ASP.NET MVC

MVC

- MVC separates the user interface of an application into three main aspects:
  - The Model
  - The View
  - The Controler

MVC

- Model
  A set of classes that describes the data you're working with as well as the business rules for how the data can be changed and manipulated

- View
  Defines how the application's UI will be displayed

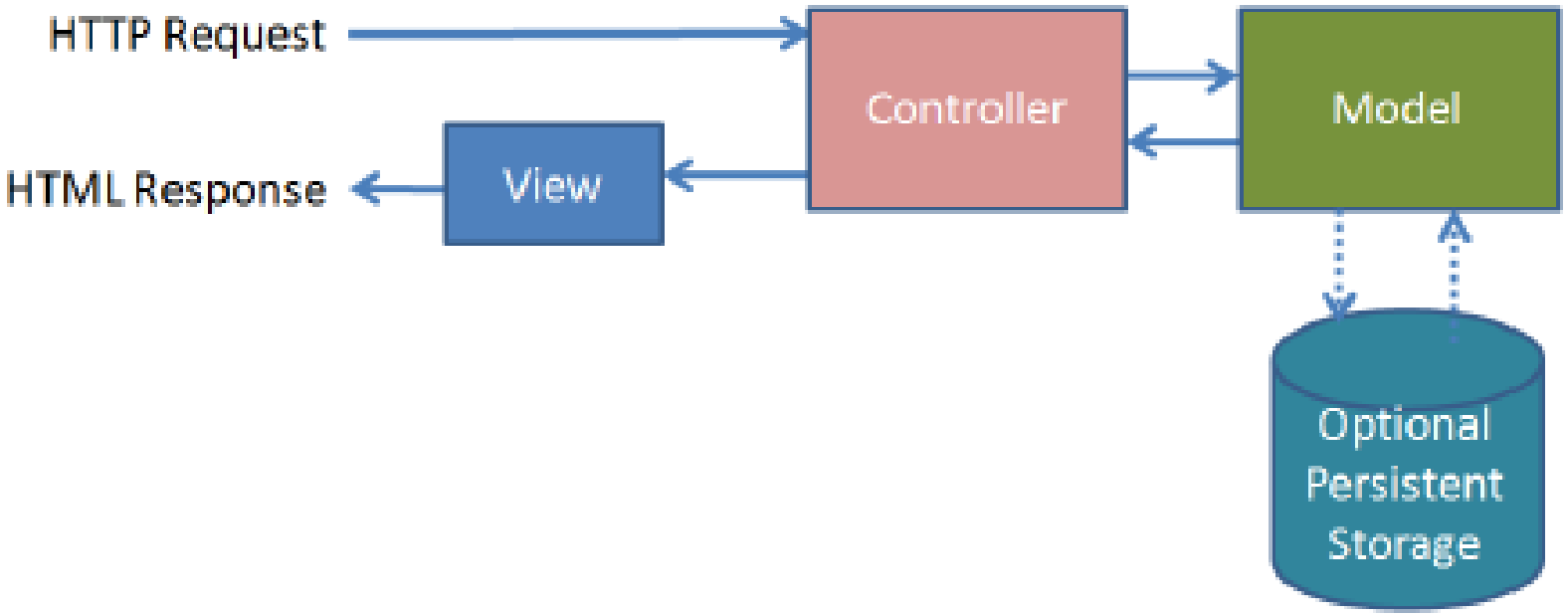
- Controller
  A set of classes that handles communication from the user, overall application flow, and application-specific logic.

MVC as Applied to Web Frameworks

- Models:
  - Represent domain objects that encapsulate:
    - data stored in a database
    - code to manipulate the data
    - code to enforce domain-specific business logic
    - Example: encapsulating Entity Framework

MVC as Applied to Web Frameworks

- View:
  - Template to dynamically generate HTML
- Controller:
  - Class that manages the relationship between the View and the Model
  - It responds to user input, talks to the model, and decides which view to render (if any).


