MVC Introduction

Objectives

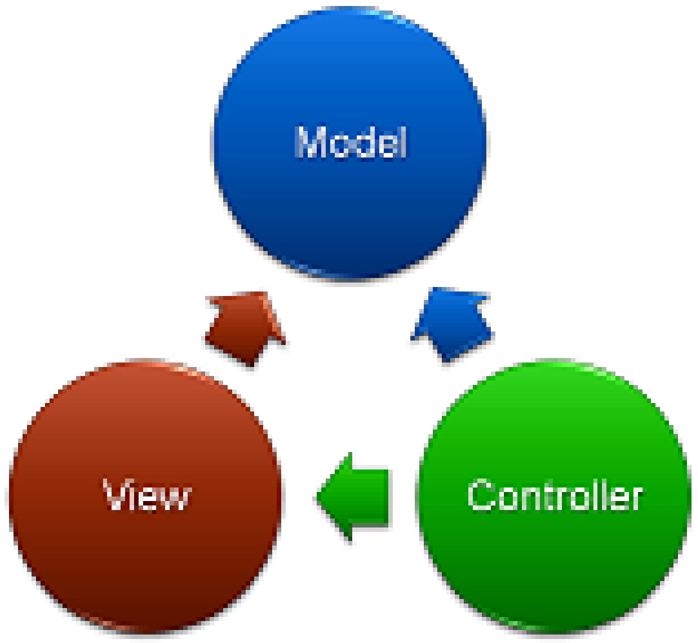
- Examine MVC project architecture
- Use routing to map requests to MVC
- Build controllers to manage requests
- Allow model binding to pass data to actions
- Use models to access and represent data
- Implement views to render results to client

Problems with ASP.NET Web Forms [motivation]

- **Lack of control over rendered markup**
  - many server controls generate ugly HTML
  - client side CSS and JavaScript sometimes cumbersome
- **Lack of modularity**
  - pages contain input processing and rendering logic
  - pages sometimes even contain data access logic
- **Lack of control over runtime environment dependencies**
  - applications are difficult to test without live web server

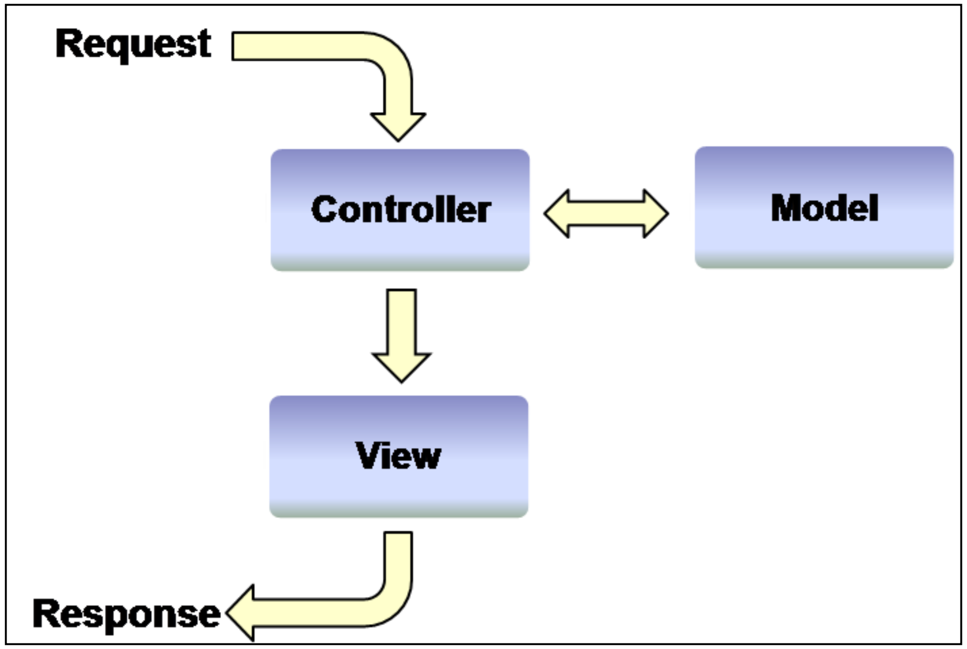
What is MVC?

- **Model-View-Controller architecture**
  - sometimes called front-controller
  - deliberate separation of concerns



What is MVC?

- **Controller receives all requests**
  - processes all input
  - accesses model
  - chooses response or view
- **Model is business layer**
  - same as WebForms (but more explicit)
  - no data access in pages
- **View emits response**
  - only contains rendering logic
  - passed data from controller



What is ASP.NET MVC?

