

COMP 2139

How to work with model binding



Agenda

- Describe how to use controller properties to retrieve primitive types from GET and POST requests
- List the order of places where MVC looks for data when its binding a parameter
- Describe how to use model binding to retrieve primitive types from GET and POST requests
- Describe how to use model binding to retrieve complex types from POST requests
- Describe how to use the name and value attribute of a submit button to POST data.
- Describe how to post an array to an action method
- Describe the use of the attributes to control the source of bound values
- Describe the use of binding to control which properties are set during model binding.

What is Model Binding

Model Binding

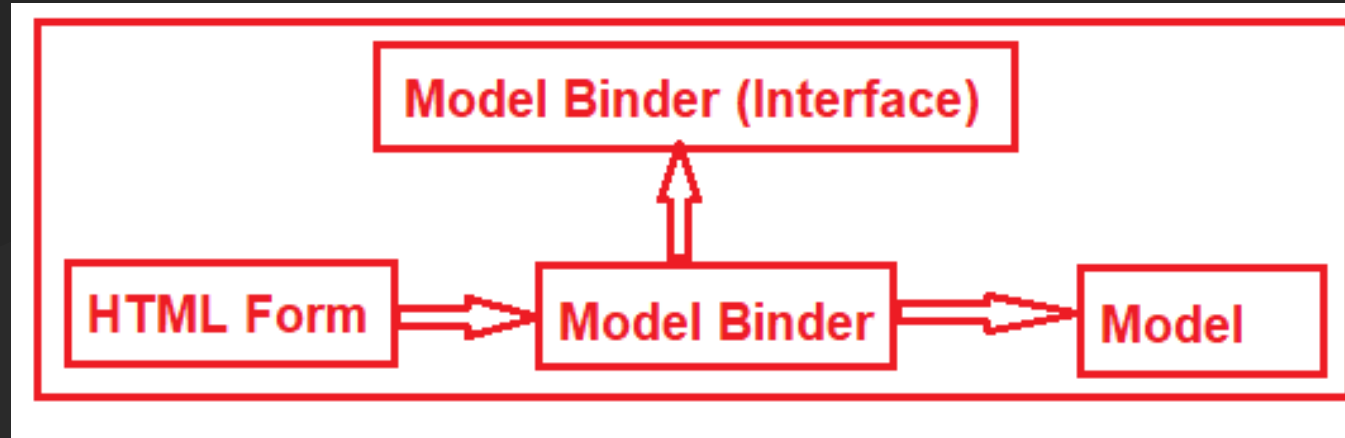
ASP.NET MVC model binding allows you to map **HTTP request data with a model**. It is the process of creating .NET objects using the data sent by the browser in an HTTP request.

In ASP.Net MVC applications, the values from a View get converted to the Model class when it reaches the Action method of the Controller class, this conversion is done by the **Model binder**.

Model Binding, is a well-designed bridge between the HTTP request and the C# action methods. It makes it easy for developers to work with data on forms (views), because POST and GET is automatically transferred into a data **model** you specify.

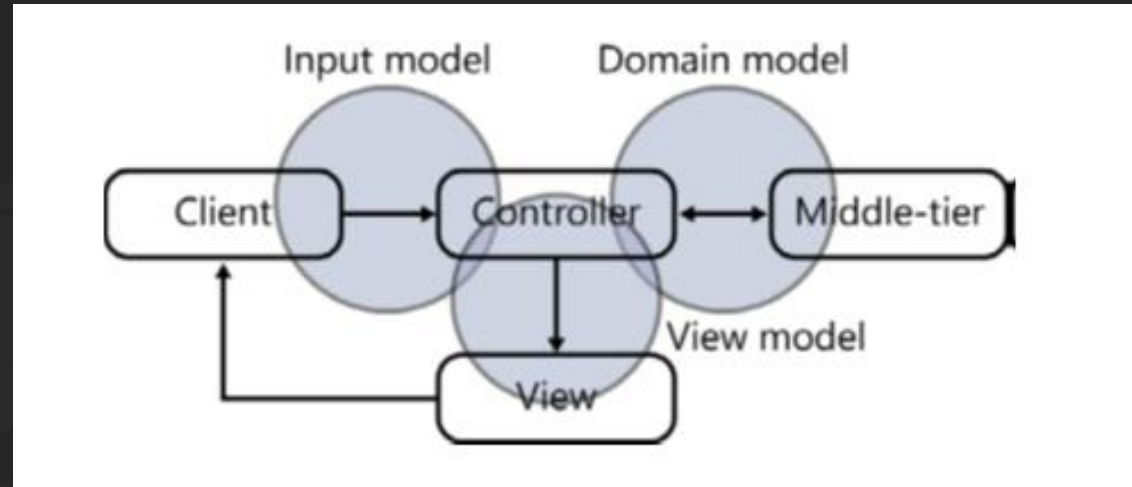
ASP.NET MVC uses default binders to complete this behind the scene.

