

Lesson 5

Exercise 1

1. Create a new **lesson05** project.
2. Create a new `Exercise01Controller` controller.
3. Create a `List` of `SelectItems` inside the `Index` Action Method of the Controller:

```
List<SelectListItem> countryList = new List<SelectListItem>();
```

4. Add elements with county names and codes, like DK for Denmark, and UK for United Kingdom. The `Text` property of the item is the name of the country and the value is country code:

```
countryList.Add(new SelectListItem { Text = "China", Value = "CN"});
```

In the UI the name of the country is shown in the dropdown list between the option tags, while the country code is in the value property of the opening option tag (hidden to the user).

Tip:

The country codes are defined in ISO 3166-1 and you'll find a list here:

http://userpage.chemie.fu-berlin.de/diverse/doc/ISO_3166.html

5. Add 3-5 countries of your own and assign the new `List` of countries to a `ViewBag` property:

```
ViewBag.Countries = countryList;
```

6. Create a View for the controller and display the countries in an html dropdown list.

You can create it by iterating over the collection write your own HTML:

```
<select id="countries" name="countries">
@foreach (SelectListItem li in ViewBag.Countries) {
    option value="@li.Value"> @li.Text</option>
}
</select>
```

or you can much easier and more convenient use the `Html.DropDownList` helper method to create the list for you:

```
@Html.DropDownList("Countries")
```

7. To send form data, all form elements must be inside the form element. The `Html.BeginForm` helper method is excellent for that:

```
@using (Html.BeginForm("Index", "Exercise01", FormMethod.Get)) {
    ...
}
```

The `Html.BeginForm` method has several overloading methods. In this example, it is the version with three parameters that is used. `Index` is the action method and `Exercise01` is the controller called when the form is submitted. The third parameter tells that form data are sent with the method GET. In this exercise, we are not changing the state of the application and therefore we submit the data using an HTTP GET, not HTTP POST.

Tip:

See the W3C section [Quick Checklist for Choosing HTTP GET or POST](#) for how to choose between POST and GET.

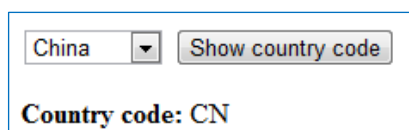
Because we are not changing the state of the application, we use the `Html.BeginForm` overload that allows us to specify action method, controller and method (**HTTP POST** or **HTTP GET**). Often views contain the `Html.BeginForm` overload that takes no parameters. The no parameter version defaults to posting the form data to the POST version of the same action method and controller.

See Pro ASP.NET MVC 5, pp. 604-05 for further information about the `BeginForm` helper method.

The dropdown list must be similar to this:



8. Add a submit button: "Show country code", and add functionality that displays the country code when the user clicks the button:



To read the country code from the URL parameter you must create and add a string parameter to the action method with the same name as the form field you want to access. In this case it's the name of the dropdown select list, and that name is `countries`:

```
public ActionResult Index(string countries)
{
    ..
    ViewBag.CountryCode = countries;
    return View();
}
```

If you save the URL parameter value representing the country code as a `ViewBag` properly you can display that in the view.

Remember to check if it is null before you display it.

```
@if (ViewBag.CountryCode != null) {  
    <strong>Country code: </strong>@ViewBag.CountryCode  
}
```

If you don't do that, you'll get a runtime error when the page is initially loaded, because the `ViewBag CountryCode` property doesn't exist.

Add Country

1. Add one more form to the view. Inside the new form there must be two text boxes and a button named "Add":



When you fill out the two text boxes with information about a new country and click the "Add" button, the new country must appear in the dropdown list as a selected element (see above).

The content of the new form must be sent by the method POST, because you are now *adding data* and thereby *changing the state* on the server.

