Hands-On Lab

What’s New in Web Forms in ASP.NET 4.5

Lab version: 1.2.0

Last updated: 3/9/2012

* + 1. 

Contents

[Overview 3](#_Toc319078511)

[Exercise 1: Model Binding in ASP.NET Web Forms 5](#_Toc319078512)

[Task 1 – Using Strongly-Typed Data-Bindings 5](#_Toc319078513)

[Task 2 – Introducing Model Binding in Web Forms 9](#_Toc319078514)

[Task 3 – Value Providers in Model Binding 12](#_Toc319078515)

[Task 4 – Using Model Binding for Filtering 14](#_Toc319078516)

[Task 5 – Using Model Binding for Update Operations 18](#_Toc319078517)

[Exercise 2: Data Validation 20](#_Toc319078518)

[Task 1 – Unobtrusive Validation 20](#_Toc319078519)

[Task 2 – Validating the Model with Data Annotations 26](#_Toc319078520)

[Task 3 – Handling Custom Database Errors with ModelState 30](#_Toc319078521)

[Task 4 – Request Validation in ASP.NET Web Forms 4.5 33](#_Toc319078522)

[Exercise 3: Asynchronous Page Processing in ASP.NET Web Forms 37](#_Toc319078523)

[Task 1 – Updating the Product Details Page to Upload and Show Images 37](#_Toc319078524)

[Task 2 – Adding Asynchronous Processing to the Product Details Page 41](#_Toc319078525)

[Appendix: Using Code Snippets 45](#_Toc319078526)

1. To give feedback please write to [VSKitFdbk@Microsoft.com](mailto:VSKitFdbk@Microsoft.com).
2. Copyright © 2012 by Microsoft Corporation. All rights reserved.

Overview

* 1. The upcoming version of ASP.NET Web Forms introduces a number of improvements focused on improving user experience when working with data.
  2. In previous versions of Web Forms, when using data-binding to emit the value of an object member, you used the data-binding expressions Bind() or Eval(). In the new version of ASP.NET, you are able to declare what type of data a control is going to be bound to by using a new ItemType property. Setting this property will enable you to use a strongly-typed variable to receive the full benefits of the Visual Studio development experience, such as IntelliSense, member navigation, and compile-time checking.
  3. With the data-bound controls, you can now also specify your own custom methods for selecting, updating, deleting and inserting data, simplifying the interaction between the page controls and your application logic. Additionally, model binding capabilities have been added to ASP.NET, which means you can map data from the page directly into method type parameters.
  4. Validating user input should also be easier with the latest version of Web Forms. You can now annotate your model classes with validation attributes from the System.ComponentModel.DataAnnotations namespace and request that all your site controls validate user input using that information. Client-side validation in Web Forms is now integrated with jQuery, providing cleaner client-side code and unobtrusive JavaScript features.

Lastly, in the request validation area, improvements have been made to make it easier to selectively turn off request validation for specific parts of your applications or read invalidated request data.

# Objectives

* 1. In this hands-on lab, you will learn how to:
  + Use strongly-typed data-binding expressions
  + Use new model binding features in Web Forms
  + Use value providers for mapping page data to code-behind methods
  + Use Data Annotations for user input validation
  + Take advange of unobstrusive client-side validation with jQuery in Web Forms
  + Implement granular request validation
  + Implement asyncrhornous page processing in Web Forms

# Prerequisites

* + [Microsoft Visual Studio 11 Ultimate Beta](http://go.microsoft.com/fwlink/?LinkId=240160)
  + [Windows PowerShell](http://support.microsoft.com/kb/968930/) (for setup scripts – already installed on Windows 7 and Windows Server 2008 R2)

# Setup

* 1. Throughout the lab document, you will be instructed to insert code blocks. For your convenience, most of that code is provided as Visual Studio Code Snippets, which you can use from within Visual Studio to avoid having to add it manually.
  2. To install the code snippets:
  3. Open a Windows Explorer window and browse to the lab’s **Source\Setup** folder.
  4. Double-click the **Setup.cmd** file in this folder to install the Visual Studio code snippets.

