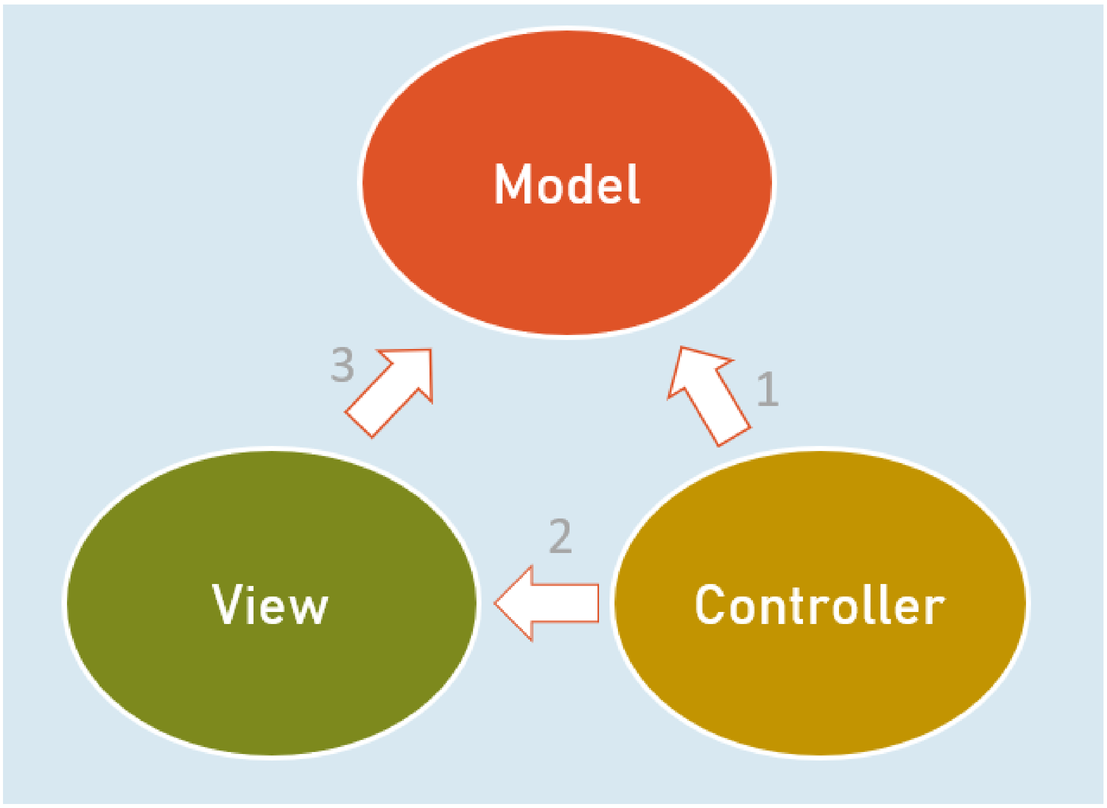
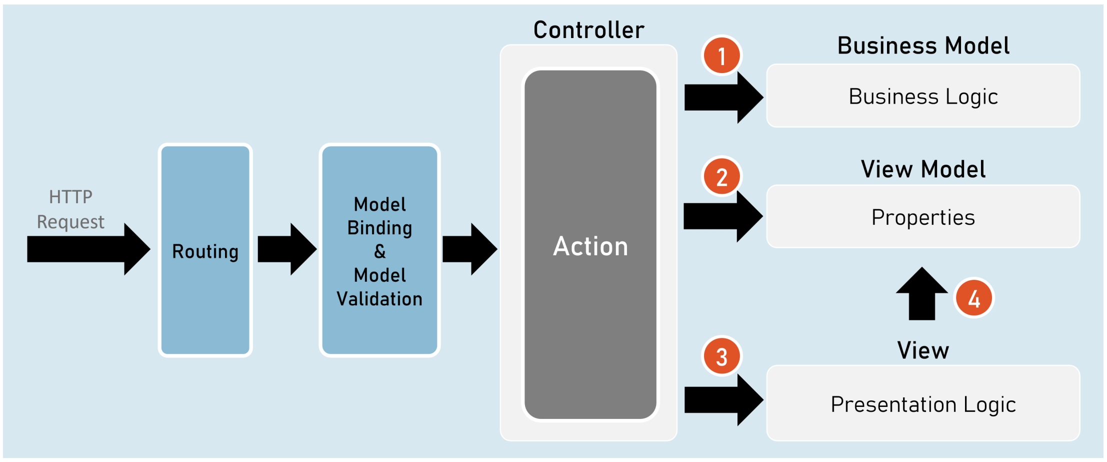
Section Cheat Sheet (PPT)