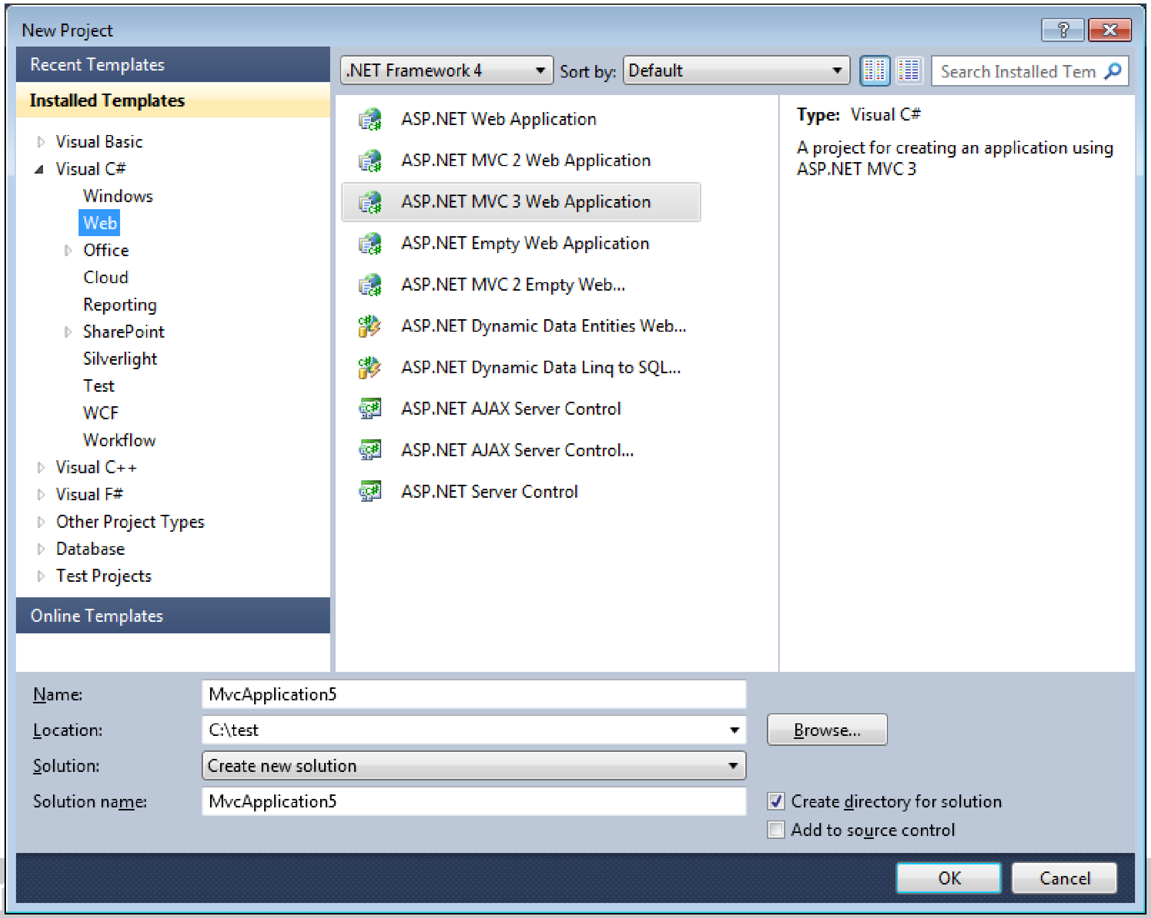
- **Framework**
  - maps requests onto controller classes and methods
  - Razor and WebForms view engines for rendering HTML templates
  - extensible, customizable and designed for testing
- **Main assemblies:**
  - System.Web
  - System.Web.Routing
  - System.Web.Mvc

ASP.NET MVC source

- **MVC is an open source project**
  - Microsoft Public License (MS-PL)
    - http://www.opensource.org/licenses/ms-pl.html
- **MVC code is available**
  - for browsing/debugging
  - for modification/redistribution
  - http://aspnet.codeplex.com/

ASP.NET MVC and Visual Studio

- **ASP.NET MVC installs a web application project template**
  - web site project types not supported



New Project

Recent Templates

Installed Templates

- Visual Basic
- Visual C#
  - Windows
  - Web
  - Office
  - Cloud
  - Reporting
  - SharePoint
  - Silverlight
  - Test
  - WCF
  - Workflow
- Visual C++
- Visual F#
- Other Project Types
- Database
- Test Projects

Online Templates

.NET Framework 4 | Sort by: Default | Search Installed Tem

- ASP.NET Web Application
- ASP.NET MVC 2 Web Application
- ASP.NET MVC 3 Web Application
- ASP.NET Empty Web Application
- ASP.NET MVC 2 Empty Web...
- ASP.NET Dynamic Data Entities Web...
- ASP.NET Dynamic Data Linq to SQL...
- ASP.NET AJAX Server Control
- ASP.NET AJAX Server Control...
- ASP.NET Server Control

Type: Visual C#

A project for creating an application using ASP.NET MVC 3

Name:	MvcApplication5
Location:	C:\test
Solution:	Create new solution
Solution name:	MvcApplication5

Browse...

