WEB524

WEB PROGRAMMING ON WINDOWS

WEEK 5 – LECTURE 2

ADD NEW ASSOCIATED DATA
AND ATTRIBUTE ROUTING

Today

- We will look at the “add new” use case
- Learn how to design and code your web app to implement the two use cases.

Resources

- Review the **AssocAddEdit** code example posted on Blackboard.
- Review the **Class Diagram** of the view model classes contained in the code example posted on Blackboard.

Checkpoint

- Recently, you learned about some design and coding features that will enable you to work with associated data:

- Using a dedicated "…Form" view model class to package data for an HTML Form
- If the HTML Form needs an item-selection element, we use a **SelectList** object to package the items

“Add new” for an object with a required association

- It is very common for your object to be dependant on another object.
- You must have a reference to the associated object.

- In the code example:
  - A Vehicle MUST be associated with a Manufacturer (in other words, a Vehicle is dependent on a Manufacturer).
  - When adding a new Vehicle, we must know or have, information about the associated Manufacturer.

- In the retail business:
  - a Product must be associated with a Supplier.
  - When adding a new Product, we must know or have, information about the associated Supplier.

- At the College:
  - An Academic Program MUST be associated with a School (Faculty).
  - When adding a new Academic Program, we must know or have, information about the associated School (Faculty).

“Add New” View Model Class

- When you have a “to-one” association, add the property to the “add new” view model class.
- The property type is int (or the data type of the unique identifier).
- The property name is <entity>Id (for example, ManufacturerId).

- For example, the VehicleAddViewModel class will have this property:

```
[Range(1, Int32.MaxValue)]
public int ManufacturerId { get; set; }
```

- In another example, Manufacturer has a “to-one” required association with Country. A ManufacturerAddViewModel class will have this property:

```
[Range(1, Int32.MaxValue)]
public int CountryId { get; set; }
```

- Setting up your view model this way will help you avoid design and coding errors.

Starting with an empty context

- During the flow of the application, it is possible that the app will allow the user to start with an empty context and begin the process of adding a Vehicle.
- In this situation, the user must choose the Manufacturer from an item-selection element.
- *See the "Create New" feature on the list-of-vehicles page.*

If we already have information…

- It is also possible that the context of the flow knows or has information about the associated object.
- In the code example, when working with the Manufacturer object, we know and have information about that Manufacturer.
- Therefore, when adding a new Vehicle for that Manufacturer we do not have to ask the user for Manufacturer info.

- *See the "Add vehicle for this manufacturer" feature on the manufacturer details page.*

How the nature of the association affects your interaction design

- When designing for the “add new” use case, the nature of the association affects the interaction design.
- Which of the following is better and more correct?
    1. Begin adding a new Vehicle. During this process, choose the Manufacturer for that Vehicle.
    2. Choose a specific Manufacturer then add a new Vehicle for that Manufacturer.
- There is no single right answer, some designs will work better in different situations.

Handle “add new”, if you must choose the associated object

1. Write a **VehicleAddViewModel** class. Include an **int** property for the identifier of the associated **Manufacturer**.
2. Write a **VehicleAddFormViewModel** class. Include **SelectList** and descriptive properties for the associated **Manufacturer**.
3. In the Manager class, write a **VehicleAdd()** method. Validate the associated **Manufacturer** object before attempting to add the new **Vehicle** object.
4. In the controller class, write the **Create()** method pair using the techniques you’ve already learned.

Write a “VehicleAdd” view model class…

- As you have done before, write a **VehicleAddViewModel** class.

- This time, you will include an **int** property for the identifier of the associated **Manufacturer**.

```
[Range(1, Int32.MaxValue)]
public int ManufacturerId { get; set; }
```

Write a “VehicleAddForm” view model class…

- Write a **VehicleAddFormViewModel** class as previously learned.
- You may inherit from the **VehicleAddViewModel** class.
- Include a **SelectList** property that will eventually hold the collection of **Manufacturer** items used as the source in an HTML Forms item-selection element.

```
public SelectList ManufacturerList { get; set; }
```

In the Manager class, write a “VehicleAdd” method…

- Write a **VehicleAdd()** method in the Manager class.
- The new method will locate and validate the associated **Manufacturer** object before attempting to add the new Vehicle object.
- The incoming **newItem** object will have a **ManufacturerId**. Use that value to look for the associated **Manufacturer** object.
- There is a code example next.