Model-View-Controller (MVC) Pattern

"Model-View-Controller" (MVC) is an architectural pattern that separates application code into three main components: Models, Views and Controllers.





1. Controller invokes Business Model.
2. Controller creates object of View Model.
3. Controller invokes View.
4. View accesses View Model.

Responsibilities of Model-View-Controller

Controller

- Receives HTTP request data.
- Invoke business model to execute business logic.

Business Model

- Receives input data from the controller.
- Performs business operations such as retrieving / inserting data from database.
- Sends data of the database back to the controller.

Controller

- Creates object of ViewModel and files data into its properties.
- Selects a view & invokes it & also passes the object of ViewModel to the view.

View

- Receives the object of ViewModel from the controller.
- Accesses properties of ViewModel to render data in html code.

- After the view renders, the rendered view result will be sent as response.

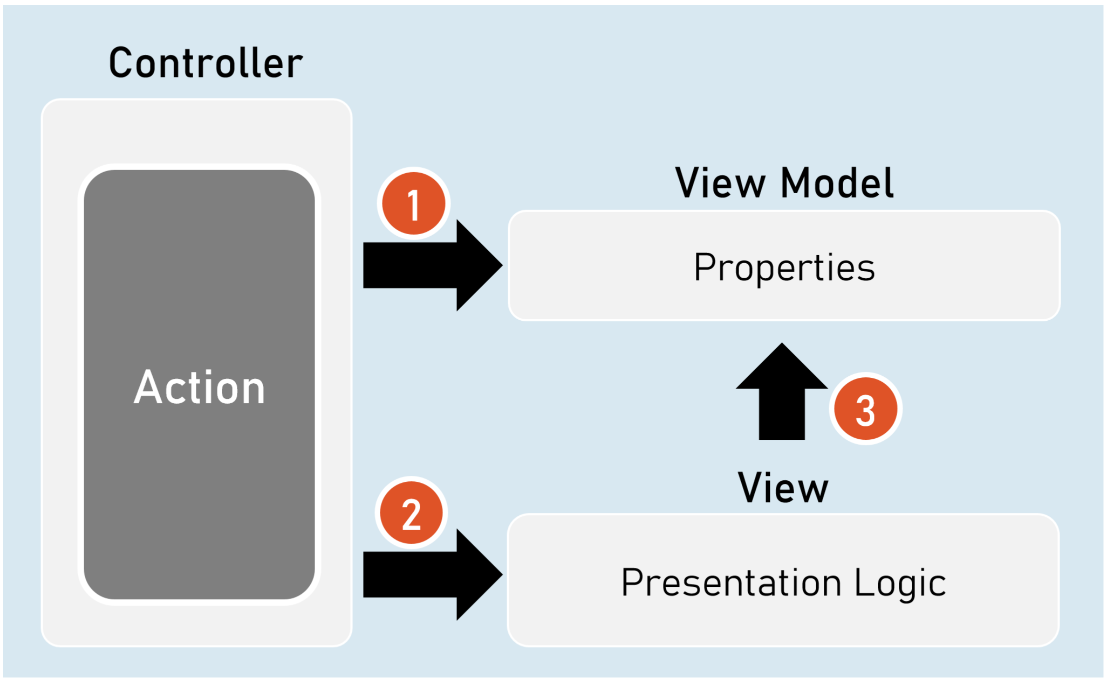
Benefits / Goals of MVC architectural pattern

- Clean separation of concerns
- Each component (model, view and controller) performs single responsibility.
- Identifying and fixing errors will be easy.
- Each component (model, view and controller) can be developed independently.
- In practical, both view and controller depend on the model.
- Model doesn't depend on neither view nor the controller.
- This is one of the key benefits of the 'clean separation'.
- This separation allows the model to be built and tested independently.
- Unit testing each individual component is easier.

