep trucking along.

Add the **IUserService** interface to **Empower.Services**

For right now, let's define two simple requirements of our **IUserService**. It needs to be able to give us a count of all users. It needs to be able to give us a count of all roles. We'll add to its capability later.

```
public interface IUserService
{
    int UserCount { get; set; }
    int RoleCount { get; set;}
}
```

### New project - Empower.NHibernate.Services

In Visual Studio, create a new C# library called **Empower.NHibernate.Services**. This is going to house all of our app-specific services that relate to NHibernate operations.

It should reference:-

- Empower.NHibernate
- Empower.NHibernate.Entities
- Empower.Services

The following projects should reference it:-

- Empower.Mvc

### Concrete implementation of IUserService

Create a new class in **Empower.NHibernate.Services** which implements **IUserService**. The initial definition should be as follows:-

```
public class UserService : IUserService
{
```

```
}
```

VS will gripe because we've said we can do **UserCount** and **RoleCount** but have provided no means of doing so. Our project won't compile right now.

## Injecting IRepository

We can make life much easier for ourselves if we use the **IRepository** service we created earlier. It has a **Count()** method our service can exploit.

We'll need to inject two **IRepository** instances, one for the **User** entity, and another to handle the **Role** entity.

```
public class UserService : IUserService
{
    private readonly IRepository<User> _userRepo;
    private readonly IRepository<Role> roleRepo;

    public UserService(
        IRepository<User> userRepo,
        IRepository<Role> roleRepo
    )
    {
        _userRepo = userRepo;
        _roleRepo = roleRepo;
    }
}
```

## Making good on the count

We still need to keep our implementation promises, but thanks to the injected repositories, that becomes trivial. Simple pass throughs will suffice.

```
public class UserService : IUserService
{
    private readonly IRepository<User> _userRepo;
    private readonly IRepository<Role> roleRepo;

    public UserService(
        IRepository<User> userRepo,
        IRepository<Role> roleRepo
    )
    {
        _userRepo = userRepo;
        _roleRepo = roleRepo;
    }

    public int UserCount => _userRepo.Count();
    public int RoleCount => _roleRepo.Count();
}
```

The demands of the **IUserService** interface have been met. Let's tell dependency injection all about it.

## Dependency injection

We'll need to head back to the **Empower.Mvc** project to make further changes to **Startup.cs**. Add the following binding to **ConfigureApplicationComponents**

```
kernel.Bind<IUserService>().To<UserService>()  
    .InScope(RequestScope);
```

## UserController

Let's stay in the **Empower.Mvc** project for now, and add a new **Controller** to **Controllers** called **UserController**.

Inject a copy of **IViewModelService** into the controller. We've made a choice to route all MVC service requests through this service. We'll continue that there.

```
private readonly IViewModelService _viewModelService;  
  
public UserController(IViewModelService viewModelService)  
{  
    _viewModelService = viewModelService;  
}
```

## Changes to IViewModelService

Let's define two new things we want **IViewModelService** to do.

```
public interface IViewModelService  
{  
    ContactViewModel Contact(ContactViewModel model);  
  
    // New stuff here.  
    int GetSiteUserCount();  
    int GetSiteRoleCount();  
}
```

## Supporting the new demands in ViewModelService.

Meeting the new demands of **IViewModelService** can be met easily if we inject an instance of **IUserService** into it.

```
private readonly IUserService _userService;  
private readonly IEmailService _emailService;  
  
public ViewModelService(IUserService userService, IEmailService emailService)  
{  
    _userService = userService;
```

```
_emailService = emailService;
}
```

We can now use **\_userService** properties here to meet our new contractual demands. Again, simple passthroughs will suffice.

```
public int UserCount => _userService.UserCount;
public int RoleCount => _userService.RoleCount;
```

## Testing our service

The big moment. We're going to test our entire dependency chain into the database.

We will call:-

- UserController
  - *which relies on **IViewModelService***
    - *which relies on **IUserService***
      - *which relies on **IRepository** and **IRepository***
        - *implemented by **NHibernateRepository***
          - *which is derived from **NHibernateBase***
            - *which depends on **IUnitOfWork***
              - *implemented by **UnitOfWork***
                - *which depends on\_ **IDatabaseSettings***
                  - *\_implemented by **DatabaseSettings***
                  - *which depends on\_ **ISettingsService***
                    - *implemented by **SettingsService***
                      - *which depends on\_ **IConfiguration***

**Whoa!**

Modify the Index of the UserController

We're going to keep things simple for now.

We're going to set a couple of **ViewData[""]** flags based on the things we get back from our service calls.

```
public IActionResult Index()
{
    ViewData["UserCount"] = $"{_viewModelService.GetSiteUserCount}";
    ViewData["RoleCount"] = $"{_viewModelService.GetRoleUserCount}";
}
```

Let's add a view that exposes the values derived through our service.

Right click the **Index** method and click **Add View**. Choose an empty view. When complete, it'll be present in **~Views/User**. Let's add the following initial markup.

```

<div class="row">
  <div class="col-md-6">
    <div class="well">
      <h1>Users <span class="badge">@ViewData["UserCount"]</span></h1>
      <p>There are <strong>@ViewData["UserCount"] users</strong> signed up</p>
    </div>
  </div>
  <div class="col-md-6">
    <div class="well">
      <h1>Roles <span class="badge">@ViewData["RoleCount"]</span></h1>
      <p>There are <strong>@ViewData["RoleCount"] roles</strong> in the system.
    </div>
  </div>
</div>

```

## Link users on the navigation

Let's return to **\_Layout.cshtml** and create a navigation link to our Users controller action.