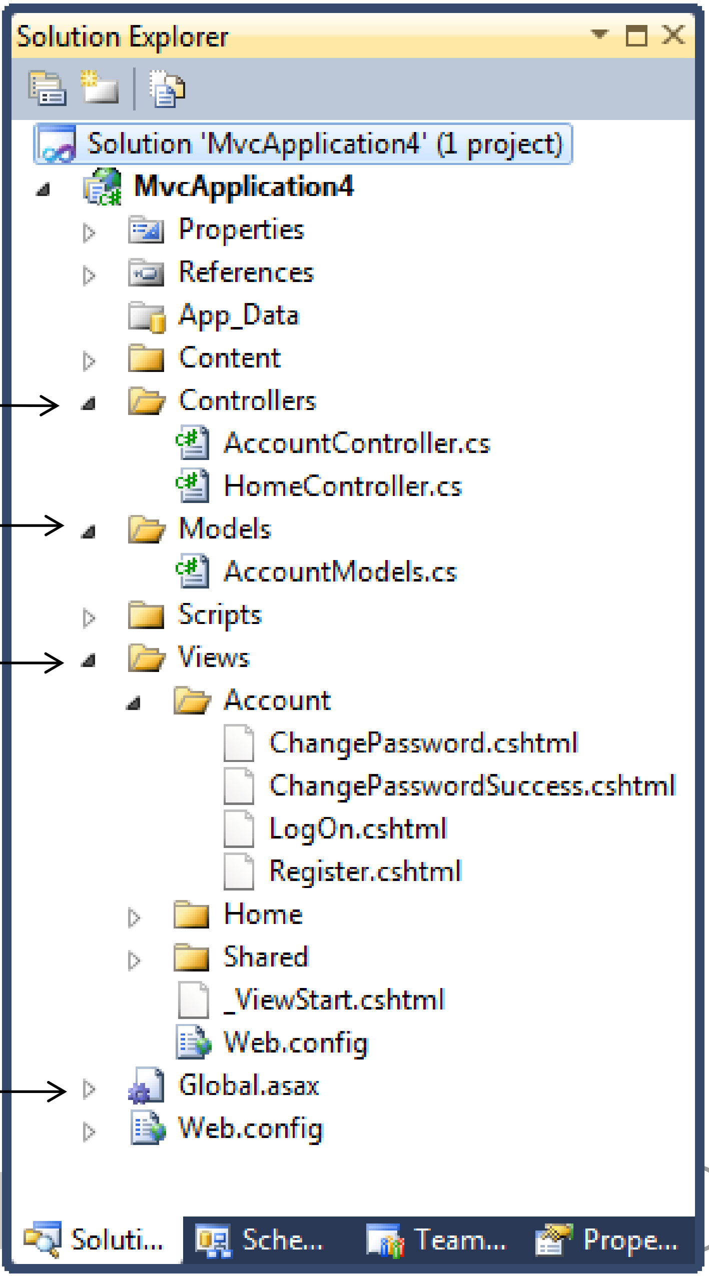
[✓] Create directory for solution

[ ] Add to source control

OK | Cancel

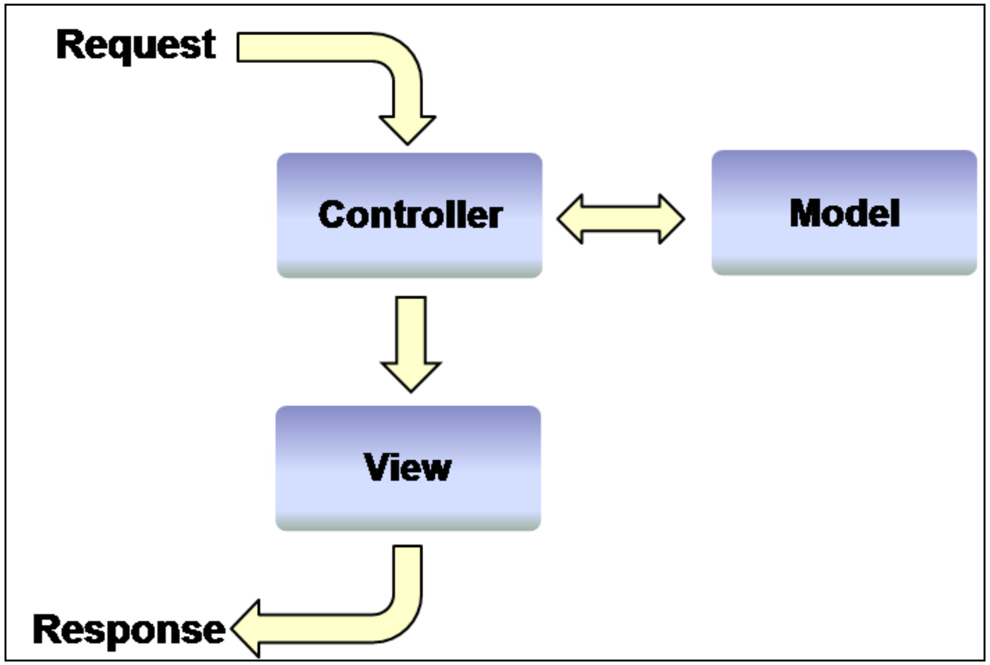
ASP.NET MVC project files

- **Controllers live in "Controllers"**
  - inherit `Controller` base
- **Models directory for data classes and code**
  - no special requirements
- **Views live in "Views"**
  - subdirectory with same name as controller
  - HTML templates with code
- **Routes defined in global.asax**
  - define URL to controller mapping