Views

View is a web page (.cshtml) that is responsible for containing presentation logic that merges data along with static design code (HTML).

- Controller creates an object of ViewModel and fills data in its properties.
- Controller selects an appropriate view and invokes the same view & supplies object of ViewModel to the View.
- View access the ViewModel.



- View contains HTML markup with Razor markup (C# code in view to render dynamic content).
- Razor is the view engine that defines syntax to write C# code in the view. @ is the syntax of Razor syntax.
- View is NOT supposed to have lots of C# code. Any code written in the view should relate to presenting the content (presentation logic).
- View should neither directly call the business model, nor call the controller's action methods. But it can send requests to controllers.

Razor View Engine

Razor Code Block

```
@{

  C# / html code here

}
```

Razor code block is a C# code block that contains one or more lines of C# code that can contain any statements and local functions.

Razor Expressions

```
@Expression
--or--
@(Expression)
```

Razor expression is a C# expression (accessing a field, property or method call) that returns a value.

Razor - If

```
@if (condition) {
  C# / html code here
}
```

Razor - if...else

```
@if (condition) {
  C# / html code here
}
else {
  C# / html code here
}
```

Else...if and nested-if also supported.

Razor - Switch

```
@switch (variable) {
  case value1: C# / html code here; break;
  case value2: C# / html code here; break;
  default: C# / html code here; break;
}
```

Razor - foreach

```
@foreach (var variable in collection ) {
  C# / html code here
}
```

Razor - for

```
@for (initialization; condition; iteration) {
  C# / html code here
}
```

Razor - Literal

```
@{
 @: static text
}
```

Razor - Literal

`<text>static text</text>`

Razor - Local Functions

```
@{
return_type method_name(arguments) {
  C# / html code here
 }
}
```

The local functions are callable within the same view.

Razor - Members

Razor - Methods, Properties, Fields

```
@functions {
 return_type method_name(arguments) {
   C# / html code here
 }

 data_type field_name;

 data_type property_name
 {
   set { … }
   get { … }
 }
}
```

The members of razor view can be accessible within the same view.

Html.Raw( )

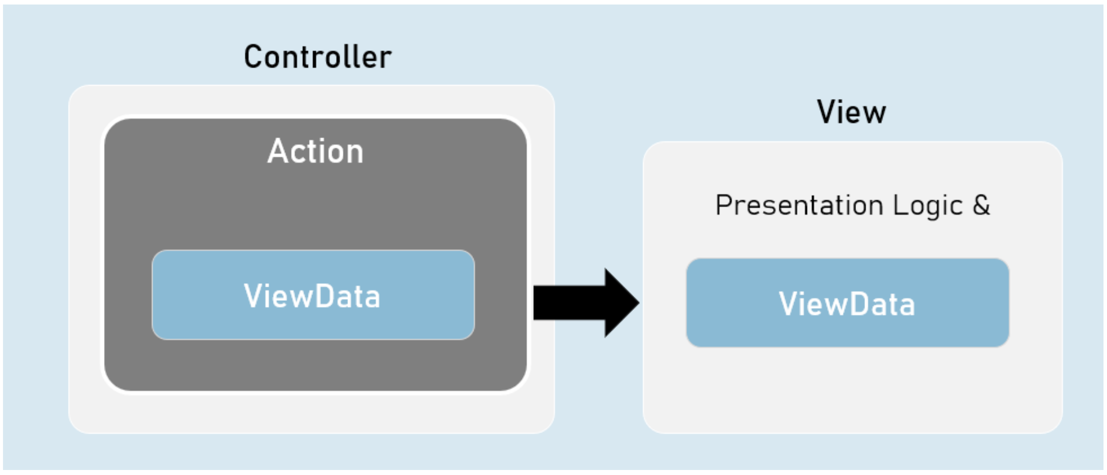
```
@{
 string variable = "html code";
}

@Html.Raw(variable) //prints the html markup without encoding
(converting html tags into plain text)
```

ViewData

ViewData is a dictionary object that is automatically created up on receiving a request and will be automatically deleted before sending response to the client.