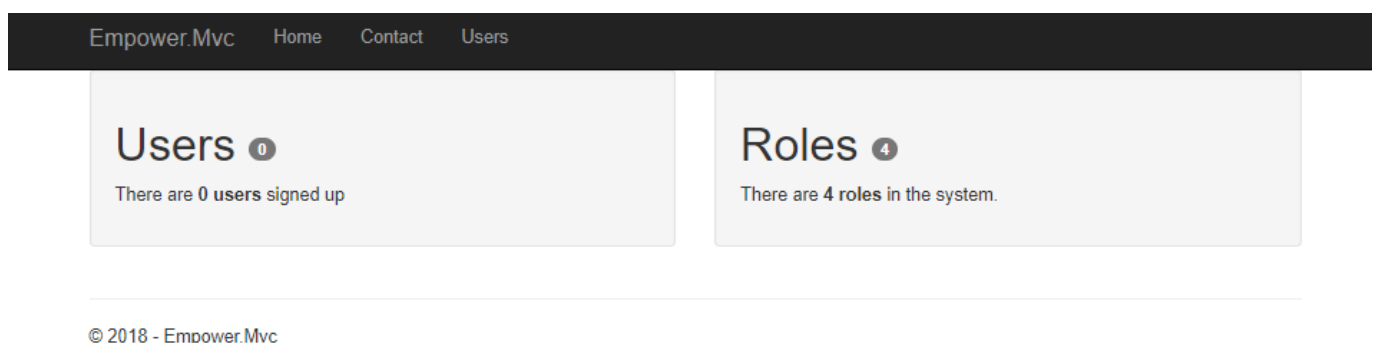
```

<ul class="nav navbar-nav">
  <li><a asp-area="" asp-controller="Home" asp-action="Index">Home</a></li>
  <li><a asp-area="" asp-controller="Home" asp-action="Contact">Contact</a></li>
  <li><a asp-area="" asp-controller="User" asp-action="Index">Users</a></li>
</ul>

```

## Run the app

Fire up Visual Studio and run the application. If all goes well, you should see a new link called Users, and on navigating there, should see a screenshot resembling the attached.



## Commit your work

Let's commit and merge **feature/nhservices** back into our master **git** branch.

## What just happened?

- We created a new project called **Empower.NHibernate.Services**, designed to run complex app-specific services that involve nHibernate.
- We defined a new interface called **IUserService**, which present supports two properties
- We create a **concrete implementation** of **IUserService** in **Empower.NHibernate.Services**.
  - This relied on the Repository work we did earlier, and was easier to implement as a result.
  - All we had to do was pass through commands to another service.
- We wired up **IUserService**, letting our DI container know what to provide when asked.
- We injected **IUserService** into ViewModelService.
- We defined two new properties on **IViewModelService**, **GetSiteUserCount** and **GetSiteRoleCount**
- We implemented those properties in **ViewModelService**, using our injected **IUserService** to satisfy them.
- We created a new Controller called **UserController**
  - Injected **IViewModelServices** into it.
  - Defined the **Index** method, setting some ViewData flags based on the outputs of **IViewModelService**
- We created a new View which shows our respective user and role counts.



## Exercise 6 - Sign up

---

While it's very nice to have some confirmation that our database interaction is working, that `UserCount` is never going to get past zero unless we provide a way to add more.

### Requirements

We need a sign up facility.

This specification for this facility is as follows:-

#### ! Requirement

- Inputs
  - User entered:-
    - Username.
    - Email address,
    - Password
    - Confirm password.
- Validate each of the fields using existing validation attributes
- Create custom validation that will dupe check usernames and email addresses
- Encrypt the password using a *password hash*
- Generate a **JoinDate** value
- When all of the data is present and valid, save a new user record in the database

### New git branch - feature/signup

Let's create a new branch to enable us to work on our signup feature independently. Create the **feature/signup** and check it out.

### New ViewModel - SignUpViewModel.cs

The first step in meeting our requirement is creating a **ViewModel**.

Once again, everything we're asked to expect from the user is a string. We have to auto-generate a `JoinDate` at some point, but it's not necessary for the **ViewModel** to convey. We can handle that elsewhere.

Let's go back to our **Empower.Domain** project, hit the `Client/ViewModels` directory and add a new one, **SignUpViewModel.cs**.

```
public class SignUpViewModel : IViewModel
{
    public string Username { get; set; }
    public string Email { get; set; }
    public string Password { get; set; }

    public string ConfirmPassword { get; set; }
}
```

Once again, we'll use the **attributes** from **DataAnnotations** to specify what sort of validation we're after.

```

public class SignUpViewModel : IViewModel
{
    [Required(ErrorMessage = "Please enter your username")]
    [RegularExpression("^[A-Za-z0-9 _-]+$", ErrorMessage = "Please enter valid
characters only")]
    [MinLength(3, ErrorMessage = "Your username should be at least three characters in
length")]
    [MaxLength(50, ErrorMessage = "Your username should be at most fifty characters in
length")]
    public string Username { get; set; }

    [DisplayName("Email address")]
    [Required(ErrorMessage = "Please enter your email")]
    [EmailAddress(ErrorMessage = "Please enter a valid email")]
    public string Email { get; set; }

    [PasswordPropertyText]
    [Required(ErrorMessage = "Please enter your password")]
    [MinLength(8, ErrorMessage = "Your username should be at least eight characters in
length")]
    [MaxLength(300, ErrorMessage = "Your username should be at most 300 characters in
length")]
    public string Password { get; set; }

    [PasswordPropertyText]
    [DisplayName("Confirm your password")]
    [Required(ErrorMessage = "Please confirm your password")]
    [Compare("Password", ErrorMessage = "Password and confirm password must match")]

    public string ConfirmPassword { get; set; }
}

```

:question\_mark: **Compare** We haven't used the Compare attribute before. It allows us to specify another property name. This property and the referenced property must be equal.



We got a sensible way to use **MaxLength** and **MinLength**. We want both our username and password field to have some complexity. A minimum length constraint is a good way to do that. We also have storage constraints in our database; there is no point allowing the user to enter a username that is longer than can be stored. Such a user would never be able to log in.

Our **SignUpViewModel** is in good shape.

