Customer "Edit Existing" Functionality

We already have some view models, namely: **CustomerBaseViewModel** and **CustomerAddViewModel**.

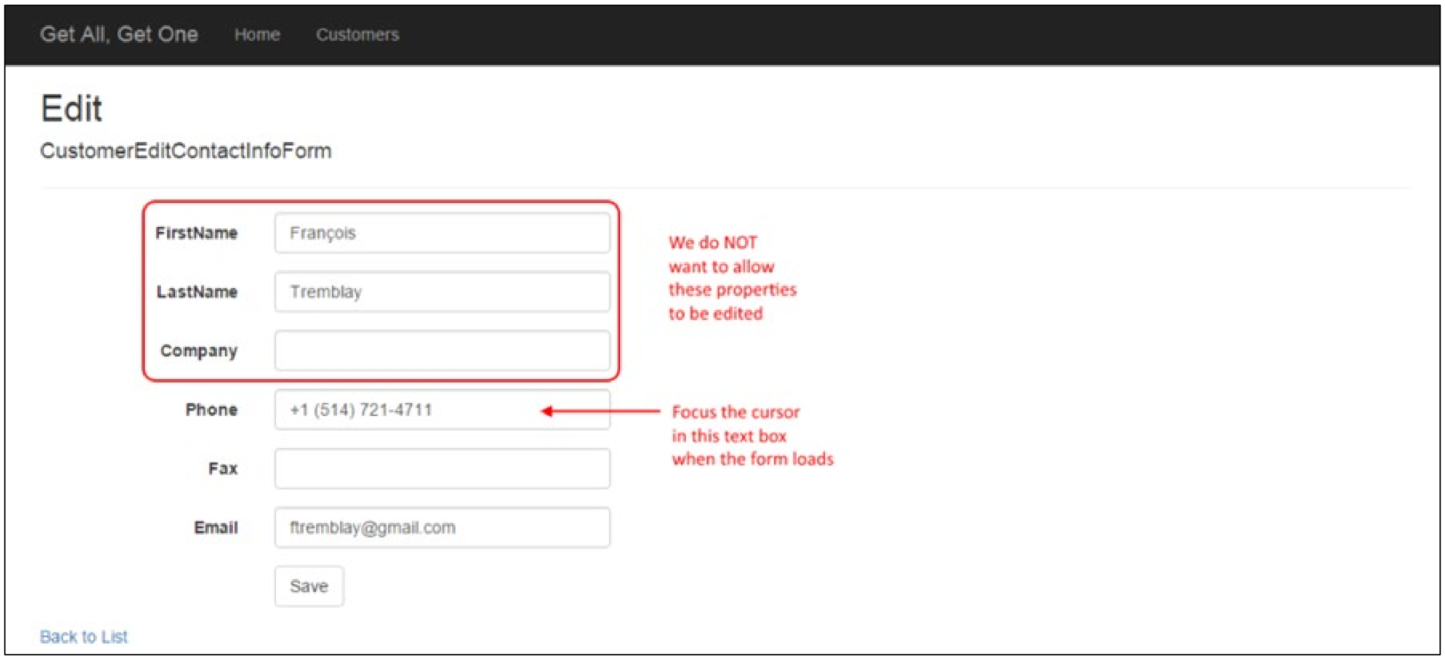
We cannot use these models for the "edit" functionality because they do not have an identifier property.

CustomerBaseViewModel could work for very simple objects but it probably does not make sense to use it here. We usually do not permit someone to edit all the properties of an existing object in a single task. Instead, we focus on the editing (some properties) task that makes sense for the use case. As such, we need to create another view model class to exactly match the shape of the editable object.