It is mainly used to send data from controller to view.



ViewData is a property of Microsoft.AspNetCore.Mvc.Controller class and Microsoft.AspNetCore.Mvc.Razor.RazorPage class.

It is of Microsoft.AspNet.Mvc.ViewFeatures.ViewDataDictionary type.

```
namespace Microsoft.AspNetCore.Mvc
{
 public abstract class Controller : ControllerBase
 {
   public ViewDataDictionary ViewData { get; set; }
```

```
    }
  }
```

- It is derived from IDictionary<KeyValuePair<string, object>> type.
- That means, it acts as a dictionary of key/value pairs.
- Key is of string type.
- Value is of object type.

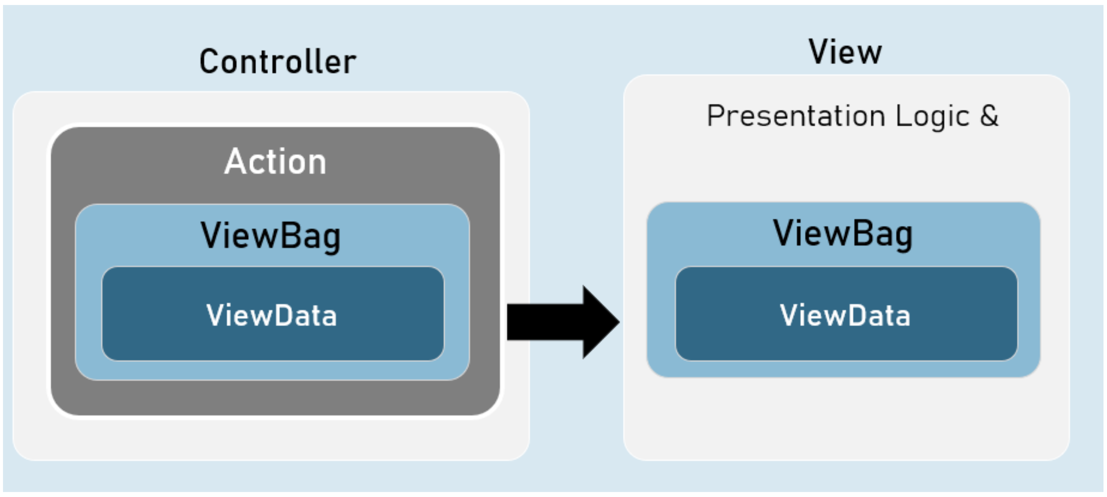
ViewData - Properties and Methods

- `int Count { get; set; }` //gets the number of elements.
- `[string Key]` //Gets or sets an element.
- `Add(string key, object value)` //Adds a new element.
- `ContainsKey(string key)` //Determines whether the specified key exists or not.
- `Clear()` //Clears (removes) all elements.

ViewBag

ViewBag is a property of Controller and View, that is used to access the ViewData easily.

ViewBag is 'dynamic' type.



ViewBag is a property of Microsoft.AspNetCore.Mvc.Controller class and Microsoft.AspNetCore.Mvc.Razor.RazorPageBase class.

It is of dynamic type.

```
namespace Microsoft.AspNetCore.Mvc
{
 public abstract class Controller : ControllerBase
 {
   public dynamic ViewBag { get; set; }
 }
}
```

The 'dynamic' type similar to 'var' keyword.

But, it checks the data type and at run time, rather than at compilation time.

If you try to access a non-existing property in the ViewBag, it returns null.

`[string Key] //Gets or sets an element.`

Benefits of 'ViewBag' over ViewData

ViewBag's syntax is easier to access its properties than ViewData.

Eg: `ViewBag.property` [vs] `ViewData["key"]`

You need NOT type-cast the values while reading it.

