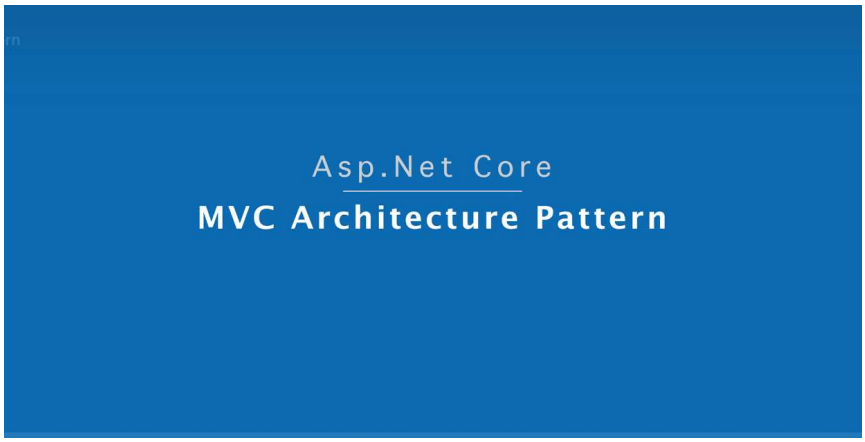
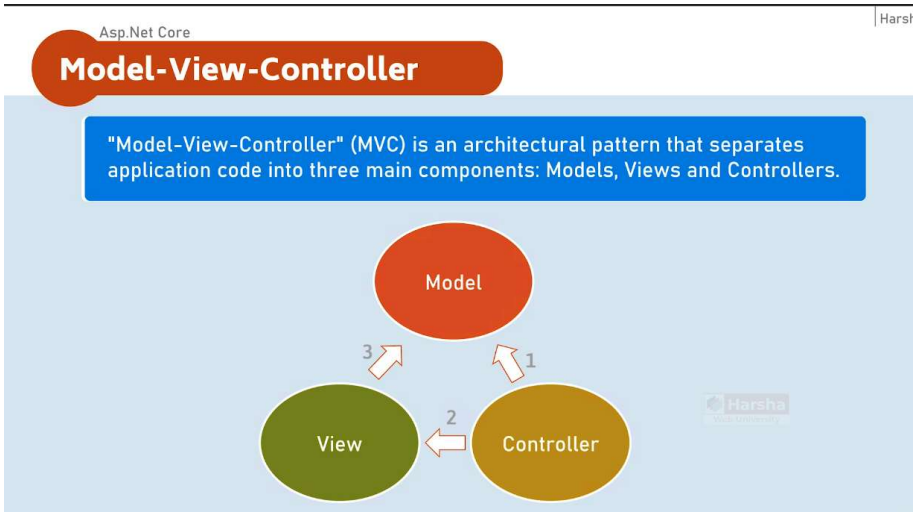
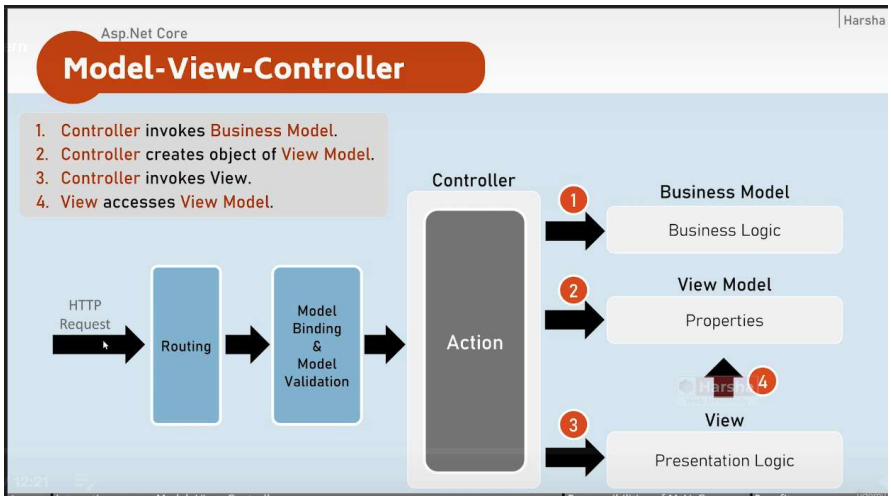
Section 08: Razor Views [MVC]

Sunday, January 12, 2025 8:11 PM







In most of the ASP.NET Core applications, we use the Model-View-Controller architecture pattern.

The Model-View-Controller pattern is an architectural pattern that dictates how you separate the code into models, views, and controllers. So far in our course, we have worked with the models and controllers, but we have not yet discussed views. This is the time to reveal the concept called views and finally implement the Model-View-Controller pattern.

Here, the thumb rule is that the controller can invoke anything. The controller can invoke the model, the controller can invoke the view, and then the view can invoke the model. These are the three roles.

But it's a bit confusing if you are a beginner, so let us try to decode the same.

First of all, there are two types of models: business model and view model. Once the request is sent, first it will be received by the routing and then eventually go to the model binding and model validation, as we have seen earlier. Then it reaches the action method.

So, the controller action method is the entry point for your MVC architecture. The controller takes the responsibility of receiving the request and deciding what response should be sent. This means whatever is returned in the action method will eventually be sent as a response body to the client.

To understand this execution flow in the MVC architecture, let's try to imagine a scenario. Let's say there is an e-commerce application and there is a login page. The goal here is that the user will enter their email address and password and then be able to see the list of products in the e-commerce application.

The request contains the email address and password as part of the request body, and then the ASP.NET routing mechanism checks the URL to see whether it matches with any one of the action methods. Since an action method is selected here, it first performs the model binding process to receive the email address and password from the request, and then it performs the basic validations, such as required validation or email address validation, etc., against the model. Then it reaches the action method.

So, what is the exact responsibility of the action method? It receives the model object, which contains the email address and password, then it has to check whether the model state is valid. This means if email and password values are not null, then `ModelState.IsValid` equals true.

Then it performs the operation number one. It invokes the business model to perform the business logic. Eventually, the business model is a service class that performs the business logic or business operations and business validations. This means it verifies whether the email address and password are correct as per the data from the database. If it is correct, then it returns true back to the action method.

So, the action method of the controller performs the second operation, which is creating a view model containing all the product information that is to be displayed in the view (that means on the page).

Yes, the controller has a list of products as a view model, for example, as a list of product model classes. But it is just a list or collection, right? So, in order to visualize the same, meaning to print it in the form of a page, you require a view, which contains your HTML code.

The controller action method supplies the view model object to the view. It says to the view: you have your design logic (that means your HTML code), and take this list of products (the view model) and print it as you want, either in the form of a table, or in the box format, or as a tree list. It is up to you. The controller doesn't involve itself in the design logic.

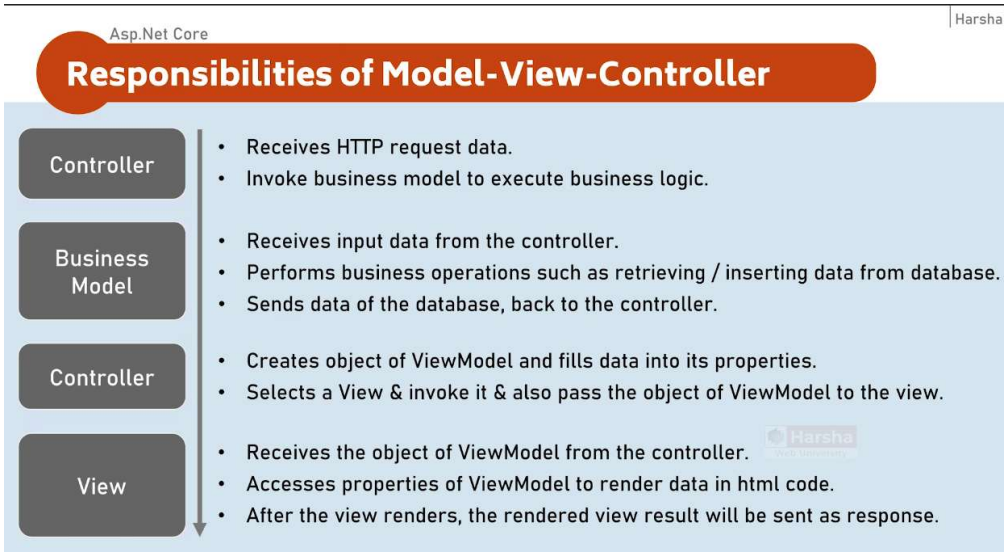
The view contains the design logic, and whenever necessary, the view reads the view model information supplied by the controller. Finally, when the view executes on the server, it generates HTML code that includes the data of the view model. The result of the view is a plain HTML page that contains the list of products in the form of a table, boxes, or side-by-side format. Then, the final view result will be sent as a response back to the browser.

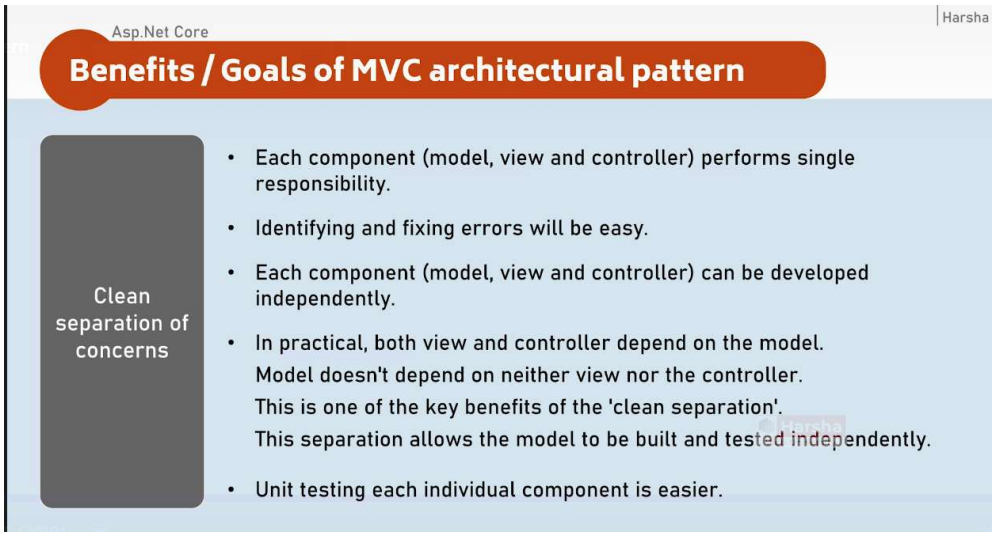
So, this is the complete process of what exactly happens in the Model-View-Controller pattern. The same type of pattern is repeated for every request, whether it is a login form, registration form, or any other type of webpage in real-world apps.