VehicleAdd() Example

```
// Find the manufacturer.
// If found, you will use it when adding the new Vehicle object.
var a = ds.Manufacturers.Find(newItem.ManufacturerId);

if (a == null)
   return null;

// Attempt to add the new item
var addedItem = ds.Vehicles.Add(Mapper.Map<VehicleAdd,
Vehicle>(newItem));

// Set the associated item property
addedItem.Manufacturer = a;
ds.SaveChanges();
```

In the controller class, write the “Create” method pair…

- As you have done before, write the “Create” method pair in the VehiclesController class using the same techniques as before.

- The “GET” method – which delivers the HTML Form to the browser – MUST initialize and configure a **VehicleAddFormViewModel** object before sending it to the view.

- The “POST” method – which handles the data submitted by the user – works the same as in other “add new” use cases.

Handle “add new”, if you know or have info about the associated object

- This situation is seen in the “Add vehicle for this manufacturer” feature on the manufacturer details page.

1. If needed, write a **VehicleAddViewModel** class.
2. If needed, write a **VehicleAddFormViewModel** class.
3. If needed, in the **Manager** class, write a **VehicleAdd()** method.
4. In the controller class, write the **Create()** method pair using the same techniques you used in the past.

Steps 1 - 3

- The first three steps are (almost) identical to the previous scenario.

1. If needed, write a VehicleAddViewModel class.
2. If needed, write a VehicleAddFormViewModel class.
   - Include one or more name/descriptive properties for the associated **Manufacturer**. This will enable you to display information in the view that will help the user identify the **Manufacturer**.

   ```
   public string ManufacturerName { get; set; }
   ```
3. If needed, in the Manager class, write a VehicleAdd() method.

In the ManufacturersController class, write the “Create” method pair…

- This task will be familiar but parts of it are different than you would expect.
- It is possible that you will want the **ManufacturersController** to have a method pair that will enable the “add new” use case for a **Manufacturer**. Plan to use the suggested default Create() method name for this purpose.
- We must use a different method name, in the **ManufacturersController** class to enable the “add new” use case for a **Vehicle**. It is suggested that you use **AddVehicle()** for the method name.
- The method MUST have an **int** parameter for the manufacturer identifier. The idea is that this value is known in the current context or flow of the app. Therefore, its presence enables us to handle a use case that would be stated like this:

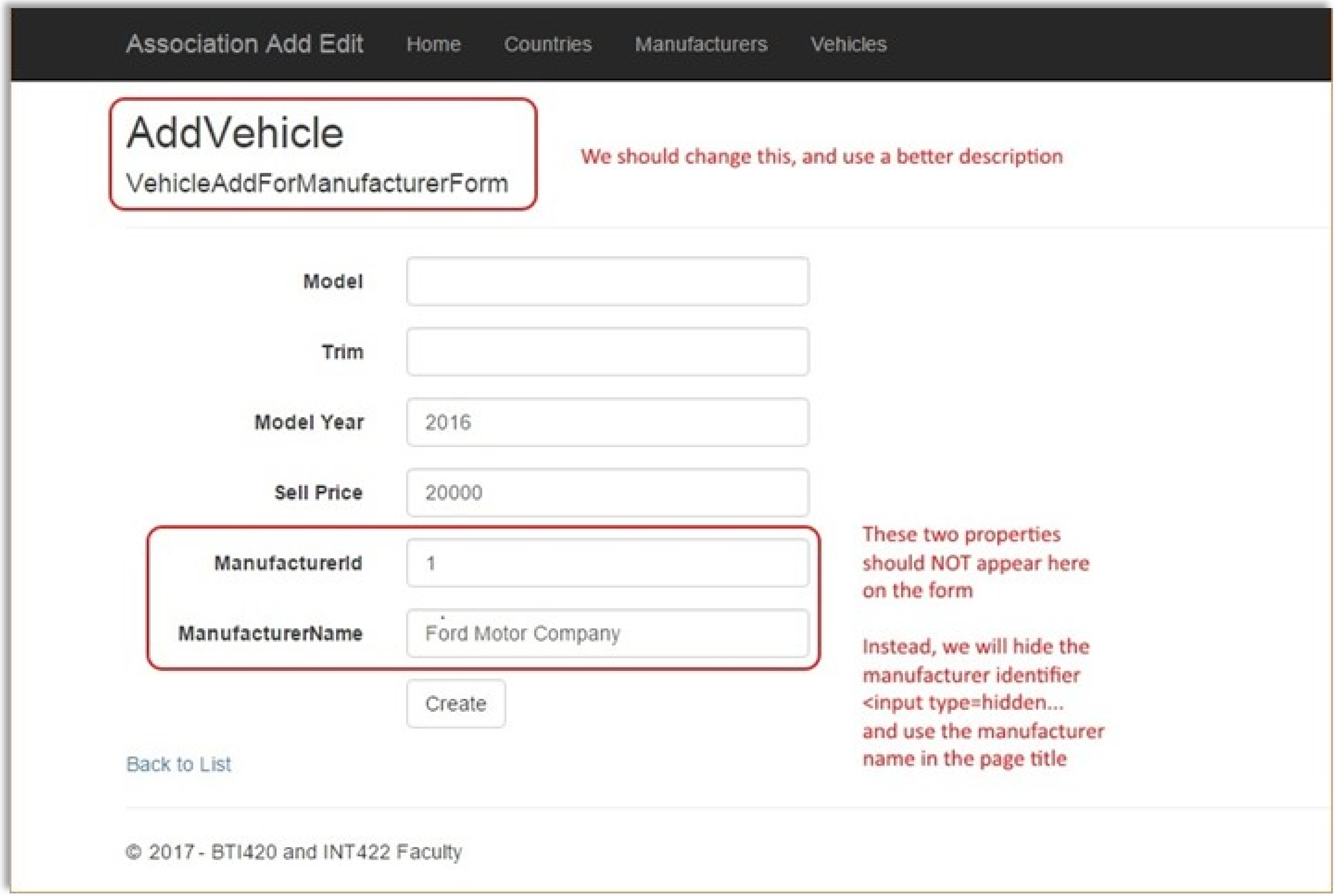
“Add a new vehicle for the manufacturer with identifier 123.”

The “GET” method

- In the “GET” method that delivers the HTML Form to the browser, we have more work to do than before.
- The new task is to locate and validate the associated **Manufacturer** object before attempting to add the new **Vehicle** object.
- After this, you can generate a view based on the **VehicleAddFormViewModel** class. It will generate a view similar to the following slide.

```
// Attempt to get the associated object, if found then we can continue.
// We MUST initialize and configure a VehicleAddFormViewModel object
// before sending it to the view.
var a = m.ManufacturerGetById(id.GetValueOrDefault());

// Create and configure a form object
var o = new VehicleAddForm();
o.ManufacturerId = a.Id;
o.ManufacturerName = a.Name;
```



Association Add Edit Home Countries Manufacturers Vehicles

AddVehicle

VehicleAddForManufacturerForm

We should change this, and use a better description

Model	
Trim	
Model Year	2016
Sell Price	20000
ManufacturerId	1
ManufacturerName	Ford Motor Company