Controllers

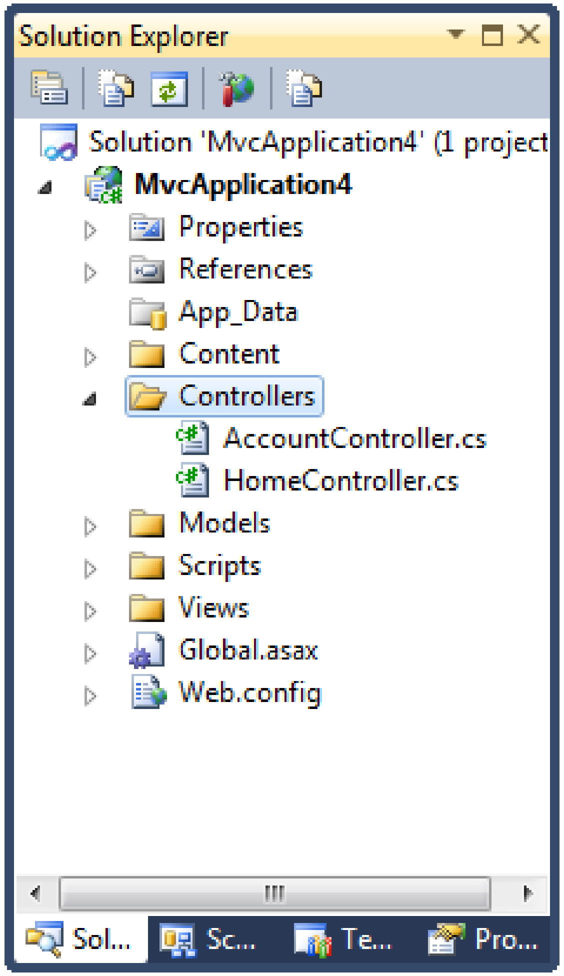
- **Controller is class that receives all requests**
  - handle all input
  - access model to perform business logic and data access
  - determine result to return to client



Controller class

- **Controller conventions**
  - class typically resides in "Controllers" directory
  - class name must end with "Controller"
  - derives from `Controller` base class

```
public class ProductController : Controller
{
   // ...
}
```



Controllers and requests

- **Requests map onto controller methods**
  - methods represent operations user will be performing
  - called "Actions" or "Action Methods"
  - typically return `ActionResult` (more on this later)

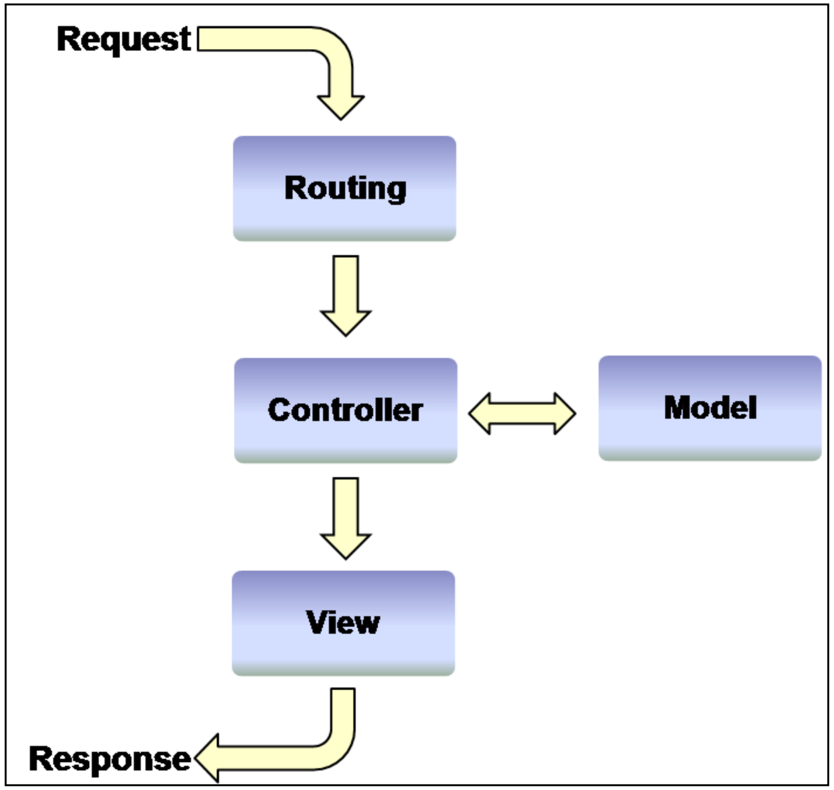
```
public class ProductController : Controller
{
   public ActionResult ShowAll() {
      // ...
   }
   public ActionResult Edit(string id) {
      // ...
   }
   public ActionResult Update(string id,
                        string description, decimal price) {
      // ...
   }
}
```