Thumb Rules of MVC Pattern:
1. **The controller invokes both types of models:**
   - It invokes the business model to perform business operations. The business logic invokes the data access layer to perform real-world database operations such as reading or writing data from the database. Business logic is about validations and calculations and invokes the data access logic, which is separate from the business logic. The business logic mainly deals with database tables and related operations.

2. **The controller creates an object of the view model or a collection:**
   - It contains either a single object or a collection, which is the data to be presented in the view. Then, it passes the view model object to the view, asking it to execute and print the view model data.

3. **The view accesses the view model object:**
   - The view does not directly access the business model. Instead, it directly deals with the data present in the view model but doesn't care where the data has been retrieved from or from which database table. It is the responsibility of the business model to handle backend logic.





So, these are the responsibilities of the controller, business model, and view, and these points execute in the same order. We have already discussed the same in the previous slide, and the same information is documented here.

In brief:
- The controller receives the request and invokes the business model.
- The business model receives the input data from the controller and performs business operations such as searching, inserting, updating, or any other database-specific operations. It also performs necessary server-side validations.
- If everything is clear, the business model sends the essential data back to the controller.
- The controller receives the data and forms it into an object of the ViewModel class.

The ViewModel class is a POCO (Plain Old CLR Object) class. As we have created model classes with validation attributes in the last section, in the ViewModel, you can add data attributes such as `Required`, `Range`, etc., which are applied to the data properties.

The model classes created in the model binding section are exactly an example of the ViewModel. The controller passes the ViewModel object to the view. Then, the view accesses the property values of the ViewModel and prints them in the desired design.

Finally, once the view renders on the server side, the result is a plain HTML code. This plain HTML code, along with the corresponding CSS and JavaScript files, is sent as a response to the browser. The browser receives the HTML code and executes the HTML, CSS, and JavaScript as per its nature. In this way, the real user gets the actual output.

We have described the complete process of the Model-View-Controller (MVC) pattern theoretically. Practically, we will implement the same in further sections. So, in case you miss any point in this lecture, don't get confused now. Continue with the further lectures. Within one or two sections, you will get a complete, clear-cut idea of how the Model-View-Controller works.

Benefits of the Model-View-Controller Pattern:
Each component—Model, View, Controller—is developed independently. This approach provides the following benefits:
- **Unit Testing**: You can unit test each component separately.
- **Team Collaboration**: You can divide the work among multiple teams. For example, one group of your development team can take

care of the model, while another team handles the controller.
- **Parallel Development**: Models are generally completed first. Then, the view and controller can be independently developed in parallel. This is the greatest advantage of the Model-View-Controller pattern.

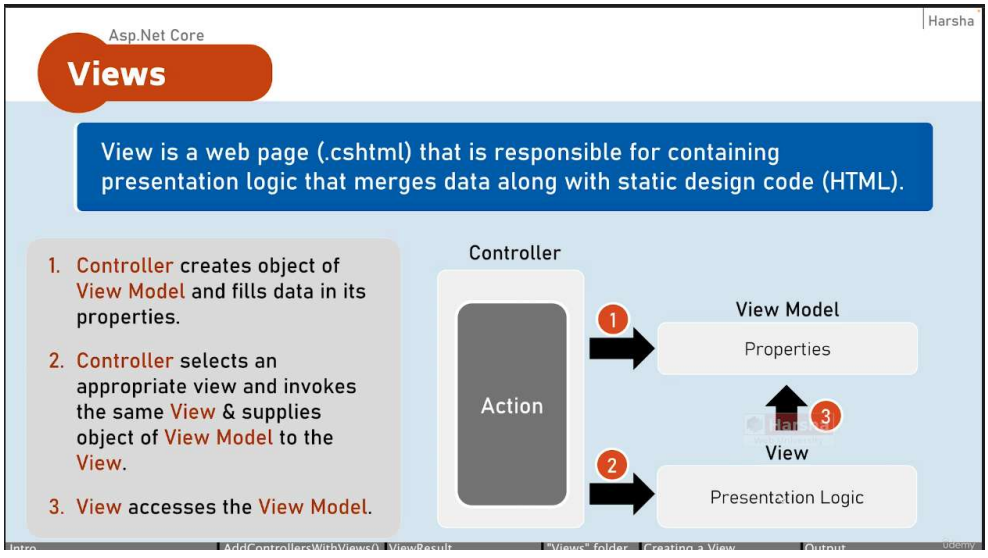
For instance, once there is clarity on:
1. What properties are to be stored in the database, and
2. What property values are to be displayed in the view,

One team can focus on view development, while another team develops the controllers, without knowing the view details.

This way, you can complete large tasks in a shorter span of time. Each component—Model, View, and Controller—performs independent tasks:
- **Models** take care of business logic.
- **Views** handle presentation logic.
- **Controllers** manage both the model and the view, invoking each as needed.

Because of this independence, the components become modular. For example, controllers and views can be developed by different teams. This modularity and flexibility are the main reasons why the Model-View-Controller pattern has become extremely popular.

In the upcoming sections, we will try to implement this pattern practically.



In ASP.NET Core, the view is a web page that is responsible to contain the design logic or presentation logic. The presentation logic can be either in the form of HTML or C#.

HTML code is called client-side code, and C# code is called server-side code. That means at first, on the server side, the C# code executes, and after that, the result of the view will be sent to the browser as a response. Then on the browser, the HTML code executes.

So in the view, if you write one statement of HTML and another statement of C# in the same file, first on the server side, the C# code executes, and on the client side, on the browser, the HTML code executes. Here, client or browser both are the same.

The view will be invoked by the controller. The main responsibility of the controller is that it sends the ViewModel object to the view.

At first, the controller creates an object of the ViewModel or a collection, then it fills up all the values that are essential to print the output.

But the controller doesn't generate any real output. The controller just passes the ViewModel object to the view. Then the view directly accesses the properties of the ViewModel.

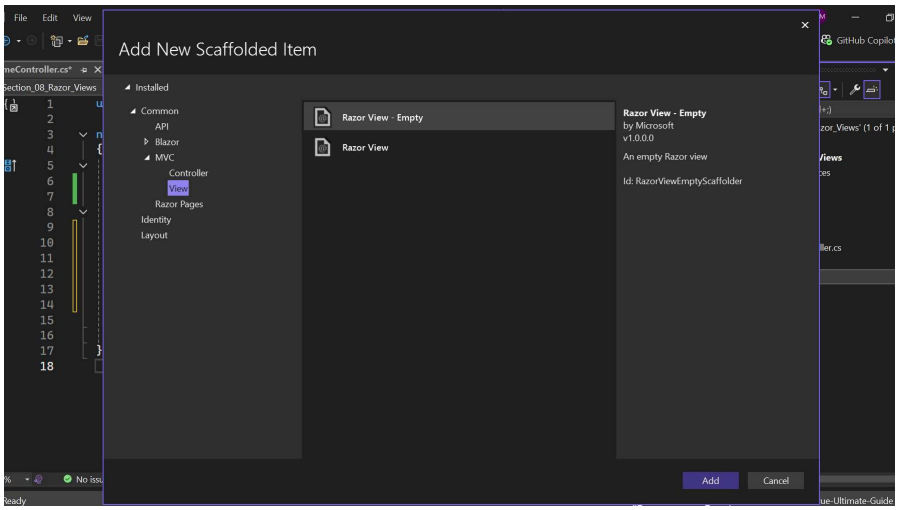
For example, the controller creates an object of the Employee ViewModel, which contains all the employee details like employee roll number, name, and work awards, etc. That ViewModel object will be given to the view, and then the controller selects which view is appropriate to execute.

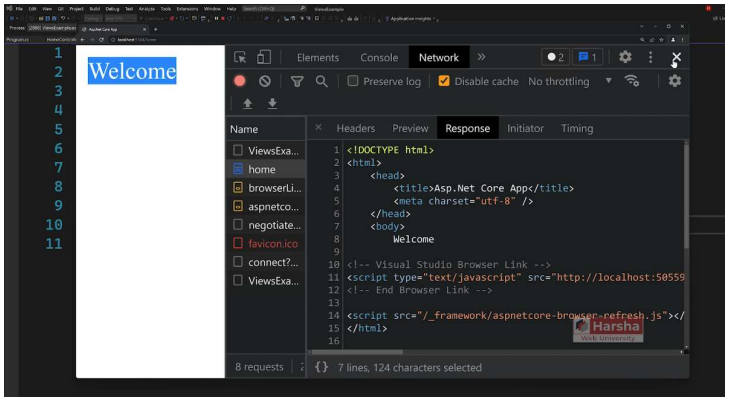
For example, there is a route parameter called display mode. It has two options: either summary or detailed. So if the route parameter value equals summary, it invokes one view. If it is equal to detailed, it invokes another view.

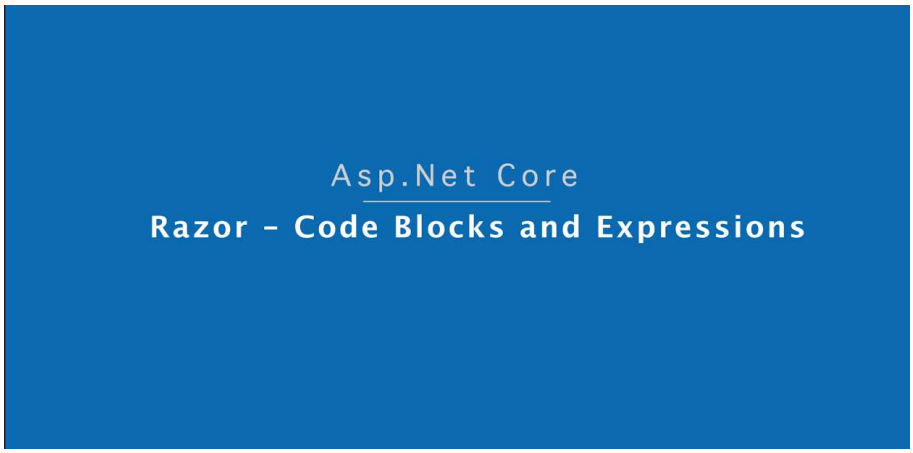
So based on the route parameter or based on the data from the database, the controller is able to decide which view should be executed based on the requirement or situation or maybe based on the user requirement. So the controller invokes the appropriate view and supplies the ViewModel object to the view.

