# Asp

Asp uses MVC as a main component of it. The views folder has the UI elements.

The controller folder has the routing elements. How it works is that; inside the controller folder there are ‘pages’ like homecontroller. They store libraries of functions that basically link the HTTP requests the user puts in the browser to the particular views.

It does this via an ‘action result’. The action result is quite clever, it will assume the page is the name of the function if it’s not specified.

To show the page you say:  
return View(“pagename”).



This describes how the controllers work. Home is a controller. The most general one is described last. Eg

{controller}/{action}.

Because MVC is essentially a framework it comes with a routeconfig class in app\_start folder. It has in there the code that makes the routing work.

Configuring the routing engine is done by copy and pasting the default and modyfing it:

routes.MapRoute(

name: "Default",

url: "{controller}/{action}/{id}",

defaults: new { controller = "Home", action = "Index", id = UrlParameter.Optional }

);

Each route maps a ‘pattern’ or family of webpages as per my understanding so far.

The URL is quite nice because we can insert our own wildcards like:

routes.MapRoute(

name: "Serial Number",

url: "serial/{letterCase}",

defaults: new { controller = "Home", action = "Serial", letterCase="upper"}

Thus, it takes ‘serial/lettercase’. Where lettercase can be upper or lower. By default it’s upper.

The routes are executed sequentially. So move them up or in the order you want things to be accessed. If you notice, the first route specified defaults that actually work. So even though you type in webpage/serial/uppercase. It will default to index?

So we finished the first exercise getting routing functional.

The big hurdle was the routing order. Any time a website is causing a page not found. It’s because of a routing error.

**Action types:**

The ActionResult function we keep calling can return a variety of things.

It can return preset error messages, it can return ‘content’ which is just text. It can return full or partial webpages, and it can return JSON, it can ‘redirect to action or redirect to route.

Eg.

return Json(new { name = "serial", value = serial }, JsonRequestBehavior.AllowGet);

This constructs a new json object, specifying the internals name, and value. It also has the argument jsonrequestbehaviour.allowget. (This seems to be a security thing. Json by default doesn’t allow get?

**Action Selector:**

Short and long term memory inside of those actionResults.

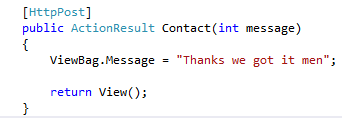
You can use ViewBag and functions of viewbag eg. ViewBag.Message = “bla” to store strings temporarily. These don’t survive redirects.

You can use TempData[“bla”]=”bla”. To survive redirects.

I think how this is to be used is like a ‘field kit’ for the view. The view you call (direct webpage) can reach into the bag and pull out anything you packed for it.

There’s also a [httpPost] you can put above functions in the controller. What this does is it indicates that the function will be executed if a post arrives from that page.

The HttpPost method, takes whatever the user posts and stores it in a variable that is an argument of the function that was called:



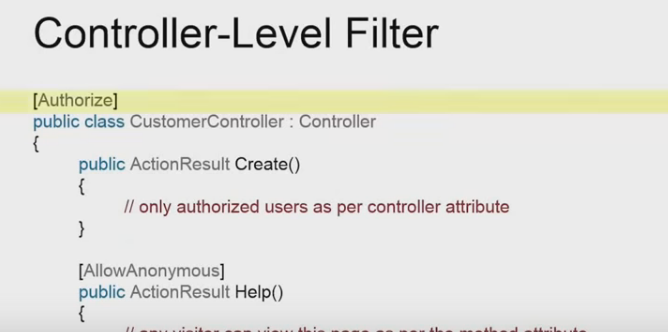
In this case message (which we made an int for lulz and testing) is returned.

# Filters

These are little tags you can add infront of the actionresult functions that have effects.

For example [Authorize] lets you control who can use the function as per the inbuilt authorization system.

This doesn’t have to just be on the action results, it can be on the whole controller class:



This can be reverted with [Allowanonymous] on individual funcitons.

Filters can be also more expansive, executing code before or after a route is taken.

# Razor Syntax

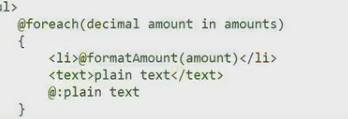
Razor is similar to the blade engine of laravel.

Use @ at any point in the html file to access C#, You can use this to get things like browser details.

For bigger blocks of C you can use @helper functionname.

It’s important to realise you can access csharp anytime inside of html with the @ sign, but you cannot do the reverse. Accessing HTMl inside fo a C# code block/line.