Model Binding – High Level



- Model binding is a well-designed bridge between the HTTP request and the C# action methods.
- It makes it easy for developers to work with data on forms (views)
- Both POST and GET request are automatically transferred into the data model we specify.
- ASP.NET MVC uses default binders to complete this behind the scene

Model Binding Architecture in ASP.NET MVC



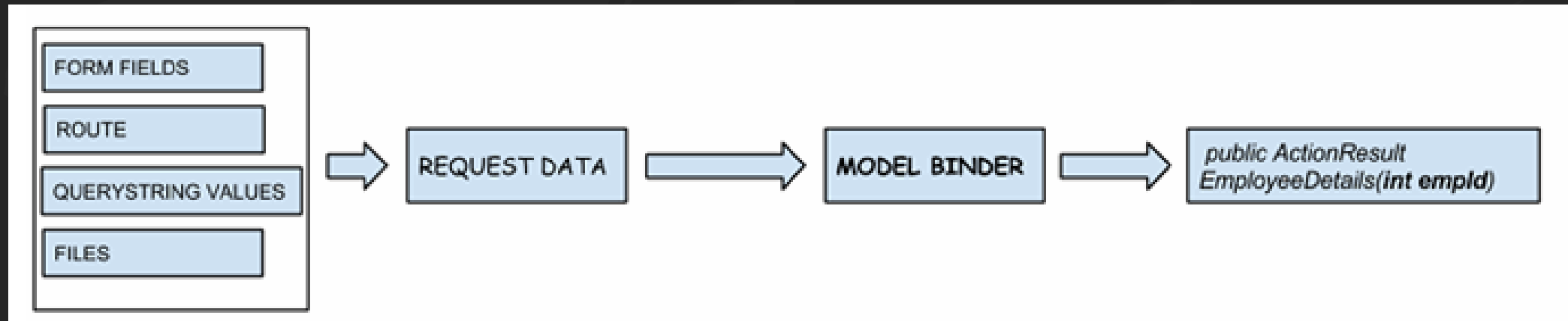
Input Model → It provides the representation of the data being posted to the controller
→ describes the request data you are provided and are working with

Domain Model → It is the representation of the domain-specific entities operating in the middle tier
→ Pushes a vision of the data that is, likely, distinct from the input mode or view layer

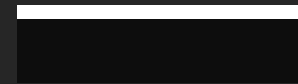
View Model → It provides the representation of the data being worked on in the view
→ Simply describes a vision of the data to the presentation layer

Model Binding Process

We can further visualize the Model Binding process with the following diagram...



Controller Properties



Controller Properties

Two properties of the Controller class

Property	Description
Request	Represents the HTTP request sent from the browser to the server
RouteData	Represents the MVC route for the current request

Two properties the Request property

Property	Description
Query	A dictionary that holds the query string parameters in the URL
Form	A dictionary that holds the form values in the body of a POST request

One property of the RouteData

Property	Description
Values	A dictionary that holds the route data for the current request, including the current controller, action method, and named route parameter.

How to use controller properties to retrieve GET and POST data

A URL with a query string parameter

`https://localhost:5001/Home/Index?page=2`

A URL with a query string parameter

```
public IActionResult Index() {  
    ViewBag.Page = Request.Query["page"];  
    return View();  
}
```

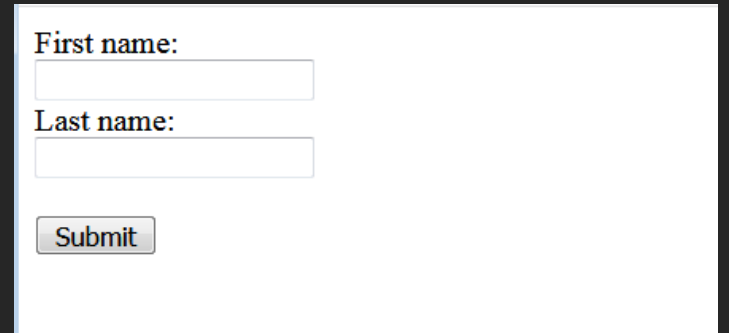
How to use controller properties to retrieve GET and POST data...