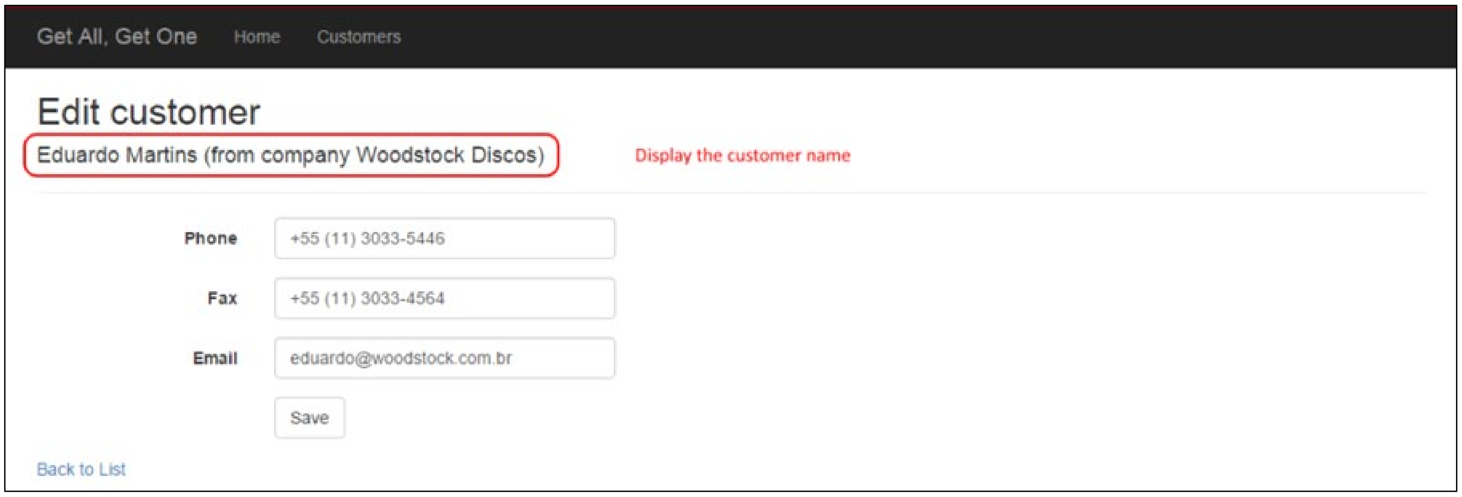
In the code example, we have decided that we will permit the editing of contact info – *phone, fax, and email* – for a specific customer. The new view model class will need these properties plus the identifier property.

Unfortunately, this is not enough! If we show an HTML Form with only the phone, fax, and email properties then the user will not have enough information to know which customer is being edited.

In the view model class, we must also include human-readable and contextually logical information. For a customer, it makes sense to include the customer's name however, we do not want to allow editing of the customer's name.



Do not worry too much as we have a strategy. After Visual Studio has scaffolded the view, we will simply edit the view, and *remove* the ability to edit the customer name.



Edit customer

Eduardo Martins (from company Woodstock Discos)

Display the customer name

Phone	+55 (11) 3033-5446
Fax	+55 (11) 3033-4564
Email	eduardo@woodstock.com.br

Let us create the view model class. Since this model will enable the user to edit the contact info for a customer in an HTML Form, we can call it **CustomerEditContactFormViewModel**. For brevity, data annotations have been removed.

```
public class CustomerEditContactFormViewModel
{
    [Key]
    public int CustomerId { get; set; }

    public string FirstName { get; set; }

    public string LastName { get; set; }

    public string Company { get; set; }

    public string Phone { get; set; }

    public string Fax { get; set; }

    public string Email { get; set; }
}
```

This approach does introduce another problem. The shape of the object that is sent to the view will be ***different*** than the shape of the object that is posted by the user from the HTML Form. In other words:

To the browser: 7 properties (CustomerId, FirstName, LastName, Company, Phone, Fax, Email)

Posted from the HTML Form: 4 properties (CustomerId, Phone, Fax, Email)

One solution is to create another view model class with only the four properties that describe the object that is posted by the user from the HTML Form. For example:

```
public class CustomerEditContactViewModel
{
    [Key]
    public int CustomerId { get; set; }

    [StringLength(24)]
    public string Phone { get; set; }

    [StringLength(24)]
    public string Fax { get; set; }

    [Required]
    [StringLength(60)]
    public string Email { get; set; }
}
```

If you wanted to, you could use inheritance here. That is, **CustomerEditContactFormViewModel** could inherit from **CustomerEditContactViewModel** but it is not necessary here.

Create a mapper

In the controller, we will fetch the object-to-be-edited from the data store. The fetch task will re-use the "get one" method in the manager object. The returned object will have the **CustomerBaseViewModel** data type.

We will need a new mapping from a **CustomerBaseViewModel** class to **CustomerEditContactFormViewModel** class. Add it now.

Do we need a mapping from **CustomerEditContactViewModel** to **Customer**?

Interestingly, the answer is "no". Instead, we will use a different mapping technique to do this, explained next.

In the Manager, create an "edit existing" method

This method will update an existing object and save the changes to the data store. The method signature needs:

An argument of type **CustomerEditContactViewModel** that can be passed in from the controller

and a return type of type **CustomerBaseViewModel** that can be passed back to the controller

Here's the method's algorithm:

1. Attempt to locate the object that is being edited.
2. If not found, return null (to the controller).
3. Otherwise, make the changes, and then save the changes.
4. Return the freshly edited object back to the controller.