## Creating the Controller actions

Go to your **Empower.Mvc** project and add two new actions to **UserController**. Now we're up to speed, you'll know we need a **GET action** that'll have the job of showing the SignUp form. We'll stub these out for now.

```

[HttpGet]
public IActionResult SignUp()
{
    return View(new SignUpViewModel());
}

```

```
[HttpPost]
public IActionResult SignUp(SignUpViewModel viewModel)
{
    return View(viewModel);
}
```

## Creating the View

Right click the SignUp **GET** action and choose **Add View**. Create a **Create** view with **SignUpViewModel** set as the type. The scaffolding will generate a form, aware that we're validating and already close to suitable for our needs.

```
<h2>SignUp</h2>

<h4>SignUpViewModel</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form asp-action="Plop">
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>
            <div class="form-group">
                <label asp-for="Username" class="control-label"></label>
                <input asp-for="Username" class="form-control" />
                <span asp-validation-for="Username" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Email" class="control-label"></label>
                <input asp-for="Email" class="form-control" />
                <span asp-validation-for="Email" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Password" class="control-label"></label>
                <input asp-for="Password" class="form-control" />
                <span asp-validation-for="Password" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="ConfirmPassword" class="control-label"></label>
                <input asp-for="ConfirmPassword" class="form-control" />
                <span asp-validation-for="ConfirmPassword" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="StatusMessage" class="control-label"></label>
                <input asp-for="StatusMessage" class="form-control" />
                <span asp-validation-for="StatusMessage" class="text-danger"></span>
            </div>
            <div class="form-group">
                <div class="checkbox">
                    <label>
                        <input asp-for="HasErrored" /> @Html.DisplayNameFor(model =>
model.HasErrored)
                    </label>
                </div>
            </div>
        </form>
    </div>
</div>
```

```

        <div class="form-group">
            <input type="submit" value="Create" class="btn btn-default" />
        </div>
    </form>
</div>
</div>

```

We don't need the **StatusMessage** as a form element. We don't need **HasErrored** to be a checkbox. Let's remove both of those.

Our new form should have the following markup

```

<div class="row">
    <div class="col-md-4">
        <form asp-action="SignUp" method="post">
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>
            <div class="form-group">
                <label asp-for="Username" class="control-label"></label>
                <input asp-for="Username" class="form-control" />
                <span asp-validation-for="Username" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Email" class="control-label"></label>
                <input asp-for="Email" class="form-control" />
                <span asp-validation-for="Email" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Password" class="control-label"></label>
                <input asp-for="Password" class="form-control" type="password"/>
                <span asp-validation-for="Password" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="ConfirmPassword" class="control-label"></label>
                <input asp-for="ConfirmPassword" class="form-control" type="password"
            />
                <span asp-validation-for="ConfirmPassword" class="text-danger"></span>
            </div>
            <div class="form-group">
                <input type="submit" value="Create" class="btn btn-default" />
            </div>
        </form>
    </div>
</div>

```

## Link the controller action to the navigation

Time to dive into **Views/Shared** and hit the **\_Layout.cshtml** file. We need to create a navigation link for our **SignUp** action.

```

<div class="navbar-collapse collapse">
    <ul class="nav navbar-nav">
        <li><a asp-area="" asp-controller="Home" asp-action="Index">Home</a></li>

```

```

<li><a asp-area="" asp-controller="Home" asp-action="Contact">Contact</a></li>
<li><a asp-area="" asp-controller="User" asp-action="Index">Users</a></li>
<li><a asp-area="" asp-controller="User" asp-action="SignUp">Sign up!</a></li>
</ul>
</div>

```

## Run the app

You should be able to start your app, hit the **sign up** link, and see your form. We've now got a UI component capable of conveying our **sign up** input parameters.

It doesn't do much right now.

## Commit your work

Commit your changes so far, but don't merge back into master just yet. We have remaining work on **feature/signup**.

## A very necessary diversion - IPasswordHashing service

We're about to create a service that will store user credentials, including a password. We cannot store those in *cleartext*



**cleartext** is a term that describes things being stored without any kind of encryption or obsfucation. While this book has *frequently* taken us down bad development paths, if only to show how they can be corrected, we're not going to do that here. It is **very bad** security practice to store passwords in **cleartext**. Most major password breaches that you head about in the news occur *precisely* because of this practice. We're encrypting from the get-go.

We need to encrypt, immediately. To do that, we're going to use a **one-way hash**. We need to create a **password hashing service** to control that for us.



**One way hashing** is a technique which puts *cleartext* user input through an encryption routine to produce a **password hash**. This is the only representation of the password that is stored. The user's original *cleartext* password never makes the database. When the user logs in, we run the encryption algorithm again. If their user input matches their original cleartext password, the encryption routine will generate the same hash as exists in the database. The **One way** specifier simply means that someone cannot work their way back from the hash to the cleartext password.

## New command classes - EncryptedPasswordRequest and EncryptedPasswordResponse

Switch into **Empower.Domain**. We're about to create two new command objects that'll help describe our service.

Open **Client > Command** in your project.

Right click the **Request** directory and add a class called **EncryptedPasswordRequest**

Then right click **Response**, and add a class called **EncryptedPasswordResponse**.

Let's get them defined immediately, they're not complex.

### EncryptedPasswordRequest

**EncryptedPasswordRequest** will contain two properties.

- Password
  - *clear text* password
- PasswordSalt
  - Additional encryption factor
  - Autogenerated by **IPasswordHashing** implementation if not set



A **password salt** is an extra piece of information which is used in conjunction with the *cleartext* password to calculate the one way hash. Once the **password** has been stored as a **one way** hash, we'll always need the **salt** and the correct *cleartext* password to generate the same hashed password. The **salt** will be stored in the database row alongside the **encrypted password**

```
namespace Empower.Domain.Client.Command.Request
{
    public class EncryptedPasswordRequest : IRequest
    {
        public string Password { get; set; }

        // Not necessary for first time creations
        public string PasswordSalt { get; set; }
    }
}
```

## EncryptedPasswordResponse

**EncryptedPasswordResponse** is designed to hold the result of the password encryption.

The **EncryptedPassword** field will house the result of the encryption attempt. **PasswordSalt** will contain the salt used in the hashing algorithm. We can use the property to persist the value to the database.

```
namespace Empower.Domain.Client.Command.Response
{
    public class EncryptedPasswordResponse : IResponse
    {
        public string EncryptedPassword { get; set; }
        public string PasswordSalt { get; set; }
        public DateTime? CompletedAt { get; set; }
    }
}
```

Open up **Empower.Services** and create a new *interface* called **IPasswordHashingService**. We can leave the definition blank for now

```
public interface IPasswordHashingService
{
    EncryptedPasswordResponse EncryptPassword(EncryptedPasswordRequest request);
}
```

We've defined what we want our service to do. Now we just need to express how it will do it.