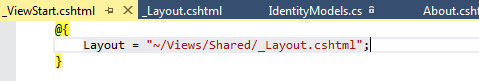
To do that you need to do things like:



Notice how the text element has a @: plaintext?

To comment @\* <code to be commented> \*@

# Layout

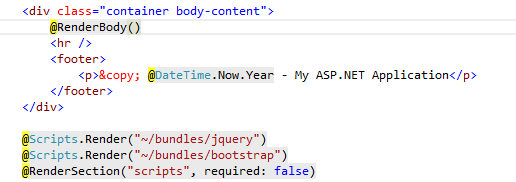


Quick way to apply the layout to all views.

This viewstart can also be copied and pasted into any view subfolder. (it’s currently in shared).

This defined the layout for anything in that folder.

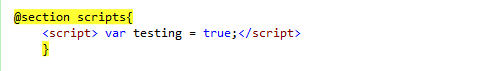
In the actual layout.cshtml file we have:



This @renderbody part is where the actual view data is ‘injected’.

The rendersection allows you to insert parts of the view for instance, the part of the view called ‘scripts’, in if it exists(required:false).

Inside the actual view file:



# HTML helpers

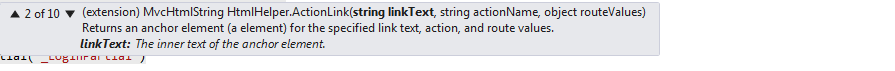
Seems to do something to make code less error prone>?

This seems to be a nice way to do things such as actually put a button in your website.

Done as follows:

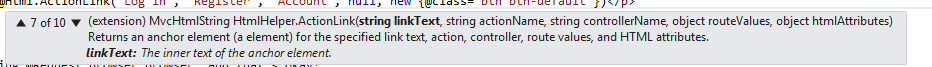


Actionlink has a lot of different overloads:



The above is telling it the link should be called ‘home’, it should call the method “action” inside the controller “Home”.

We can also do things like configure what the link looks like through CSS:

**TEXT COLORS:**

@helper formatAmount(decimal number)

{

var color = "green";

if (number < 0)

{

color = "red";

}

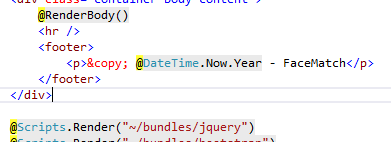
<span style="color:@color">@String.Format("{0:c}", number)</span>

}

Another thing we can do is the “partial” section.



This is a lot like:



Render body.

It renders whatever’s in the arugment as a partial view:

In this case \_LoginPartial which is in the shared folder.

We can set up forms:



Logoff is the action, Account is the ‘controller’. FormMethod.post is the HTTP method.

We can:

**HTML.action**

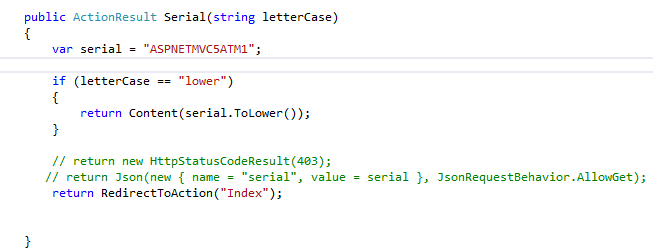
This allows you to return not views like the above, but return actual string as per some function..

Html.Action(“Serial”,”Home”)

This calls the value of action Serial in home controller.



However Serial looks like this:



*As a side note; you can only print strngs via the Content function.*

*So you can’t say return “yadingus”. You have to say return Content(“yadingus”). This is because the whole function is of type ‘actionresult’ and actionresult can’t return a string. It can return only another action redirect or a content.*

That is; IF the parameter is ‘lower’ then it triggers the string. If however the lettercase isn’t lower, (Or it’s null) it tries to return a page when it expects as string. (Yes it will literally print the html out).

We have to pass the function the variable lettercase as an argument:



**Bundling and minification?**

Minifying seems to be a method to improve performance by shortening variable names and spaces.

Inside of the layout we see:

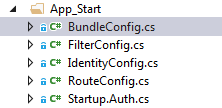


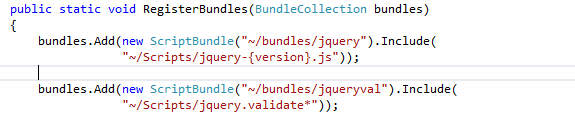
These are calling styles and bundles.

Where do they come from?