HTTP Request
HTML Response
View
Controller
Model
Optional
Persistent
Storage

A Pattern

- MVC is a pattern that can be applied to different frameworks

- ASP.NET MVC
  - The MVC pattern has been applied to ASP.NET. This does not mean that it is the same as MVC running in other places.

ASP.NET MVC 3

- 10 months after MVC 2
  - The Razor view engine
  - Support for .NET 4 Data Annotations
  - Improved model validation
  - Greater control and flexibility with support for dependency resolution and global action filters
  - Better JavaScript support with unobtrusive JavaScript, jQuery Validation, and JSON binding
  - Use of NuGet to deliver software and manage dependencies throughout the platform

Razor View Engine

- Code-focused templating for HTML generation
- Compact, expressive, and fluid
- Not a new language
- Easy to learn
- Works with any text editor
- IntelliSense
- No XML-like heavy syntax

ASP.NET MVC 4

- Features include:
  - ASP.NET Web API
  - Enhancements to the default project templates
  - Mobile project template using jQuery Mobile
  - Display Modes
  - Task Support for Asynchronous Controllers
  - Bundling and Minification

ASP.NET Web API

- Referred to as Web API
- A framework that offers the ASP.NET MVC development style but is tailored to writing HTTP services. (service-oriented design)
- Several MVC features have been adopted for HTTP Service domain:
  - Routing
  - Model Binding and Validation
  - Filters
  - Scaffolding
  - Easy Unit Testability

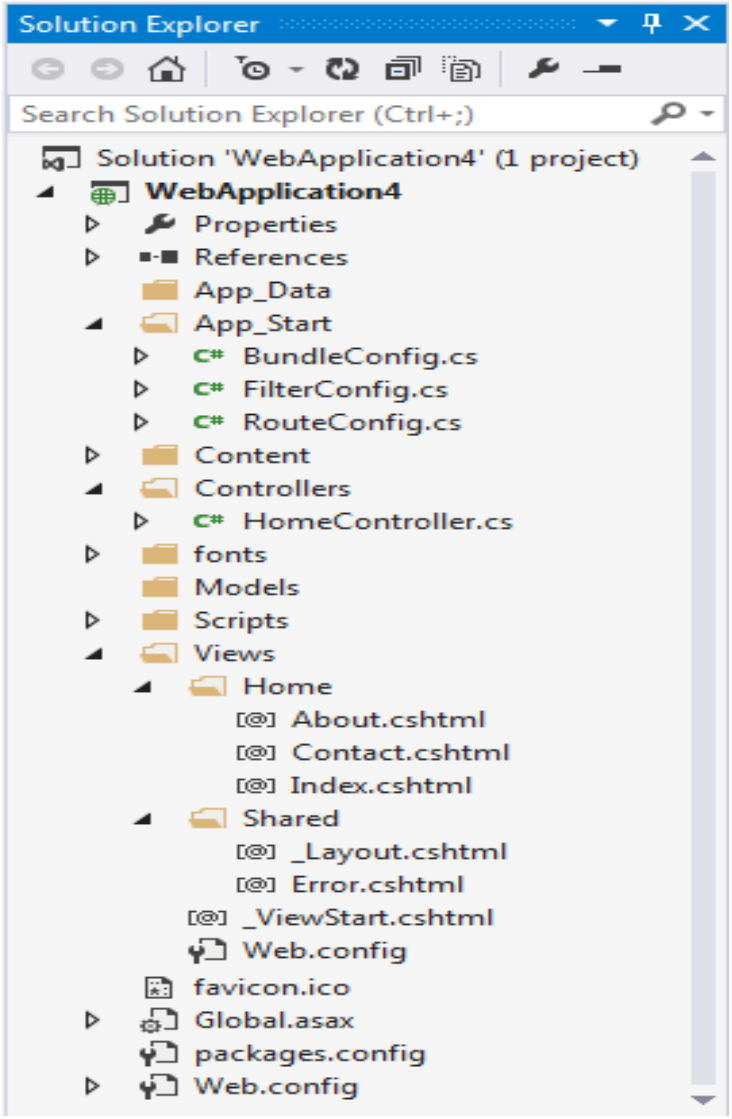
ASP.NET MVC 5

- http://www.asp.net/mvc/mvc5

Layout of an MVC project

When you create a new MVC project, your solution should have the following structure in your Solution Explorer.

