`public static string DropDownList(this HtmlHelper htmlHelper, string name, IEnumerable<SelectListItem> selectList, string optionLabel).`

The first parameter will set the name and id attributes of the HTML select element. If the second parameter is null, then Razor will also look for an IEnumerable<SelectListItem> in the ViewData with a key matching that name. In other words, if the name parameter is "WorksiteID," Razor is expecting an IEnumerable<SelectListItem> in ViewData["WorksiteID"] or ViewBag.WorksiteID (which are two different ways of expressing the same thing)

The final parameter accepts a generic object that we can use to set HTML attributes. In this example, we're using the parameter to set the class attribute to a Bootstrap CSS class. You could add properties to the object to set other attributes as well.

Using @Html.DropDownList()

This method works similar to first approach (DropDownListFor) but only difference is we are explicitly naming the DropDownList to match the Model property name (Employee.DepartmentId).

Bind using IEnumerable<SelectListItem>

```
List<SelectListItem> items = new List<SelectListItem>();
items.Add(new SelectListItem { Text = "IT", Value = "1" });
items.Add(new SelectListItem { Text = "HR", Value = "2" });
items.Add(new SelectListItem { Text = "Payroll", Value = "2" });
ViewBag.Departments = items;
```

Bind using List<T>

```
   @Html.DropDownList("DepartmentId", @ViewBag.Departments as IEnumerable<SelectListItem>, "Select One", new { @class = "form-control" })
```

```
<select class="form-control" id=" DepartmentId " name=" DepartmentId ">
        <option value="">Select Department</option>
        <option value="1">IT</option>
        <option value="2">HR</option>
        <option value="2">Payroll</option>
    </select>
```

na post

`DepartmentId: 1`

```
var selected = new[] { 1, 3, 7 };


List<SelectListItem> items = db.Subjects.Select(g => new SelectListItem
            {
                Text = g.name,
                Value = g.id.ToString()
                //,Selected = g.id == 2 // currently selected item
                //or
                //Selected = selected.Contains(c.CategoryID)
            }).ToList();

            items.Add(new SelectListItem { Text = "-Select Item-", Value = "-1", Selected
= true });


            ViewBag.SingleSubSub = items;
@Html.DropDownList("Department", @ViewBag.SingleSubSub as List<SelectListItem>, "Select
One", new { @class = "form-control" })

////////////////////////////////////////////////////////////////////////////////////////
```

Bind using List<T>

```
 //List<T>
            var subjects = db.Subjects.Select(c => new {
                id = c.id,
                name = c.name
            }).ToList();

//We need an IEnumerable<SelectListItem> in our Razor view, and creating a new instance
of the SelectList class does exactly this
                                                        //value  //Text od gore id i name
            ViewBag.SingleSub = new SelectList(subjects, "id", "name");


@Html.DropDownList("Department", @ViewBag.SingleSubSub as IEnumerable<SelectListItem>,
"Select One", new { @class = "form-control" })
or
@Html.DropDownList("DepartmentId", new SelectList(ViewBag.DepartmentList,
"DepartmentId","DepartmentName", Model.DepartmentId), "Select")
```

Using @Html.DropDownListFor()

Having a strongly-typed ViewModel makes your code more robust. Let's take a look at how we would do that. Here's our ViewModel class:

```
public class EmployeeViewModel
{
public int ID { get; set; }
public string Name { get; set; }
public int WorksiteID { get; set; }
public IEnumerable<Worksite> Worksites { get; set; }
}
```

In the controller, we'll want to populate the Worksites property with the Worksites from the database.

```
// GET: Employees/Create
public ActionResult Create()
{
var viewModel = new EmployeeViewModel();
viewModel.Worksites = db.Worksites;
return View(viewModel);
}
```

Finally, in the view, we'll want to change the model type from

`Employee` to `EmployeeViewModel` at the top of the file. Then, we'll change the line with `@Html.DropDownList` to this:

```
@Html.DropDownListFor(model => model.WorksiteID, new SelectList(Model.Worksites,
"ID", "Name"))
```

The first parameter is a lambda expression that identifies which property in the model we're going to bind to. This will set the `name` and `id` attributes to "WorksiteID". The second parameter is an `IEnumerable<SelectListItem>` that comes from our ViewModel.