Located in the same place as routeconfig. Infact all the configs seem to be in the same place.





This Is how bundles are registered. Notice that the version is not specified, it’s left as a placeholder. This lets usupdate the bundle if we want and not have to change the code.

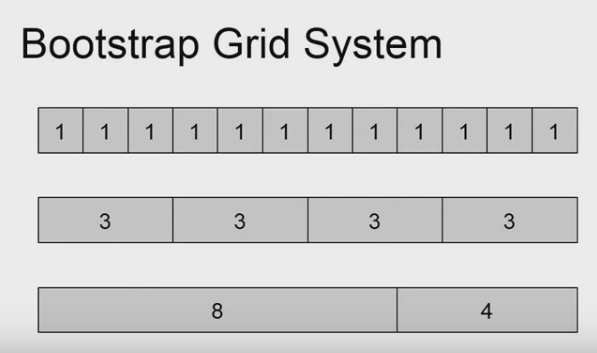
The validate has a wild card at the end to cover extensions. .js and .unobtrsuivejs.

Why do we want bundles:

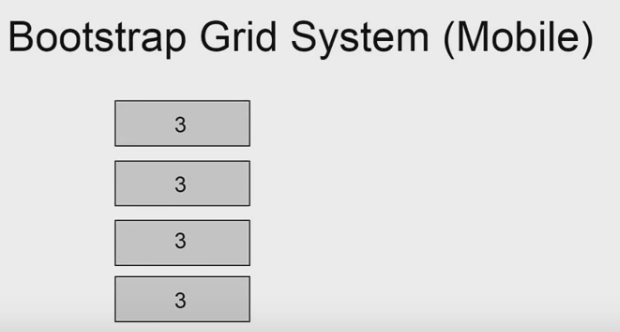
With bundles you can minify and non-minify files on the fly depending on if you’re in release mode or debug mode.

>> This has recently changed from the tutorial and wel’l ignore this section

**BootSTRAP**

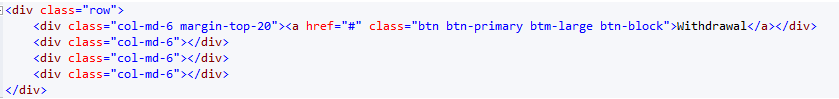


Has a 12 collumn system



bootstrap organises all content into 12 horizontal boxes. IF the sceeen size decreases it puts the boxes vertically as per default.

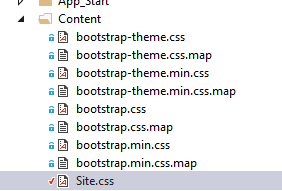
So bootstrap explains what this ‘col’ thing actually is; it’s a prebuilt class that’s prebuilt to react to mobile devices and device screen size changes naturally:



Notice, we also added a button to the firstcollumn. When defening a class for these elements you can just keep pressing spacebar and added new class names.

For instance btn is the first class, btm-primary is the second, btn-large is the third.

Similarly, we added a class in the CSS file for the whole website:



Site.CSS.

And added a class that adds a 20px margin infront of everything.

The website for boostrap is added to bookmarks that explains what a lot of these prebuilt classes do.

**Glyphs:**

Glyphs are icons that can be used; they are built into bootstrap:



To use these glyphicons you create an empty span with the class Glyphicon <glyphiconname>

Where the name is given above on the website:



**Some boostrap things:**

<div class = “row”>

This seems to be each ‘row’ level thing. Remember these things are ALWAYS under each other, they never slide to the side.

</div>

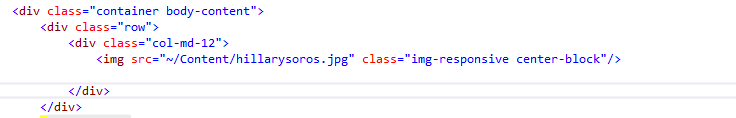
<div class="col-lg-6 margin-top-20">

Adds properties to the element that makes it auto resize, and align.