Form data in the body of a POST request

```
firstname=Grace
```

An action method that retrieves the form data

```
public IActionResult Index() {  
    ViewBag.FirstName = Request.Form["firstname"];  
    return View();  
}
```



First name:

Last name:

How to use controller properties to retrieve GET and POST data...

A URL with a value for the id parameter of the default route

`https://localhost:5001/Home/Index/all`

```
endpoints.MapControllerRoute(  
    name: "default",  
    pattern: "{controller=Home}/{action=Index}/{id?}");
```

An action method that retrieves the value of the route parameter named id

```
public IActionResult Index() {  
    ViewBag.Id = RouteData.Values["id"];  
    return View();  
}
```

How to use model binding to retrieve GET and POST data

An action that uses model binding to get a string value

```
public IActionResult Index(string id)
{
    ViewBag.Id = id;
    return View();
}
```

Three types of data this method can retrieve

1. A Form parameter in the body of a POST request

`id=2`

2. A Route parameter

`https://localhost:5001/Home/Index/2`

3. A query string parameter

`https://localhost:5001/Home/Index?id=2`

Highest



Lowest

The order in which MVC looks for data to bind to a parameter

1. The body of the **POST** request.
2. The **route** values in the URL.
3. The **query string** parameters in the URL.

The benefits of model binding

- You don't have to write repetitive code to retrieve values.
- MVC automatically casts the value to match the data type of the action method parameter.
- Model binding is NOT case sensitive.
- You can change how you pass data to an action without having to change its code.

An action method and view that bind to primitive types

The action method

```
[HttpPost]
public IActionResult Add(string description, DateTime dueDate) {
    ToDo task = new ToDo {
        Description = description,
        DueDate = dueDate
    };
    // rest of code
}
```

The view

```
<form asp-action="Add" method="post">
    <label for="description">Description:</label>
    <input type="text" name="description">
    <label for="dueDate">Due Date:</label>
    <input type="text" name="dueDate">
    <button type="submit">Add</button>
</form>
```

You must make sure the name of the action method parameters matches the names posted by the view

How to use model binding for complex types

The class for a complex types

```
public class ToDo
{
    public int Id { get; set; }
    public string Description { get; set; }
    public DateTime? DueDate { get; set; }
}
```


An action method and view that bind to a complex type

The action method

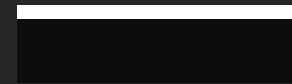
```
[HttpPost]
public IActionResult Add(ToDo task) {
    // rest of code
}
```

The view

```
@model ToDo
...
<form asp-action="Add" method="post">
    <label asp-for="Description">Description:</label>
    <input asp-for="Description">
    <label asp-for="DueDate">Due Date:</label>
    <input type="text" asp-for="DueDate">
    <button type="submit">Add</button>
</form>
```

ASP.Net MVC automatically initializes the object when you bind to a complex type. To do that, it looks for request or route parameters with the same names as the object properties.

Model Binding in Chapter Source Project (Ch10aNFLTeams)



Ch10aNFLTeams

The custom route in the NFL Teams (Ch10a) application

```
app.UseEndpoints(endpoints =>
{
    endpoints.MapControllerRoute(
        name: "",
        pattern: "{controller=Home}/{action=Index}/conf/{activeConf}/div/{activeDiv}");

    endpoints.MapControllerRoute(
        name: "default",
        pattern: "{controller=Home}/{action=Index}/{id?}");
});
```

The base class for the TeamListViewModel

```
namespace NFLTeams.Models
{
    6 references
    public class TeamViewModel
    {
        10 references
        public Team Team { get; set; }
        9 references
        public string ActiveConf { get; set; } = "all";
        9 references
        public string ActiveDiv { get; set; } = "all";
    }
}
```

- Make sure the complex type's, property names match the custom route parameters names.
- This makes it easy to create/update the action method so it binds to the model type.