Controllers and routing

- **MVC heavily utilizes ASP.NET URL Routing**
  - URL indicates controller and action
    - *`~/Controller/Action`*
  - configurable in **`global.asax`**

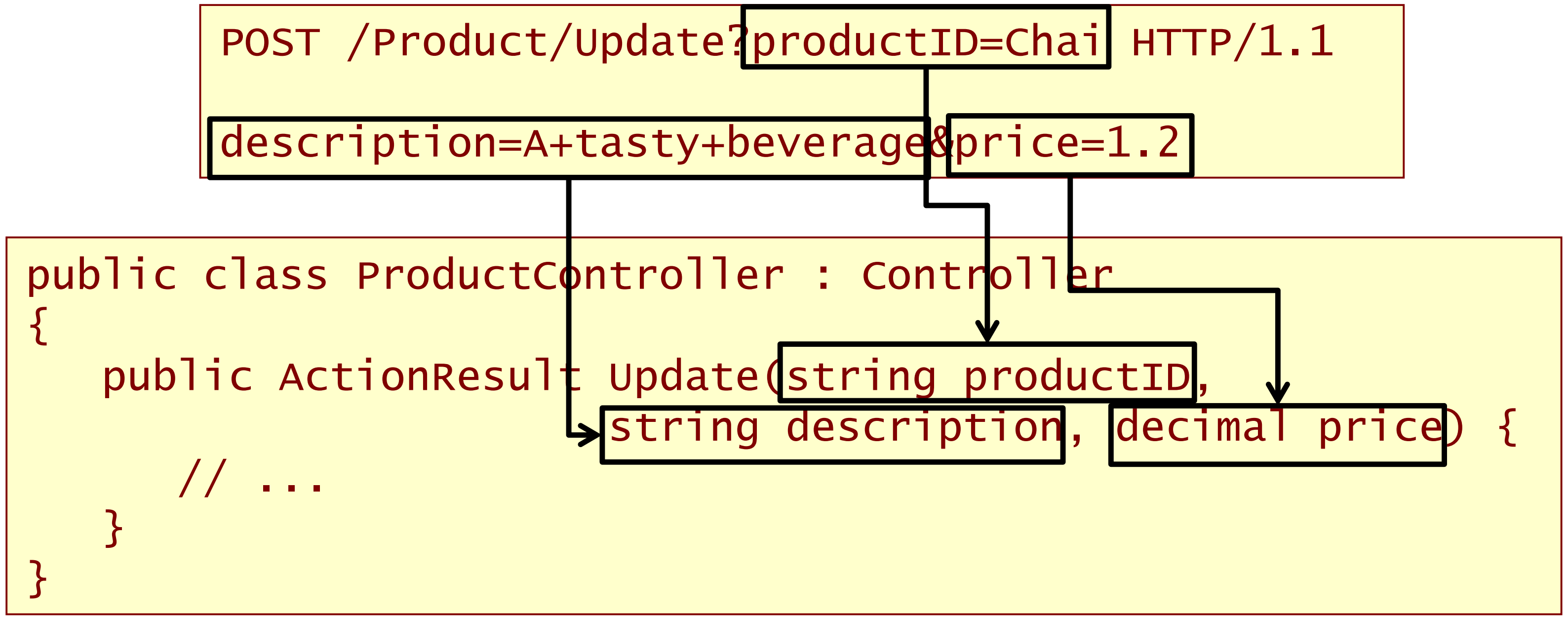
```
~/Product/ShowAll
~/Product/Edit/Chai
~/Product/Update/Chai
```

```
public class ProductController : Controller
{
   public ActionResult ShowAll() {
      // ...
   }
   public ActionResult Edit(string id) {
      // ...
   }
   public ActionResult Update(string id,
                        string description, decimal price) {
      // ...
   }
}
```



Controllers and input

- **Model binding maps inputs to action method parameters**
  - query string
  - POST data
  - route parameters
- **Input value must match name of action method parameter**



```
POST /Product/Update?productID=Chai HTTP/1.1

description=A+tasty+beverage&price=1.2
```

```
public class ProductController : Controller
{
   public ActionResult Update(string productID,
                          string description, decimal price) {
      // ...
   }
}
```