Layout of an MVC project



Layout of an MVC project

- **App_Data** –it's meant to hold data for your application (just as the name says). A couple of examples would include a portable database (like SQL Server Compact Edition) or any kind of data files (XML, JSON, etc.). I prefer to use SQL Server.
- **App_Start** – The App_Start folder contains the initialization and configuration of different features of your application.
  - BundleConfig.cs – This contains all of the configuration for minifying and compressing your JavaScript and CSS files into one file.
  - FilterConfig.cs – Registers Global Filters.
  - RouteConfig.cs – Configuration of your routes.

  There are other xxxxConfig.cs files that are added when you apply other MVC-related technologies (for example, WebAPI adds WebApiConfig.cs).
- **Content** – This folder is meant for all of your static content like images and style sheets. It's best to create folders for them like "images" or "styles" or "css".

Layout of an MVC project

- **Controllers** – The controllers folder is where we place the controllers.
- **Models** – This folder contains your business models. It's better when you have these models in another project, but for demo purposes, we'll place them in here.
- **Scripts** – This is where your JavaScript scripts reside.
- **Views** – This parent folder contains all of your HTML "Views" with each controller name as a folder. Each folder will contain a number of cshtml files relating to the methods in that folder's controller. If we had a URL that looked like this:

  http://www.xyzcompany.com/Products/List

  we would have a Products folder with a List.cshtml file. We would also know to look in the Controllers folder and open the ProductsController.cs and look for the List method.

Layout of an MVC project

- **Views/Shared** – The Shared folder is meant for any shared cshtml files you need across the website.
- **Global.asax** – The Global.asax is meant for the initialization of the web application. If you look inside the Global.asax, you'll notice that this is where the RouteConfig.cs, BundleConfig.cs, and FilterConfig.cs are called when the application runs.
- **Web.Config** – The web.config is where you place configuration settings for your application. For new MVC developers, it's good to know that you can place settings inside the <appsettings> tag and place connection strings inside the <connectionstring> tag.

Controller

- “Convention over configuration”
- ASP.NET MVC is heavily convention-based
  - Uses Directory-naming structure when resolving view templates.
    - Allows you to omit the location path when referencing views from within a Controller class.

View

- Each controller's class name ends with *Controller*
  - *LegalController, LegalServicesController, HomeController*
- One Views directory for all the views of your application
- Views that controllers use live in a subdirectory of the Views main directory and are named according to the controller name(minus the *Controller* suffix)
  - Views for the LegalController would live in /Views/Legal
- Reusable UI elements live in a similar structure, but in a Shared directory in the Views folder

Controller Basics

- Model View Controller:
  - 1st Controller basics, then the other stuff.
  - Have a look at the HomeController
    - Responsible for deciding what will happen when you browse to the homepage of the website.

The Controller's Role

- Responsible for responding to user input, making changes to the model in response to user input if needed.
- **Controllers** are concerned with the **flow of the application**, working with data that comes in and providing data going out to the relevant view.