## New project - Empower.Security

In Visual Studio, create a new project called **Empower.Security.Services**

When the project has been created, create a class called **PasswordHashingService**, and get it to implement **IPasswordHashingService**.

Let's get the following utility functions in place.

- **CreateSalt** creates a salt value matching a specified **size**
- **GenerateSaltedHash** creates a hashed password using cleartext and a salt.

```
using Empower.Services.Interfaces;
using System;
using System.Collections.Generic;
using System.Text;
using Empower.Domain.Client.Command.Request;
using Empower.Domain.Client.Command.Response;
using System.Security.Cryptography;

namespace Empower.Security.Services
{
    public class PasswordHashingService : IPasswordHashingService
    {
        /// <summary>
        /// Create a salt for the password hash (just makes it a bit more complex)
        /// </summary>
        /// <param name="size"></param>
        /// <returns></returns>
        private static string CreateSalt(int size)
        {
            // Generate a cryptographic random number.
            var rng = new RNGCryptoServiceProvider();
            var buff = new byte[size];
            rng.GetBytes(buff);

            // Return a Base64 string representation of the random number.
            return Convert.ToBase64String(buff);
        }

        /// <summary>
        /// Generate a hash for a password, adding a salt value
        /// </summary>
        /// <param name="plainText"></param>
        /// <param name="salt"></param>
        /// <returns></returns>
        private static string GenerateSaltedHash(string plainText, string salt)
        {
            // http://stackoverflow.com/questions/2138429/hash-and-salt-passwords-in-
            c-sharp

            var plainTextBytes = Encoding.UTF8.GetBytes(plainText);
```

```

        var saltBytes = Encoding.UTF8.GetBytes(salt);

        // Combine the two lists
        var plainTextWithSaltBytes = new List<byte>(plainTextBytes.Length +
saltBytes.Length);
        plainTextWithSaltBytes.AddRange(plainTextBytes);
        plainTextWithSaltBytes.AddRange(saltBytes);

        // Produce 256-bit hashed value i.e. 32 bytes
        HashAlgorithm algorithm = new SHA256Managed();
        var byteHash = algorithm.ComputeHash(plainTextWithSaltBytes.ToArray());
        return Convert.ToBase64String(byteHash);
    }
}

```

Now we've got those utility functions in place, we can satisfy the demands of **IPasswordHashingService**.

```

public const int SALT_SIZE = 24;

public EncryptedPasswordResponse EncryptPassword(
    EncryptedPasswordRequest request
)
{
    // Either keep the existing salt or
    // use GenerateSalt command to generate a new one
    var passwordSalt = request.PasswordSalt ?? CreateSalt(SALT_SIZE);

    // Use GenerateSaltedHash to generate
    // a hashed password
    var passwordHash = GenerateSaltedHash(request.Password, passwordSalt);

    // Return the results.
    return new EncryptedPasswordResponse
    {
        EncryptedPassword = passwordHash,
        PasswordSalt = passwordSalt,
        CompletedAt = DateTime.UtcNow
    };
}

```

## Wiring up dependency injection

As always, we need to let our dependency injection container know how it can resolve a request for an **IPasswordHashingService**. Let's return to **Empower.Mvc** and add the binding to **Startup.cs**

```

kernel.Bind<IPasswordHashingService>().To<PasswordHashingService>()
    .InSingletonScope();

```

One copy will do for the entire app. We've therefore declared its scope as **Singleton**.



We can now inject this **password hashing service** into anything that needs it.



We'll never store a cleartext password! Not once!

## New commands for the sign up process.

We're going to create a couple more **Command** objects, following our Response and Request adapter, separate from the ViewModel, that UserService will employ to perform its work.

### SignUpRequest.cs

Open the **Empower.Domain** library, and add a new class in Client/Commands/Request called **SignUpRequest.cs**

```
public class SignUpRequest : IRequest
{
    public string Email { get; set; }
    public string Username { get; set; }
    public string Password { get; set; }
    public string ConfirmPassword { get; set; }
}
```

It'll contain all we need to sign up a new user. We'll also say it is an **IRequest**, so we can use it in our adapters.

### SignUpResponse.cs

Let's head into Response territory. Create a new class called **SignUpResponse.cs** in the Client/Command/Responses directory.

This will contain the result of our request.

```
namespace Empower.Domain.Client.Command.Response
{
    public class SignUpResponse : IResponse
    {
        public DateTime? CompletedAt { get; set; }
        public string ErrorMessage {get; set;}
        public bool HasErrored => !string.IsNullOrEmpty(ErrorMessage);
    }
}
```

## Refactoring in inheritance

We've only implemented two **IResponse** commands. Copying and pasting the ErrorMessage and HasErrored properties to meet the contractual demands of **IResponse** is not only getting old, it's also bad practice. We're violating the DRY principle when we don't need to.



**DRY** = Do not repeat yourself. Another **TLA**.

We *know* that every IResponse needs to implement **ErrorMessage**, **HasErrored** and **CompletedAt**, but do we have to do it every time?

We can use inheritance to simplify our classes.

Let's handle **IRequest** first. This interface doesn't have any contractual requirements. We've merely used it to constrain what can be used, where. It's not causing us any additional pain right now, but the requirements of **IRequest** may change in the future. We're going to create a placeholder base class which could be very useful for later defining behaviour common to every request.

In Client/Command/Requests, create a new class called **BaseRequest.cs**. Implement **IRequest**, and mark the class **abstract**.

```
public abstract class BaseRequest : IRequest
{
}
```

**BaseResponse** is a tad more interesting. Create it in Command/Responses.

```
public abstract class BaseResponse : IResponse
{
    public bool HasErrored => !string.IsNullOrEmpty(ErrorMessage);
    public string ErrorMessage { get; set; }
}
```

We are going to add properties to this. We're going to implement **HasErrored** and **ErrorMessage** for any class that derives from **BaseResponse**.

```
public abstract class BaseResponse : IResponse
{
    public bool HasErrored => !string.IsNullOrEmpty(ErrorMessage);
    public string ErrorMessage { get; set; }
}
```

Now comes the fun part. Let's revisit our existing command objects and change them to directly inherit from either **BaseResponse** or **BaseRequest**

The request objects don't change much. They now inherit from **BaseRequest** instead of declaring that they are an **IRequest**. They still fit that profile through the relationship with **BaseRequest**.

