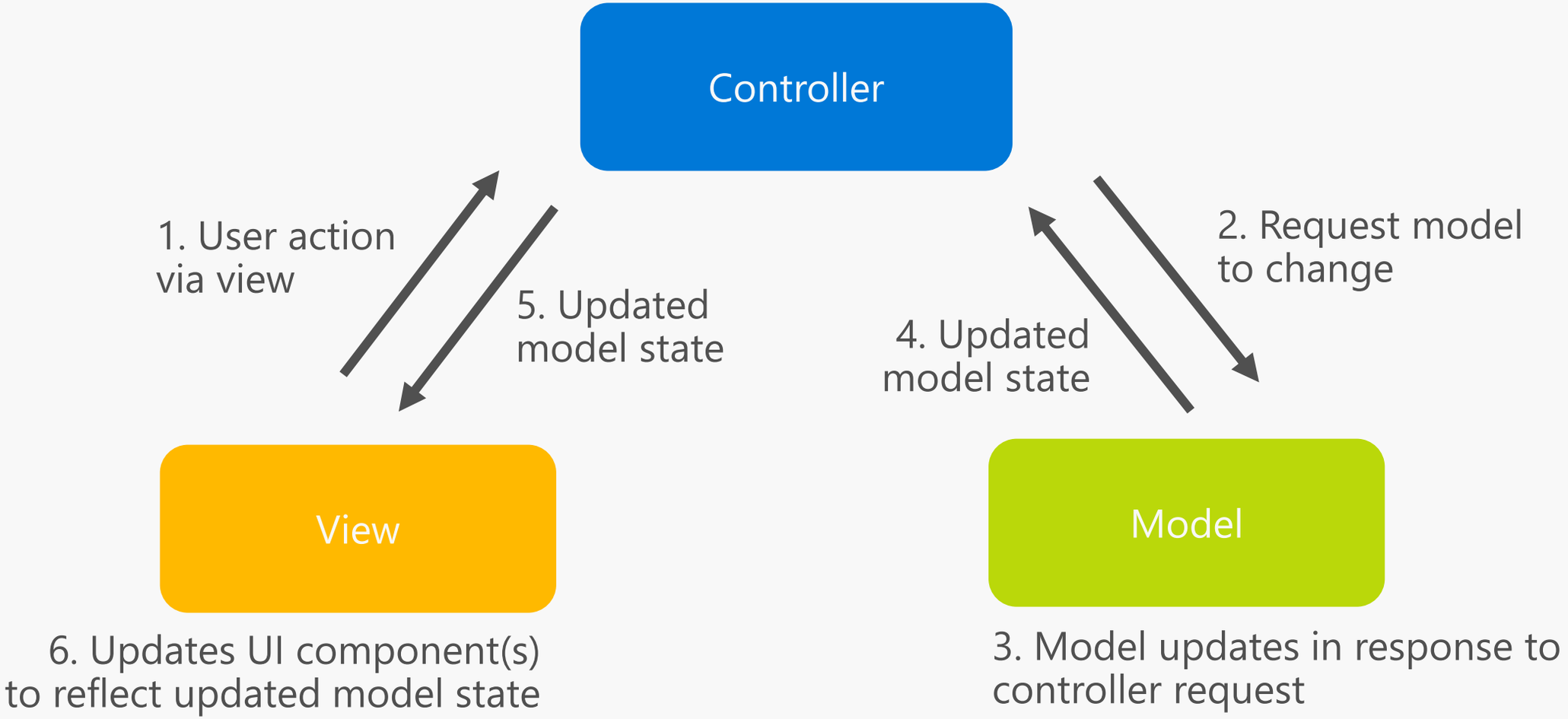
Microsoft

# Model View Controller (MVC) Design Pattern

Crystal Tenn
Crystal.Tenn@microsoft.com

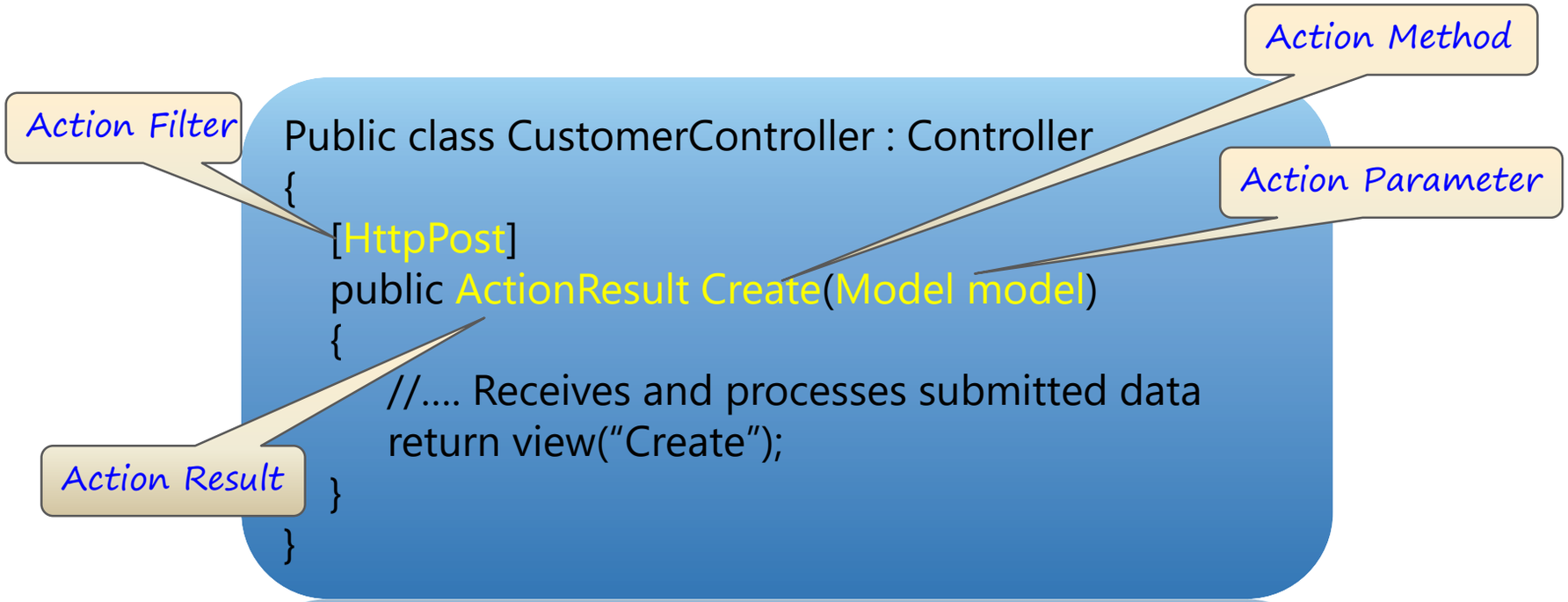
# What is the Model View Controller (MVC) design pattern?