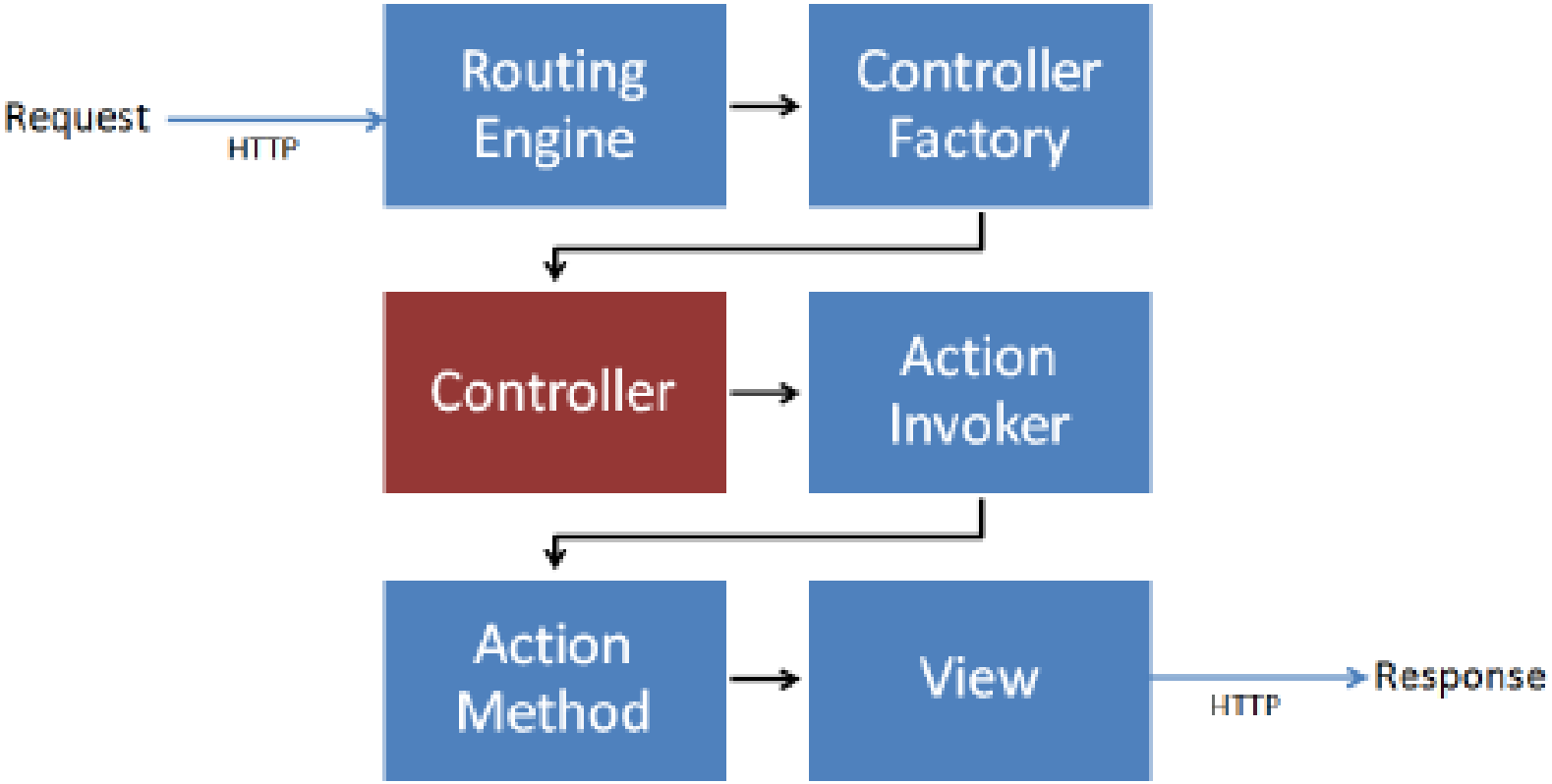
The Controller's Role

- The URL tells the **routing mechanism** (later chapter) which **controller class** to instantiate and which action method to call, and supplies the required arguments to that method.
- The **controller's** method then decides which **view** to use, and that **view** then renders the HTML.
- The **controller's** method could return something other than a **view**.