Eg: `ViewBag.object_name.property`

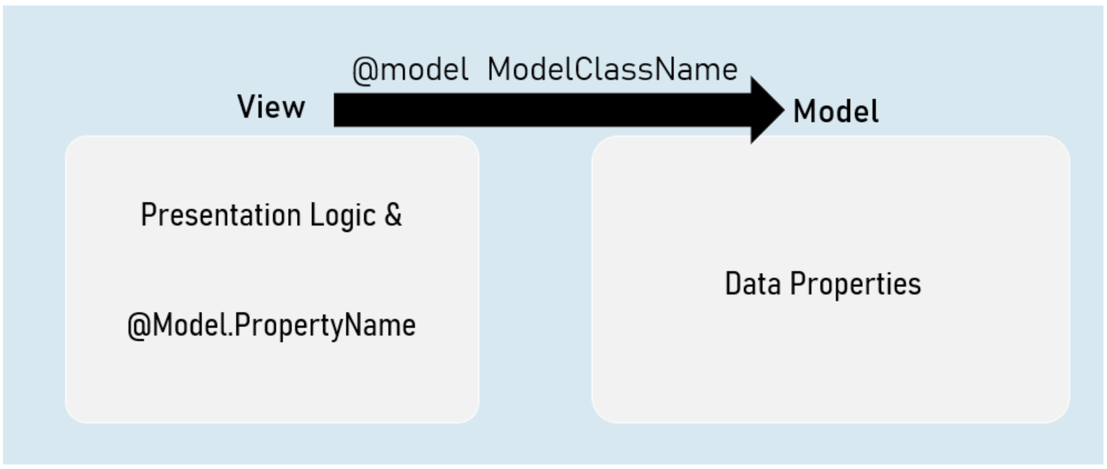
[vs]

`(ViewData["key"] as ClassName).Property`

Strongly Typed Views

Strongly Typed View is a view that is bound to a specified model class.

It is mainly used to access the model object / model collection easily in the view.



Benefits of Strongly Typed Views

- You will get Intellisense while accessing model properties in strongly typed views, since the type of model class was mentioned at @model directive.
- Property names are compile-time checked; and shown as errors in case of misspelled / non-existing properties in strongly typed views.

- You will have only one model per one view in strongly typed views.
- Easy to identify which model is being accessed in the view.

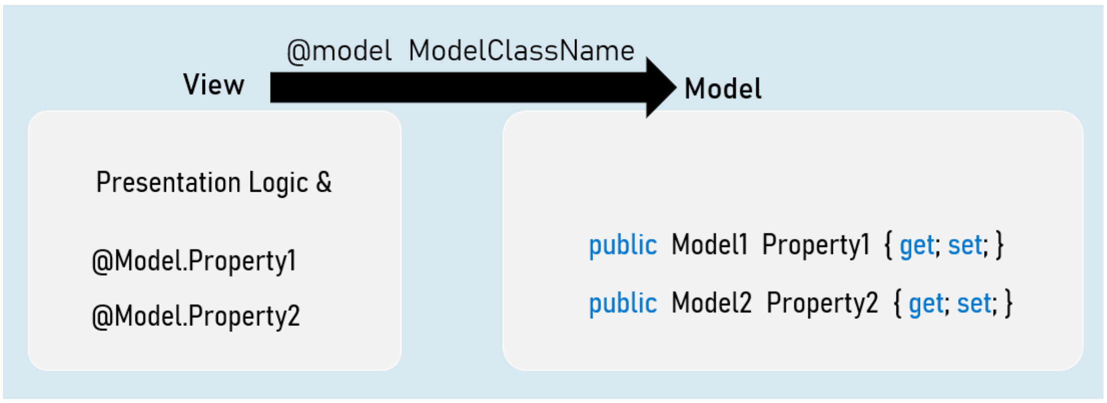
Helper methods in Controller to invoke a View

- `return View( );` //View name is the same name as the current action method.
- `return View(object Model );` //View name is the same name as the current action method & the view can be a strongly-typed view to receive the supplied model object.
- `return View(string ViewName);` //View name is explicitly specified.
- `return View(string ViewName, object Model );` //View name is explicitly specified & the view can be a strongly-typed view to receive the supplied model object.

Strongly Typed Views

Strongly Typed View can be bound to a single model directly.