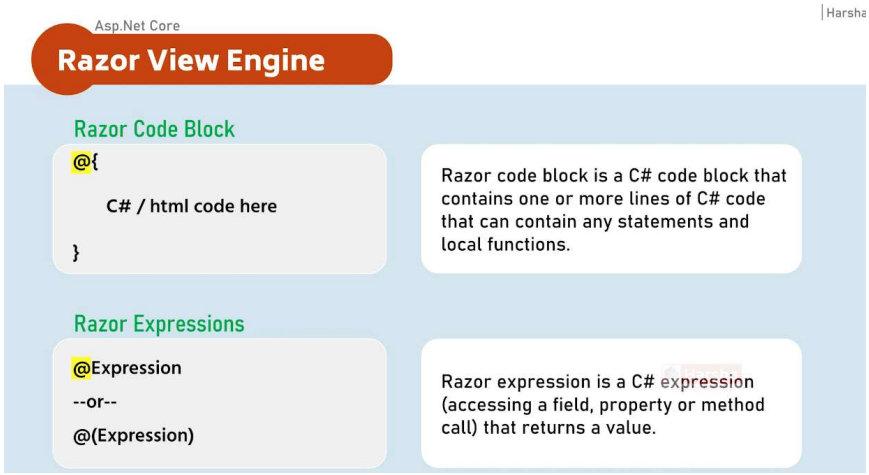
Then the view reads the data from that ViewModel. For example, the employee view reads the data from the Employee ViewModel that is supplied by the controller. Then as a part of the presentation logic, the actual details of the employee will be populated in the respective place wherever that ViewModel properties are accessed.

In that way, the view result executes, and you will get the plain HTML code, and that will be the response that is sent to the browser.

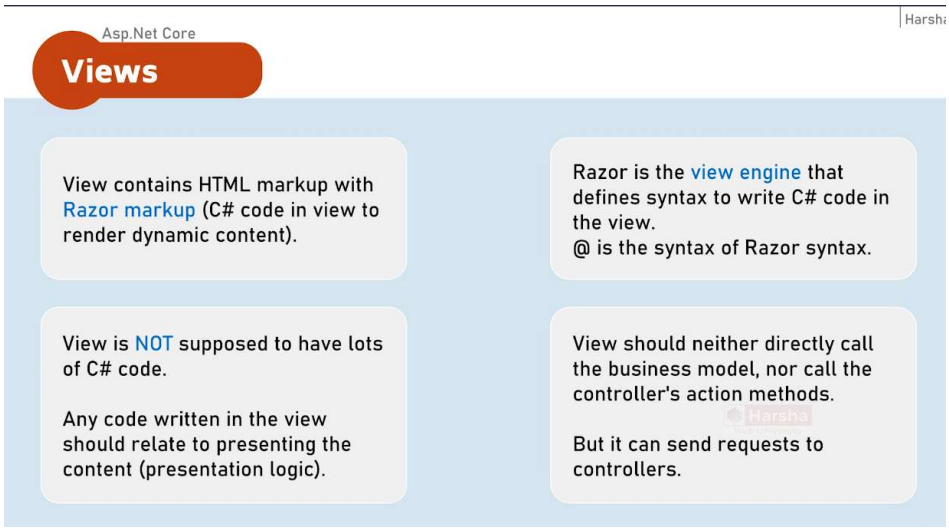






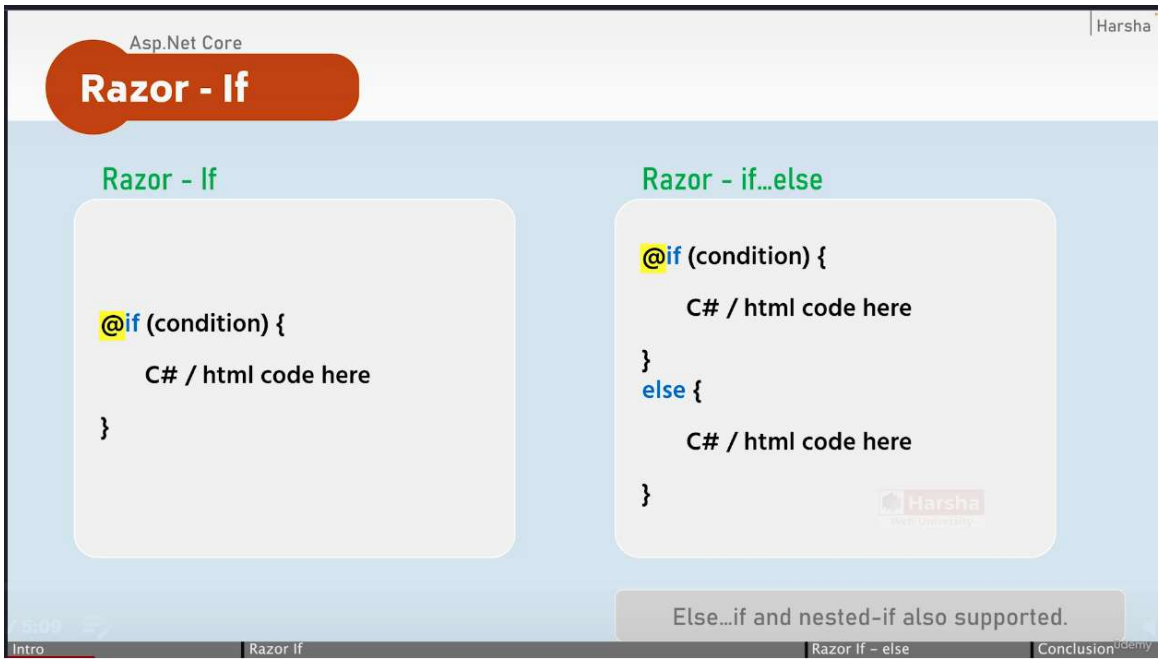


In order to write the server-side presentation logic in the ASP.NET Core Razor View, you have to use Razor code blocks and Razor expressions.
When you want to declare one or more C# variables or any other type of C# statements, you can use these Razor code block.



Asp.Net Core

Razor – If



Asp.Net Core

Razor – Switch-Case

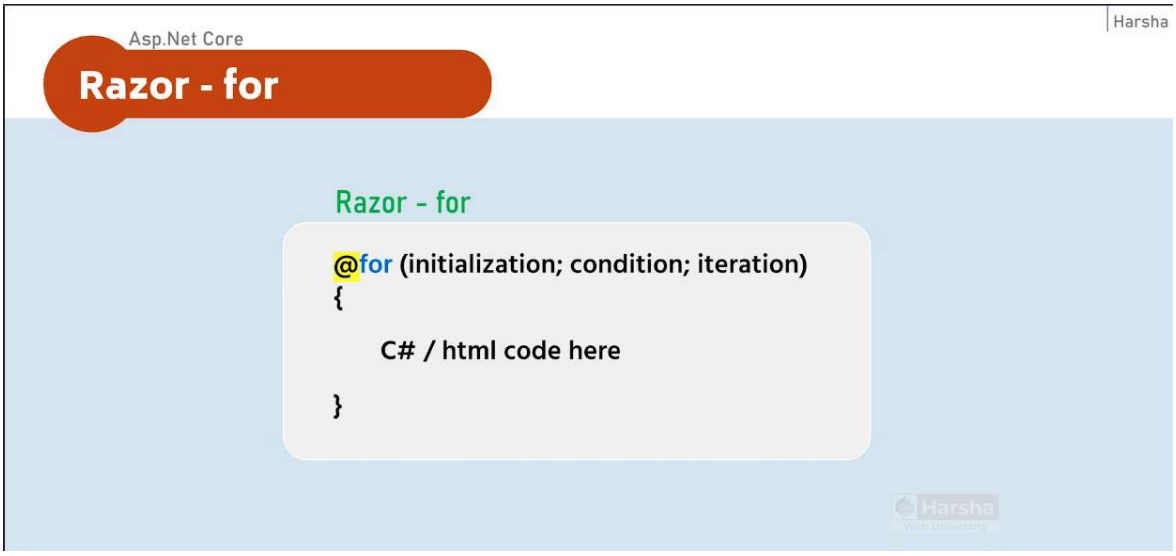

Harsha
Asp.Net Core
Razor - Switch
Razor - Switch
@switch (variable) {
case value1: C# / html code here; break;
case value2: C# / html code here; break;
default: C# / html code here; break;
}


Asp.Net Core
Razor – Foreach


Harsha
Asp.Net Core
Razor - foreach
Razor - foreach
@foreach (var variable in collection) {
C# / html code here
}