# Controller

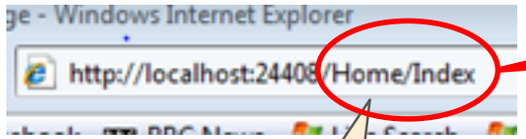
# What is a Controller?

- Simple .NET class that *orchestrates* entire request
- Exposes 1-to-many public methods called *Action Methods…*
- Each Action Method returns an *Action Result…* (HTML, JSON, string, file, etc.)



# Invoked from Routing Engine

- Controllers/Actions are URL-Addressable…
  - Routing engine parses URL: /home/index
  - Maps to corresponding controller and action method
  - Returns an "action result"




Both controller and action are URL Addressable

```
public class HomeController : Controller
{
    public ActionResult Index()
    {
        ViewBag.Message = "Welcome!";
        return View();
    }

    public ActionResult Retrieve(int id)
    {
        // Code that retrieves customer by id
        return View("Retrieve", dataFile);
    }
}
```

# Action Methods

- Controller exposes 1 to many *Action Methods*
  - Operations that controller can perform
  - Each method corresponds to an operation

```
public class CustomerController : Controller
   {
      public ActionResult Get(int Id)
      public ActionResult GetAll()
      public ActionResult Add(Customer customer)
      public ActionResult Edit(Customer customer)
      public ActionResult Delete(Customer customer)
   }
```

- Each Action Method returns *Action Result* type

# Action Result

- Base type for variety of common return types
- Action method returns required child action result

| Derived Type | Behavior | Payload |
|---|---|---|
| ViewResult<br>PartialViewResult | View engine prepares response | HTML |
| ContentResult | Renders raw content | String |
| FileContentResult | Returns file content | File |
| JsonResult | Renders data as Json | JSON |
| JavaScriptResult | Returns script to execute | JavaScript |
| RedirectResult | Redirects to another URL | Redirect |
| HttpNotFoundResult | Returns HTTP 404 status code | 404 Code |
| HttpUnauthorizedRequest | Returns HTTP 403 status code | 403 Code |
| EmptyResult | No response | Nothing |

# Action Selectors

- Updateable views expose multiple ActionMethods
- Differentiate with HTTP Accept Verbs (Filters)
- Specifies ActionMethod a controller should invoke
  - HTTP GET: *Retrieves* data only (*read only*)
  - HTTP POST: *Submits* data to server (*changes state* on server)

```
[HttpGet]
public ActionResult Create()
{
    //.... Returns blank form for input (think of as read-only operation)
 }

[HttpPost]
public ActionResult Create(Model model)
{
    //.... Receives and processes submitted data
 }
```