If you are not familiar with the Visual Studio Code Snippets, and want to learn how to use them, you can refer to the appendix from this document ‘Appendix: Using Code Snippets’.

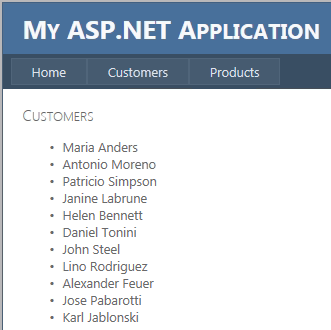
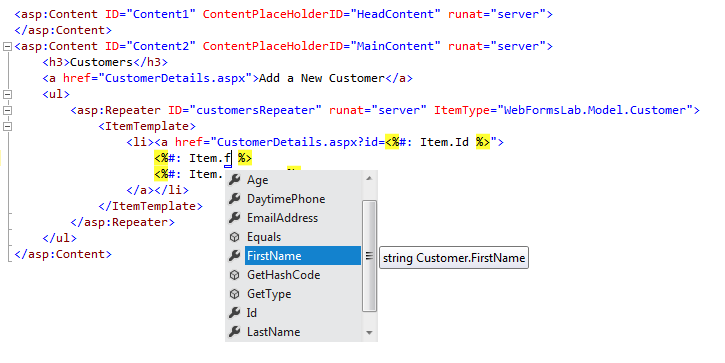
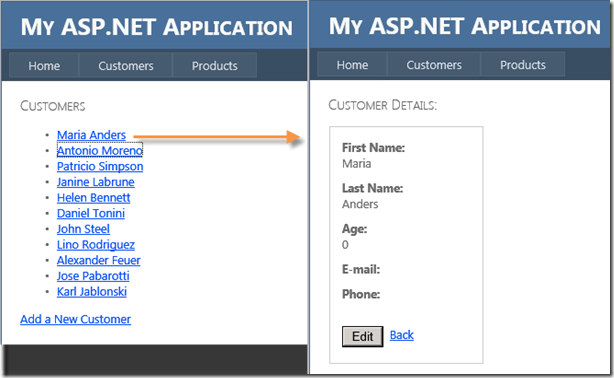
# Exercises

* 1. This hands-on lab includes the following exercises:
  2. Exercise 1: Model Binding in ASP.NET Web Forms
  3. Exercise 2: Data Validation
  4. Exercise 3: Asynchronous Page Processing in ASP.NET Web Forms
  5. **Note:** Each exercise is accompanied by a starting solution—located in the Begin folder of the exercise—that allows you to follow each exercise independently of the others. Please be aware that the code snippets that are added during an exercise are missing from these starting solutions and that they will not necessarily work until you complete the exercise.
  6. Inside the source code for an exercise, you will also find an End folder containing a Visual Studio solution with the code that results from completing the steps in the corresponding exercise. You can use these solutions as guidance if you need additional help as you work throughout this hands-on lab.

Exercise 1: Model Binding in ASP.NET Web Forms

* 1. The new version of ASP.NET Web Forms introduces a number of enhancements focused on improving the experience when working with data. Throughout this exercise, you will learn about strongly typed data-controls and model binding.

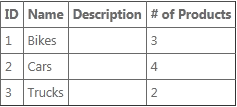
Task 1 – Using Strongly-Typed Data-Bindings