Asp.Net Core

Razor – for



Asp.Net Core

Razor – Literal

Harsha

Asp.Net Core

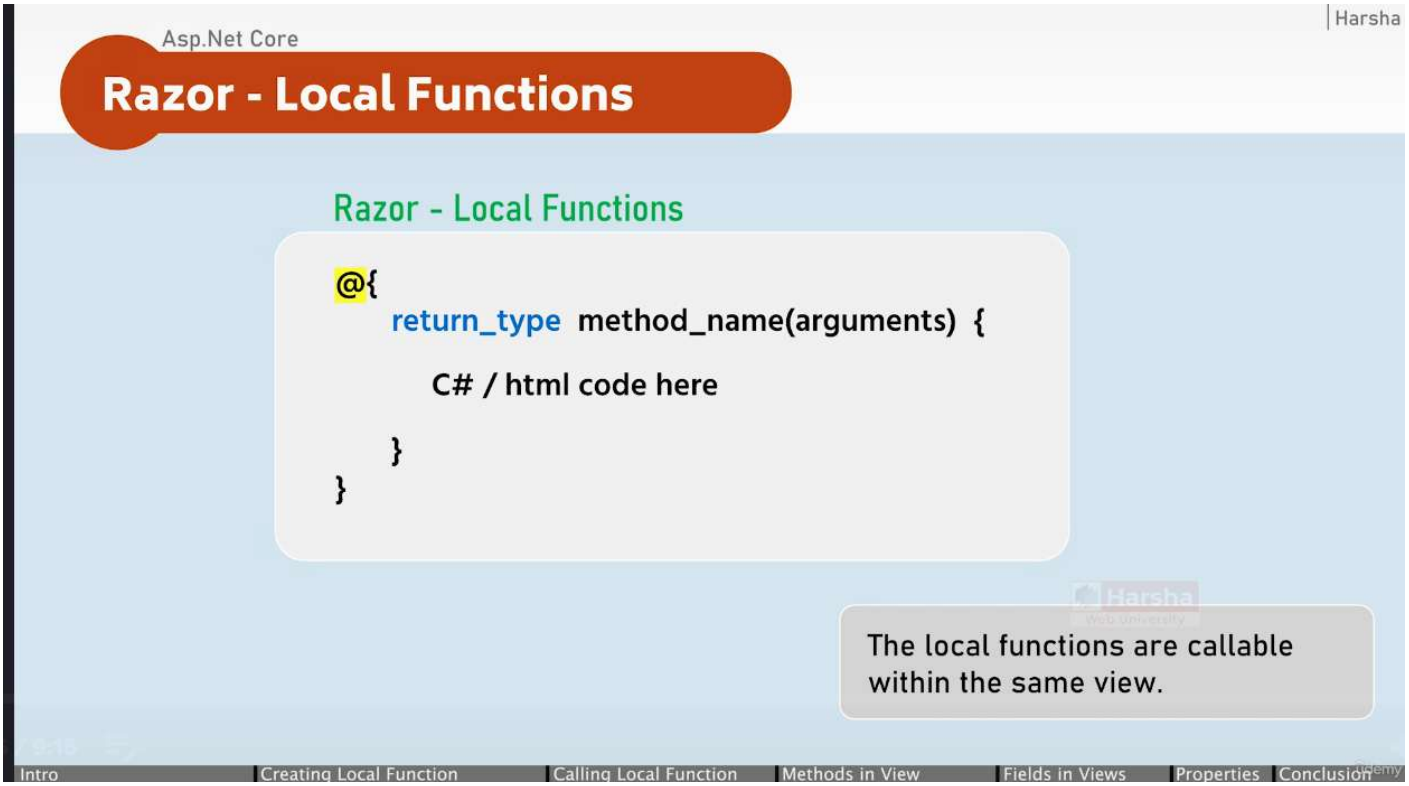
Razor - Literal

Razor - Literal

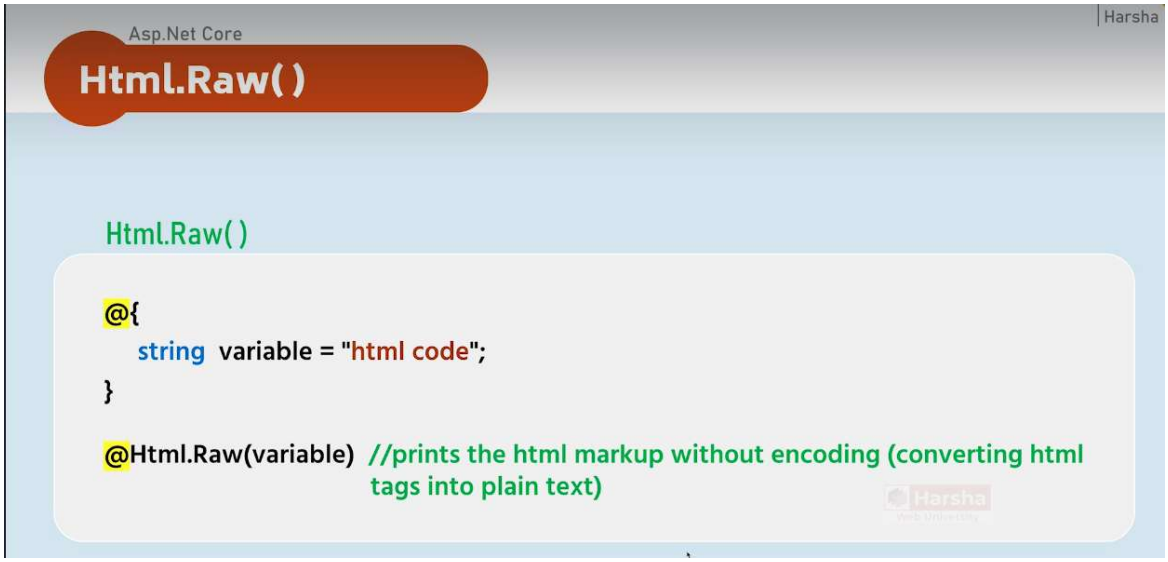
```
@{
    @: static text
}
```

Razor - Literal

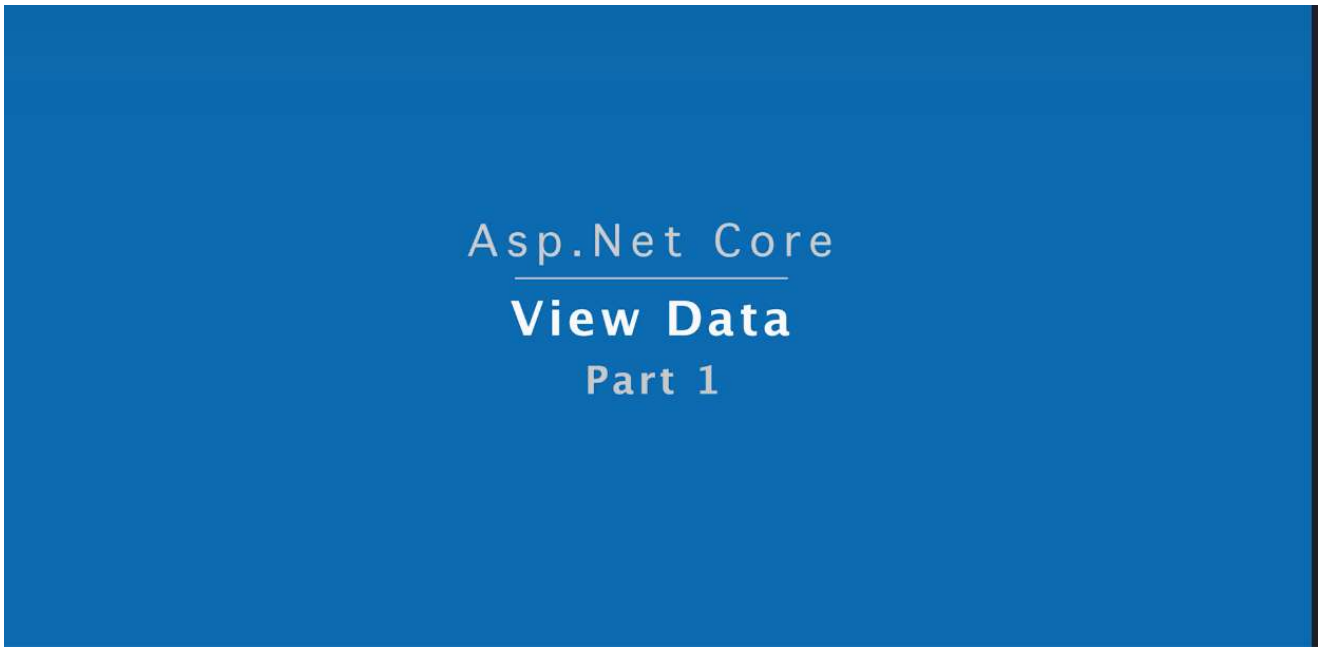
```
<text>static text</text>
```

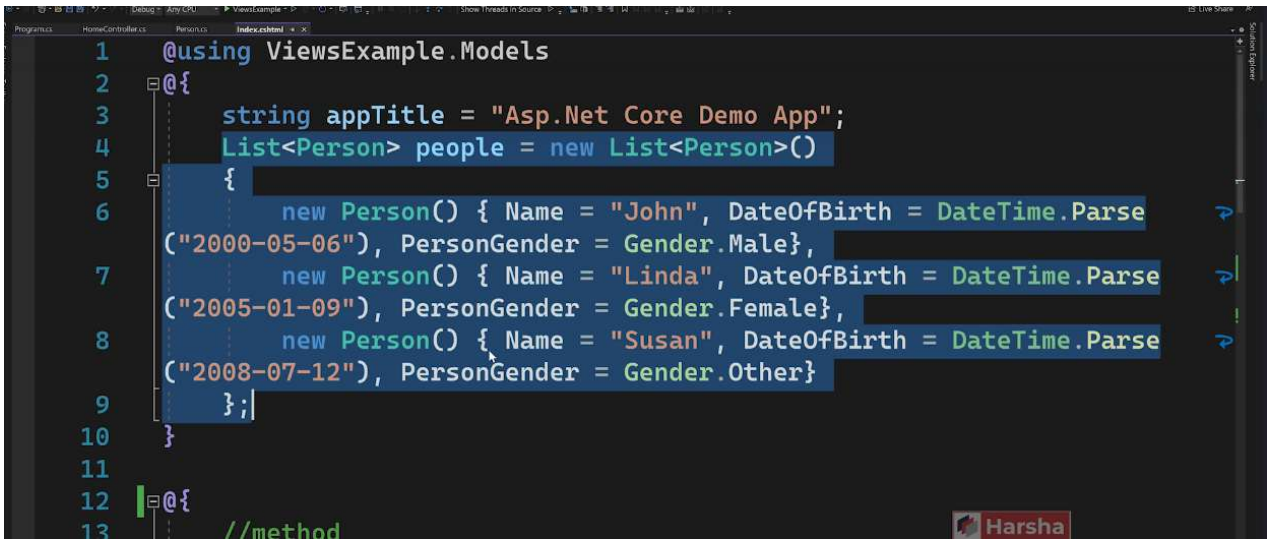


Razor View internally converted as C# class

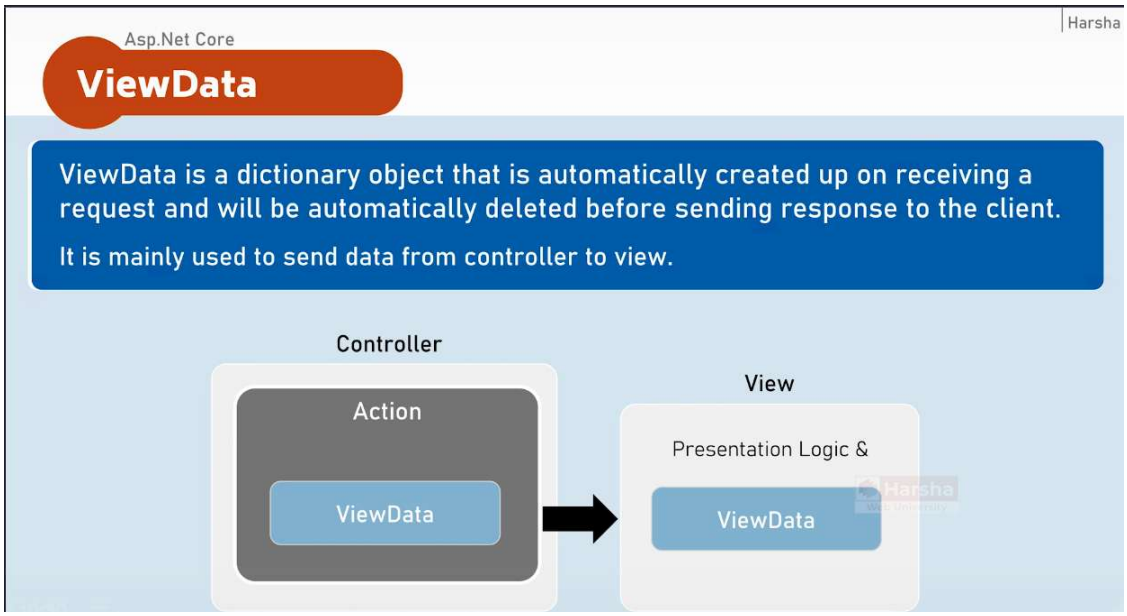


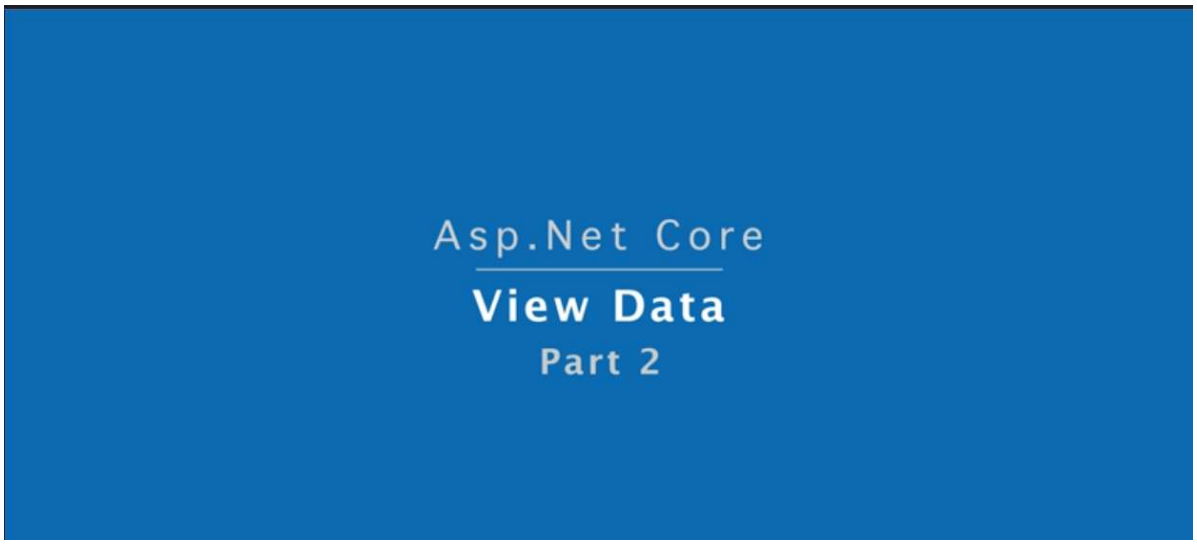
Suppose you have stored the HTML code or JavaScript code in a variable, and you would like to print it, meaning you would like to execute that HTML or JavaScript code.
By default, it is not supported. ASP.NET Core will sanitize that HTML or JavaScript code, which means it converts the code into a non-executable format and prints the same code as it is instead of executing it.
However, if you want to execute that HTML or JavaScript code, you have to use Html.Raw and pass the variable that contains your HTML or JavaScript code.
This way, instead of sanitizing it, the code directly executes, and you will get the corresponding effect or output.