Create

Back to List

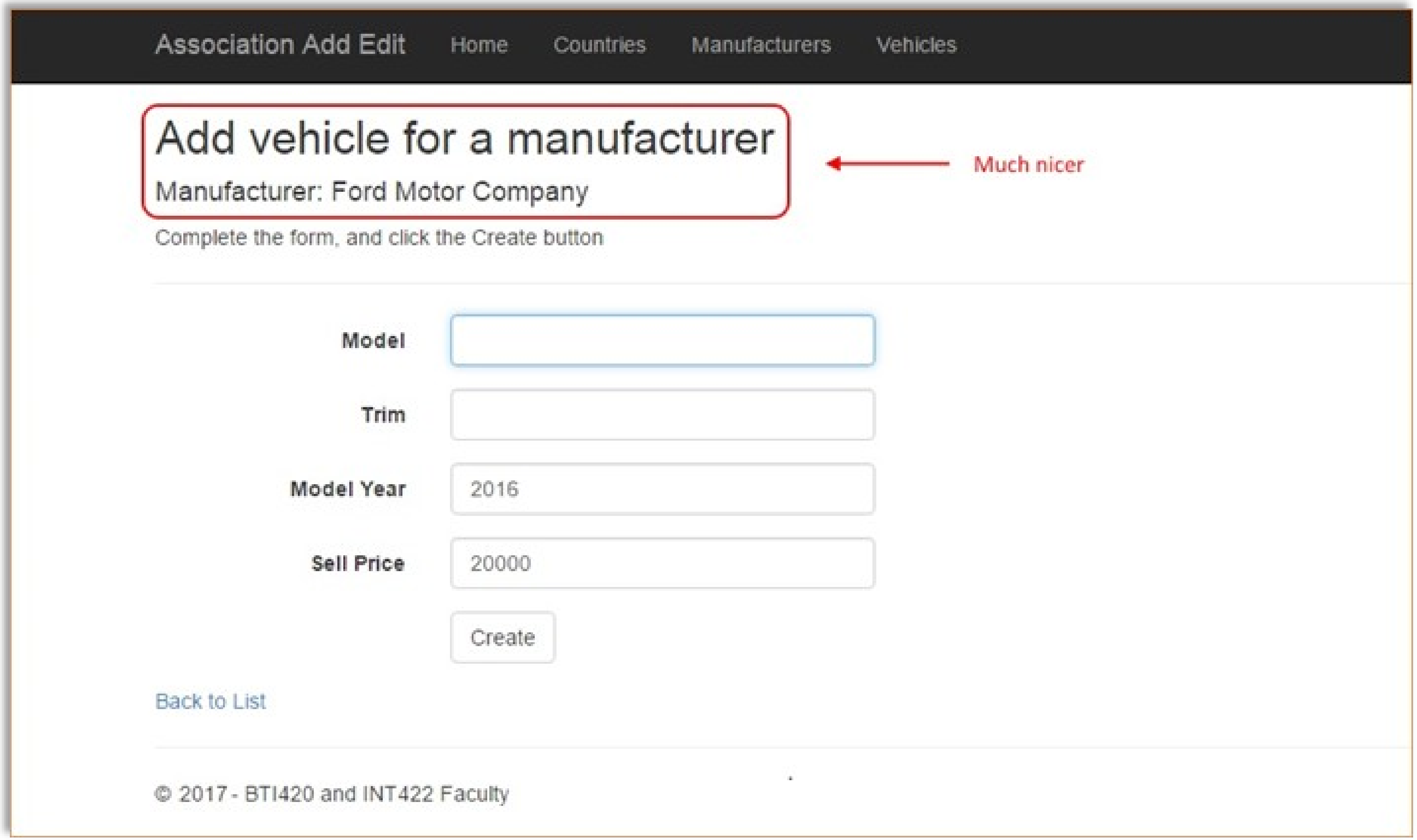
These two properties should NOT appear here on the form

Instead, we will hide the manufacturer identifier <input type=hidden... and use the manufacturer name in the page title

© 2017 - BTI420 and INT422 Faculty

Problems

- The generated view has a few problems mostly because we already know or have some information about the manufacturer.
- We must NOT allow the user to edit the data.
- As stated in the image (previous slide), the two manufacturer-related properties should not appear in editable elements on the HTML Form.
- Instead, we will edit the form:
    1. Convert the **ManufacturerId** to an <input type=hidden… element.
    2. Use the value of **ManufacturerName** in the page's visible title.
- For example, you could do something like the following slide.