The Home/Index() action that binds to a complex type

```
0 references
public IActionResult Index(TeamListViewModel model)
{
    model.Conferences = context.Conferences.ToList();
    model.Divisions = context.Divisions.ToList();

    var session = new NFLSession(HttpContext.Session);
    session.SetActiveConf(model.ActiveConf);
    session.SetActiveDiv(model.ActiveDiv);

    // if no count value in session, use data in cookie to restore fave teams in session
    int? count = session.GetMyTeamCount();
    if (count == null) {
        var cookies = new NFLCookies(HttpContext.Request.Cookies);
        string[] ids = cookies.GetMyTeamIds();

        List<Team> myteams = new List<Team>();
        if (ids.Length > 0)
            myteams = context.Teams.Include(t => t.Conference)
                                   .Include(t => t.Division)
                                   .Where(t => ids.Contains(t.TeamID)).ToList();
        session.SetMyTeams(myteams);
    }

    IQueryable<Team> query = context.Teams;
    if (model.ActiveConf != "all")
        query = query.Where(
            t => t.Conference.ConferenceID.ToLower() == model.ActiveConf.ToLower());
    if (model.ActiveDiv != "all")
        query = query.Where(
            t => t.Division.DivisionID.ToLower() == model.ActiveDiv.ToLower());
    model.Teams = query.ToList();

    return View(model);
}
```

MVC in this example creates a new **TeamListViewModel** parameter and populates it with **ActiveConf** and **ActiveDiv** properties with the matching route parameters (if not passed, default “all” is used for **ActiveDiv** and **ActiveConf** instead).

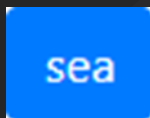
After that, the action method code populates more of the properties of the object and before sending to the view.

Two ways to code a Submit button

An `<input>` element for a button that posts a team ID to the action method

```
@model Team;
...
<form asp-action="Add" method="post">
    <input type="submit" asp-for="TeamID" class="btn btn-primary" />
</form>
```

How the button looks in the browser



→ limitation


- The `<input>` element allows you to use the `asp-for` tag to generate the name and the value attributes
- Automatically displays the value attribute as the text for the button

Two ways to code a Submit button...

An <button> element for a button that posts a team ID to the action method

```
@model Team;
...
<form asp-action="Add" method="post">
    <button type="submit" name="TeamID" value="@Model.TeamID"
        class="btn btn-primary">@Model.Name</button>
</form>
```

How the button looks in the browser



- The <button> element can provide greater control over the text (or image) that's displayed on the button
- Requires you to:
 - manually code the name and value attributes
 - make sure the value in the name attribute matches the name of the action method parameter/property

Two ways to code a Submit button...

The `<input>` element

- Allows you to use the `asp-for` tag helper to generate the name and value attributes.
- Automatically displays the **value attribute** as the text for the button.

The `<button>` element

- Gives you control over the text (or image) that's displayed on the button.
- Requires you to manually code the `name` and `value` attributes.
- Requires you to make sure the value in the `name` attribute matches the name of the action method parameter / property.

How to post an array to an action method

An action method that accepts a string array

```
[HttpPost]
public IActionResult Filter(string[] filter)
{
    // code that does the filtering
    return View();
}
```

- When elements of a form, have the same name, the values of the elements automatically **post to the server as an array**
- Useful is when you want to filter data.

How to post an array to an action method

```
<h3>Filter By:</h3>
<form asp-action="Filter" method="post">
  <!-- make sure each select element has the same name
       and that it matches the action method parameter name -->
  <label>Price</label>
  <select name="filter">
    <option value="all">All</option>
    <option value="lt10">Under $10</option>
    <option value="10to50">$10 to $50</option>
    <option value="gt50">Over $50</option>
  </select>
  <label>Color</label>
  <select name="filter">
    <option>All</option>
    <option>Red</option>
    <option>Blue</option>
    <option>Yellow</option>
    <option>Green</option>
    <option>Purple</option>
  </select>
  <button type="submit">Filter</button>
</form>
```

- Both combo boxes in the form have the same name.
- As a consequence, when the form posts, MVC collects the selected values of **each** `<select>` element in a string array.

How to post an array to an action method

<input type="checkbox"/>	filter	{string[2]}
<input type="checkbox"/>	filter[0]	⌵ "lt10"
<input type="checkbox"/>	filter[1]	⌵ "Blue"

← example passed (submitted) parameters for both dropdowns

Summary

- You can pass an array to an action method by coding multiple HTML elements
 - The elements must that have the same **name** within a form
- The name of ther HTML elements needs to match the name of the action method parameter/property
- The action method parameter or property needs to be of the IEnumerable type, such as an array or a List<T> type.