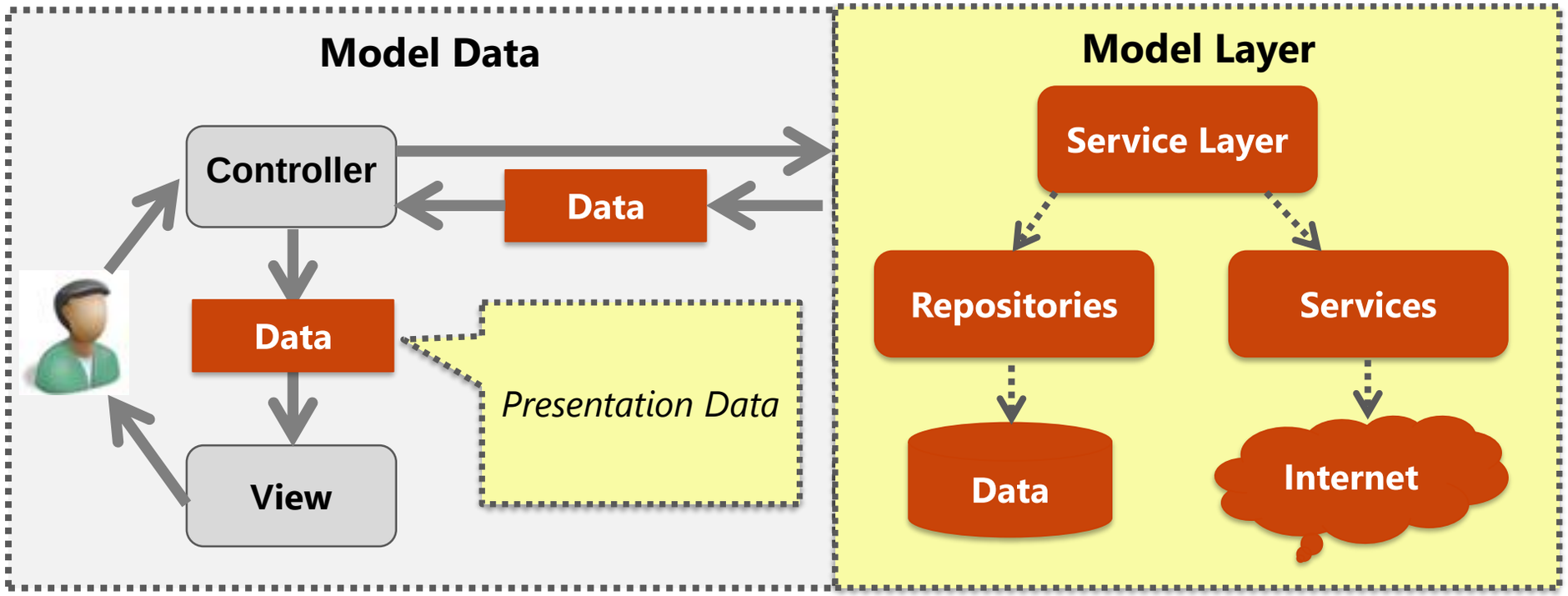
*Action method invoked for Get operation*

*Action method invoked for Post operation*

# Model

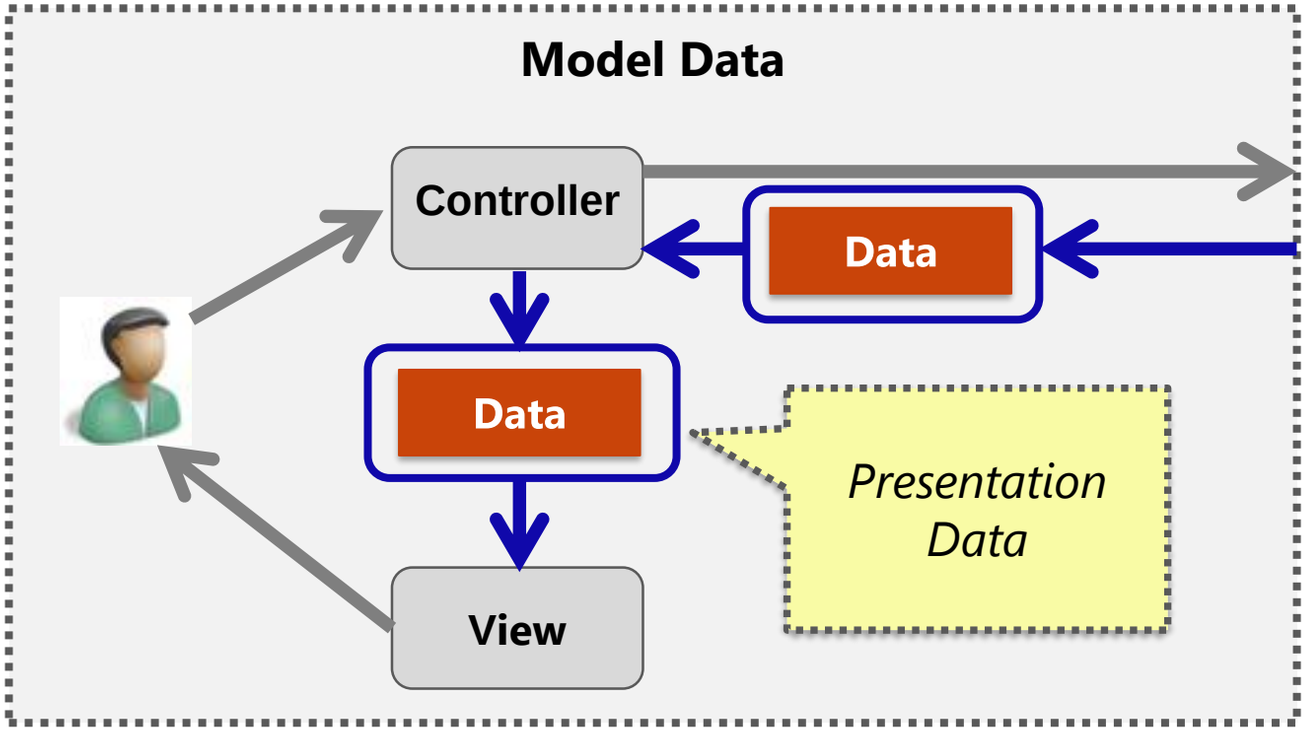
# What is the Model?