Action results

- **Controller logic determines result sent to client**
  - action methods must return **`ActionResult`**
  - **`ActionResult`** is base class to emit response to client
    - derived **`ViewResult`** is most common
    - other derived action results possible
  - **`Controller`** provides helper APIs to create action results

```
public class ProductController : Controller
{
   public ActionResult ShowAll()
   {
      // ...
   }
}
```

ContentResult

- **`ContentResult` and `Controller.Content` return string**
  - string is passed to **`Response.Write`**

```
public class ProductController : Controller
{
    public ActionResult ShowAll() {
        return Content("<h1>This is all!</h1>");
    }
}
```

RedirectResult

- **`RedirectResult` returns 302 HTTP status code**
  - ctor accepts URL
- **`RedirectToAction` helper to redirect to controller/action**
  - pass action and controller names

```
public class ProductController : Controller
{
   public ActionResult Edit(int id)
   {
      Product p = Products.Load(id);

      if (p == null)
         return RedirectToAction("Index", "Product");

      // ...
   }
}
```

ViewResult

- `ViewResult` **and** `Controller.View` **indicate view to render**
  - pass name of view to render

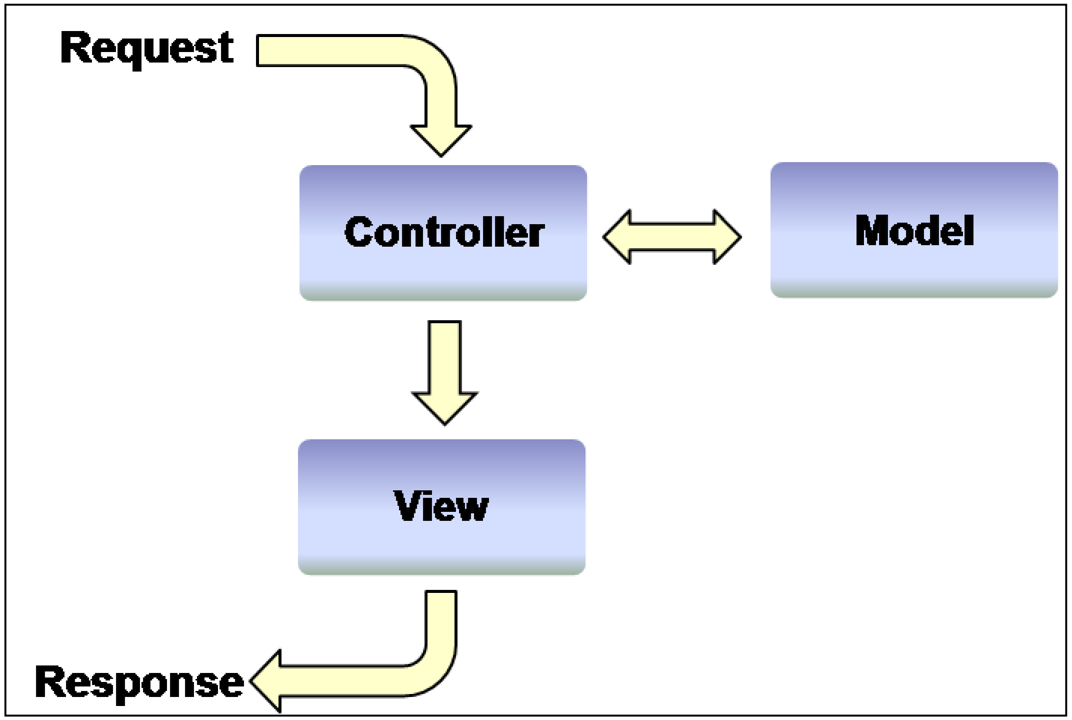
```
public class ProductController : Controller
{
   public ActionResult ShowAll() {
      return View("ShowAll");
   }
}
```

  - if no name chosen then view with same name as action used

```
public class ProductController : Controller
{
   public ActionResult ShowAll() {
      return View();
   }
}
```