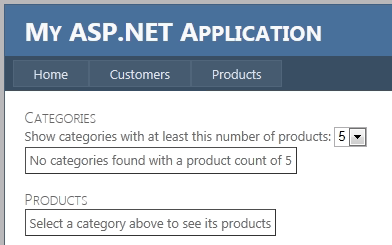
* 1. In this task, you will discover the new strongly-typed bindings available in ASP.NET 4.5.
  2. Open **Visual Studio 11** and open the **WebFormsLab-Ex1-Begin.sln** solution located in the **Source\Ex1-ModelBinding\Begin** folder of this lab.
  3. Open the **Customers.aspx** page. Place an unnumbered list in the main control and include a repeater control inside for listing each customer. Set the repeater name to **customersRepeater** as shown in the following code.
     1. In previous versions of Web Forms, when using data-binding to emit the value of a member on an object you’re data-binding to, you would use a data-binding expression, along with a call to the Eval method, passing in the name of the member as a string.
     2. At runtime, these calls to Eval will use reflection against the currently bound object to read the value of the member with the given name, and display the result in the HTML. This approach makes it very easy to data-bind against arbitrary, unshaped data.
     3. Unfortunately, you lose many of the great development-time experience features in Visual Studio, including IntelliSense for member names, support for navigation (like Go To Definition), and compile-time checking.
     4. (Code Snippet – *Web Forms Lab - Ex01 – Customers Repeater*)
     5. HTML
     6. <asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">
     7. ...
     8. **<ul>**
     9. **<asp:Repeater ID="customersRepeater" runat="server">**
     10. **<ItemTemplate>**
     11. **<li>**
     12. **<%# Eval("FirstName") %>**
     13. **<%# Eval("LastName") %>**
     14. **</li>**
     15. **</ItemTemplate>**
     16. **</asp:Repeater>**
     17. **</ul>**
     18. <a href="CustomerDetails.aspx"> Add a New Customer</a>
     19. </asp:Content>
  4. Open the **Customers.aspx.cs** file.
  5. In the **Page\_Load** method, add code to populate the repeater with the list of customers.
     1. (Code Snippet – *Web Forms Lab - Ex01 – Bind Customers Data Source*)
     2. C#
     3. protected void Page\_Load(object sender, EventArgs e)
     4. {
     5. **using (var db = new WebFormsLab.Model.ProductsContext())**
     6. **{**
     7. **this.customersRepeater.DataSource = db.Customers.ToList();**
     8. **this.customersRepeater.DataBind();**
     9. **}**
     10. }
     11. The solution uses EntityFramework together with CodeFirst to create and access the database. In the following code, the customersRepeater is bound to a materialized query that returns all the customers from the database.
  6. Press **F5** to run the solution and go to the **Customers** page to see the repeater in action. As the solution is using CodeFirst, the database will be created and populated in your local SQL Express instance when running the application.
     1. 
     2. Figure
     3. Listing the customers with a repeater
     4. **Note:** In Visual Studio 11, IIS Express is the default Web development server.
  7. Close the browser and go back to Visual Studio.
  8. Open the **Customers.aspx** page and use the new **ItemType** attribute in the repeater to set the **Customer** type as the binding type.
     1. HTML
     2. <asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">
     3. <ul>
     4. <asp:Repeater ID="customersRepeater"
     5. **ItemType="WebFormsLab.Model.Customer"**
     6. runat="server">
     7. <ItemTemplate>
     8. ...
     9. </ItemTemplate>
     10. </asp:Repeater>
     11. </ul>
     12. </asp:Content>
     13. The ItemType property enables you to declare which type of data the control is going to be bound to and allows you to use strongly-typed binding inside the data-bound control.
  9. Replace the ItemTemplate content with the following code.
     1. (Code Snippet – *Web Forms Lab - Ex01 – Costumer List Item*)
     2. HTML
     3. <asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">
     4. ...
     5. <ul>
     6. <asp:Repeater ID="customersRepeater" ItemType="WebFormsLab.Model.Customer" runat="server">
     7. <ItemTemplate>
     8. **<li>**
     9. **<a href="CustomerDetails.aspx?id=<%#: Item.Id %>">**
     10. **<%#: Item.FirstName %> <%#: Item.LastName %>**
     11. **</a>**
     12. **</li>**
     13. </ItemTemplate>
     14. </asp:Repeater>
     15. </ul>
     16. </asp:Content>
     17. Setting the ItemType property causes two new typed variables to be generated in the scope of the data-binding expressions: **Item** and **BindItem**. You can use these strongly typed variables in the data-binding expressions and get the full benefits of the Visual Studio development experience.
     18. The “**:**” used in the expression will automatically HTML-encode the output to avoid security issues (for example, cross-site scripting attacks). This notation was available since .NET 4 for response writing, but now is also available in data-binding expressions.
     19. **Note:** The Item member works for one-way binding. If you want to perform two-way binding use the **BindItem** member.
     20. 
     21. Figure 2
     22. IntelliSense support in strongly-typed binding
  10. Press **F5** to run the solution and go to the Customers page to make sure the changes work as expected.
      1. 
      2. Figure 3
      3. Listing customer details
  11. Close the browser and go back to Visual Studio.