The request objects inherit from **BaseRequest**. **BaseRequest** is an **IRequest**. Therefore, our request objects are still **IRequest**, and will still satisfy the demands of any generic constraints on our adapter classes that need them to be so.

```
public class SignUpRequest : BaseRequest
{
    public string Email { get; set; }
    public string Username { get; set; }
    public string Password { get; set; }
    public string ConfirmPassword { get; set; }
}
```

```
public class SendEmailRequest : BaseRequest
{
    // These will come from the Adapter.
    public string FromEmail { get; set; }
    public string FromName { get; set; }
    public string Message { get; set; }

    // This will be provided in the service.
    public EmailDestination Destination { get; set; }
}
```

```
public class EncryptedPasswordRequest : BaseRequest
{
    public string Password { get; set; }

    // Not necessary for first time creations
    public string PasswordSalt { get; set; }
}
```

Here's the really fun part. **BaseResponse**. It actually handles **ErrorMessage** and **HasErrored** for us, so we can *remove* all those copied and pasted properties.

```
public class EncryptedPasswordResponse : BaseResponse
{
    public string EncryptedPassword { get; set; }
    public string PasswordSalt { get; set; }
}
```

```
public class SendEmailResponse : BaseResponse
{
    // The ? means that the type can be NULL
    public DateTime? CompletedAt { get; set; }
}
```

```
public class SignUpResponse : BaseResponse
{
    public DateTime? CompletedAt { get; set; }
}
```

Notice anything missing? None of the Response classes have an **ErrorMessage** or **HasErrored** property defined anymore. They don't need to. They inherit that functionality from **BaseRequest** now, and *still* satisfy any demand that they be an **IRequest**.

## Adapting SignUpViewModel into a SignUpRequest

Now our command objects are back in place, we need to establish how a `SignUpViewModel` gets turned into a `SignUpRequest` and how a `SignUpResponse` gets turned back into a `ViewModel`.

It's another implementation of `IRequestResponseAdapter`.

All of our objects are in place, and will satisfy the type constraints of the adapter.

```
public class SignUpAdapter : IRequestResponseAdapter<SignUpRequest, SignUpResponse,
SignUpViewModel>
{
    public SignUpRequest AdaptRequest(SignUpViewModel viewModel, SignUpRequest
request)
    {
        return new SignUpRequest
        {
            Password = viewModel.Password,
            ConfirmPassword = viewModel.ConfirmPassword,
            Email = viewModel.Email,
            Username = viewModel.Username
        };
    }

    public SignUpViewModel AdaptResponse(SignUpViewModel viewModel, SignUpResponse
response)
    {
        viewModel.HasErrored = response.HasErrored;
        viewModel.StatusMessage = (response.HasErrored) ? response.ErrorMessage :
"Success";
        viewModel.CompletedAt = response.CompletedAt;

        return viewModel;
    }
}
```

It follows the same pattern as before. The class essentially acts as a mapper for requests, and will set **StatusMessage** according to the state of **HasErrored**. We're repeating ourselves again in **AdaptResponse**.

Every single one of our Adapters is using the same, copied and pasted code to handle errors. We can improve on that.

Let's add another base class, **BaseRequestToResponseAdapter**, and refactor. Add the class in the **Empower.Adapters** library in the ViewModels directory.

First off, it has to be a generic, with the same type constraints of the interface it is going to implemented.

```
public abstract class BaseRequestResponseAdapter<TRequest, TResponse, TViewModel>
    where TRequest : IRequest
    where TResponse : IResponse
    where TViewModel : IViewModel
{
}
```

```
}
```

Next, we need to declare that it can do the work of an **IRequestResponseAdapter**.

```
public abstract class BaseRequestResponseAdapter<TRequest, TResponse, TViewModel>
    : IRequestResponseAdapter<TRequest, TResponse, TViewModel>
    where TRequest : IRequest
    where TResponse : IResponse
    where TViewModel : IViewModel
{
}

```

Because we've declared that, we need to implement the required methods. Our **AdaptResponse** method is different for each ViewModel and command request combination.

We can both satisfy the immediate contractual demands of **IRequestResponseAdapter** and hand off implementation by marking the **AdaptResponse** method as **abstract**.

```
public abstract class BaseRequestResponseAdapter<TRequest, TResponse, TViewModel>
    : IRequestResponseAdapter<TRequest, TResponse, TViewModel>
    where TRequest : IRequest
    where TResponse : IResponse
    where TViewModel : IViewModel
{
    public abstract TRequest AdaptRequest(TViewModel viewModel, TRequest request);
}

```



An **abstract** method or property is a way of telling C#'s compiler that the required behaviour will be implemented by a derived class. **abstract** methods can only be defined in **abstract** classes, which is why the compiler doesn't complain. It'll never be expected to new up an abstract class, so the lack of a proper implementation does not matter.

For **AdaptResponse**, we're going to provide an implementation inside the class.

```
public abstract class BaseRequestResponseAdapter<TRequest, TResponse, TViewModel>
    : IRequestResponseAdapter<TRequest, TResponse, TViewModel>
    where TRequest : IRequest
    where TResponse : IResponse
    where TViewModel : IViewModel
{
    public abstract TRequest AdaptRequest(TViewModel viewModel, TRequest request);

    public virtual TViewModel AdaptResponse(TViewModel viewModel, TResponse response)
    {
        viewModel.HasErrored = response.HasErrored;
        viewModel.StatusMessage = (response.HasErrored) ? response.ErrorMessage :
            "Success";
    }
}

```

```
        return viewModel;
    }
}
```

As implemented, that will serve all our response classes, performing most of what they need. We **always** want to set **HasErrored** to true in the **ViewModel** if that is true in the command object.

We'll still need to do some specific adaptation, but we can use our base class to do the every time donkey work, leaving specific implementations to fill in the blanks that only they can.