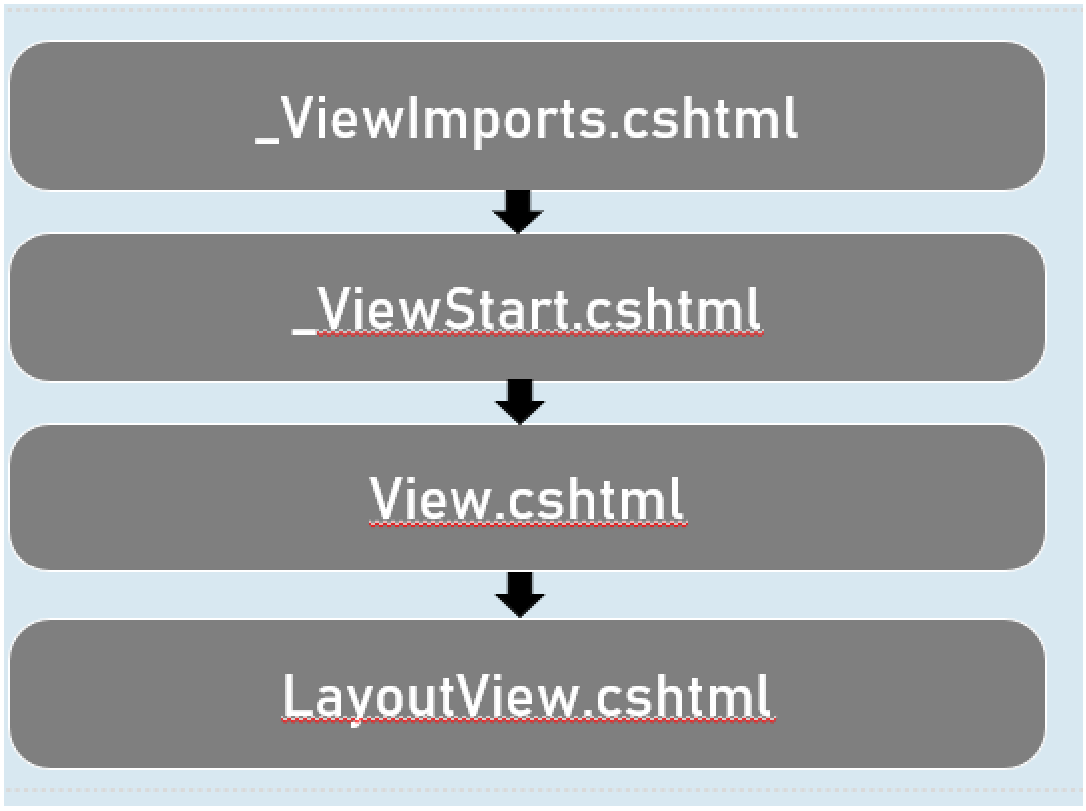
But that model class can have reference to objects of other model classes.