Task 2 – Introducing Model Binding in Web Forms

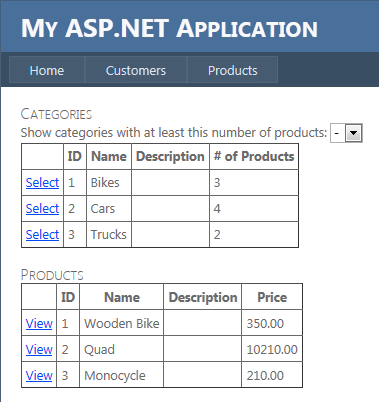
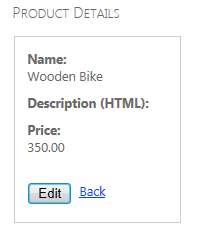
* 1. In previous versions of ASP.NET Web Forms, when you wanted to perform two-way data-binding, both retrieving and updating data, you needed to use a Data Source object. This could be an Object Data Source, a SQL Data Source, a LINQ Data Source and so on. However if your scenario required custom code for handling the data, you needed to use the Object Data Source and this brought some drawbacks. For example, you needed to avoid complex types and you needed to handle exceptions when executing validation logic.
  2. In the new version of ASP.NET Web Forms the data-bound controls support model binding. This means that you can specify select, update, insert and delete methods directly in the data-bound control to call logic from your code-behind file or from another class.
  3. To learn about this, you will use a GridView to list the product categories using the new **SelectMethod** attribute. This attribute enables you to specify a method for retrieving the GridView data.
  4. Open the **Products.aspx** page and include a **GridView**. Configure the GridView as shown below to use strongly-typed bindings and enable sorting and paging.
     1. (Code Snippet – *Web Forms Lab - Ex01 – Categories GridView*)
     2. HTML
     3. <asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">
     4. **<asp:GridView ID="categoriesGrid" runat="server"**
     5. **AutoGenerateColumns="false"**

**ItemType="WebFormsLab.Model.Category" DataKeyNames="CategoryID">**

* + 1. **<Columns>**
    2. **<asp:BoundField DataField="CategoryId" HeaderText="ID" SortExpression="CategoryId" />**
    3. **<asp:BoundField DataField="CategoryName" HeaderText="Name" SortExpression="CategoryName" />**
    4. **<asp:BoundField DataField="Description" HeaderText="Description" />**
    5. **<asp:TemplateField HeaderText="# of Products">**
    6. **<ItemTemplate><%#: Item.Products.Count %></ItemTemplate>**
    7. **</asp:TemplateField>**
    8. **</Columns>**
    9. **</asp:GridView>**
  1. Use the new **SelectMethod** attribute to configure the GridView to call a **GetCategories** method to select the data.
     1. HTML
     2. <asp:GridView ID="categoriesGrid" runat="server"
     3. AutoGenerateColumns="false"
     4. ItemType="WebFormsLab.Model.Category" DataKeyNames="CategoryId"
     5. **SelectMethod="GetCategories">**
     6. <Columns>
     7. <asp:BoundField DataField="CategoryId" HeaderText="ID" SortExpression="CategoryId" />
     8. <asp:BoundField DataField="CategoryName" HeaderText="Name" SortExpression="CategoryName" />
     9. <asp:BoundField DataField="Description" HeaderText="Description" />
     10. <asp:TemplateField HeaderText="# of Products">
     11. <ItemTemplate><%#: Item.Products.Count %></ItemTemplate>
     12. </asp:TemplateField>
     13. </Columns>
     14. </asp:GridView>
  2. Open the **Products.aspx.cs** code-behind file and add the following using statements.
     1. (Code Snippet – *Web Forms Lab - Ex01 – Namespaces*)
     2. C#
     3. **using System.Data.Entity;**
     4. **using System.Data.Entity.Infrastructure;**
     5. **using WebFormsLab.Model;**
  3. Add a private member in the **Products** class and assign a new instance of **ProductsContext**. This property will store the Entity Framework data context that enables you to connect to the database.
     1. C#
     2. public partial class Products : System.Web.UI.Page
     3. {
     4. **private ProductsContext db = new ProductsContext();**
     5. ...
  4. Create a **GetCategories** method to retrieve the list of categories using LINQ. The query will include the **Products** property so the GridView can show the amount of products for each category. Notice that the method returns a raw IQueryable object that represent the query to be executed later on the page lifecycle.
     1. (Code Snippet – *Web Forms Lab - Ex01 – GetCategories*)
     2. C#
     3. **public IQueryable<Category> GetCategories()**
     4. **{**
     5. **var query = this.db.Categories**
     6. **.Include(c => c.Products);**
     7. **return query;**
     8. **}**
     9. **Note:** In previous versions of ASP.NET Web Forms, enabling sorting and paging using your own repository logic within an Object Data Source context, required to write your own custom code and receive all the necessary parameters. Now, as the data-binding methods can return IQueryable and this represents a query still to be executed, ASP.NET can take care of modifying the query to add the proper sorting and paging parameters.
  5. Press **F5** to start debugging the site and go to the Products page. You should see that the GridView is populated with the categories returned by the GetCategories method.
     1. 
     2. Figure 4
     3. Populating a GridView using model binding
  6. Press **SHIFT**+**F5** Stop debugging.