Open **ContactAdapter** and **SignUpAdapter**. Instead of **implementing** the **IRequestResponseAdapter** *interface*, we're going to **inherit** from **BaseRequestResponseAdapter**. We can replace all of the common error handling with one line.

```
viewModel = base.AdaptResponse(viewModel, response);
```

Our adapters have no unnecessarily copied and pasted code in them. They are now using a new keyword, **override**.

In both cases, it means that we're overriding an implementation in the base class. In the case of **AdaptRequest**, we have to override. The base class does not have its own implementation. The app will not compile if we do not.

In the case of **AdaptResponse**, we can optionally override the method if we want to provide our own specific adaptation in the Response. Your app will still build if it is not provided, falling back on the base class method, which is implemented.

Another keyword is **base**, used in these classes to call the base class version of **AdaptResponse**.

Contact adapter.

```
public class ContactAdapter :
BaseRequestResponseAdapter<SendEmailRequest, SendEmailResponse, ContactViewModel>
{
    public override SendEmailRequest AdaptRequest(ContactViewModel viewModel,
SendEmailRequest request)
    {
        request.FromEmail = viewModel.Email;
        request.FromName = viewModel.Name;
        request.Message = viewModel.Message;

        return request;
    }

    public override ContactViewModel AdaptResponse(ContactViewModel viewModel,
SendEmailResponse response)
    {
        viewModel = base.AdaptResponse(viewModel, response);

        viewModel.CompletedAt = response.CompletedAt;

        return viewModel;
    }
}
```

```
}
}
```

Sign up adapter.

```
public class SignUpAdapter : BaseRequestResponseAdapter<SignUpRequest, SignUpResponse,
SignUpViewModel>
{
    public override SignUpRequest AdaptRequest(SignUpViewModel viewModel,
SignUpRequest request)
    {
        return new SignUpRequest
        {
            Password = viewModel.Password,
            ConfirmPassword = viewModel.ConfirmPassword,
            Email = viewModel.Email,
            Username = viewModel.Username
        };
    }

    public override SignUpViewModel AdaptResponse(SignUpViewModel viewModel,
SignUpResponse response)
    {
        viewModel = base.AdaptResponse(viewModel, response);
        viewModel.CompletedAt = response.CompletedAt;

        return viewModel;
    }
}
```

## Dependency injection

Time to wire up the new Adapter. Let's return to **Startup.cs** and specify that the **SignUpAdapter** is the thing that will turn ViewModels into commands, and back again.

```
kernel.Bind<IRequestResponseAdapter<SignUpRequest, SignUpResponse, SignUpViewModel>>()
    .To<SignUpAdapter>().InSingletonScope();
```



As usual, the DI binding is essentially saying "if anything asks for something that can do **IRequestResponseAdapter<SignUpRequest, SignUpResponse, SignUpViewModel>** functionality, give it a **SignUpAdapter**"

## New project - Empower.NHibernate.Adapters

Create a new C# class library project called **Empower.NHibernate.Adapters**. This will house classes that adapt **Commands** in **Domain.Client** to nHibernate **Entities**.



Dependencies

This project needs to link to:-

- Empower.NHibernate
- Empower.NHibernate.Entities
- Empower.Domain
- Empower.Services



Projects that depend on this.

The following project need to link to **Empower.NHibernate.Adapters**

- Empower.Mvc
- Empower.NHibernate.Services

## New directory - Interfaces

This is going to be another case where we keep our interfaces close to our classes. Create a new directory called **Interfaces** in **Empower.NHibernate.Adapters**.

## New interface - IRequestToEntityAdapter

One of the most common things we're going to need to do is translate **Command** objects, specifically **Request** objects, into **Entities**. We're going to define a generic service to do this.

Create a new interface called **IRequestToEntityAdapter**

```
public interface IRequestToEntityAdapter
{
}
```

We want this to be able to adapt any **Request** command into any **Entity**, but we don't know what we're dealing with in advance.

```
public interface IRequestToEntityAdapter<TRequest, TEntity>
{
    TEntity Adapt(TRequest request);
}
```

As usual, we should place some constraints on what types our adapter will deal with, using the **where** keyword to constrain the types of **TRequest** and **TEntity**.

```
public interface IRequestToEntityAdapter<TRequest, TEntity>
    where TRequest : IRequest
    where TEntity : Entity
{
}
}
```

Finally, we should define what behaviour we'd like the service to have. Our needs are easy to express.



```
public interface IRequestToEntityAdapter<TRequest, TEntity>
    where TRequest : IRequest
    where TEntity : Entity
{
    TEntity Adapt(TRequest request);
}
```

Anything saying that it is an **IRequestToEntityAdapter** must provide a method called **Adapt**, taking a **TRequest** object and returning **TEntity**.

## Concrete implementation - SignUpRequestToUserAdapter.cs

Let's return to our **Empower.NHibernate.Adapters** project. We may adapt other kinds of things for **NHibernate**-related objects, so let's create a subdirectory specifically marked to handle our entity mappings. Create a directory called **Entities**.

Add a new class to that directory. Call it **SignUpRequestToUserAdapter.cs** and *implement* **IRequestToEntityAdapter**. As the name might suggest, this class is going to adapt **SignUpRequest** commands into **User** entities.

### Inject an IPasswordHashingService

We've still not injected the **IPasswordHashingService** we defined earlier. At the time, the **UserService** might have seemed the ideal place to inject the service. Now that we have an adapter that has the specific job of adapting **SignUpRequest** objects into **User** entities, it makes more sense to put that here.

💡 One of the best ways not to store a *cleartext* password is to ensure no field in the **User** entity ever gets set to one. We're going to inject **IPasswordHashingService** here to create a robust adaptation mechanism that'll automatically turn the *cleartext* password in the **Command** object into an encrypted password in the **User** entity object.

Let's include a readonly reference variable and **constructor injection** to populate it.

```
public class SignUpRequestToUserAdapter : IRequestToEntityAdapter<SignUpRequest, User>
{
    private readonly IPasswordHashingService _passwordHashingService;

    public SignUpRequestToUserAdapter(IPasswordHashingService passwordHashingService)
    {
        _passwordHashingService = passwordHashingService;
    }
}
```

We still owe the technical debt taken on when we said that this class could do **IRequestToEntityAdapter** stuff. We don't have an **Adapt** implementation specified yet. The compiler *demands* satisfaction. Better get it satisfied.