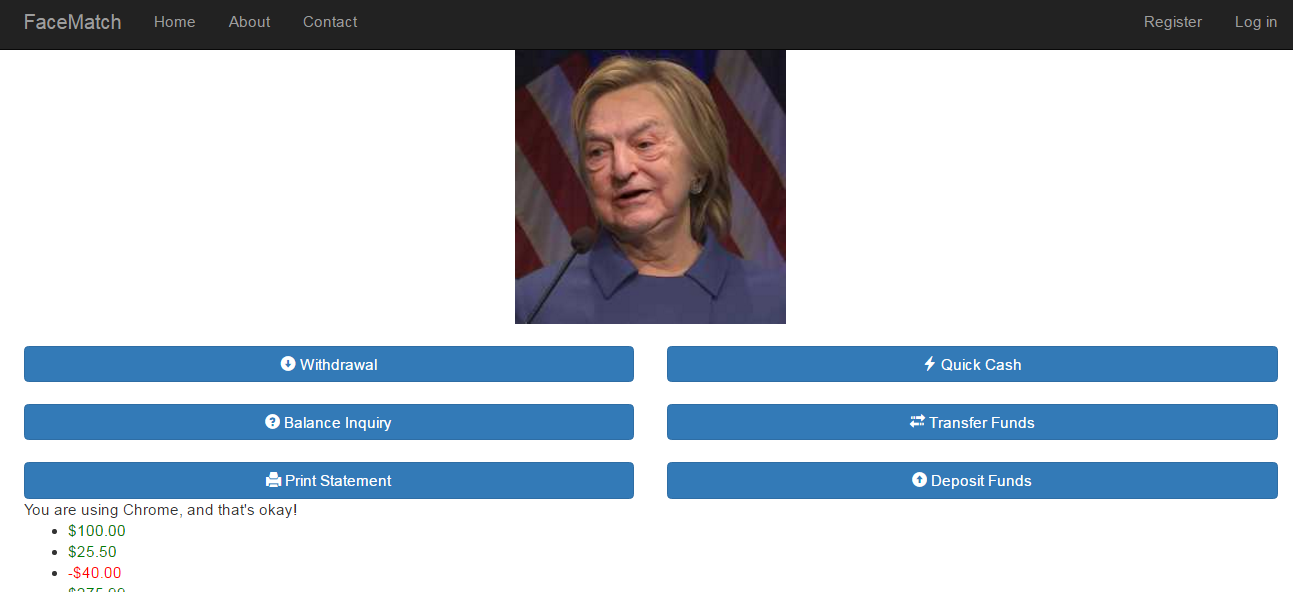
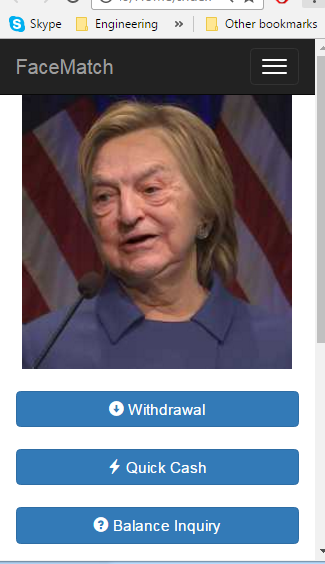
All divs must be inside a parent container element.

**Banners:**

For a banner, we drag images into the content folder in visual studio, and then drag and drop them in the code to insert it:



Note the properties: Class=”img-responsive center-block” This is what allows it to automatically resize and to always be centered.

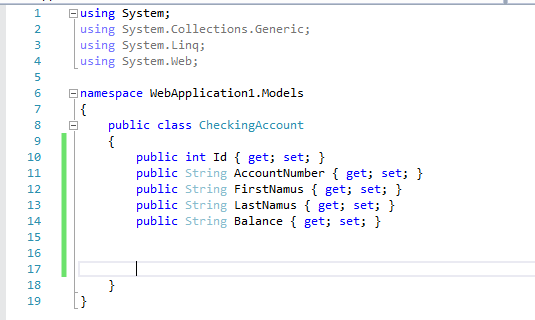
# Models and Actions views:

We have the ability to create classes and such by right clicking and clicking new.

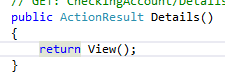
What we did first was to create a model called ‘checkingAccount’.

Inside this model we made a few variables; In MVC models seem to be representations of abstract data bodies that normally wouldn’t be collected together in a website, but would be collected together in everyday logic, and database logic.

We have the following variables in model:



We then created a view that automatically inherited from this model.