- *Overloaded term* -- refers to…
  - *Model Layer* – Service layers (BAL, DAL, DB) that provide compute, business and data storage operations
  - *Model Data* – Domain data object returned from operation and passed to View for presentation



# Presentation Model

- For now…
  - Think about the model as a *data object*
  - Passed back from the *service layer*
  - Passed into *view*
  - With view, *strongly type* to model



# Loosely-Coupled Approach

- Controller can pass small amounts of data to View
- Assign data in Controller and access in View
  - *ViewData* – Dictionary of objects
  - *ViewBag* – Dynamic object

## ViewData

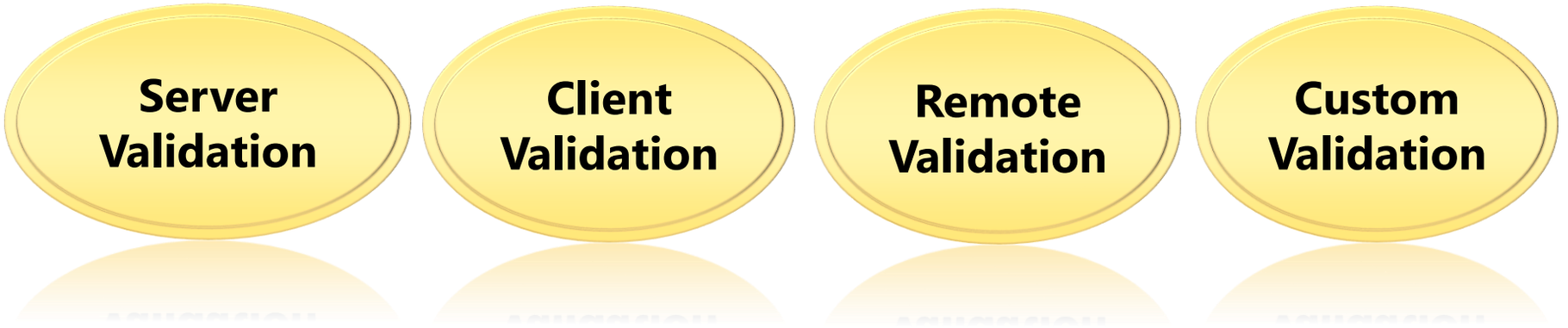
- Name/Value Dictionary
- No compile time check
- Key Names are strings
- Requires casting/boxing

## ViewBag

- Supports *dynamic* types
- Predefined classes not needed
- No compile time check
- No IntelliSense
- Only supported in .NET 4

# MVC Input Validation

- Exposes *built-in* layer of *declarative validation services*
- Leverage *without writing any code*
- Centralize by decorating *model* with validation attributes…
  - Validation code automatically generated…
  - Both client- and server-side…
- Built-in remote validation plumbing
- Extensible – *"roll your own"* custom validation services



# Validation by Annotation

- Add by applying built-in validation *attributes* to model properties

```
public class StudentViewViewModel
{
    [Required(ErrorMessage = "First Name is required.")]
    [Display(Name = "Student First Name")]
    [MinLength(2, ErrorMessage = "First name must be at least 2 characters")]
    [MaxLength(35, ErrorMessage = "First name cannot exceed 35 characters")]
    public string FirstName { get; set; }
}
```

# Built-In Validation Types

- Large number of validation types are built into MVC framework:

| Attribute | Description |
|---|---|
| Required | Makes data item required |
| Display | Applies friendly name to labels |
| Min/Max Length | Applies max/min length to data |
| DisplayFormat | Applies DataFormatString to data |
| RegularExpression | Applies regular expression to data |
| DataType | Applies specific data type to data *(DataType.Password)* |
| ReadOnly | Makes data item read-only |
| AllowAnonymous | Negates [Authorize] and allows access – Login/Register |

# View

# What is a View?

- A *template* - contains no logic or computations
- Single responsibility: *Generating HTML*
- Combines markup/data, generates HTML

```
@model ArticleViewModel

<h2>Welcome to MVC Blog!</h2>
@foreach (var item in Model.Articles){
    <div>
          <h3>@item.Title</h3>
          <div>
            @item.Body
          </div>
          Posted:
             @item.Published.ToLongDateString()
             @item.Published.ToShortTimeString()
             @item.Author
             @item.CommentCount Comments
    </div>
  }
```