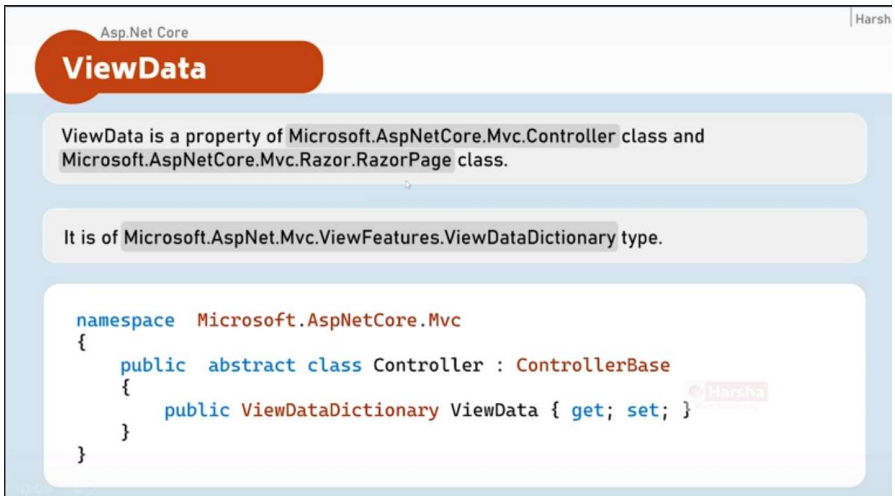


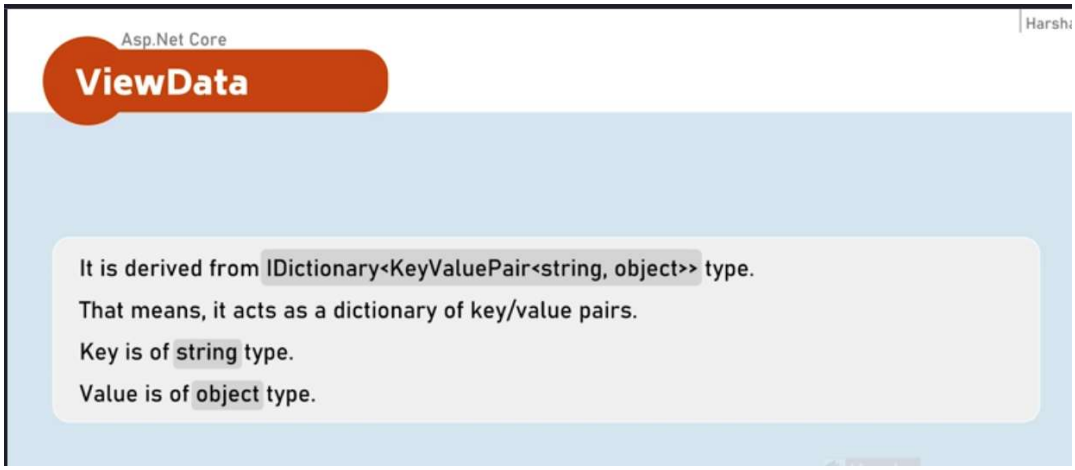
So, what do you think? Is it a good way to initialize the variables, collections, or model objects in the same view like this?
Certainly not.
The typical relationship between the controller and view is: **controller supplies, view uses.** This means the view receives the data from the controller and prints it in the appropriate place in the design.
Whether it is a model object, a collection of model types, or any other useful data, all such data should be supplied from the controller to the view.
But what is the way to supply the data from the controller to the view? That is through the **ViewData object.**





localhost:port-number





The only problem with ViewData is that when you're retrieving a value, you have to typecast it into the appropriate type.

If it is a single object of a particular model class, you have to convert it into the model class type.

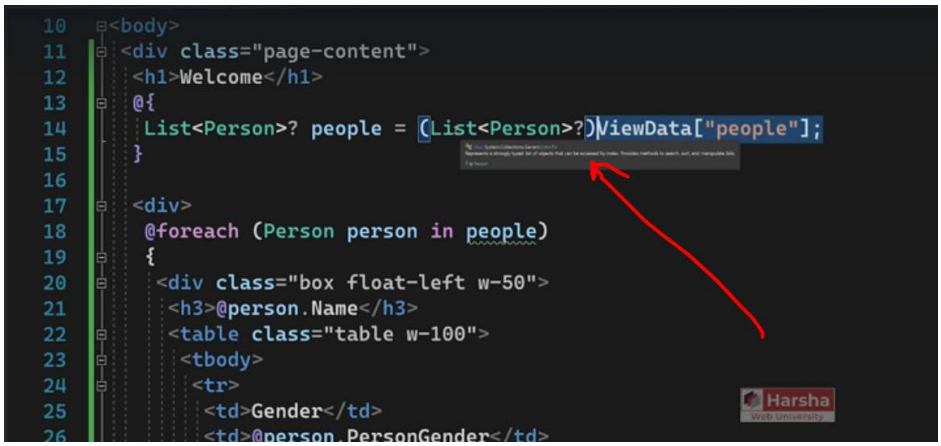
If it is a collection, you have to write something like List<Person>.
So, typecasting is a must.