Multiple

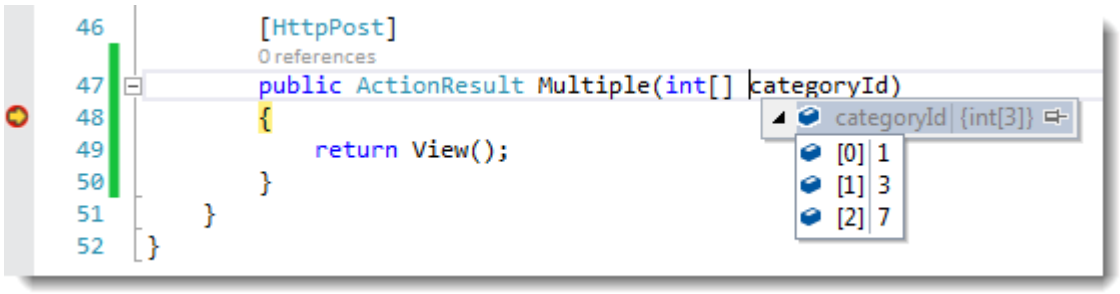
```
var subjects = db.Subjects.Select(c => new {
                id = c.id,
                name = c.name
            }).ToList();


            ViewBag.Sub = new MultiSelectList(subjects, "id", "name");
or (its same)
                                                  //value  //Text
            ViewBag.SingleSub = new SelectList(subjects, "id", "name");


@Html.DropDownList("CategoryId", ViewBag.Sub as IEnumerable<SelectListItem>, new { @class
= "form-control", multiple = "multiple" })
same as
@Html.ListBox("CategoryId", (MultiSelectList)ViewBag.Categories)

new MultiSelectList(categories, "CategoryID", "CategoryName", new[]{1,3,7});// selected will
be 1 3 7
```

get Multiple values



Enum

```
public enum Gender
{
Male,
Female
}
```

Copy and paste the following code in the index view

```
@using HTML_HELPER.Models
@Html.DropDownList("EmployeeGender",
new SelectList(Enum.GetValues(typeof(Gender))),
"Select Gender",
```

```
new { @class = "form-control" })
```

When we run the application, it will generate the following HTML

```
<select class="form-control" id="EmployeeGender" name="EmployeeGender">
<option value="">Select Gender</option>
<option>Male</option>
<option>Female</option>
</select>
```

ASP.NET MVC 5.1 saw the introduction of a new helper: `Html.EnumDropDownListFor`. As a strongly typed helper, this works with `enum` properties in view models. It takes the members of the enum and produces a select list with them, assigning the value to the option's `value` attribute and the enumerator to the text. Here's an example of an `enum`:

```
public enum Power
{
    Coal = 1,
    Gas,
    Hydro,
    Nuclear,
    Solar,
    Wave,
    Wind
}
```

This might be used to represent a selection of power generation options. Here's how it might appear as part of a view model:

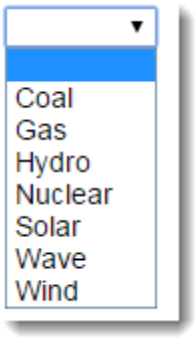
```
public class PowerViewModel
{
    public Power Power { get; set; }
}
```

And this is how the helper is used to render the options in the view:

```
@model PowerViewModel

<form method="post">
    @Html.EnumDropDownListFor(model => model.Power)
    <div>
        <input type="submit" />
    </div>
</form>
```

By default, the option with a value of 0 is selected. In this case, the enum values were intialised at 1 so the helper added an option with a value of 0 and empty text and made it selected:



If you want another option to be selected, you can set that in the view model:

```
var model = new PowerViewModel();
model.Power = Power.Hydro;
return View(model);
```