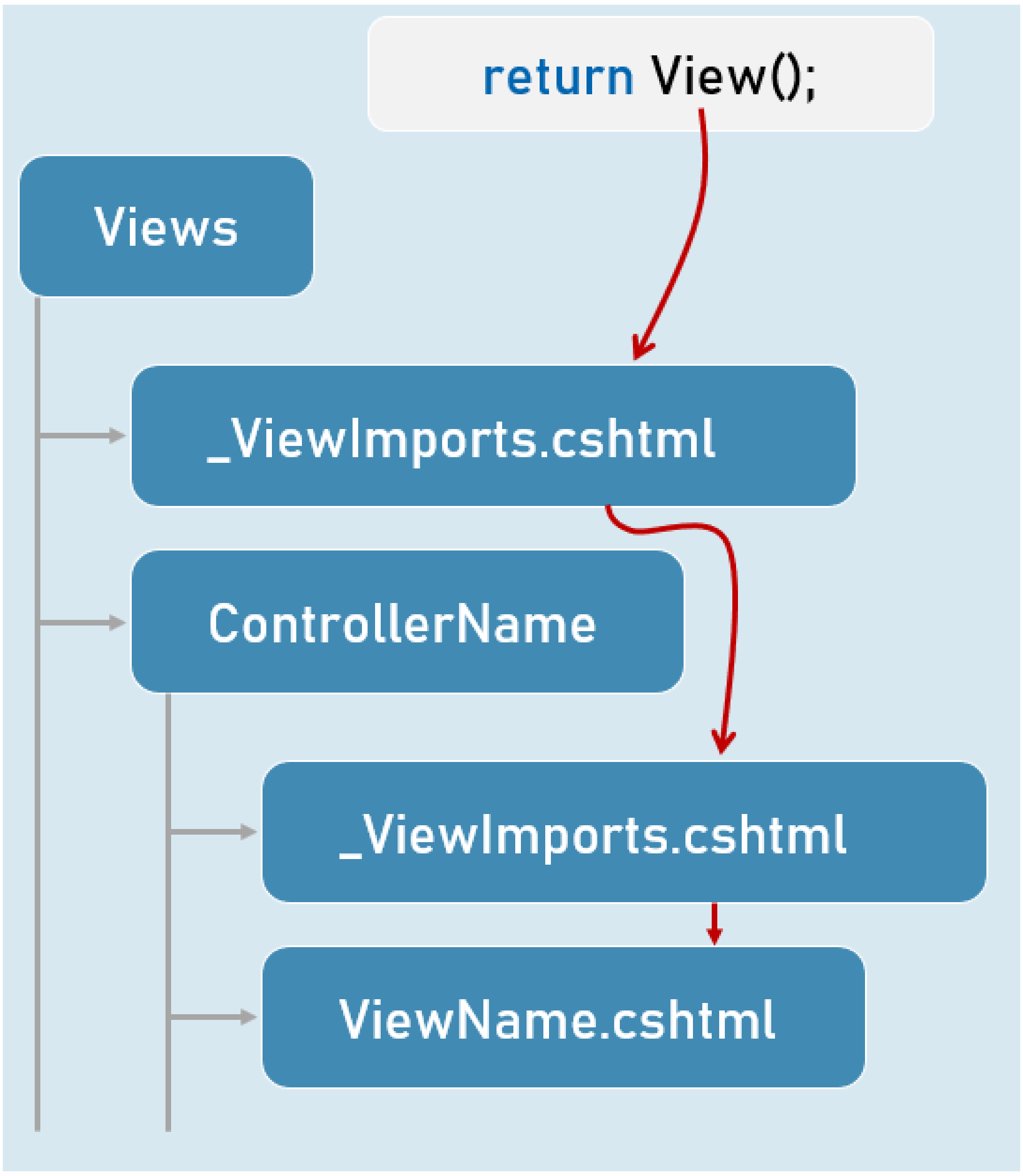


ViewImports.cshtml

ViewImports.cshtml is a special file in the "Views" folder or its subfolder, which executes automatically before execution of a view.

It is mainly used to import common namespaces that are to imported in a view.