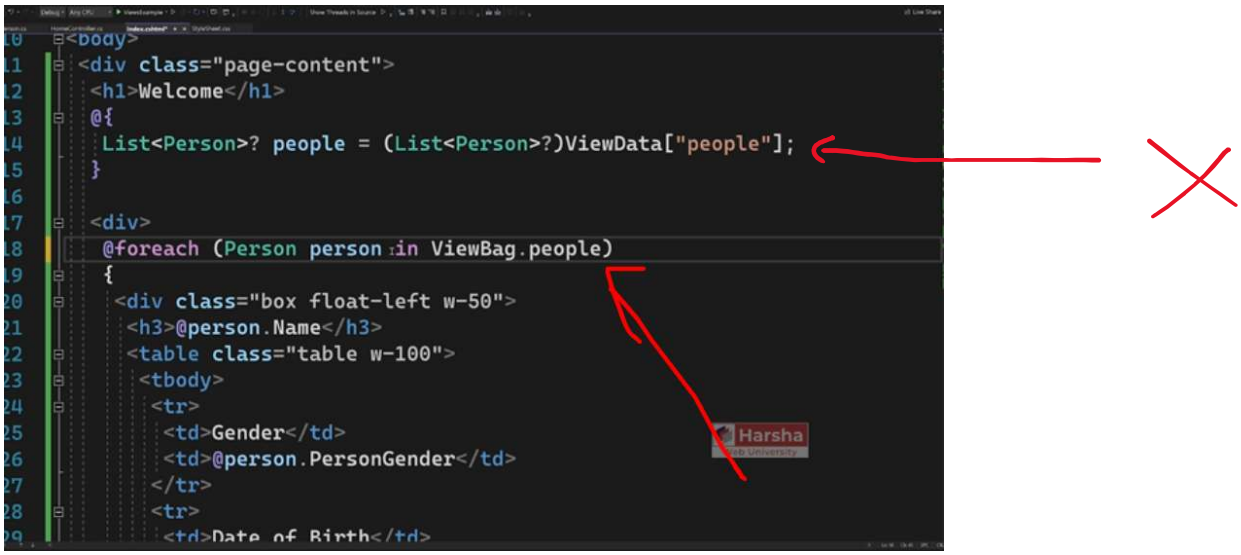
To fix this issue, ViewBag has been introduced. With the help of ViewBag, you can directly access data since it is a predefined property.

For example, you can write:
ViewBag.<KeyName>

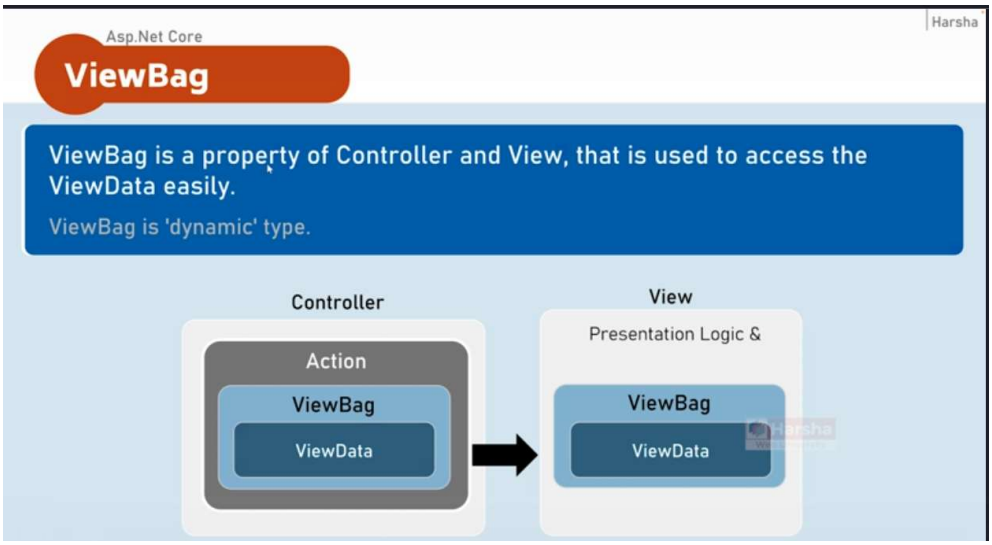
Here, the key of the ViewData becomes a property of ViewBag.

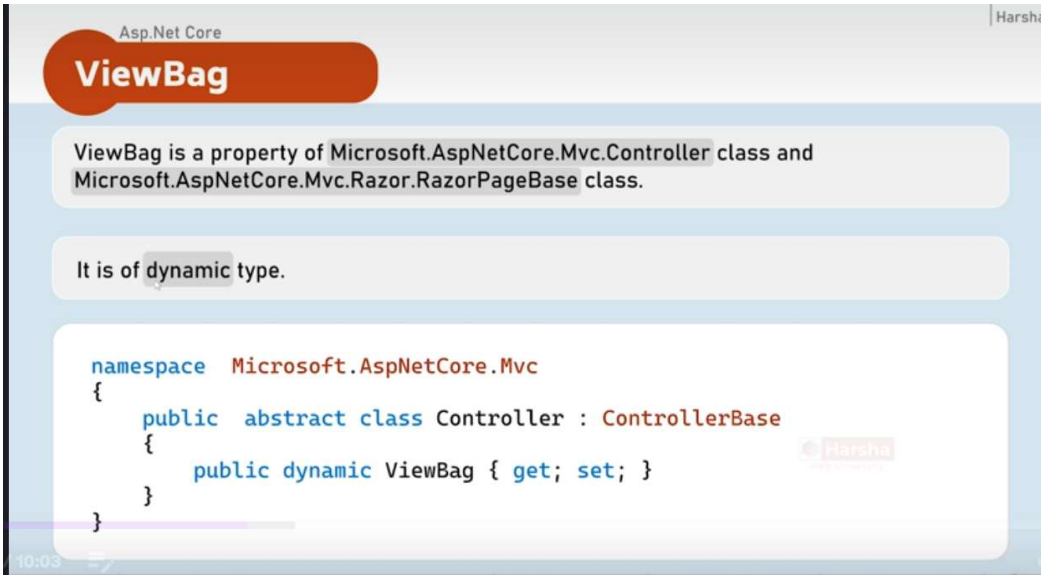
That's all!