A typical "edit existing" method could look like this …

```
public CustomerBaseViewModel CustomerEditContactInfo(CustomerEditContactViewModel customer)
{
    // Attempt to fetch the object.
    var obj = ds.Customers.Find(customer.CustomerId);

    if (obj == null)
    {
        // Customer was not found, return null.
        return null;
    }
    else
    {
        // Customer was found.  Update the entity object
        // with the incoming values then save the changes.
        ds.Entry(obj).CurrentValues.SetValues(customer);
        ds.SaveChanges();

        // Prepare and return the object.
        return mapper.Map<Customer, CustomerBaseViewModel>(obj);
    }
}
```

How does this statement work? ds.Entry(obj).CurrentValues.SetValues(customer);

Let us start at the beginning. Please check out the MSDN links where appropriate.

ds.Entry(obj)

This returns an object which provides access to information about an entity (*obj* in this situation) and allows control of the properties. You must ask for an Entry that is contained in the data context in-memory temporary workspace. Here's the MSDN document.

CurrentValues

The Entry object has a CurrentValues property, which gets the current values for the entity's properties as a collection of DbPropertyValues. Here's its MSDN document.

SetValues(customer);

The CurrentValues property has a SetValues() method. This sets (changes) the CurrentValues properties to those in a passed-in object (customer). From this MSDN document:

> *The [passed-in] object can be of any type.*
> *Any property on the [passed-in] object with a name that matches a property name in the [data store object] and can be read will be read.*
> *Other properties will be ignored.*

The SetValues() function ignores navigation properties. Overall, it is a very safe way to make changes to objects. We will use this for "edit existing" tasks and possibly other kinds of tasks.

In the controller, modify the Edit() methods

As you have seen several times already, we use two methods for “add new”, “edit existing”, and “delete item” in ASP.NET MVC web apps.

The first method responds to HTTP **GET**. Its job is to prepare the data that is needed for an HTML Form and pass the data to the view, which appears in the user’s browser.

The other method responds to HTTP **POST** when the user submits the HTML Form. Its job is to validate the incoming data, process it, and handle the result.

Controller Edit() method, for HTTP GET

Its job is to prepare the data that is needed for an HTML Form and pass the data to the view, which appears in the user’s browser. We need to provide enough information to enable the user to know the context or environment for the work to be done and the editable properties.

Notice that the signature of the scaffolded Edit() method has an int argument for the object identifier (the details method had an int too).

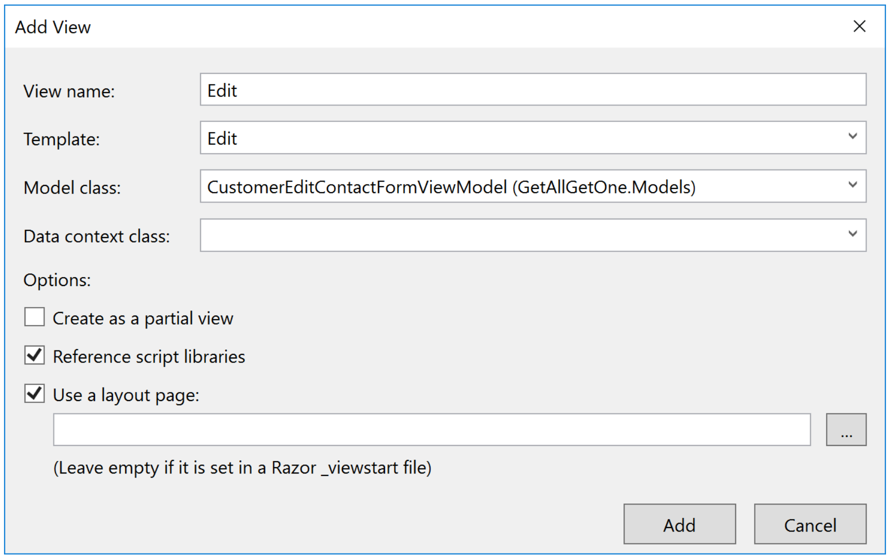
This is a best practice – we must get a reference to the object-to-be-edited and the most reliable reference is its unique identifier. You may have noticed that the scaffolder for the “List” and “Details” views automatically creates an “Edit” link that includes the unique identifier in the URL.