To handle the input from the new form elements, you must create a new Action method in the controller with the same name as the Action methods already used. This new Action method must be called on submit of the form. You can use the `[HttpPost]` attribute for that:

```
[HttpPost]  
public ActionResult Index(FormCollection fc) { ... }
```

By decorating the Action method `Index` with the `[HttpPost]` attribute, you tell the controller to use this version of `Index` whenever content is sent to the server by the HTTP method POST. When you use `FormCollection` as datatype of the action method parameter, you can then reference each element of the `FormCollection` inside the code block by using this syntax:

```
fc["fieldName"]
```

2. Write the code that adds *country* and *code* to the dropdownlist.

Keep state: Store the newly added countries as a session variable

Right now, you can only add a single country. Whenever you add one more than one country, the former added country will be lost. That's because the HTTP protocol is stateless and don't "remember" actions taken between requests.

You can change that by using Sessions. Session management is built into the ASP.NET framework and is very similar to PHP sessions.

1. To store the list of countries in a session you must reference the same list in both Action methods. Therefore, you should declare the list of countries as a private field in the Controller class which makes it accessible from both Action methods:

```
public class [ControllerName]Controller : Controller
{
    private List<SelectListItem> countryList = new List<SelectListItem>();
    public ActionResult Index(string Countries) { ... }

    [HttpPost]
    public ActionResult Index(FormCollection fc) { ... }
```

2. Having that list as a field member of the Controller class, you can refer to it and use it in both versions of the Action method. In the first Action method you'll declare it as a Session variable:

```
// if the session variable isn't set you add elements to the existing countryList
// and store it in a session
if (Session["countryList"] == null) {
    countryList.Add(new SelectListItem { Text = "China", Value = "CN" });
    countryList.Add(new SelectListItem { Text = "Denmark", Value = "DK" });
    countryList.Add(new SelectListItem { Text = "France", Value = "FR" });
    countryList.Add(new SelectListItem { Text = "USA", Value = "US" });

    Session["countryList"] = countryList;
}
else {
    // if the sessionvariable is set, you'll make a reference to it
    // note that session variable is not strongly typed
    // that's because all types of variables can be stored in sessions
    // therefore you must cast it to the right type when you assign it to a
    variable

    countryList = (List<SelectListItem>) Session["countryList"];
}
```

Because the session variable is always accessible when the page is initially loaded, you can refer to it from the Action method that is decorated with the [HttpPost] attribute:

```
[HttpPost]
public ActionResult Index(FormCollection fc) {
    countryList = (List<SelectListItem>)Session["countryList"];
    // add the new country to the list
    ...

}
```

With that knowledge, it should be relatively easy for you to add session handling to the code and store information about the newly added countries as long as the session is active (i.e. as long as the browser window is open):

Countries



The screenshot shows a web form with a dropdown menu. The dropdown is open, showing a list of countries: China, Denmark, France, USA, Romania, Sweeden, Norway, and Gernamy. The word 'Gernamy' is also displayed above the dropdown, indicating it is the selected item. Below the dropdown is a button labeled 'Add'.

Sort the list (optional)

When you add an element, it is added as the last element to the list. Wouldn't it be great to have the list updated as a sorted list whenever you add new country element to it?

You can do that by creating a helper method that handles the sorting for you. Follow these steps to do so:

1. Create a new folder in your ASP.NET MVC project and name it **Infrastructure**.
2. Create a new `Utilities` class to the folder.
3. Add a static method named `SortSelectList` to that class:

```
public static void SortSelectList(List<SelectListItem> selectList) {
    ArrayList textList = new ArrayList();
    ArrayList valueList = new ArrayList();

    foreach (SelectListItem li in selectList) {
        textList.Add(li.Text);
    }
    textList.Sort();

    foreach (object item in textList) {
        SelectListItem li = selectList.Find(x => x.Text.Contains(item.ToString()));
        valueList.Add(li.Value);
    }
    selectList.Clear();

    for (int i = 0; i < textList.Count; i++) {
        if (valueList[i].ToString() == selectedCode.ToString()) {
            selectList.Add(new SelectListItem { Text = textList[i].ToString(),
            Value = valueList[i].ToString(), Selected = true });
        }
        else {
            selectList.Add(new SelectListItem { Text = textList[i].ToString(),
            Value = valueList[i].ToString() });
        }
    }
}
```

If you don't understand every detail of how the `SortSelectList` works, it's okay. What's important is that you know how to use it.