Most of our adaptation is actually **mapping**, the simple art of copying one property's value into another. We're definitely adapting too. The data that goes in is not the same that comes out. A cleartext password gets encrypted. A password salt is generated. Both adapted values are moved into the **User** entity.

We're also providing a value for **JoinDate**, a value not carried in **SignUpRequest**. We're just setting it to a default, the current **DateTime.UtcNow** value.

```
public User Adapt(SignUpRequest request)
{
    // Use the hashing service to get a more secure
    // password.
    var encryptedPasswordResponse = _passwordHashingService.EncryptPassword(
        new EncryptedPasswordRequest
        {
            Password = request.Password
        }
    );

    return new User
    {
        Username = request.Username,
        Password = encryptedPasswordResponse.EncryptedPassword,
        PasswordSalt = encryptedPasswordResponse.PasswordSalt,
        Email = request.Email,
        // This will need to be included later.
        Role = null,
        // Set the JoinDate to now
        JoinDate = DateTime.UtcNow
    };
}
```

:question\_mark: **Why is the Role left empty?** We're going to get the calling service to provide that for now. We'll explore strategies whereby adapters might resolve this information themselves.

For now, we'll return a **User** that is *almost* ready for saving.

## Wiring up dependency injection

We need to tell the **dependency injection container** that there is a new service in town. It's a generic service, so once we've navigated back to our **Empower.Mvc** project, we can go to the **Startup.cs** class to get that resolved.

We're keeping our binding **specific** at the moment. Our **adapter** knows what types it needs. Our **adapter service** will only be provided if those two types are the ones in play.

```
kernel.Bind<IRequestToEntityAdapter<SignUpRequest, en.User>>()
    .To<SignUpRequestToUserAdapter>();
```

If anything needs a **IRequestToEntityAdapter<SignUpRequest, en.User>**, we've configured the DI Container to return a **SignUpRequestToUserAdapter**.

## Change to IUserService

We're adding some functionality to **IUserService**. From now on, anything claiming to be able to perform this functionality must perform an additional service.

Add the following method to IUserService

From now on, anything claiming to be able to do **IUserService** things must be able to handle a **SignUp**, taking in a **SignUpRequest** and returning a **SignUpResponse**.

```
SignUpResponse SignUp(SignUpRequest request);
```

### Concrete class changes

We're going to need to make sure that our **UserService** fulfils its newly foisted obligation to handle a **SignUpRequest**

Let's open **Empower.NHibernate.Services** and make the change.

Our sign up implementation is going to need to do the following things:-

- Check to see that the supplied email and username do not already exist in the database
  - If true
    - Return an error message, indicating the sign up did not happen
  - If false
    - Adapt the response into an entity
    - Create a new User in the database
      - The user should be in the "Sign up" role
      - The user's password should be encrypted
      - A password salt, to enable encryption checks, should also be stored.

Most of the implementation is already done. We've got an adapter that can turn our command into a saveable entity. We've also got NHibernate plumbing, such as a repository, that we can call upon for our persistence needs. The implementation is mostly a case of stitching all of that stuff together.

We do have two requirements that we've not met. We need to make sure that we don't already have a user with the same email or username as an existing user.

Let's define a new method on the concrete implementation to do just that.

For now, no other part of the application needs to know about this method apart from **UserService** itself. We can define it as a **private** method.

```
private SignUpResponse PerformDupeCheck(SignUpRequest request)
{
    // Search the DB for a user with the same email or username
    // that is provided in the request.

}
```

Let's flesh it out.

```
private SignUpResponse PerformDupeCheck(SignUpRequest request)
{
```

```

// New up a response object.
// This is presently the only piece of validation performed at this
// level. The rest is handled at model binding level
//
// We're okay to start constructing our response here.
// if we need to change that in future, we can simply pass an existing response
// object in.
var response = new SignUpResponse();

// Search the DB for a user with the same email or username
// that is provided in the request.
//
// We're making one call to the DB here, asking if there is a row where either is
true.
var existingUser =
    _userRepository
        .GetQueryOver()
        .Where(c => c.Email == request.Email || c.Username == request.Username)
        .SingleOrDefault();

// .SingleOrDefault returns null if no row is found
// If existingUser is not null, we have a problem
if (existingUser != null)
{
    response.ErrorMessage =
        // This is a ternary statement. It's another way of expressing
conditional
        // functionality
        (existingUser.Email == request.Email && existingUser.Username ==
request.Username)
        ? "A user with the provided email and username already exists"
        : (existingUser.Email == request.Email)
            ? $"The email {request.Email} belongs to another user. Is it you?"
            : "This username is unavailable";
}

return response;
}

```

The method will create a new response object, perform the check and return a response, setting an error message if anything went wrong.

The second requirement we have to meet is to be able to set the user's role.

To do that, we need to load a Role entity which models the "Sign up" role, and assign it to the user. Again, we'll write a helper method to assist. We may need this ability in future inside **UserService**.

We're going to write a method that given a string alias, returns a Role entity we can use.

```

private Entities.Role GetRoleByAlias(string alias)
{
    return _roleRepository.GetQueryOver()
        .Where(c => c.Alias == alias)

```

```
        .SingleOrDefault();  
    }
```

With those helpful methods in place, we've got all the pieces we need to implement our overall contractual requirement, and its surprisingly easy.

```
public SignUpResponse SignUp(SignUpRequest request)  
{  
    var response = PerformDupeCheck(request);  
  
    // Does the username exist?  
    // Does the email exist?  
    if (!response.HasErrored)  
    {  
        // Adapt the sign up to an entity.  
        var entity = _signUpEntityAdapter.Adapt(request);  
  
        // Set the role. All signups will go into the signup role.  
        entity.Role = GetRoleByAlias("signup");  
  
        // Save through the repository  
        _userRepository.Add(entity);  
    }  
  
    return response;  
}
```

The complexity has been hidden away in the following ways:-

- Error handling
  - Errors detected in the PerformDupeCheck
  - We only attempt to persist when no errors are found
- Adaptation
  - We're turning a SignUpRequest into a saveable User entity.
    - The adapter makes use of **IPasswordHashingService** to automatically encrypt the entered cleartext password.
- Persistence
  - Our repository is handling the complexity of saving.