1. Task 3 – Value Providers in Model Binding
   1. Model binding not only enables you to specify custom methods to work with your data directly in the data-bound control, but also allows you to map data from the page into parameters from these methods. On the method parameter, you can use value provider attributes to specify the value’s data source. For example:
   * Controls on the page
   * Query string values
   * View data
   * Session state
   * Cookies
   * Posted form data
   * View state
   * Custom value providers are supported as well
   1. If you have used ASP.NET MVC, you will notice the model binding support is similar. Indeed, these features were taken from ASP.NET MVC and moved into the System.Web assembly to be able to use them on Web Forms as well.
   2. In this task, you will update the GridView to filter its results by the amount of products for each category, receiving the filter parameter with model binding.
   3. Go back to the **Products.aspx** page.
   4. At the top of the GridView, add a **Label** and a **ComboBox** to select the number of products for each category as shown below.
      1. (Code Snippet – *Web Forms Lab - Ex01 – Categories DropDownList*)
      2. HTML
      3. **<h3>Categories</h3>**
      4. **<asp:Label ID="Label1" runat="server" AssociatedControlID="minProductsCount">Show categories with at least this number of products:</asp:Label>**
      5. **<asp:DropDownList runat="server" ID="minProductsCount" AutoPostBack="true">**
      6. **<asp:ListItem Value="" Text="-" />**
      7. **<asp:ListItem Text="1" />**
      8. **<asp:ListItem Text="3" />**
      9. **<asp:ListItem Text="5" />**
      10. **</asp:DropDownList>**
      11. **<br/>**
   5. Add an **EmptyDataTemplate** to the GridView to show a message when there are no categories with the selected number of products.
      1. (Code Snippet – *Web Forms Lab - Ex01 – No Categories Message*)
      2. HTML
      3. <asp:GridView ID="categoriesGrid" runat="server"
      4. AutoGenerateColumns="false"
      5. ItemType="WebFormsLab.Model.Category" DataKeyNames="CategoryId"
      6. SelectMethod="GetCategories">
      7. <Columns>
      8. <asp:BoundField DataField="CategoryId" HeaderText="ID" />
      9. <asp:BoundField DataField="CategoryName" HeaderText="Name" />
      10. <asp:BoundField DataField="Description" HeaderText="Description" />
      11. <asp:TemplateField HeaderText="# of Products">
      12. <ItemTemplate><%#: Item.Products.Count %></ItemTemplate>
      13. </asp:TemplateField>
      14. </Columns>
      15. **<EmptyDataTemplate>No categories found with a product count of <%#: minProductsCount.SelectedValue %></EmptyDataTemplate>**
      16. </asp:GridView>
   6. Open the **Products.aspx.cs** code-behind and add the following using statement (shown in **bold**).
      1. C#
      2. using System.Data.Entity;
      3. using System.Data.Entity.Infrastructure;
      4. using WebFormsLab.Model;
      5. **using System.Web.ModelBinding;**
   7. Modify the **GetCategories** method to receive an integer **minProductsCount** argument and filter the returned results. To do this, replace the method with the following code.
      1. (Code Snippet – *Web Forms Lab - Ex01 – GetCategories 2*)
      2. C#
      3. **public IQueryable<Category> GetCategories([Control]int? minProductsCount)**
      4. **{**
      5. **var query = this.db.Categories**
      6. **.Include(c => c.Products);**
      7. **if (minProductsCount.HasValue)**
      8. **{**
      9. **query = query.Where(c => c.Products.Count >= minProductsCount);**
      10. **}**
      11. **return query;**
      12. **}**
      13. The new **[Control]** attribute on the **minProductsCount** argument will let ASP.NET know its value must be populated using a control in the page. ASP.NET will look for any control matching the name of the argument (minProductsCount) and perform the necessary mapping and conversion to fill the parameter with the control value.
      14. Alternatively, the attribute provides an overloaded constructor that enables you to specify the control from where to get the value.
      15. **Note:** One goal of the data-binding features is to reduce the amount of code that needs to be written for page interaction.
      16. Apart from the [Control] value provider, you can use other model-binding providers in your method parameters. Some of them are listed in the task introduction.
   8. Press **F5** to start debugging the site and go to the Products page. Select a number of products in the drop-down list and notice how the GridView is now updated.
      1. 
      2. Figure 5
      3. Filtering the GridView with a drop-down list value
   9. Stop debugging.
2. Task 4 – Using Model Binding for Filtering