How to control the source of bound values



How to control the source of bound values

As we learned earlier, MVC model binding automatically checks for names in multiple places and in a specific sequence. To start, it checks **posted data**, then **route data**, then **query string data**... However in some instances you might want more control over this sequence.. Or you simply might want to bind data that is sent in other ways.

Attributes that specify the source of the value to be bound

Attribute	
[FromForm]	From the form parameters in the body of POST request
[FromRoute]	From the route parameters of the URL
[FromQuery]	From the query string parameters of the URL
[FromHeader]	From the HTTP request header
[FromServices]	From a service that is injected into the application
[FromBody]	From the body of the HTTP request . This is often used when a client-side script sends JSON data to an action method. This attribute can only decorate one parameter per action method.

An action method that specifies the source of its parameters

```
public IActionResult Index([FromRoute] string id, [FromQuery] int pagenum)
{
    ViewBag.Id = id;
    ViewBag.Page = pagenum;
    return View();
}
```

In this example,

- the action method tells MVC to get the value of the `id` from the `URLs Route segment`.
- the action method tell MVC to get the value for the `pagenum` from `the URLs query string`.

An action method passes an argument to an attribute

```
public IActionResult Index([FromHeader(Name = "User-Agent")] string agent)
{
    ViewBag.UserAgent = agent;
    return View();
}
```

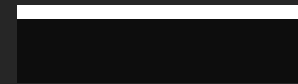
- In this example, the action method tells MVC to get the value for the agent action parameter from the **HTTP Request Header, User-Agent** property.

An class that applies an attribute to a property

```
public class Browser
{
    [FromHeader(Name = "User-Agent")]
    public string UserAgent { get; set; }
    ...
}
```

- This example shows that you can apply “*From*” attributes to class properties as well as action method.
- Here MVC always looks in the **HTTP Request Header**, “**User-Agent**” for the value of the UserAgent property and no where else.

How to control which values are bound



How to control which values are bound

Previously , we learned how to control *where* MVC looks when binding values to parameters, in an action method. In this section we will review how to control *which* properties of a complex type are bound.

Two attributes that determine which values are bound

Attribute	Description
[Bind(names)]	Allows you to list the names of the properties that can be set during model binding. This attribute can be applied to a <u>model</u> or to a <u>parameter of an action method</u> .
[BindNever]	Indicates that a property should <u>never</u> be set during model binding. This attribute can <u>ONLY</u> be applied to <u>model properties</u> .

The namespace of Attributes


Attribute	Namespace
[Bind]	Microsoft.AspNetCore.Mvc
[BindNever]	Microsoft.AspNetCore.Mvc.ModelBinding

How to control which values are bound

Example:

Three different ways to prevent the **IsManager** property from being bound

```
public class Employee {  
    public string Name { get; set; }  
    public string JobTitle { get; set; }  
    public bool IsManager { get; set; }  
}
```



Preventing a Property from being bound (Ex 1)

EXAMPLE 1 of 3

With the Bind attribute in the parameter list

```
[HttpPost]
public IActionResult Index([Bind("Name", "JobTitle")] Employee employee) {
    if (ModelState.IsValid) {
        if (employee.JobTitle == "Boss")
            employee.IsManager = true;    // can be set in code
    }
    return View(employee);
}
```

- Notice how the IsManager property is explicitly removed from the Bind() listing.

Preventing a Property from being bound (ex 2)

EXAMPLE 2 of 3

With the Bind attribute on the class

```
[Bind("Name", "JobTitle")]  
public class Employee {  
    public string Name { get; set; }  
    public string JobTitle { get; set; }  
    public bool IsManager { get; set; }  
}
```

- Annotate the class, explicitly omitting the IsManager property from the Bind() property listing

Preventing a Property from being bound (ex 3)

EXAMPLE 3 of 3

With the **BindNever** attribute on the **IsManager** property

```
public class Employee {  
    public string Name { get; set; }  
    public string JobTitle { get; set; }  
  
    [BindNever]  
    public bool IsManager { get; set; }  
}
```

- Annotate the **IsManager** property with the **BindNever** attribute in model class

Questions?