To continue, we will use the passed-in object identifier and fetch the desired object. If not found, return “not found” otherwise create a view model object that will be used in the view to create the HTML Form. Above, we created the **CustomerEditContactFormViewModel** class. We will map the source **CustomerBaseViewModel** object to this new “form” object then pass it to the view.

```
// GET: Customers/Edit/5
public ActionResult Edit(int? id)
{
    // Attempt to fetch the matching object
    var obj = m.CustomerGetById(id.GetValueOrDefault());

    if (obj == null)
        return HttpNotFound();
    else
    {
        // Create and configure an "edit form"

        // Notice that obj is a CustomerBaseViewModel object so
        // we must map it to a CustomerEditContactFormViewModel object
        // Notice that we can use AutoMapper anywhere,
        // and not just in the Manager class.
        var formObj = m.mapper.Map<CustomerBaseViewModel, CustomerEditContactFormViewModel>(obj);

        return View(formObj);
    }
}
```

Create an “edit existing” view, using the scaffolder

Create the “edit existing” view. Complete the dialog as shown below:



View name: Edit

Template: Edit

Model class: CustomerEditContactFormViewModel (GetAllGetOne.Models)

Data context class:

Options:

- [ ] Create as a partial view
- [x] Reference script libraries
- [x] Use a layout page:

(Leave empty if it is set in a Razor _viewstart file)

Modify the view so that we are only allowing the user to edit the appropriate properties. Comment out the code related to the unnecessary properties.

```
@*<div class="form-group">
    @Html.LabelFor(model => model.FirstName, htmlAttributes: new { @class = "control-label col-md-2" })
    <div class="col-md-10">
        @Html.EditorFor(model => model.FirstName, new { htmlAttributes = new { @class = "form-control" } })
        @Html.ValidationMessageFor(model => model.FirstName, "", new { @class = "text-danger" })
    </div>
</div>

<div class="form-group">
    @Html.LabelFor(model => model.LastName, htmlAttributes: new { @class = "control-label col-md-2" })
    <div class="col-md-10">
        @Html.EditorFor(model => model.LastName, new { htmlAttributes = new { @class = "form-control" } })
        @Html.ValidationMessageFor(model => model.LastName, "", new { @class = "text-danger" })
    </div>
</div>

<div class="form-group">
    @Html.LabelFor(model => model.Company, htmlAttributes: new { @class = "control-label col-md-2" })
    <div class="col-md-10">
        @Html.EditorFor(model => model.Company, new { htmlAttributes = new { @class = "form-control" } })
        @Html.ValidationMessageFor(model => model.Company, "", new { @class = "text-danger" })
    </div>
</div>*@
```

Show the value of the “commented out” properties as the subtitle of the view. Instead of:

```
<h4>CustomerEditContactFormViewModel</h4>
```

We can use the following:

```
<h4>@Model.FirstName @Model.LastName (@Model.Company)</h4>
```

Controller Edit() method, for HTTP POST

The job of this method is to validate the incoming data, process it, and handle the result.

Just like the Create() method that handled the HTTP POST, we will change the FormCollection argument so that we can take advantage of strong typing, model binding, and validation.

```
public ActionResult Edit(int? id, CustomerEditContactViewModel model)
```

We should always perform integrity checks on the data:

1. Model binding and validation
2. Data tampering

The first check is done with the same kind of logic that was used in the "add new" situation. Simply check whether model state is valid.

If this test fails, then we must display the HTML Form again. To keep the code simple, and the concept clear, we will simply re-display the HTML Form. Later, we will learn a better and more complete way of handling this situation.

The second check compares the identifier in the URL and checks whether it matches the identifier that is inside the package of data that was sent by the browser.

If this test fails, then the data (sent by the browser) was tampered with so we should redirect to the list-of-customers view.

If both tests pass, then we can attempt to do the update using the method in the **Manager** object. If the **Manager** returns null, then there was a problem with the data so our initial approach (like above) will be to re-display the HTML Form. Later, we will learn a better and more complete way of handling this situation.

If the update succeeds, then we should redirect to the "Details" view so that the user can see and visually confirm that the edit was done correctly.