If you prefer a default option that says "Pick One" or similar, you can pass that value into the second argument for the helper method:

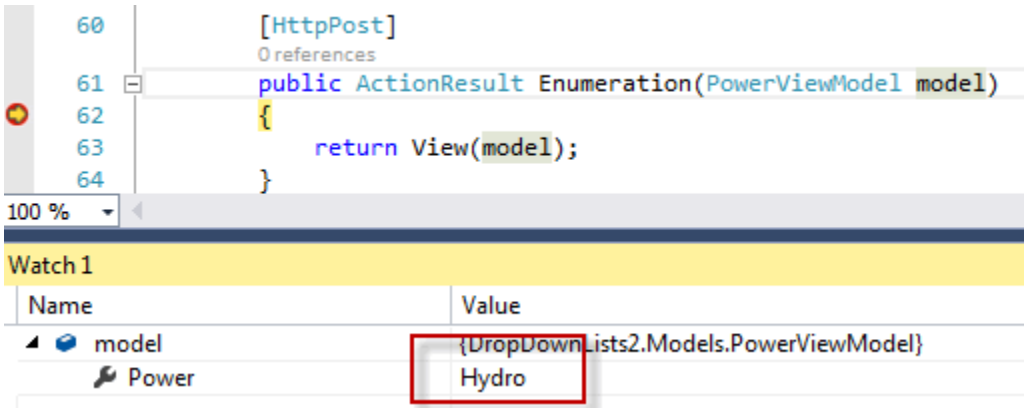
```
@Html.EnumDropDownListFor(model => model.Power, "Pick One")
```

If your enum starts at 0, this will result in an extra option with an empty value appearing in the list. However, the first enum (in my example, `Coal`) will still be selected by default. If you start your enum from 1, the "Pick One" option will be generated with a value of 0, which will be selected by default and will also form a valid selection if you make the property required via the DataAnnotation `[Required]` attribute. So what if you don't want the "Pick One" option being assigned a value and/or don't want the option with a value of 0 to be selected by default? You can solve both these problems by making the enum nullable in your view model:

```
public class PowerViewModel
{
    public Power? Power { get; set; }
}
```

And of course, you can still force the user to pick a valid value by adding the `[Required]` attribute to the property in the view model.

Finally, to retrieve the selected value, you ensure that the enum is included in the parameter list belonging to the action method that the form posts to, either on its own or as part of a view model. Here's the selected value being caught by the debugger when the form is posted back to the controller:



link

@Html.ActionLink("View", "details", new { id = employee.Id })

`@html.BeginForm()`

It's not a stupid question. `@html.BeginForm()` works like this. It has some parameters you could add to it like `Action` `Controller` `FormType` `htmlAttributes`. The way it works is that if you leave it empty it will look for a post action with the same name that on the page you are now, for example if you are in on the login page, it will look for a login post action. I always write what `action` and `controller` I want it to access.

@Html.BeginForm("Action", "Controller", FormMethod.Post, new { @class = "my_form"}) {

}

by using `@using (Html.BeginForm("Action", "Controller", FormMethod.Post))`
. you don't need to care about closing your form tag and it prevents accidental issue if somebody forgets to close the form.

```
public ActionResult Index() {
    return RedirectToAction("actionName");
    // or
    return RedirectToAction("actionName", "controllerName");
    // or
    return RedirectToAction("actionName", "controllerName", new {/* routeValues, for example:
}
```

and in `.cshtml` view:

```
@Html.ActionLink("linkText","actionName")
```

OR:

```
@Html.ActionLink("linkText","actionName","controllerName")
```

OR:

```
@Html.ActionLink("linkText", "actionName", "controllerName",
    new { /* routeValues forexample: id = 6 or leave blank or use null */ },
    new { /* htmlAttributes forexample: @class = "my-class" or leave blank or use null */ })
```

Notice using `null` in final expression is not recommended, and is better to use a blank `new {}` instead of `null`