4. Call the `SortSelectList` method from the Action method in order to sort new elements into the list before the dropdown list is generated inside the view as html. The method is declared as `static`. That is important to notice, because it means you must call the method directly through the class name without instantiating the class.
5. In this new version, the newly added element is not selected. Change the code so newly added elements are displayed as selected.

Tip: You can do that by giving the method an extra string parameter with information about the country code.

Exercise 2, mandatory

In this exercise you'll work with the `MbmStore` project. We want to enhance the invoice page by filtering invoices for a given customer:

Invoices		
<div>Tina Petterson ▾ Show invoices</div>		
Customer	Product	Price
Tina Petterson	Forrest Gump (Movie)	154,50
	With a Little Help from My Friends: The Making of Sgt. Pepper (Book)	180,00
	Total	334,50
© 2016 - My ASP.NET Application		

To accomplish that we need a `SelectListItemList` with customer information to populate the dropdownlist and when the user clicks the **Show invoices** button, invoices for that customer must be displayed.

1. Open the `MbmStore` project
2. Open the Invoice controller and add this code that generates the list from the list of invoices instantiated in the `Repository` class:

```
// declare the list
List<SelectListItem> customers = new List<SelectListItem>();

// generate the dropdown list
foreach (Invoice invoice in repository.Invoices)
{
    customers.Add(new SelectListItem { Text = invoice.Customer.Firstname + " " +
        invoice.Customer.Lastname, Value = invoice.Customer.PersonId.ToString() });
}
```

3. As a customer can have more than one order, we need to remove duplicates. You can do that by using a LINQ expression. It groups the list by value (`Customer.PersonId`) and then takes the first element in each group, and sort by `Text` (`Customer.Firstname + " " + invoice.Customer.Lastname`).

The `GroupBy` function returns an `IEnumerable` and we typecast that back to a `List` of `SelectListItem`:

```
// removes duplicate entries with same ID from a IEnumerable
customers = customers.GroupBy(x => x.Value).Select(y => y.First()).OrderBy(z =>
z.Text).ToList<SelectListItem>();
```

4. To send the list to the view assign a `ViewBag` property `CustomerId` to the list of customers.

```
ViewBag.CustomerId = customers;
```

5. Add form to the view that shows the dropdownlist of customers:

```
@using (Html.BeginForm()) {
    @Html.DropDownList("CustomerId", "Select customer")
    <button type="submit">Show invoices</button>
}
```

6. With that in place we need an overload method of `Index` that reads the value of the selected `CustomerId` and filter that list of invoiced by that id:

```
[HttpPost]
public ActionResult Index(int? CustomerId)
{
    ...
}
```

Most of the code in this action method is the same. It also declares a `SelectListItem List` and populate that with customers data from the `Invoice List` that is generated in the `Repository` class. However, instead of having two declarations of the `SelectListItem List`, you could declare the list as a private field inside the class.

The extra code you need is the filtering on invoices based on the selected `CustomerId`. You'll use LINQ for that:

```
if (CustomerId != null ) {
    // select invoices for a customer with linq
    invoices = repository.Invoices.Where(r => r.Customer.PersonId == CustomerId);
}
```

7. Send the list of invoices to the view as a `ViewBag` property and test the filtering function and make sure it works as expected.

Exercise 3

Close the **MbmStore** project and reopen the **lesson05** project. Create a new `Exercise03Controller` controller with a corresponding view based on the following HTML-code:

```
<!DOCTYPE html>
<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>Breakfast order</title>
</head>

<body>
<h1>Breakfast order</h1>
<form action="POST">
<p>Your name: <input type="text" name="fullname" /></p>
<p>Room number: <input type="number" name="roomnumber" /></p>

<p>What would you like to eat for breakfast?<br /><br />
  <input type="checkbox" name="menuitem" value="cornflakes" /> Cornflakes<br/>
  <input type="checkbox" name="menuitem" value="egg" /> Egg<br/>
  <input type="checkbox" name="menuitem" value="toast" /> Toast<br/>
  <input type="checkbox" name="menuitem" value="juice" /> Juice<br/>
  <input type="checkbox" name="menuitem" value="milk" /> Milk<br/>
  <input type="checkbox" name="menuitem" value="coffee" /> Coffee<br/>
  <input type="checkbox" name="menuitem" value="tea" /> Tea<br/>
</p>

<p>When: <input type="datetime" name="time" /> </p>

<p>
  <input type="submit" value="Order" />
</p>
</form>
</body>
</html>
```

Breakfast order

Your name:

Room number:

What would you like to eat for breakfast?