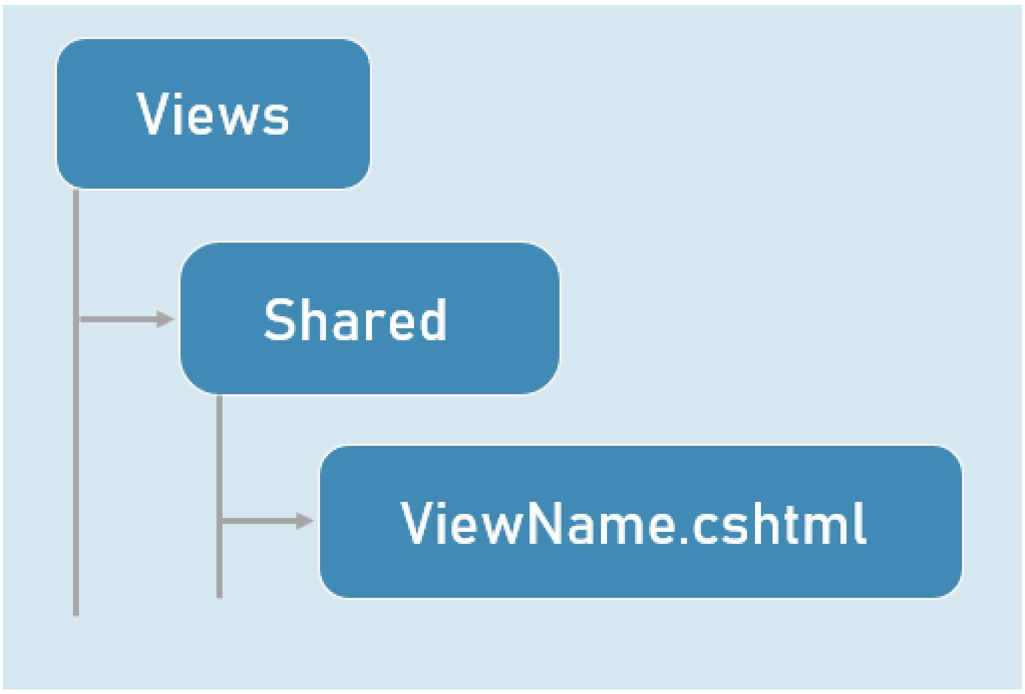




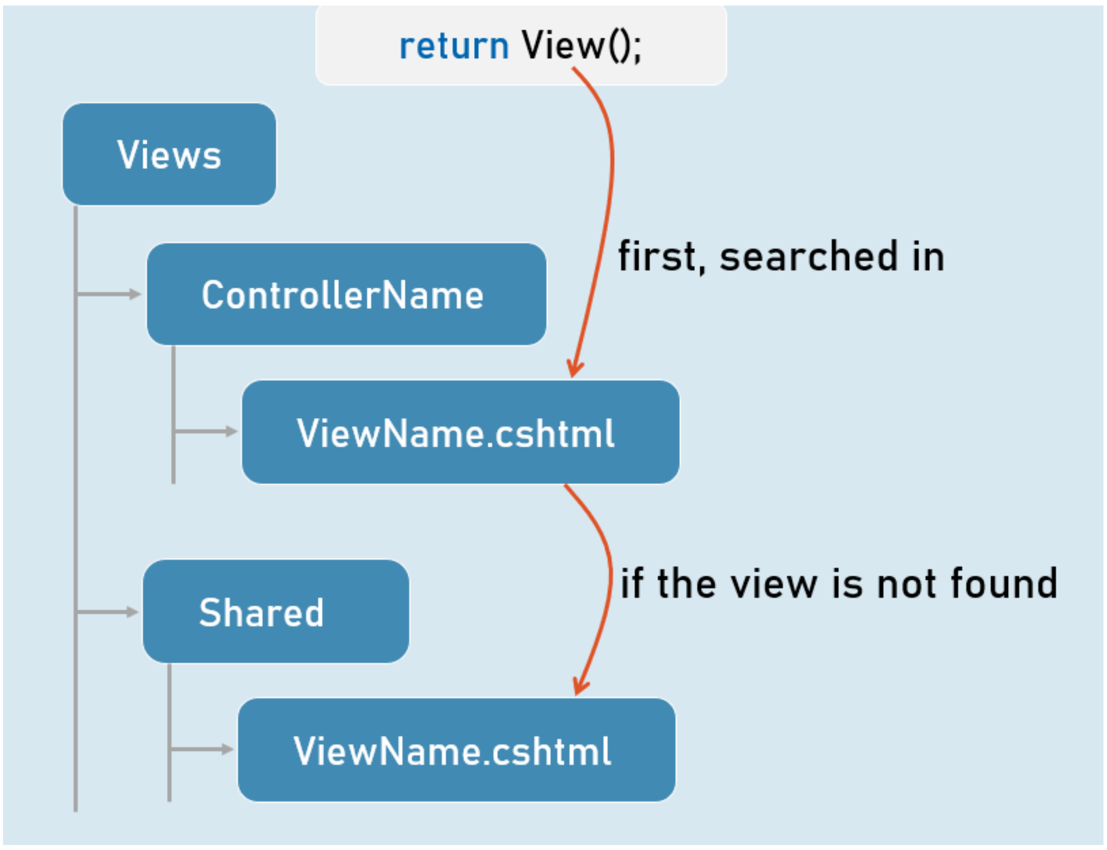
Shared Views

Shared views are placed in "Shared" folder in "Views" folder.

They are accessible from any controller, if the view is NOT present in the "Views\ControllerName" folder.



View Resolution


return View();
Views
ControllerName
ViewName.cshtml
first, searched in
Shared
ViewName.cshtml
if the view is not found