Add vehicle for a manufacturer

Manufacturer: Ford Motor Company

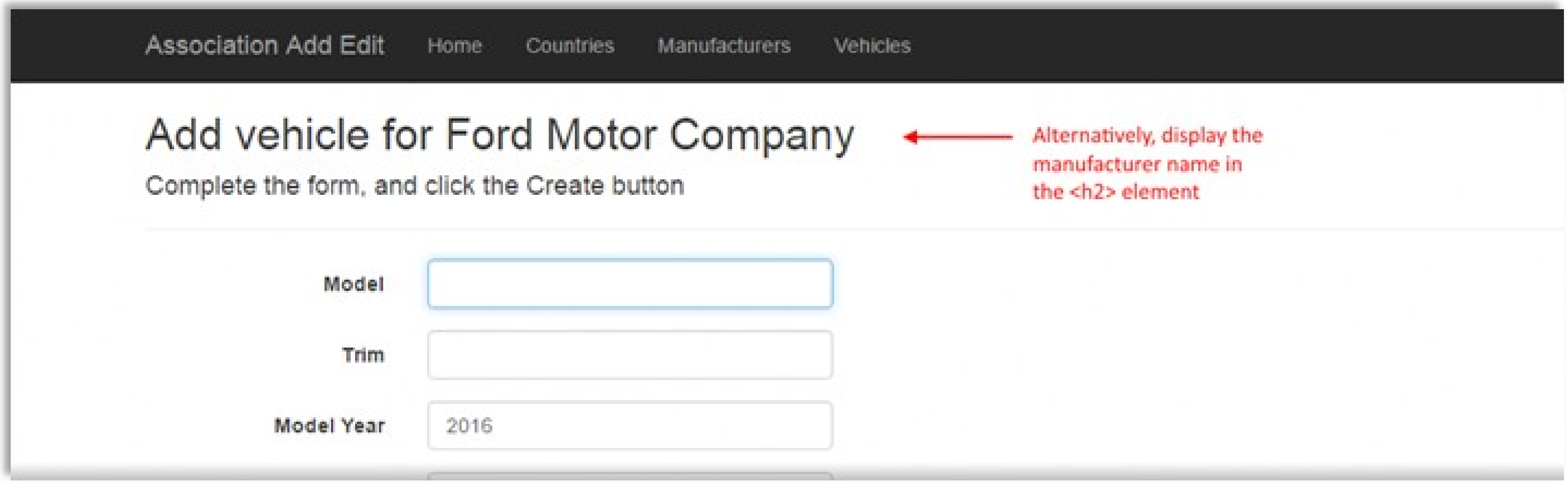
Much nicer

Complete the form, and click the Create button

Model	
Trim	
Model Year	2016
Sell Price	20000

© 2017 - BTI420 and INT422 Faculty

You could make the manufacturer info even more prominent


Association Add Edit   Home   Countries   Manufacturers   Vehicles

Add vehicle for Ford Motor Company

Complete the form, and click the Create button

Alternatively, display the manufacturer name in the <h2> element

Model	
Trim	
Model Year	2016

The “POST” method

- The “POST” method handles the data submitted by the user.
- It works almost the same as in other “add new” use cases.
- The difference is that the PRG pattern destination will be the **Details** view of the **VehiclesController**. Since it exists, we can reuse the method.

Attribute Routing

Introduction to Attribute Routing

- In the last example, the URL to the "add new" task in the **ManufacturersController** was:

  https://host.example.com/manufacturers/addvehicle/56

- In the URL above, the order or sequence of the segments suggests a different task or operation. In other words, someone could argue that we are attempting add a vehicle with identifier "56" to the manufacturers collection.
- Another way to express this is: "add a vehicle to manufacturer number 56". As a resource URL, that would be:

  https://host.example.com/manufacturers/56/addvehicle

Enabling Attribute Routing

- A feature named "Attribute Routing" enables us to do what we want.
- This article by Mike Wasson (on the official ASP.NET Web API web site) does a good job to explain attribute routing.
- In summary:
    - It is an easy way to define and use custom resource URIs that go beyond the default format.
    - It can be used alongside convention-based routing.
- When you use Attribute Routing, you must enable that feature in the app.
    - In the RouteConfig class (in the App_Start folder), add this to the RegisterRoutes method: routes.MapMvcAttributeRoutes();

Creating a route using Attributes

- With Attribute Routing, we can add an attribute to the “action” method and specify the route that it will serve.
- We can add this attribute to both **AddVehicle()** methods in the **ManufacturersController**:

```
[Route("manufacturers/{id}/addvehicle")]
public ActionResult AddVehicle(int? id)
```