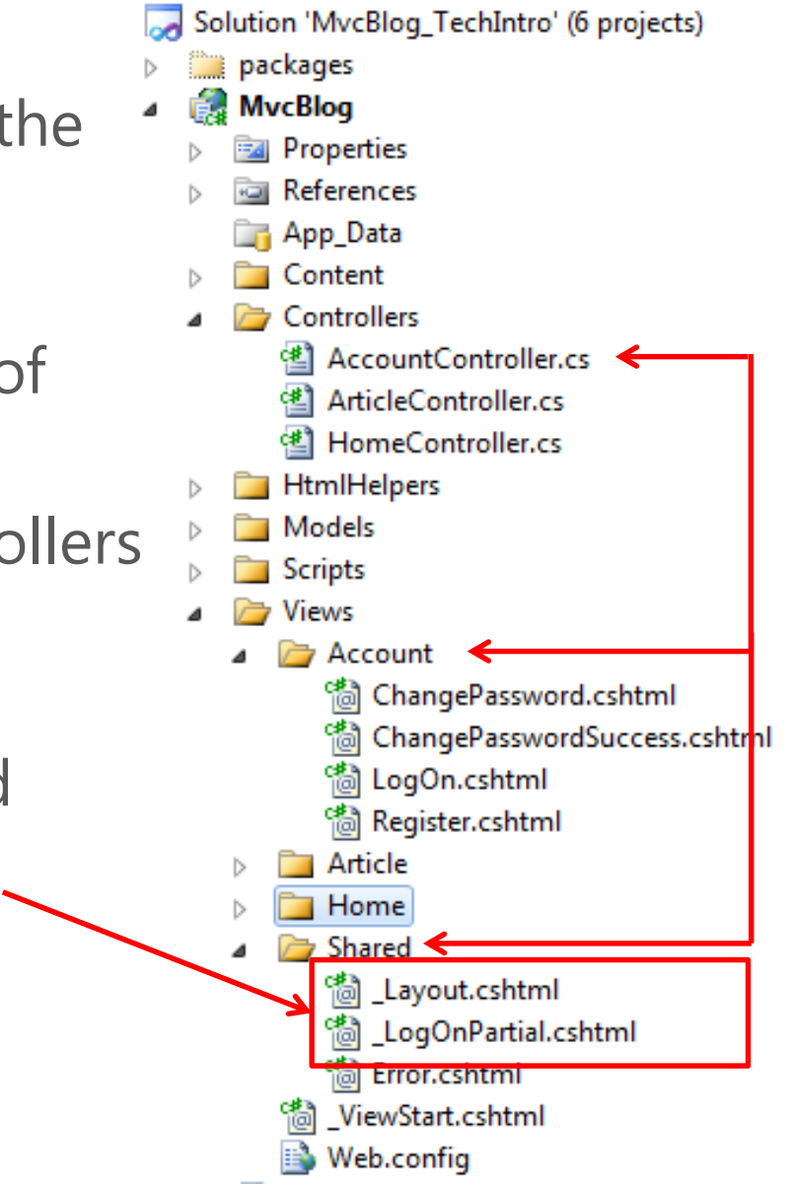
*@model directive: Specifies strongly-type model file*

*Mixes HTML and code*

*Data elements bind to properties from the model file*

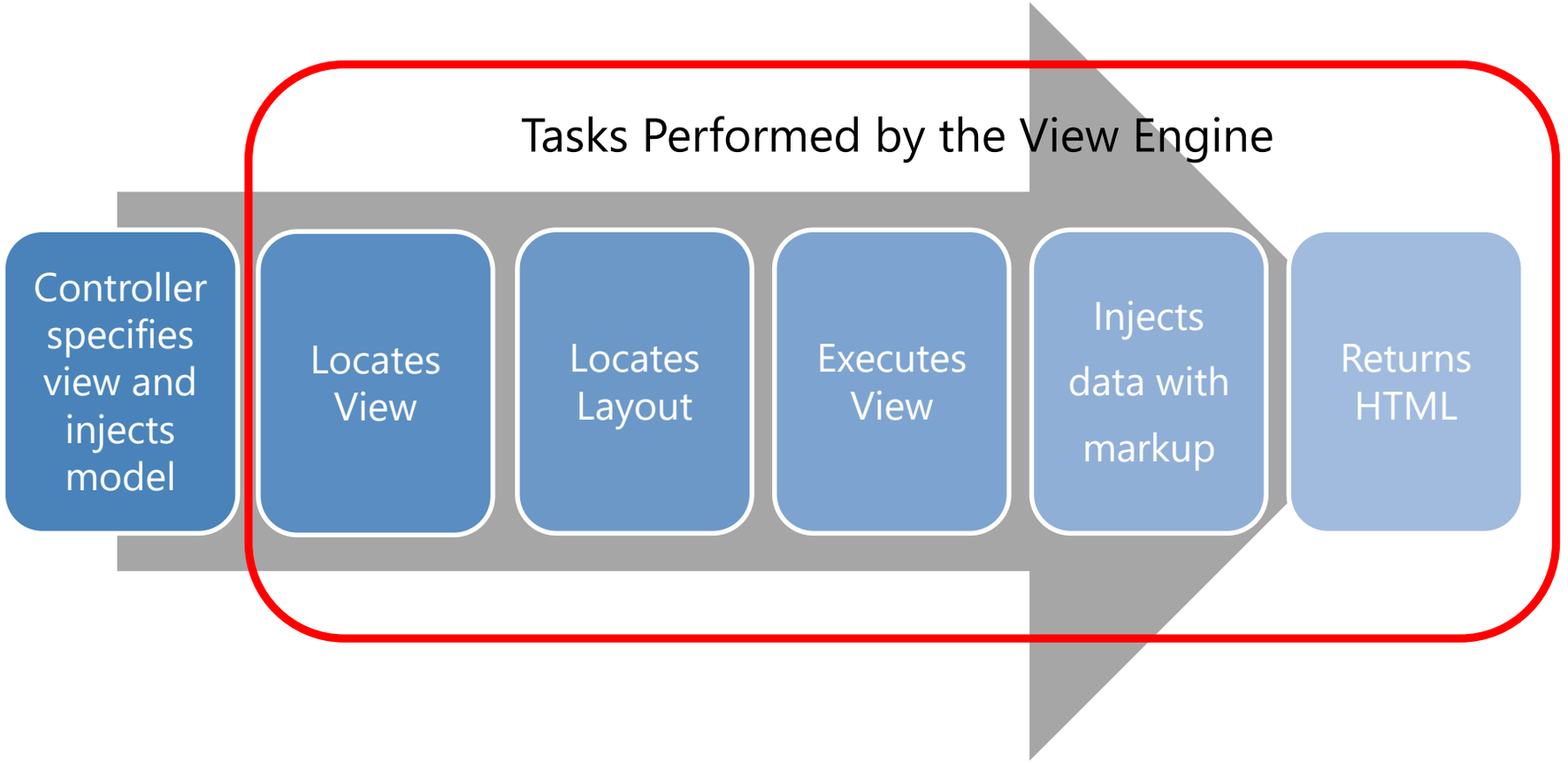
# View Conventions

- Views live in sub-folders, beneath the view folder, that correspond to controller names
- Each view corresponds to a name of an action method
- Views that are shared across controllers live in the *shared* directory
- View names that being with an underscore are *partial views* and must be called from within a view