The Controller's Role

- There is a **relationship** between the **URL** and the **METHOD** on a controller class. **Not** a relationship **between** the **URL** and a **FILE** on the web server.
- MVC serves up the results of method calls, not dynamically generated pages.
- (More about routing in later)

MVC Request processing pipeline



Modifying the LegalServicesController

- The Browse and Details methods to cover additional scenarios.

```
public class LegalServicesController : Controller
{
    //
    // GET: /LegalServices/

    public string Index()
    {
        return "What up yo?";
    }


    //
    // GET: /LegalServices/Browse

    public string Browse()
    {
        return "Browsing Legal options yo";
    }


    //
    // GET: /LegalServices/Details/{ID}

    public string Details(int ID)
    {
        return "Looking at details for " + ID;
    }
```

Observation

- Browsing to /LegalServices/Browse caused the browse method of the LegalServicesController to be executed.
  - NO extra configuration was needed.
  - This is routing in action. More later.
- We returned text to the browser without a **model** or a **view.**

Oversimplified

- The previous example was not common and was oversimplified.
  - You will almost always use Views
  - You will most often return ActionResult instead of strings
  - And routing is more complex that shown.

Mapping controller

- Controller selection based on URL
- Default URL routing logic:/[Controller]/[ActionName]/[Parameters]
- Format for routing in*App_Start/RouteConfig.cs*

```
public static void RegisterRoutes(RouteCollection routes)
{
    routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

    routes.MapRoute(
        name: "Default",
        url: "{controller}/{action}/{id}",
        defaults: new { controller = "Home", action = "Index", id = UrlParameter.Optional }
    );
}
```

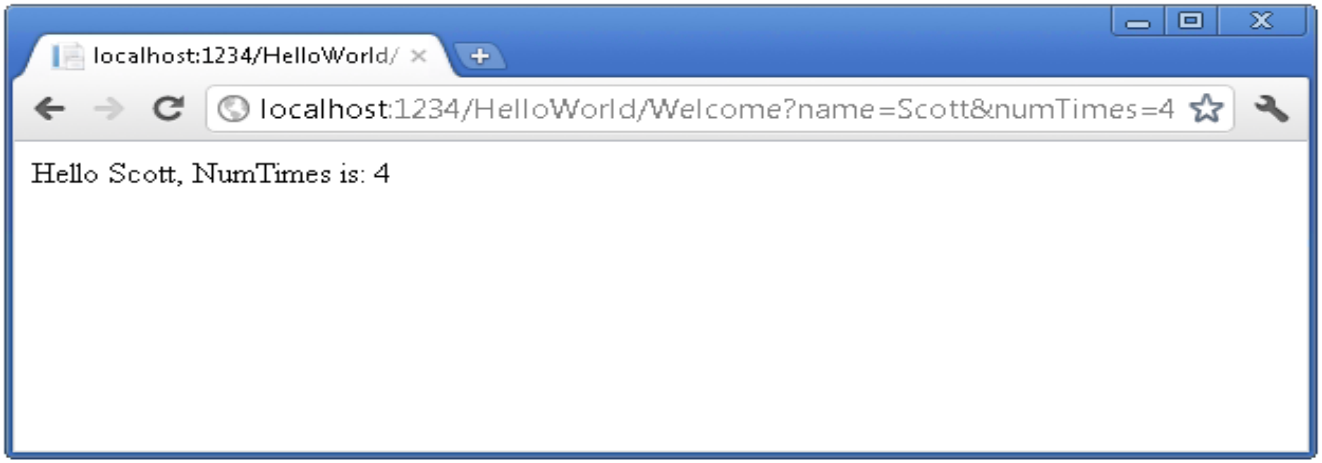
URL routing

- WebappURL without URL segments => HomeController::Index()
- Index() –default method of a controller
- /HelloWorld=> HelloWorldController
- /HelloWorld/Index => HelloWorldController::Index()
- http://webapp:port/HelloWorld/Welcome =>HelloWorldController::Welcome()

Parameters

- /HelloWorld/Welcome?name=Scott&numtimes=4
- Introducing 2 parameters to Welcome method
- Parameters passed as query strings!

```
public string Welcome(string name, int numTimes = 1) {
     return HttpUtility.HtmlEncode("Hello " + name + ", NumTimes is: " + numTimes);
}
```



