

Lesson 13 – Exercises.

Exercise 1

Create a new `exercise01` controller.

Create a list of `SelectItems`:

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

and add elements with country 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:

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

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

Assign the new List of countries to a `ViewBag` property:

```
ViewBag.Countries = countries;
```

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

The easy way to create the dropdown list is to use the `Html.DropDownList` helper method to create the list for you:

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

Or you can create it by iterating over the collection and write your own HTML:

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

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)) {
    ...
}
```

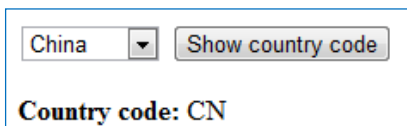
The `Html.BeginForm` method has several overloading methods. In this example, it is the version with three parameters that is used. `Index` is the action method and `Exercise01` the controller called when

the form is submitted. The third parameter tells that form data are send with the method post. In this exercise we are not changing the state of the application, so we will submit the data using an HTTP GET, not HTTP POST. 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.

The dropdown list must be similar to this:



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



To read the country code from the URL parameter you must add a string parameter to the action method:

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

You might have to change the routeConfig file also

If you save the URL parameter value representing the country code as a ViewBag property 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 `CountryCode` property doesn't exists.

Add Country

Add one more form to the view. There must be two text boxes and a button named “Add”:

A screenshot of a web form. At the top, there is a dropdown menu showing 'Romania' and a button labeled 'Show country code'. Below this, the text 'Country code: CN' is displayed. Further down, there are two input fields: 'Country:' with 'Romania' entered, and 'Code:' with 'ro' entered. At the bottom of the form is an 'Add' button.

When you fill out the text boxes 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 formData) { ... }
```

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. You can reference `FormCollection` elements by this syntax:

```
formData["fieldname"]
```

Write the code that adds the two field input elements to the dropdownlist (i.e. the name of the country and the country code).

Right now, you can only add a single country. Whenever you add one more 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.

To store the list of countries in a session you must reference the same list in both Action methods. So, you should declare the list of countries as a private field accessible from both methods:

```
public class [ControllerName]Controller : Controller
{
```

```

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

public ActionResult Index(string Countries) { ... }
}

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

```

Having that list as a class field member, you can make references to it and use it in both Action methods. In the first Action method you'll declare it as a Session variable:

```

// if the session variable isn't set you create the list 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 typecast it to the right type when you assign it to a variable
    countryList = (List<SelectListItem>)Session["countryList"];
}

```

Because the session variable is always set 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 formData) {
    countryList = (List<SelectListItem>)Session["countryList"];
    ...
}

```

With that knowledge it should be easy you to add the necessary session handling code that stores information about added countries as long as the session is active:

Countries



Germany ▼

- China
- Denmark
- France
- USA
- Romania
- Sweeden
- Norway
- Germany

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 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.

```
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 all details of how the `SortSelectList` works, it's alright. What's important is that you know how to use it.

3. 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`. What does that mean? Which consequences does it have when you call the method?
4. 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

In this exercise we'll work with *Templated Helper Methods* which 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

1 kr

2 kr

5 kr

10 kr

20 kr

Confirm

Cancel

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 Model folder with fields:

- `minutesPrKr: int`
- `coinsToInsert: int[]`
- `amountInserted: int`
- `timeNow: DateTime`

and properties:

- CoinsToInsert: List<int>, read-only
- TimeNow: datetime, read-only
- PaidUntil: datetime, read-only
- 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 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

1 kr 2 kr 5 kr 10 kr 20 kr

Confirm Cancel

The next thing you should do, is program the controller and update the fields in accordance to the amount given, when the user click the “kr” submit buttons. Remember that the name/values of buttons are only send to the server if someone actually clicked 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 formData) {

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

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

    // if the form field representing the amount is not null
    // --- write code ---

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

    if (formData["cancel"] != null) {
        // reset the model to its initial state
        ptm = new ParkingTicketMachine();
    }
    if (formData["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 imported changes 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
---------	--------

Exercise 3

Brug HTML helpers i din gennemgående opgave

Exercise 4 (optional)

You must create a new 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:

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:

```
@{
    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>();
```

You can use this syntax for adding elements to the list:

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

Receipt

When 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.**

You 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 with `ViewBag`.

Give price as part of the receipt (optional)

As the last step in this exercise, you must add improvements to your page: Write a 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.