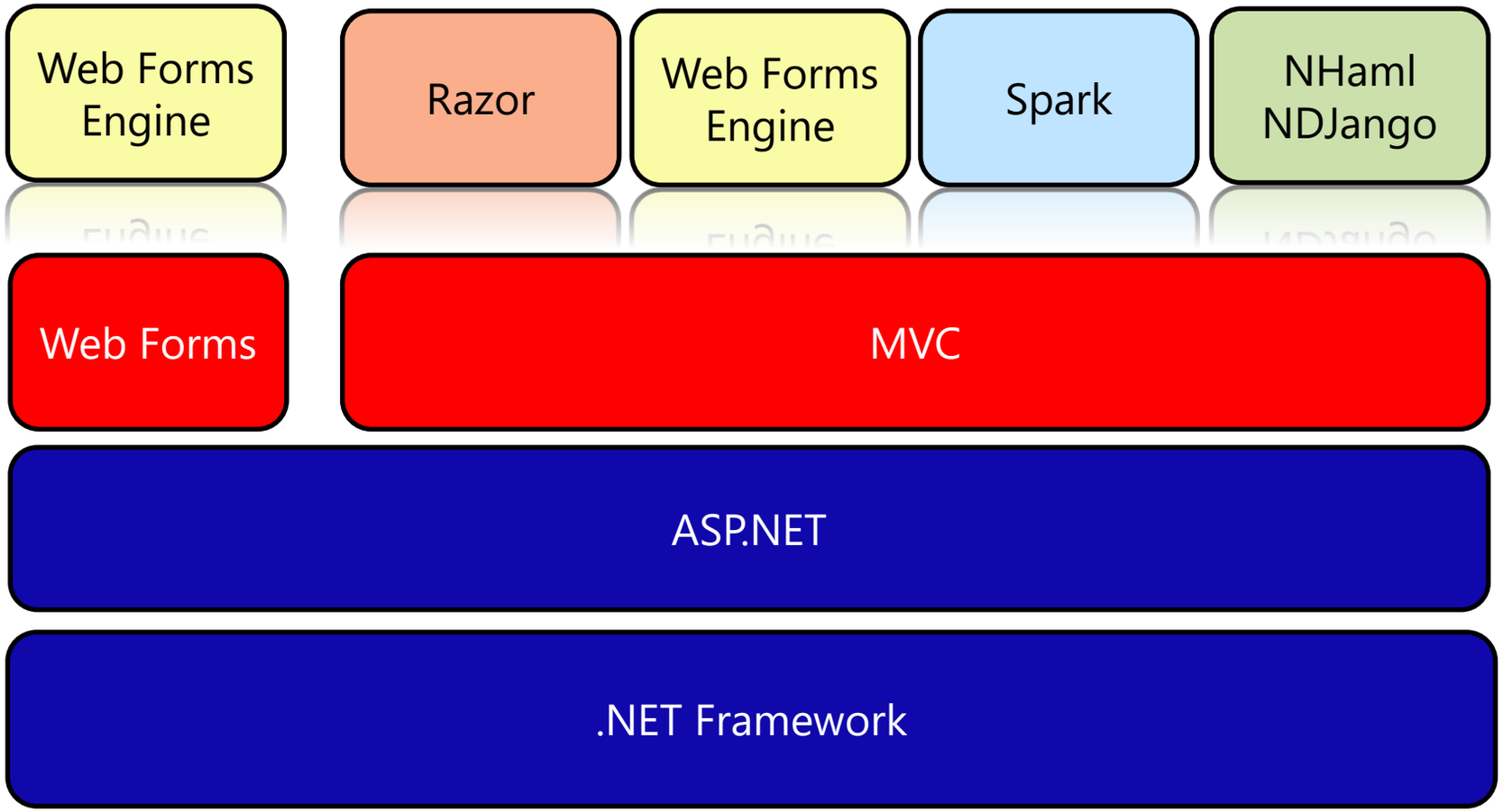
# What is a View Engine?