The result is a very simple method. Stripped of comments, it looks even simpler.

```
public SignUpResponse SignUp(SignUpRequest request)  
{  
    var response = PerformDupeCheck(request);  
  
    if (!response.HasErrored)  
    {  
        var entity = _signUpEntityAdapter.Adapt(request);  
        entity.Role = GetRoleByAlias("signup");  
        _userRepository.Add(entity);  
    }  
}
```

```
    return response;
}
```

## Putting the final pieces together.

There's a tiny bit of plumbing left to connect our **SignUp** service request with the **ViewModel**. For better or worse, we've given **IViewModelServices** the job of dealing with **ViewModel** objects. Let's ask it to handle our **SignUpViewModel**.

### Change to IViewModelService

Add the following declaration of behaviour to **IViewModelService**.

```
SignUpViewModel SignUp(SignUpViewModel viewModel);
```

```
public interface IViewModelService
{
    ContactViewModel SendContactEmail(ContactViewModel request);
    SignUpViewModel SignUp(SignUpViewModel viewModel);
}
```

### Implementing the change in ViewModelService

We'll need to give ViewModelService the things it needs, be it services or adapters, to do its job.

Let's get **IUserService** and **IRequestResponseAdapter** services on the required list with **constructor injection** and create variables so that we can reference them.

```
public class ViewModelService : IViewModelService
{
    private readonly IUserService _userService;
    private readonly IRequestResponseAdapter<SignUpRequest, SignUpResponse,
SignUpViewModel> _signUpAdapter;
    private readonly
IRequestResponseAdapter<SendEmailRequest, SendEmailResponse, ContactViewModel>
_contactAdapter;
    private readonly IEmailService _emailService;

    public ViewModelService
    (
        IRequestResponseAdapter<SendEmailRequest, SendEmailResponse, ContactViewModel>
contactAdapter,
        IRequestResponseAdapter<SignUpRequest, SignUpResponse, SignUpViewModel>
signUpAdapter,
        IEmailService emailService,
        IUserService userService
    )
    {
    }
```

```

    {
        // Assign our injected services, allowing them to access them.
        _emailService = emailService;
        _contactAdapter = contactAdapter;
        _signUpAdapter = signUpAdapter;
        _userService = userService;
    }
}

```

Now we've got our tools, we can meet the requirements.

We follow the same steps in implementation that we did before.

- Adapt the ViewModel into a request
- Send the request to a service to be completed
- Adapt the response from the service into a ViewModel
- Return the ViewModel to the caller

```

public SignUpViewModel SignUp(SignUpViewModel viewModel)
{
    var request = _signUpAdapter.AdaptRequest(viewModel, new SignUpRequest());
    var response = _userService.SignUp(request);
    viewModel = _signUpAdapter.AdaptResponse(viewModel, response);

    return viewModel;
}

```

## Controller changes

We've got two main jobs to do on **UserController**. We need to make sure it has an instance of **IViewModelServices** to work from. As usual, we can use **constructor injection** and a reference variable to **inject** the service.

```

private readonly IViewModelService _viewModelService;

public UserController(IViewModelService viewModelService)
{
    _viewModelService = viewModelService;
}

```

With that in place, we can now call **SignUp** on **\_viewModelService**, knowing nothing more than it processes a **SignUpViewModel**.

It's beautifully simple, and it'll work. Running the POST form will now create a new user in the signup role.

```

[HttpPost]
public IActionResult SignUp(SignUpViewModel viewModel)
{
    if (ModelState.IsValid)

```

```

    {
        viewModel = _viewModelService.SignUp(viewModel);
    }

    return View(viewModel);
}

```

## One last thing

We should probably change the view to let people know they've successfully signed up.

Open up **SignUp.cshtml** in Views/User. Let's put some conditional display to work.

```

@if (Model.StatusMessage != "Success")
{
    <h4>Please enter your details below</h4>
    <hr />
    <div class="row">
        <div class="col-md-4">
            <form asp-action="SignUp" method="post">
                <div asp-validation-summary="ModelOnly" class="text-danger"></div>
                <div class="form-group">
                    <label asp-for="Username" class="control-label"></label>
                    <input asp-for="Username" class="form-control" />
                    <span asp-validation-for="Username" class="text-danger"></span>
                </div>
                <div class="form-group">
                    <label asp-for="Email" class="control-label"></label>
                    <input asp-for="Email" class="form-control" />
                    <span asp-validation-for="Email" class="text-danger"></span>
                </div>
                <div class="form-group">
                    <label asp-for="Password" class="control-label"></label>
                    <input asp-for="Password" class="form-control" type="password"/>
                    <span asp-validation-for="Password" class="text-danger"></span>
                </div>
                <div class="form-group">
                    <label asp-for="ConfirmPassword" class="control-label"></label>
                    <input asp-for="ConfirmPassword" class="form-control" type="password" />
                    <span asp-validation-for="ConfirmPassword" class="text-danger"></span>
                </div>
                <div class="form-group">
                    <input type="submit" value="Create" class="btn btn-default" />
                </div>
            </form>
        </div>
    </div>
} else
{
    <div class="alert alert-success">
        <h2>Success</h2>
        <p>You have signed up with Sightsource!</p>
    </div>
}

```



```
}
```

## Run the app

Run the app and sign up.

## Commit your work

**feature/signup** is complete. Check in your work and merge the whole branch back into **master**

## What's just happened?

- We've introduced our first persisting service, **IUserService**
- We created new command objects and view models to handle the concept of a sign up request.
- We wrote a new kind of adapter, one that turns command objects into NHibernate saveable entities.
- We introduced a password hashing service, and used that in our entity adapter to automatically generate a hashed password from a cleartext one.
- We continued to use **IViewModelService** at the controller level.
- Our controller methods are now entirely dumb, don't know **what** happens during any of these requests, and doesn't **care** how the requests are implemented.