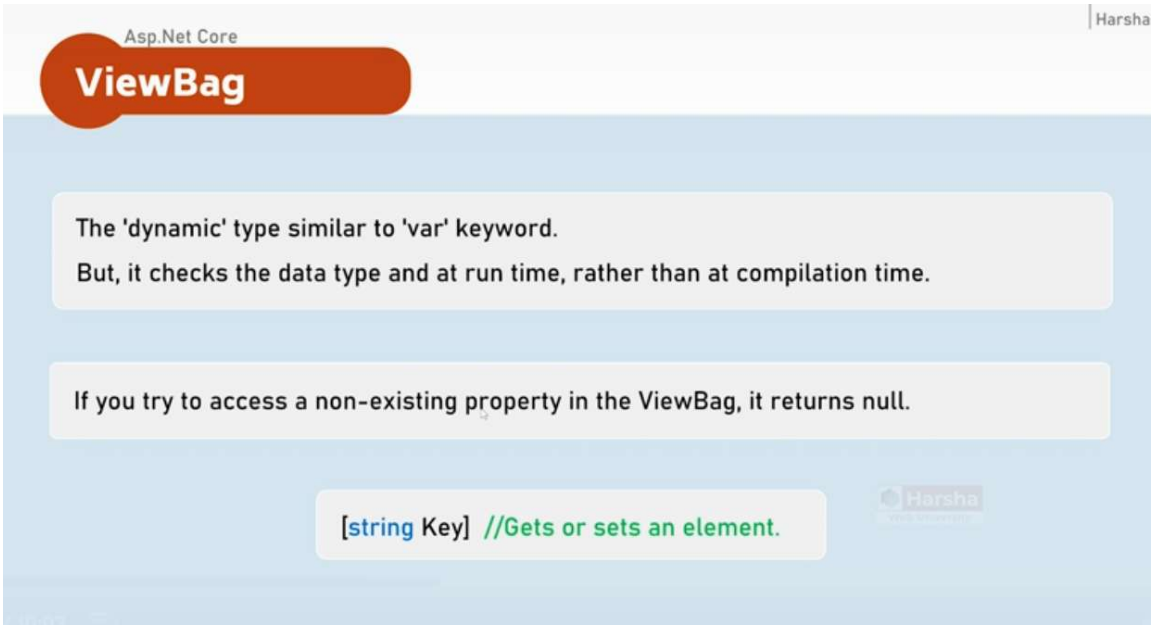


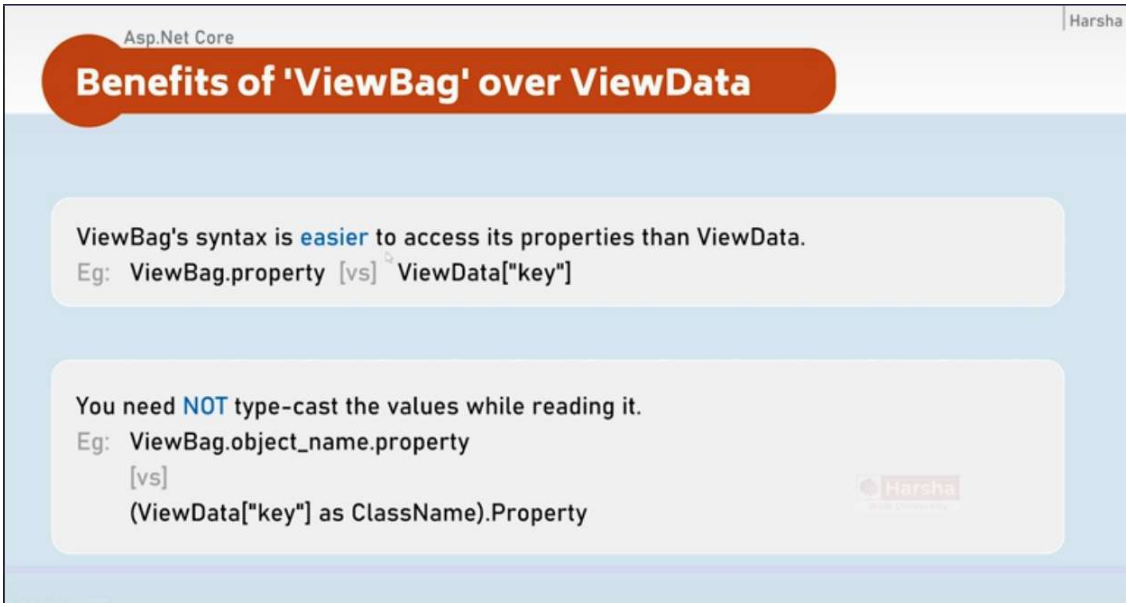


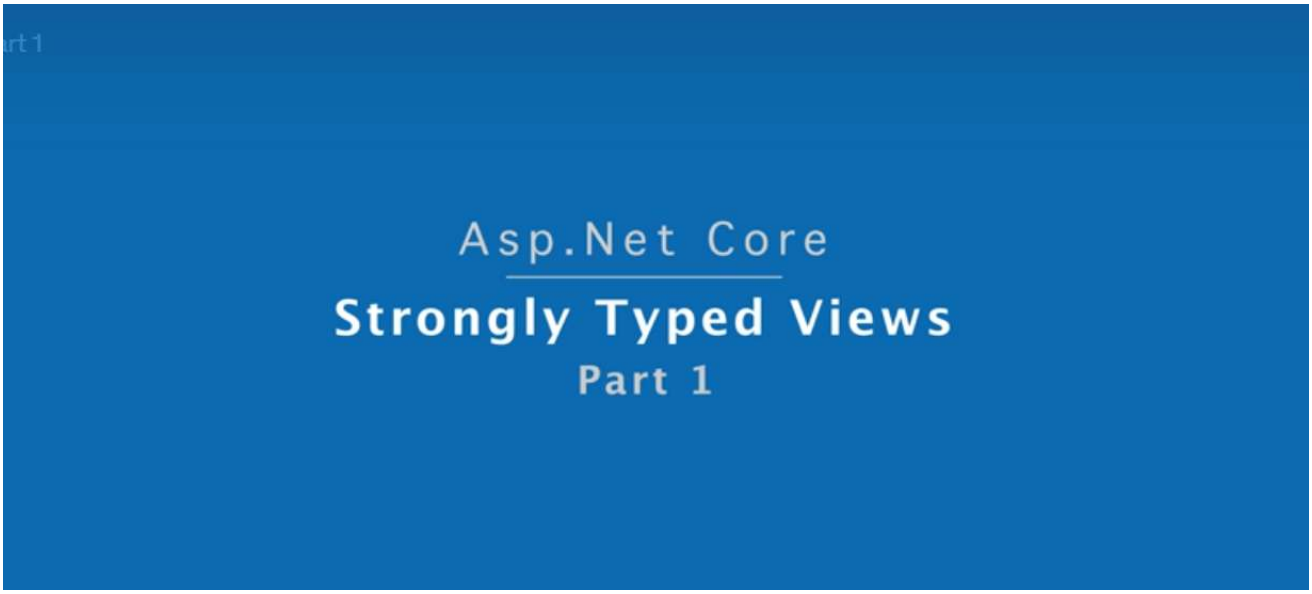
Internally, ***ViewBag*** reads data from ***ViewData.*** **ViewBag is not a separate object. It is another way of reading data from VeiwData.**

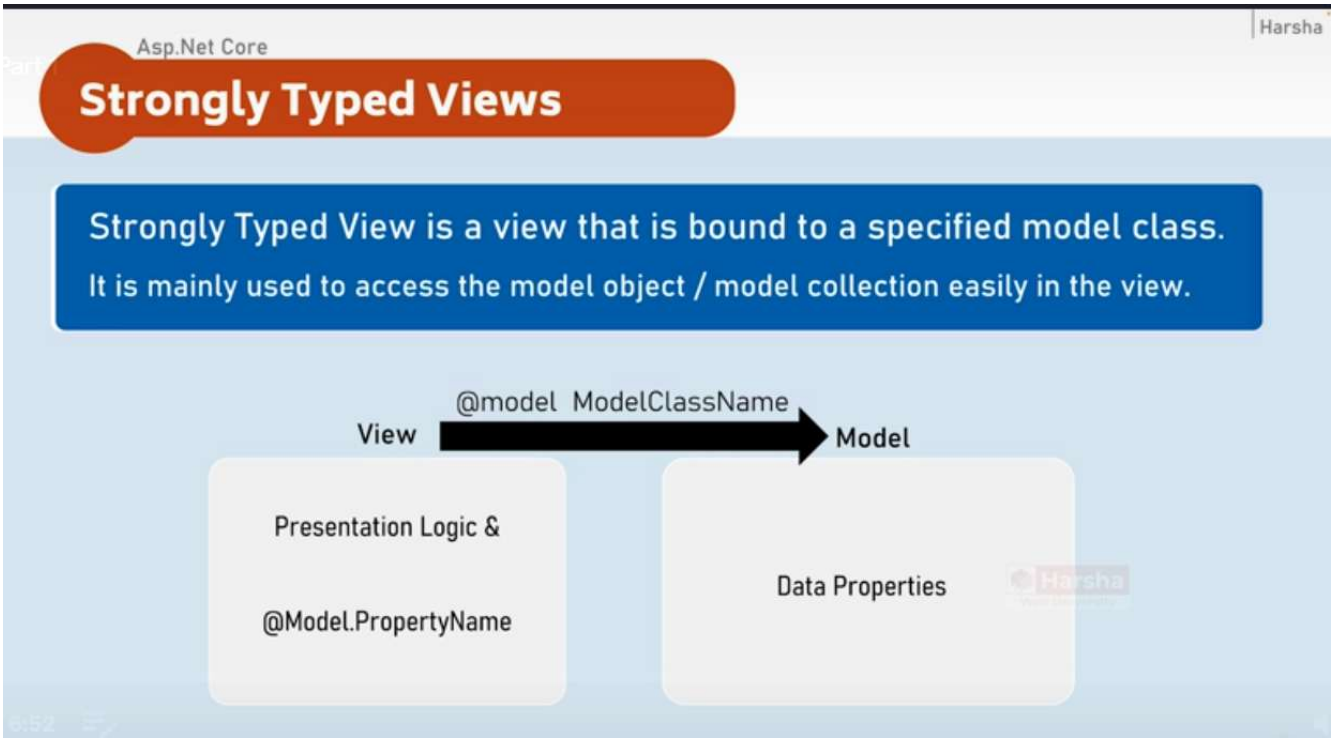







Harsha
Asp.Net Core
Benefits of 'ViewBag' over ViewData
ViewBag's syntax is easier to access its properties than ViewData.
Eg: ViewBag.property [vs] ViewData["key"]
You need NOT type-cast the values while reading it.
Eg: ViewBag.object_name.property
[vs]
(ViewData["key"] as ClassName).Property
Harsha


Asp.Net Core
Strongly Typed Views
Part 1



To make it easier to access the model object or model collection in the view, the strongly typed views concept is introduced.

A strongly typed view is tightly bound to a specific model class.

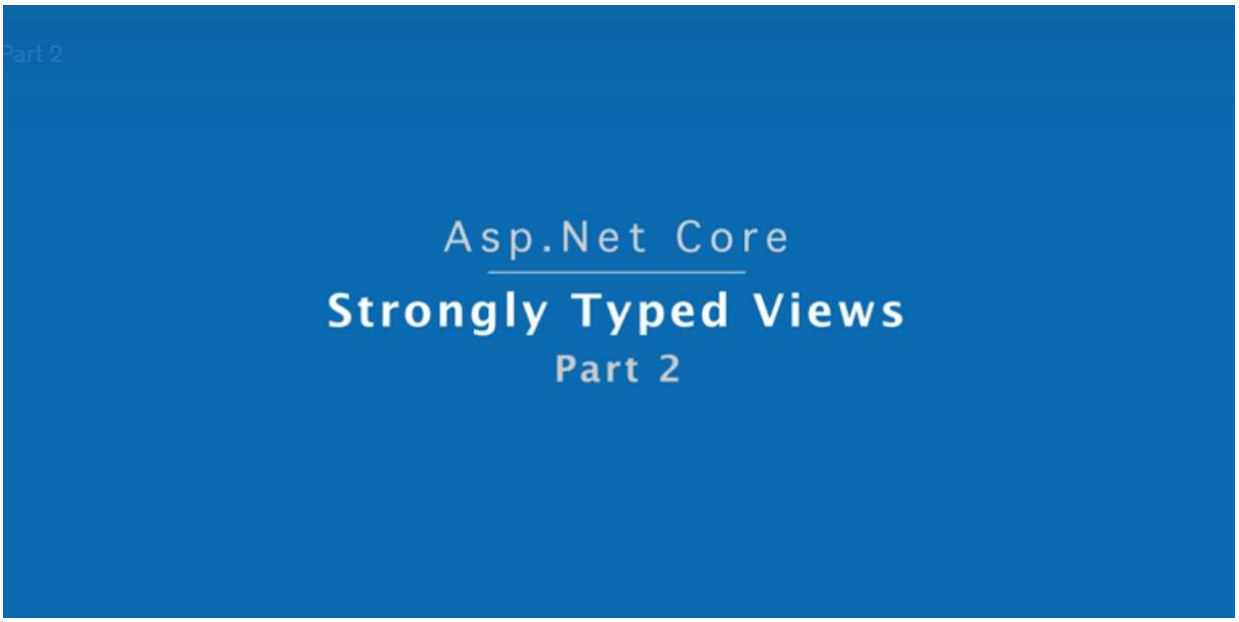
For example, consider a view that needs to display a list of persons. It can bind to a specific model class called Person.

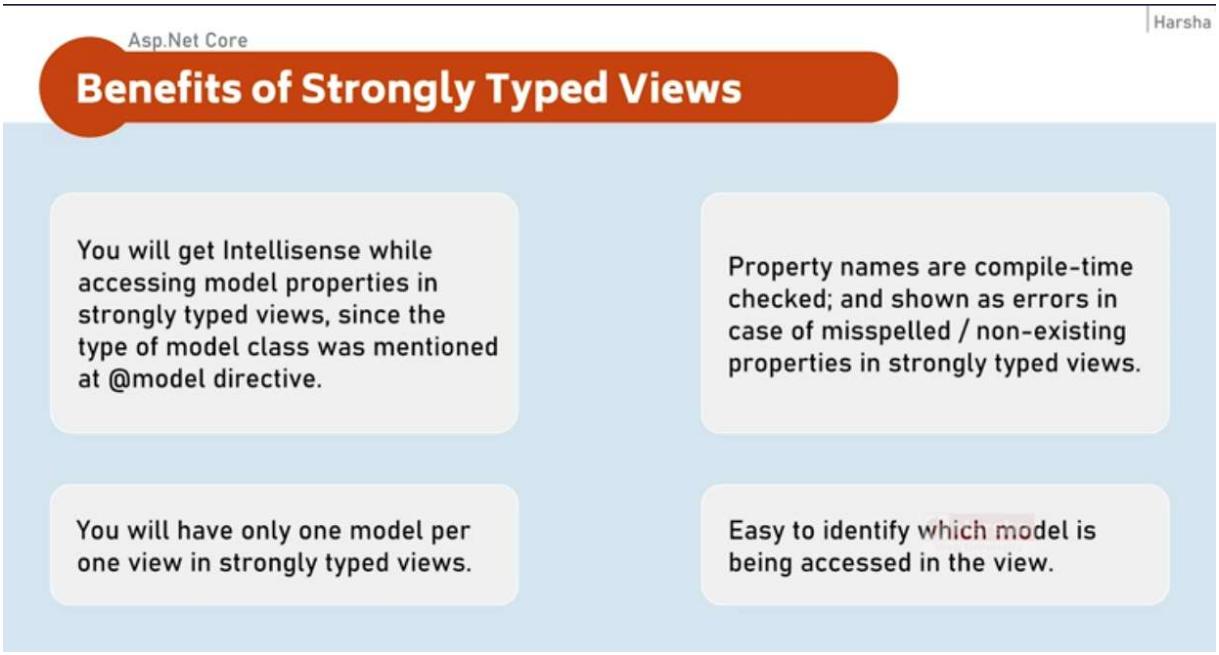
To declare this binding, you use the @model directive.