- Component that locates and parses a view template file, physically building output for the browser



# Pluggable View Engines

- View Engine is *pluggable component* in the MVC architecture



# Filters

# Action Filters

- Extensibility point…
  - Intercept the execution of an action
  - Inject behavior before or after code execution
  - Done with .NET Attributes
  - Similar to Aspect-Oriented programming
- Additional behavior, built-in or custom, attached to controllers or action methods that *declaratively* extend functionality without changing code…

# Filters

- Additional behavior, built-in or custom, that attach to controllers or action methods that *declaratively* extend functionality without changing code…
  - Authorization Filters
    - Make security decisions before executing action method code
  - Action Filters
    - Invoked before/after action method code execution
    - Inspect operation allowing for conditional or additional processing
  - Result Filters
    - Invoked before/after action result code execution
    - Inspect return values, cancel operation or perform additional processing
  - Action Selectors
    - Selects the action method that will be invoked
  - Exception Filters
    - Executes when unhandled exception is raised
    - Great for implementing logging and rendering a custom error view

# Authorization Filter

```
[Authorize]
public ActionResult Create() {...}
```

- Built-in attribute – attach to controller or action method
- Restricts access to only authenticated users
- Redirects unauthenticated users to a logon page

```
<authentication mode="Forms">
    <forms loginUrl="~/Account/LogOn" timeout="2880" />
</authentication>
```

- Optionally restrict access to users in specific role
  - [Authorize(Roles="Admin")]
  - User must be authenticated and member of Admin role

# More Built-In Filters

- Extend infrastructure plumbing

```
[HandleError(ExceptionType = typeof(NullReferenceException), View = "NullError")]
[HandleError(ExceptionType = typeof(SecurityException), View = "SecurityError")]
public class HomeController : Controller
{
    [OutputCache(Duration = 15)]
    public ActionResult Index()
    {
        ViewData["Message"] = DateTime.Now;
        return View();
    }

    [Authorize(Roles="Admin, SalesReps")]
    public ActionResult About()
    {
        return View();
    }
}
```

HandleError Filter
Redirect to custom error views

OutputCache Filter
Caches action results

Authorize Filter
Restrict Access

# Custom Action Filters

- Add pre-/post-action behavior to controller actions/results
- Extend and manipulate behavior
- Ideal for implementing cross-cutting functionality

```
public class LogMessageAttribute : ActionFilterAttribute
{
    public string Message { get; set; }

    public override void OnActionExecuting(ActionExecutingContext filterContext)
    { … }

    public override void OnActionExecuted(ActionExecutedContext filterContext)
    { … }

    public override void OnResultExecuting(ResultExecutingContext filterContext)
    { … }

    public override void OnResultExecuted(ResultExecutedContext filterContext)
    { … }
}
```

Fires before action

Fires after action

Fires before result

Fires after result

```
[LogMessage(Message = "Hello from Microsoft")]
public class ArticleController : Controller
```

# Filter Scope…

- Single Action Method:

```
[Authorize(Roles="Posts")]
public ActionResult Display(int id)
{
```

- Entire Controller

```
[Authorize(Roles = "Posts")]
public class ArticleController : Controller
{
```

- Entire Application - Global filter

```
RegisterGlobalFilters(GlobalFilters.Filters);


public static void RegisterGlobalFilters(GlobalFilterCollection filters)
{
    filters.Add(new HandleErrorAttribute());
}
```

# Global Filters

- Global filter executes on every action request
- RegisterGlobalFilters() method exposed in global.asax

```
[HandleError]
[HandleLogging]
public class StoreController : Controller {
```

```
void RegisterGlobalFilters(GlobalFilterCollection filters)
{
    filters.Add(new HandleErrorAttribute());
    filters.Add(new HandleLoggingAttribute());
}

void Application_Start()
{
    RegisterGlobalFilters(GlobalFilters.Filters);
    RegisterRoutes(RouteTable.Routes);
}
```

# Model Binding

## For HTTP "GET" Requests

GET request (via routing):


All About URI Segments

Map to URL Patterns

```
routes.MapRoute(
      "Default",
       // Route name
      "{controller}/{action}/{id}", // URL with parameters
       new { controller = "Home", action = "Index", id = UrlParameter.Optional }
);
```

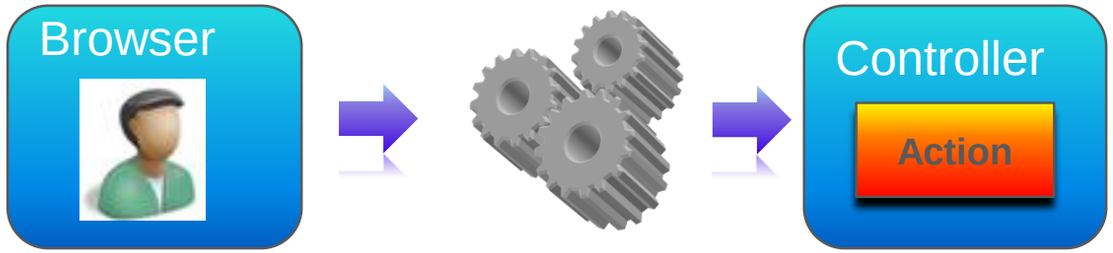
Map to controller, action

```
public class ProductsController : Controlle
r{
      public ActionResult Show(int id)
      {
            //.....
            return View();
      }
}
```

# MVC Routing Engine

## Routing Engine

- The *mechanism* with which MVC *maps* an incoming URL to a controller and action for request processing




- Enables you to *decouple* your URL from the underlying directory structure and file names (.aspx).
  - Create meaningful, intuitive URLs that make sense for the user:
    - /Customer/Oregon/Edit/1001
  - WebForms highly dependent on directory structure and .aspx files
- Performs the grunt work of matching URLs to appropriate application components