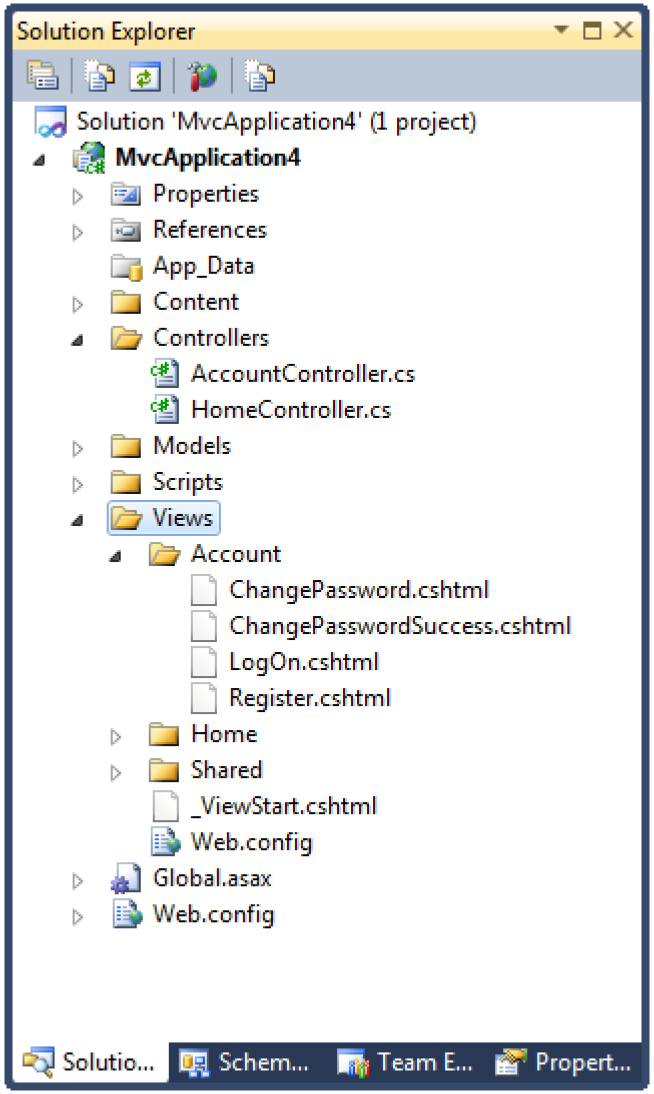
Views

- **Templates which generate content to return to client**
  - contains markup and code for rendering logic
  - typically accept data from controller



Razor view engine

- **MVC uses Razor view engine**
  - new in MVC3
  - contains streamlined syntax for code and markup
- **Views reside in "Views" directory**
  - subdirectories for each controller
  - view files named for corresponding controller action
- **Other view engines possible**
  - WebForms used prior to MVC3



Razor views

- **Views are templates for dynamically rendering markup**
  - `.cshtml` or `.vbhtml` files contain markup and code
    - file extension controls programming language
  - `@expression` syntax used for emitting dynamic values
  - `@statement` used for control flow constructs
  - `@{ code }` used for multi-line statements or code blocks

```
<h1>Hello it is @DateTime.Now</h1>
<ul>
   @for (int i = 0; i < 5; i++) {
      <li>The value is: @i</li>
   }
</ul>
@{
   int x = 1;
   int y = 2;
   int z = x + y;
}
<h2>The sum is: @z</h2>
```

Passing dynamically typed data to view

- `Controller.ViewData` **property holds data to pass to view**
  - supports dictionary-like access (key/value)

```
public class ProductController : Controller
{
   public ActionResult ShowAll()
   {
      Product[] list = Products.GetAll();

      // pass data via dictionary
      ViewData["Products"] = list;

      return View();
   }
}
```

Accessing dynamically typed data in view

- `ViewData` **provides access to data from controller**
  - `ViewData` implements indexer for dictionary data
  - must downcast values

```
<ul>
@foreach (var prod in (Product[])ViewData["Products"]) {
   <li>@prod.ProductName</li>
}
</ul>
```

Passing dynamically typed data to view

- `Controller.ViewBag` **property can pass data to view**
  - `dynamic` property for data
  - wrapper for `ViewData` dictionary

```
public class ProductController : Controller
{
   public ActionResult ShowAll()
   {
      Product[] list = Products.GetAll();

      // pass data via dynamic
      ViewBag.Products = list;

      return View();
   }
}
```

Accessing dynamically typed data in view

- **`ViewBag` provides access to data from controller**
  - `ViewBag` is `dynamic` property
  - wrapper for `ViewData` dictionary

```
<ul>
@foreach (var prod in ViewBag.Products) {
   <li>@prod.ProductName</li>
}
</ul>
```

Passing statically typed data to view

- `ViewData.Model` **holds strongly typed data to pass to view**
    - `Controller.View` method accepts model as parameter

```
public class ProductController : Controller
{
    public ActionResult ShowAll()
    {
        Product[] list = Products.GetAll();

        ViewData.Model = list;

        return View();
    }
}
```

```
public class ProductController : Controller
{
    public ActionResult ShowAll()
    {
        Product[] list = Products.GetAll();

        return View(list);
    }
}
```

Accessing statically typed data in view

- **`Model` property provides strongly typed data from controller**
  - defined by using `@model` ***`ModelType`*** directive

```
@model IEnumerable<Product>

<ul>
@foreach (var prod in Model) {
   <li>@prod.ProductName</li>
}
</ul>
```

Rendering common markup

- **Common snippets of markup tedious to code manually**
  - **`HtmlHelper`** class generates common HTML
    - mostly via extension methods
  - accessible from **`Html`** property in view

```
<body>
   @using (Html.BeginForm("Update", "Product")) {
   <fieldset>
      <legend>Product ID: @Model.ProductID</legend>
      <p>
         <label for="ProductName">Name:</label>
         @Html.TextBox("ProductName", Model.ProductName)
      </p>
      <p><input type="submit" value="Save Changes" /></p>
   </fieldset>
   }
   <p>@Html.ActionLink("View All Products", "Index", "Product")</p>
</body>
```