(continued next page)

Here is the code for the method:

```
// POST: Customers/Edit/5
[HttpPost]
public ActionResult Edit(int? id, CustomerEditContactViewModel model)
{
    // Validate the input
    if (!ModelState.IsValid)
    {
        // Our "version 1" approach is to display the "edit form" again
        return RedirectToAction("Edit", new { id = model.CustomerId });
    }

    if (id.GetValueOrDefault() != model.CustomerId)
    {
        // This appears to be data tampering, so redirect the user away
        return RedirectToAction("Index");
    }

    // Attempt to do the update
    var editedItem = m.CustomerEditContactInfo(model);

    if (editedItem == null)
    {
        // There was a problem updating the object
        // Our "version 1" approach is to display the "edit form" again
        return RedirectToAction("Edit", new { id = model.CustomerId });
    }
    else
    {
        // Show the details view, which will show the updated data
        return RedirectToAction("Details", new { id = model.CustomerId });
    }
}
```

Customer “Delete Existing” Functionality

Like “add new” and “edit existing”, this pattern is implemented with two controller methods and one manager method. For this use case, we must deliver an HTML Form to the user. The HTML Form will show the user some information about the item-to-be-deleted and give them an opportunity to confirm the delete request (or navigate away). If the user confirms the delete intention, then we must accept the posted data and process it.

In the Manager, create an “delete item” method

This will be a simple method. It needs an argument – the object identifier.

Should it return a result? In this course we will use a bool return result. The “true” value will mean that the item was deleted.

The method’s logic is straightforward. First, attempt to fetch the item to be deleted. If the item is not found then return false otherwise call the Remove() method on the collection and SaveChanges(). Return “true”.

Here’s the method’s code:

```
public bool CustomerDelete(int id)
{
    // Attempt to fetch the object to be deleted
    var itemToDelete = ds.Customers.Find(id);

    if (itemToDelete == null)
        return false;
    else
    {
        // Remove the object
        ds.Customers.Remove(itemToDelete);
        ds.SaveChanges();

        return true;
    }
}
```

Controller Delete() method, for HTTP GET

Its job is to prepare the data that is needed for an HTML Form and pass the data to the view, which appears in the user’s browser.

What data do we need for the form? We need enough data to enable the user to know the context or environment for the work to be done.

Notice the signature of the scaffolded Delete() method. It has an int argument for the object identifier. Similar to the scaffolded Details() method.

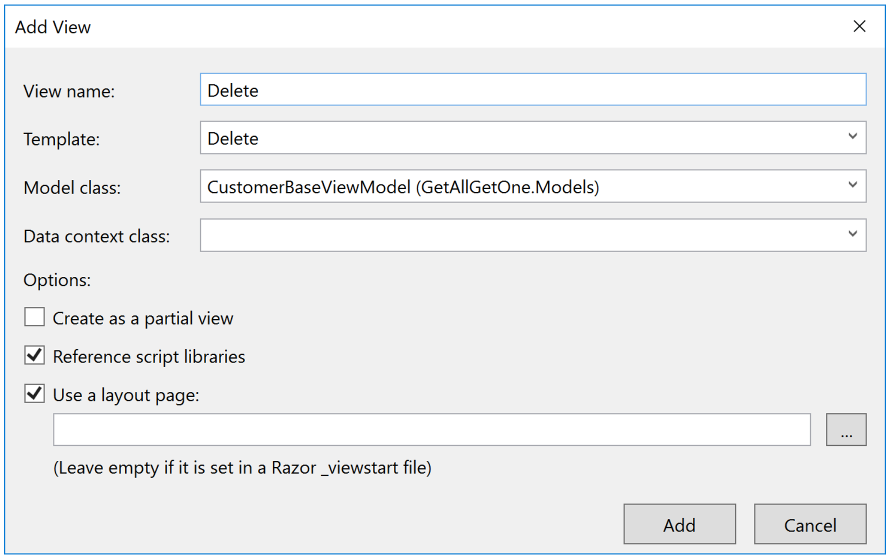
This is a best practice – we must get a reference to the object-to-be-deleted and the most reliable reference is its unique identifier. You may have noticed that the scaffolder for the "List" view automatically creates an "Delete" link that includes the unique identifier in the URL.

To continue, we will use the passed-in object identifier and fetch the desired object. If the object is not found, then redirect back to the list. Why? We don't want to give the user information that would enable them to "guess" which objects exist, and which do not. If the object is found, we can pass the fetched **CustomerBaseViewModel** object to the view.

```
// GET: Customers/Delete/5
public ActionResult Delete(int? id)
{
    var itemToDelete = m.CustomerGetById(id.GetValueOrDefault());

    if (itemToDelete == null)
    {
        // Don't leak info about the delete attempt
        // Simply redirect
        return RedirectToAction("Index");
    }
    else
        return View(itemToDelete);
}
```

Create a "delete item" view, using the scaffolder

Create the "delete item" view. Complete the dialog as shown below:



Add View

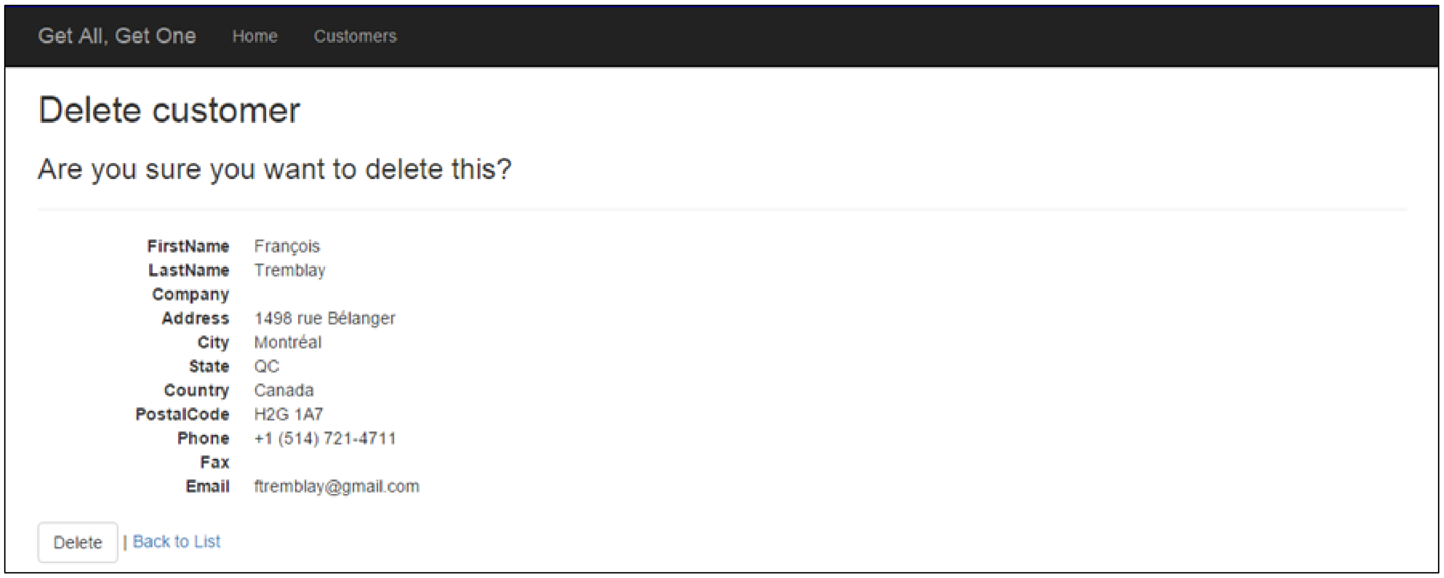
View name:	Delete
Template:	Delete
Model class:	CustomerBaseViewModel (GetAllGetOne.Models)
Data context class:	

Options:

- ☐ Create as a partial view
- ☑ Reference script libraries
- ☑ Use a layout page:

(Leave empty if it is set in a Razor _viewstart file)

If you test the delete functionality, you will see something like the following:



Delete customer

Are you sure you want to delete this?

FirstName	François
LastName	Tremblay
Company	
Address	1498 rue Bélanger
City	Montréal
State	QC
Country	Canada
PostalCode	H2G 1A7
Phone	+1 (514) 721-4711
Fax	
Email	ftremblay@gmail.com

Controller Delete() method, for HTTP POST

The job of this method is to validate the incoming data, process it, and handle the result.

The incoming data package is effectively empty – we do not need any data from the user to handle a delete request since the object identifier is in the URL. Confirm this by studying the view and the rendered HTML Form.

The controller method will call the new **Manager** method and pass the object identifier as the argument.

We do not care what the return result is – true or false – because the result will not trigger any further processing. At the end, we will just redirect to the list-of-customers.

The method code follows:

```
// POST: Customers/Delete/5
[HttpPost]
public ActionResult Delete(int? id, FormCollection collection)
{
    var result = m.CustomerDelete(id.GetValueOrDefault());

    // "result" will be true or false
    // We probably won't do much with the result, because
    // we don't want to leak info about the delete attempt

    // In the end, we should just redirect to the list view
    return RedirectToAction("Index");
}
```

When you test this method… please note you may receive an error while attempting to delete a customer that has invoice references. This error stems from the integrity rules on the relational database. Do not worry about this for now, later in the course we will learn how to deal with this.