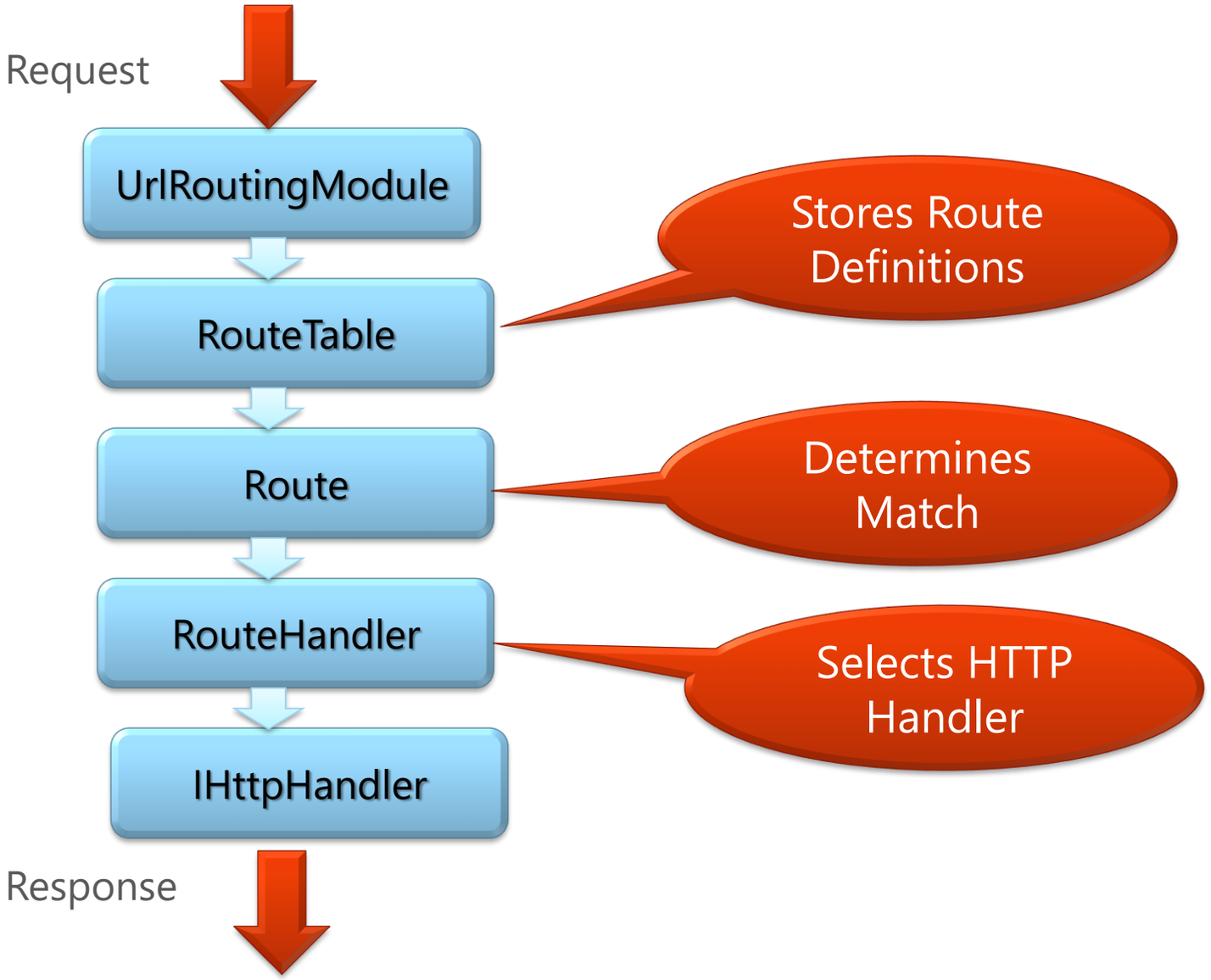
# MVC Routes

- A Route is a predefined *URL pattern*, whose elements can be dynamic or static
  - Dynamic elements defined with placeholders enclosed in braces { }
  - Static elements are constant strings
- An incoming URL is matched against each URL pattern, segment-by-segment, until a match is found

| Route Definition | Example of Matching URL |
|---|---|
| {controller}/{action}/{id} | /Products/show/beverages |
| {table}/Details.aspx | /Products/Details.aspx |
| blog/{action}/{entry} | /blog/show/123 |
| {reporttype}/{year}/{month}/{day} | /sales/2008/1/5 |
| {locale}/{action} | /en-US/show |
| {language}-{country}/{action} | /en-US/show |

# MVC Routing

## Execution Pipeline



# What are the benefits of MVC?

- **Separation of concerns**
  - Between view and model
  - A fundamental software engineering principle
  - Code that is
    - Easier to test
    - Easier to maintain
    - Easier to plug model into different views (e.g. for different clients like web vs. mobile)

# MVP Design Pattern

# Model View Presenter Web Forms Framework

- The simplest way to explain the pattern is to consider a checkout at a supermarket.
- Visualise that you, the customer, are the View, the cashier is the Presenter and the cash register is the Model.


Example Reference: http://www.bradoncode.com/blog/2012/04/mvp-design-pattern-survival-kit.html

# Model View Presenter Web Forms Framework


- Key point: The view will never access the model directly.
- Key point: We need to be able to replace the View without having to change the Presenter.
- Key point: The Presenter will make all the decisions about interactions and the Model will make the business logic decisions.

Example Reference: http://www.bradoncode.com/blog/2012/04/mvp-design-pattern-survival-kit.html

# What are the benefits of MVP?

- View layer decoupled from the model (separation of concerns)
  - Easier to maintain
  - Easier to test
  - Easier to replace/refactor
- No more code behind page!!!

Microsoft

# Thank you!