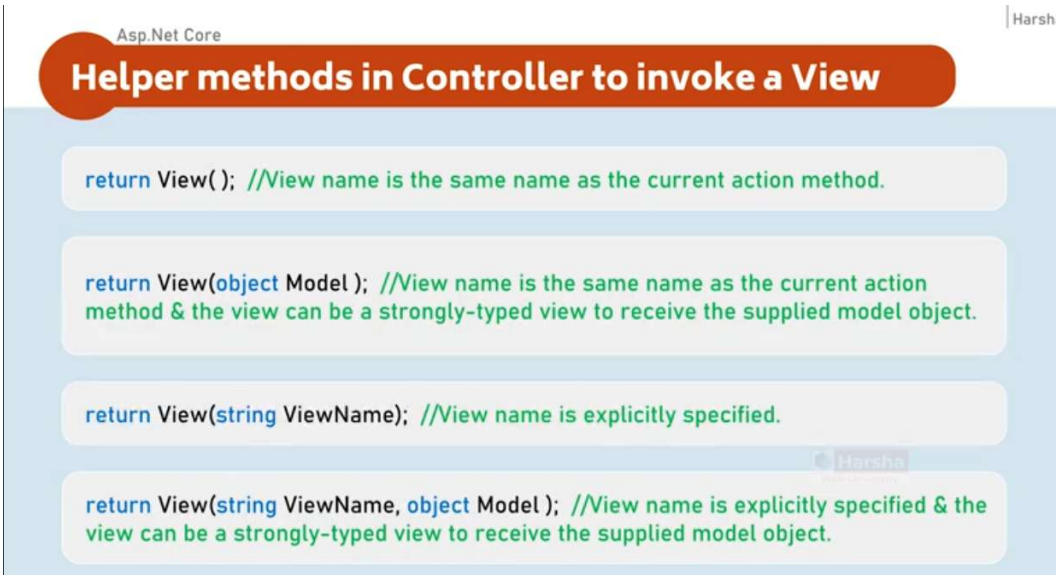
At the very top of the view, as the first statement, you can write:
@model <ModelClassName>

This indicates that the view has been tightly coupled with the specified model class.

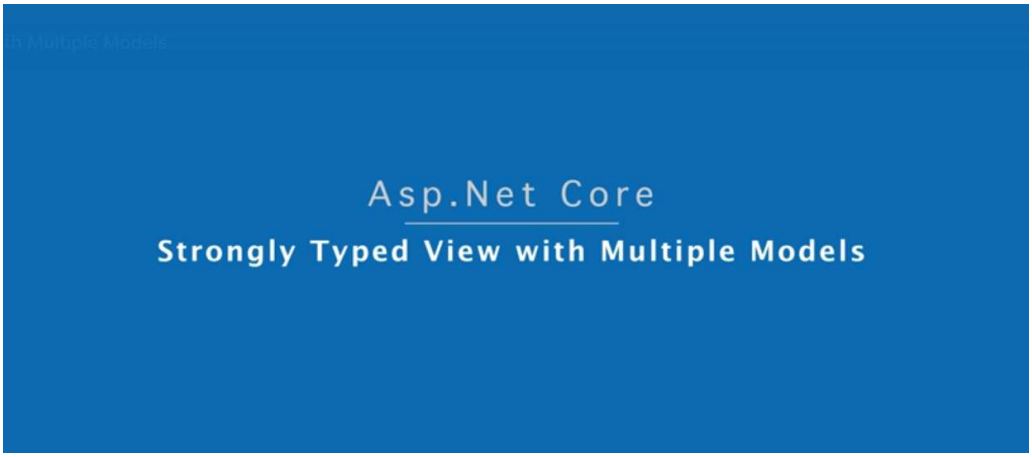
Benefits:
You can directly access the model object that is supplied from the controller in the view.
For instance, in the view, you can declare: @model List<Person>
Then, you can access all the corresponding model properties.
Here, the @Model is a predefined property in the view that contains the model object supplied by the controller.


Part 2
Asp.Net Core
Strongly Typed Views
Part 2


Harsha
Asp.Net Core
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.



There is a problem with strongly typed views.
A **strongly typed view** can be bound to only a single model class. This means it is designed to receive only the object of the specified type (e.g., Person).
However, in some cases, the same view might require information from two different models at the same time.
Technically, this is not possible with a strongly typed view.
Workaround:
Fortunately, there is a workaround to address this limitation. We will explore this solution in the next lecture.

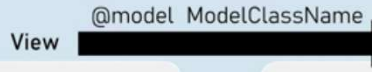


Asp.Net Core

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.



Presentation Logic &

@Model.Property1

@Model.Property2

```
public Model1 Property1 { get; set; }
public Model2 Property2 { get; set; }
```

Asp.Net Core

_ViewImports

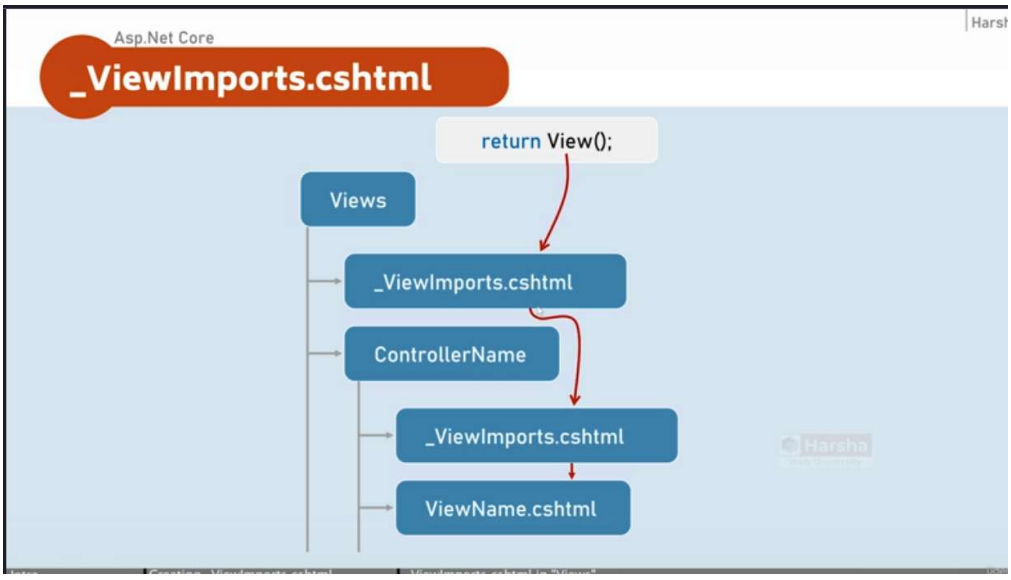
```
1  @using ViewsExample.Models
2  @model Person
3
4  <!DOCTYPE html>
5  <html>
6  <head>
7      <title>Person Details</title>
8      <meta charset="UTF-8" />
9      <link href="~/StyleSheet.css" rel="stylesheet" />
10 </head>
11 <body>
12     <div class="page-content">
13         <h1>Person Details</h1>
14         <div class="box w-50">
15             <h3>@Model.Name</h3>
16             <table class="table w-100">
```

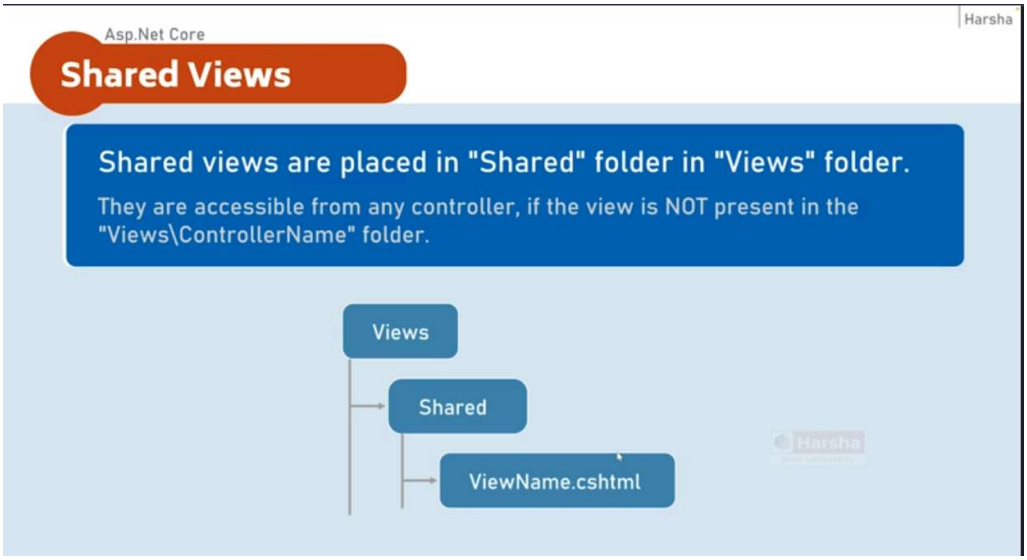
Here's the refined version of your text:

In almost all views, we are commonly importing a namespace called the **model's namespace**.
However, we are writing the using statement to import this namespace **manually in every view**, such as:

- **Details view**
- **Index view**
- **PersonWithProduct view**

Repetition Problem:
Manually repeating the same namespace in every view can be redundant and error-prone.
Solution:
To avoid this repetition, you can import the namespace **once for all views** by using the _ViewImports.cshtml file.





If you place a view inside the **Shared** folder within the **Views** folder, that view becomes **accessible from any controller** in the entire application.
This means the view becomes a **shared view** that can be used across **all controllers** in the project. Here, folder name 'Shared' is fixed.