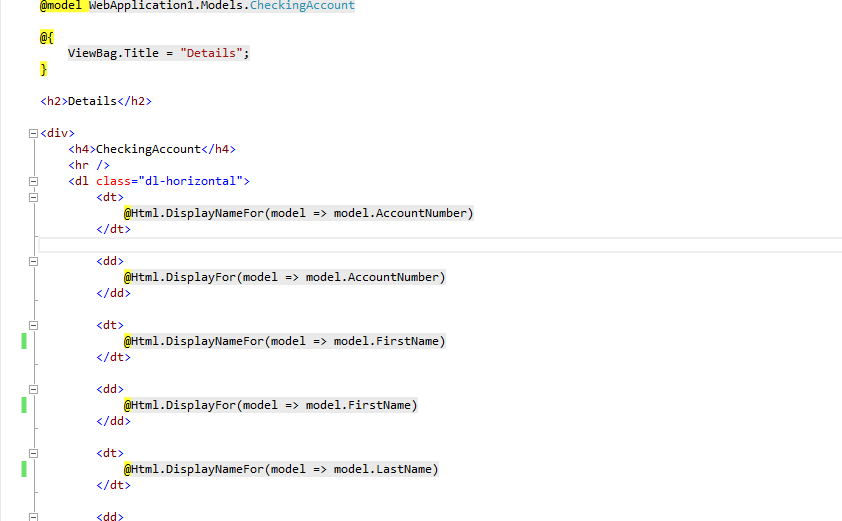
We also added a controller for the model, this comes with a set of pre-built functions like 

We do this by right clicking view folder and clicking ‘add view’.

If we click the action result we can add a ‘bound’ view to that:



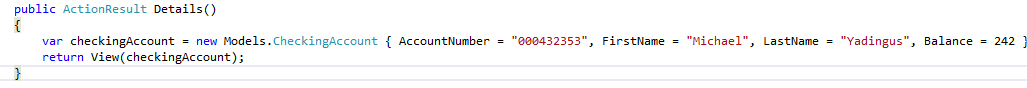
The view looks like this:



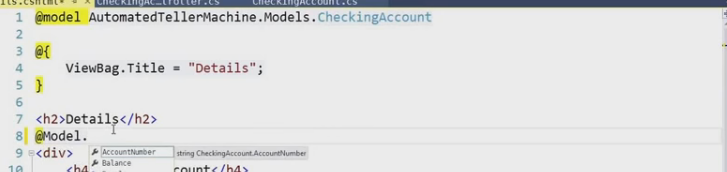
With the displays preset to show the model properties.

We can link this up together and make it work by:

The controller is showing he view like this:



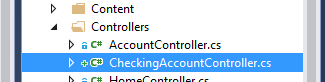
So the overall workflow is like this:

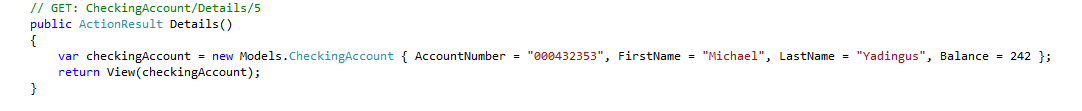
1. Create the model which has all the properties you want.
   1. (To make properties inside the model, right click->insert snipped->Prop)
2. Create a controller which can act as a point where buttons can ‘call’ for the model.
3. On the actionresult of the controller create a view that displays theinformation.
4. The view will automatically be set to show the properties of the model (all properties).
   1. 
   2. With the view, and the @model thing, anything with @Model (note capital M) refers to the model that was passed to the view when it was called).
5. Link the button from index to call the controller’s url via @url.action:

<a href="@Url.Action("Details","CheckingAccount")"

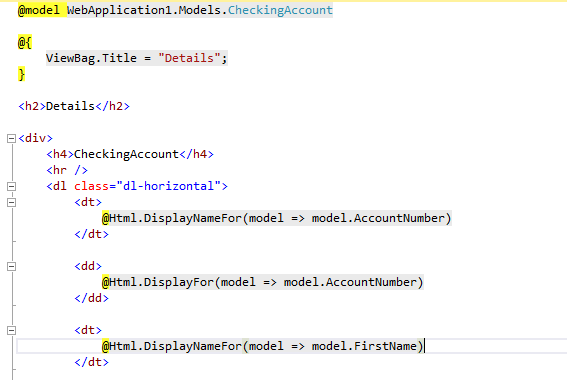
The first parameter represents the function name in the controller

The second represents the controller itself: EXLUDE THE WORD “CONTROLLER” FROM THE NAME. NOTICE HOW ITS CHECKING ACCOUNT AND NOT CHECKING ACCOUNT CONTROLLER.





In the view you can deifne the link to a model like so:



*When we last left off we had implemented the number 10 showing in our Mr Bean themed youwin youlose website.*

*We also created methods inside the model to simulate the rolling logic, but can’t find any way to actually use them. It’s predicted that the view’s buttons would call a url.action on the controller that would then trigger the methods on the model; but perhaps not.*