URL Parameters

- http://webapp/HelloWorld/Welcome/3?name=Rick

```
public string Welcome(string name, int ID = 1)
{
    return HttpUtility.HtmlEncode("Hello " + name + ", ID: " + ID);
}
```


```
public static void RegisterRoutes(RouteCollection routes)
{
    routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

    routes.MapRoute(
        name: "Default",
        url: "{controller}/{action}/{id}",
        defaults: new { controller = "Home", action = "Index", id = UrlParameter.Optional }
    );
}
```

Basic Controllers Summary

- Controllers orchestrate the interactions of the user, the model objects and the views.
- They are responsible for:
  - responding to the user input
  - Manipulating the appropriate model objects
  - And then selecting the appropriate view to display back to the user in response to the initial input

ActionResult

- Before talking about Views, let's look at ActionResults

ActionResult type

ActionResult Type	Helper Method	Description
ContentResult	Content()	Returns raw textual data to the browser, including strings, XML, comma-separated values, etc.
EmptyResult	--	Returns nothing to the browser.
FileResult	File()	Abstract base class for sending a file to the browser. Derived classes can send the file, the contents of the file, or a file stream.
HttpUnauthorizedResult	--	Returns an HTTP 401 Unauthorized Status error as the result of an unauthorized HTTP request.
JavaScriptResult	JavaScript()	Returns JavaScript commands to the browser, usually in response to an Ajax request from the browser.
JsonResult	Json()	Specialized form of ContentResult to return JavaScript Object Notation data, a lightweight, textual data format.
PartialViewResult	PartialView()	Similar to ViewResult, but sends a snippet of HTML rather than a full page, usually in response to an Ajax request.
RedirectResult	Redirect()	Rather than generating a response itself, the action method redirects to another URL entirely using an HTTP 302 status code.
RedirectToRouteResult	RedirectToRoute() RedirectToAction()	Redirects either to a specific route defined in the application or to another action method, using an HTTP 302 status code. Provides flexible options for navigating to different parts of the application.
ViewResult	View()	Returns a full HTML Web page to the user. Probably the result type you'll use most often.

The View

- The user's first impression of your site starts with the View
- Responsible for providing the user interface to the user.
- The view transforms a model into a format ready to be presented to the user.
- The view examines the model object and transforms the contents to HTML

The View

- Not all views render HTML, but HTML is the most common case
- More info on alternate types of content later.
- Views created using Razor view engine
- Controller method returns View object
- Controller method return type is ActionResult

Razor View Engine

- What is Razor?
  - Introduced in ASP.NET MVC 3 and is the default view engine moving forward
  - Provides a clean, lightweight, simple view engine.
  - Provides a streamlined syntax for expressing views
  - Minimizes the amount of syntax and extra characters.

Example

```
public ActionResult Details(int ID)
{
    ViewBag.Message = ID;
    return View("Details");
}
```

```
@{
    ViewBag.Title = "Details";
    Layout = "~/Views/Shared/_Layout.cshtml";
}

<h2>Details</h2>
<div>
    @ViewBag.Message
</div>
```

Convention

- The **Views** directory contains a folder for your controller, with the same name as the controller, but without the *controller* suffix.
- **HomeController** has views in the **Home** directory
- Within the view folder (also called controller folder) there's a view file for each action method, named the same as the action method.
- This is how views are associated to an action method.
- An **action method** can return a **ViewResult** via the **View()** method.

View() Method

- When the view name isn't specified, the ViewResult returned by the action method applies a convention to locate the view.
  - It first looks for a view with the same name as the action within the /Views/ControllerName directory. "Ex: /views/home/Index.cshtml"
- The View() method is overloaded. You can supply a view name to render a different view. "ex: View("MyView")"
  - It will still look in the same directory, but will look for the "MyView.cshtml" file.
- View("~/Views/DifferentDirectory/Index.cshtml")
  - To get to a different directory, provide the full path to that directory. You must provide the file extension as well. This bypasses the internal lookup mechanism.