Html.ActionLink

- **Renders `<a>`**
  - **`Html.ActionLink("text", "action", "controller")`**
  - URL generated from routing

```
@Html.ActionLink("View All Products", "Index", "Product")

<a href='/Product/Index'>View All Products</a>
```

- **Parameters passed via anonymous type**
  - property names become parameter names
  - passed as query string or routing parameters

```
@Html.ActionLink("View Chai",
   "View", "Product", new { id = 1,  edit = true })

<a href='/Product/View/1?edit=true'>View Chai</a>
```

Html.TextBox

- **Renders `<input type='text'>`**
  - **`Html.TextBox("name", *value*)`**
    - "name" parameter designates **`name`** and **`id`** attributes

```
@Html.TextBox("productName", Model.ProductName)

<input type='text' name='productName'
       id='productName' value='Chai' />
```

- **HTML attributes passed via anonymous type**
  - property names become attributes

```
@Html.TextBox("productName", Model.ProductName,
               new { maxlength=10 })

<input type='text' name='productName'
       id='productName' value='Chai' maxlength='10' />
```

Html.BeginForm

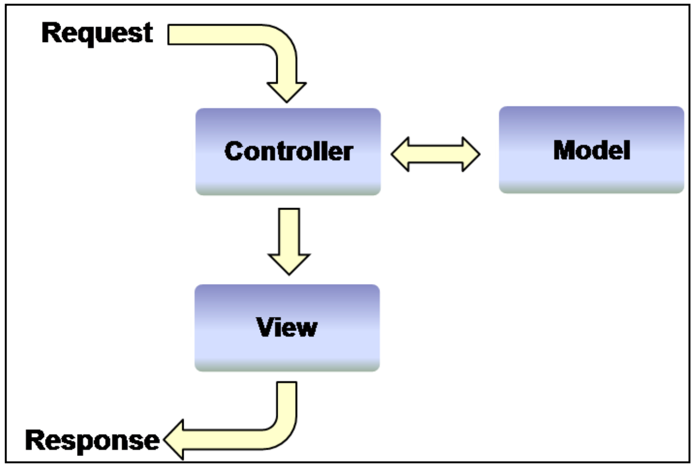
- **Renders `<form>`**
  - `Html.BeginForm("action", "controller")`
    - use with `using` to emit `</form>`

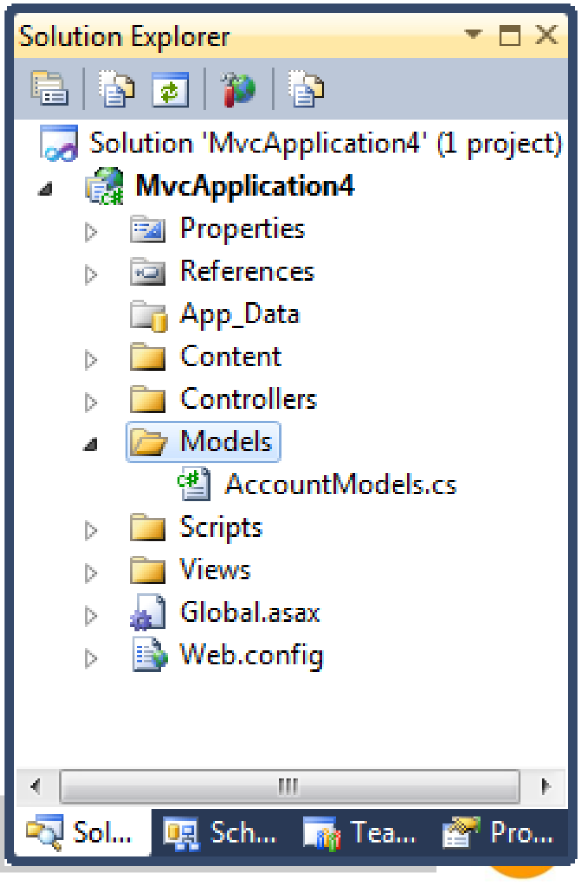
```
@using(Html.BeginForm("Update", "Product"))
{
    ...
}

<form action="/Product/Update" method="post">
    ...
</form>
```

Models

- **Model is an overloaded term**
  - can be business logic and data access
  - can be object model passed from controller to view
  - can be object model to describe inputs
- **Proper design facilitates unit testing and/or TDD**
  - repository/service pattern
  - dependency injection/inversion of control





Summary

- ASP.NET MVC is an alternative approach to WebForms
- MVC promotes deliberate separation of concerns
- MVC promotes maintainable software
- MVC allows high degree of control over markup