- ☐ Cornflakes
- ☒ Egg
- ☒ Toast
- ☒ Juice
- ☐ Milk
- ☐ Coffee
- ☒ Tea

When:

Rewrite the html code and use appropriate form helper methods to generate the view.

Tip: As default, the form helper methods will insert validation information as part of the html markup. You can see that by inspection the source code inside the browser. We will not use validation in this exercise, and to get rid of that extra code you can disable it altogether in the view file by using the `Html` helper `EnableClientValidation` method:

```
@{
    ViewBag.Title = "Parking Ticket";
    Layout = null;
    Html.EnableClientValidation(false);
}
```

The data for the check box list must come from a strongly typed `SelectListItem` collection that implements the `IEnumerable` interface. `List` does that:

```
List<SelectListItem> breakfastItems = new List<SelectListItem>();
```

As in the former exercise, you can use this syntax for creating new elements to be added to the list:

```
new SelectListItem { Text = "Cornflakes", Value = "Cornflakes" }
```

Receipt

After you have created the view for ordering, you must create a view that displays a receipt to confirm the order:

Breakfast Order

Good morning Susan!

You have ordered: **Egg, Toast, Juice, Tea.**

Your order will be prepared and delivered
to room 224, Friday, September 12, 2014.

Tip: You retrieve the checkbox elements as a simple comma-separated list with “false” inserted for elements that are not selected. To display the receipt you must convert the string to an array, loop through that array and extract all elements with values different from “false”.

For this exercise I suggest that you create the string of breakfast items inside the controller and pass it to the view as a `ViewBag` property.

Give price as part of the receipt (optional)

As the last step in this exercise, you can add improvements to the page by enhancing the receipt with the price of each ordered item, and total for the whole order:

Breakfast Order

Good morning Susan!

You have ordered: Egg (15.75) , Toast (12.50) , Juice (18.00) , Tea (12.50) .
Total Price: **58.75**

You order will be prepared and delivered
to room 224, Friday, September 12, 2014.

Tip: You can use a dictionary collection with keys and values for storing the price list. The keys must be the name of the breakfast items of type string and the values must be the price of type decimal.

Exercise 4, optional

In this exercise we'll work with *Templated Helper Methods* that generates visual elements for editing and displaying data based on a strongly typed model. If you have a person object for example, an editable field for a birthday property will be displayed as an input type of datetime, whereas firstname will be displayed as a text input type.

We'll create a webpage that simulates a *Parking Ticket Machine*. The UI should be like this:

Parking Ticket Machine

Time now	10:40
Paid until	12:10
Info display	30 kr is paid

Coin Insert

Create the Model

Before you start with the controller and view you must create the model. You do that by declaring a class `ParkingTicketMachine` inside the Models folder with *fields*:

- `minutesPrKr: int`
- `coinsToInsert: int[]` (declared as: `private int[] coinsToInsert = {1, 2, 5, 10, 20};`)
- `amountInserted: int`
- `timeNow: DateTime`

and *properties*:

- `TimeNow`: `datetime`, read-only
- `PaidUntil`: `datetime`, read-only
- `Info`: `string` (automatic property)
- `AmountInserted`, `int`, readonly

A constructor in which default values for the fields `amountInserted` and `timeNow` is set to default values (0 and now).

You must also declare a method:

- `public void insertCoin(int kr) { ... }`

which adds the parameter value given to the `amountInserted` field variable. The cost is 2 kr for 6 minutes

Tip:

You can use the static property `Now` of the `DateTime` class to get the current time.

