JOHN ABBOTT
CEGEP/COLLEGE

(420-PS4-AB)
Building Forms – ASP .NET MVC

Summer 2018

Outline

- Forms – Introduction
- Helper Methods
- Form Markup Tags
- Labels
- Check Boxes
- Drop Down Lists
- Submitting Forms
- Edit Data
- Formatting Output in Forms
- Updating Data
- Exercise Time
- Summery

Forms – Introduction

- A form in HTML is like a container that we can add various elements in it.

- Forms use two actions to operate:
  1. First action is called through the URL to return a view that contains the form (with fields.. etc.)
  2. Second action is called by the form upon *submit* to save the form data in the database.

- To add a form in a view:
  - Use regular HTML markup
  - Or, use a helper methods from C#

Form Helper Methods

- Html.BeginForm
  - Requires two parameters: target action and controller name
  - Requires keyword “using” and a code block
    → so it would place required tags *<form> </form>*

- Html.LabelFor
  - Parameter: The field from used model → use to display the label text.
  - Uses lambda expression, (example c => c.Name)

- Html.TextBoxFor
  - Same as previous.
  - To add a look and feel, add an anonymous object parameter

  new{ @class = “form-control”}

Form Markup Tags

- There are special markup tags needed to render modern and responsive forms.
- These markup tags are used by bootstrap.
- Each input in a form needs to be rapped in div tag.
  - Add a class with the value “form-group”
  - Add a **label** and an **input** fields within each div.
    - Can be added using HTML or using helper methods.

Demo: Add New Customer

- In the Customer controller
  - Create an action named “New”.
  - Add a view corresponding to the action. (name the view “CustomerForm” as it will be used to editing later)
  - Add a form using the helper methods.
  - Add Name & address labels and corresponding textboxes.
  - Add a check box for IsSubscribedToNewLetter
    - Check the layout, and think how to fix it.
    - Check getbootstrap.com/css to fix it.

Notes on Labels

- The form will pick the label values from the corresponding property in our class.

- Sometimes this is not presentable.

(example: property customerName → want to display “Name”

- Two Solutions:
  - Add a data annotation: [Display(Name =“New Display Name”)]
  - User raw HTML label tag.

  What is the difference?? **Focus & Magic string use**

Drop Down Lists

- Need to check if the values of the drop down list are prefixed or loaded from database.
- Loading from database is better and easier to maintain.
- The action returning the view with a drop down list needs to pass the list values to view.

Demo: Drop Down Lists

- Add drop down list to select membership type for the customer.
  - Get the membership type from the database.
  - **Issue:** we need two objects
    - Customer: for adding (and updating later)
    - MembershipTypes
    - Solution? Add a ModelView
  - Add a form group (div tag) and add it in.
  - Use help method HtmlDropDownListFor

```
@Html.DropDownListFor(m => m.Customer.MembershipTypeId,
 new SelectList(Model.MembershipTypes, "Id", "Name"), "Select
Membership Type", new { @class = "form-control" })
```

  - Change the display name using data annotations.

Submitting A Form

- We need a button with type submit. (HTML)
- The button will call the action stated in the helper method Html.BeginForm.

- Actions handling submit requests require:
  - [HttpPost] An attribute with HTTP post makes sure it can only be called using post request, not a get request.
  - Passed object parameter: MVC framework will automatically map values from the form to this object, know as Model Binding.

Demo: Submit

- Add a button at the end of the form with “Save” label.

- Add “Save” action.
  - Add attribute for http post.
  - Pass required parameter to it.
  - Lets inspect.

  - Add customer to context and save.
- Redirect back to the list of customers.

Editing (with Demo)

- In my Customer page, once we click on a customer, it directs us to the details view.
  - Change the link under each customer to go to an “Edit” action.
  - In the edit, retrieve the customer from the context.
  - Reuse the same form we created for adding to edit.
    - Create an object of the ViewModel with the desired values.
    - Pass it to the customer form view.

Updating Records

- First get requested entity record from the database.
- DbContext can track changes in that entity.

- Modify required properties using by:
  - TryUpdateModel(ObjectFromContext):
    - The framework will compare the key value pairs in the entity and request data.
    - Can be overloaded to provide only attributes that need to be updated.
    - Has some security holes.
  - Or Manually map each attribute
    - Or use a mapper.

Demo: Updating Records

- The action Save is used to submit from the customer form
  - As we are using it for both adding and updating
  - To identify if the action is adding or updating
    - Check the id value (0 value indicated new customer)
  - Add logic to accommodate the update request.
  - Customer Id is required to be added to the form so we can update.
    - We don't want to display it though.
    - Use hidden field.

Exercise Time

Exercise:

- In the list of Medias add a link at the top of the table to create a new media.
  - Can you make it look like a button using bootstrap classes?
- On click, we get a form to add new media with the following form fields:
  - Name – Textbox
  - Release Date – Textbox
  - Media Type – Drop down list
  - Genre – Drop down list
  - Number in Stock – Textbox
  - Remember that all the fields are required.
- Modify the link under each media to see the media details populated in the media form.
  - Modify the heading of the page to show that this is an edit action.
  - Implement update to update in the database.

Notes on Formatting Output in Forms

- Fix Date and Time Format
- Helper method “Html.TextBoxFor” can accept a format string to display the day in the way that we like.
- Example: to adjust the date field so it would not show the time, use:

```
Html.TextBoxFor(m => m.Customer.Birthdate,
"{0:d MMM yyyy}", new { @class = "form-control" })
```

Building Forms Summery

View

```
@using (Html.BeginForm("action", "controller"))
{
    <div class="form-group">
        @Html.LabelFor(m => m.Name)
        @Html.TextBoxFor(m => m.Name, new {
        @class = "form- control")
    </div>
    <button type="submit" class="btn btn-primary">
        Save
    </button>
}
```

Markup for Checkbox Fields

```
<div class="checkbox">
    @Html.CheckBoxFor(m => m.IsSubscribed)
    Subscribed?
</div>
```

Drop-down Lists

```
@Html.DropDownListFor(m => m.TypeId, new
SelectList(Model.Types,"Id","Name"),
"Select an element", new { @class = "form-
control"}
```

Overriding Labels

```
Display(Name = "Date of Birth")
public DateTime? Birthdate { get; set; }
```

Saving Data

```
[HttpPost]
public ActionResult Save(Customer customer)
{
        if (customer.Id == 0)
                _context.Customers.Add(customer);
        else
        {
                var customerInDb = _context.Customers.Single(c.Id ==
                customer.Id);
                //… update properties
        }
        _context.SaveChanges();
        return RedirectToAction("Index", "Customers")
}
```

Hidden Fields

- Required when updating data.

```
@Html.HiddenFor(m => m.Customer.Id)
```