Create the user interface

You are now ready to code the user interface. Create a controller (`Exercise04Controller`) and a view. Instantiate the `ParkingTicketMachine` class inside the controller and create a strongly typed view by including the model in the view:

```
@model Lesson05.Models.ParkingTicketMachine
@{
    ViewBag.Title = "Parkin Ticket";
    Layout = null;
}
```

You can now insert a few lines of code:

```
@using(Html.BeginForm()) {
    @Html.EditorForModel()
}
```

and based on the model the helper method will generate a form similar to this:

Parking Ticket Machine

TimeNow

PaidUntil

Info

AmountInserted

Although we could do some customizing with CSS and by applying DataAttributes to the properties in ParkingTicketMachine, we will use single field template helper methods to get more fine-grained control of the layout.

Use the `HTML.LabelFor` and `HTML.EditorFor` helper methods to create the “Time Now”, and “Paid until” fields.

You must also create a field called “amountInserted” which holds the total amount inserted into the parking meter.

In order to get full control of the output given in the “Info” field, you might want to use an ordinary HTML input field of type text for that.

When you’re done, you must come up with a UI similar to this:

Parking Ticket Machine

TimeNow 7/28/2014 20:32:21
PaidUntil 7/28/2014 21:32:21

Info 20 kr inserted

AmountInserted 20

Coin Insert

The next thing you should do, is program the controller and update the fields in accordance with the amount given, when the user clicks at one of the “kr” submit buttons. Remember that the name/values of buttons are only sent to the server if someone actually clicks the button.

If you name the buttons “1”, “2”, “5”, “10”, and “20” you can program the controller to react to button clicks:

```
[HttpPost]
public ActionResult Index(FormCollection fc) {

    // create a new instance of ParkingTicketMachine
    // --- write code ---

    // declare a variable “AmountInserted” of type int which keeps track of the
    // amount inserted
    // --- write code ---

    // if the form field representing the amount is not null
    // read the value the amount inserted and assign it to AmountInserted
    // --- write code ---

    // else set the initial value of AmountInserted to 0
    // --- write code ---

    if (fc["1"] != null) {
        // call the insertCoin method with 1+AmountInserted as parameter
    }
    else if (fc["2"] != null) {
        // call the insertCoin method with 2+AmountInserted as parameter
    }
    ...

    if (fc["cancel"] != null) {
        // reset the model to its initial state
        ptm = new ParkingTicketMachine();
    }
    if (fc["confirm"] != null) {
        ptm.insertCoin(AmountInserted);
        // load the receipt view named "confirm" with the model as parameter
        return View("confirm", ptm);
    }

    // load the default view with the model as parameter
    return View(ptm);
}
```

After pressing “Confirm” a Parking Ticket must be displayed with information such as:

Parking Ticket
Time issued: 21:03
Parking paid until: 21:03

After pressing “Cancel” the Info display should show a text like “6 kr is paid back”.

Refine the user interface

We want to polish the user interface a bit:

1. Instead of “TimeNow”, “PaidUntil” etc. the labels should be “Time now”, “Paid until”, and “Display Info”
2. In the time fields we will remove the date part and only display the time part as HH:MM (hours:minutes)
3. The “AmountInserted” label and fields are not relevant for end users, and therefore we want to hide it.
4. The input fields for TimeNow, PaidUntil and Info shouldn’t actually be editable. Therefore, you might prefer to use the `HTML.DisplayFor` helper instead of the `HTML.EditorFor` helper inside the view.

To create these minor but important improvements you can use model metadata to instruct the MVC framework how to display model attributes. You do this by adding metadata attributes above the properties in the class file.

Tip: See *ASP.NET MVC 5*, chapter 22 for further instructions of how to use metadata attributes.

Make the changes and create a user interface similar to this:

Parking Ticket Machine

Time now	10:37
Paid until	10:46
Info display	3 kr is paid

Coin Insert

1 kr	2 kr	5 kr	10 kr	20 kr
------	------	------	-------	-------

Confirm	Cancel
---------	--------

/ Jes Arbov, 29-09 2016.