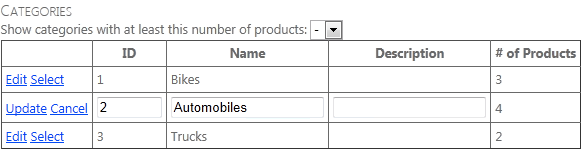
In this task, you will add a second, child GridView to show the products within the selected category.

* 1. Open the **Products.aspx** page and update the categories GridView to auto-generate the Select button.
     1. HTML
     2. <asp:GridView ID="categoriesGrid" runat="server"
     3. AutoGenerateColumns="false"
     4. ItemType="WebFormsLab.Model.Category" DataKeyNames="CategoryId"
     5. SelectMethod="GetCategories"
     6. **AutoGenerateSelectButton="true">**
  2. Add a second **GridView** named **productsGrid** at the bottom. Set the **ItemType** to **WebFormsLab.Model.Product**, the **DataKeyNames** to **ProductId** and the **SelectMethod** to **GetProducts**. Set **AutoGenerateColumns** to **false** and add the columns for ProductId, ProductName, Description and UnitPrice.
     1. (Code Snippet – *Web Forms Lab - Ex01 – Products GridView*)
     2. HTML
     3. **<h3>Products</h3>**
     4. **<asp:GridView ID="productsGrid" runat="server"**
     5. **CellPadding="4"**
     6. **AutoGenerateColumns="false"**
     7. **ItemType="WebFormsLab.Model.Product"**
     8. **DataKeyNames="ProductId"**
     9. **SelectMethod="GetProducts">**
     10. **<Columns>**
     11. **<asp:BoundField DataField="ProductId" HeaderText="ID" />**
     12. **<asp:BoundField DataField="ProductName" HeaderText="Name" />**
     13. **<asp:BoundField DataField="Description" HeaderText="Description" HtmlEncode="false" />**
     14. **<asp:BoundField DataField="UnitPrice" HeaderText="Price" />**
     15. **</Columns>**
     16. **<EmptyDataTemplate>**
     17. **Select a category above to see its products**
     18. **</EmptyDataTemplate>**
     19. **</asp:GridView>**
  3. Open the **Products.aspx.cs** code-behind file. Implement the **GetProducts** method to receive the category ID from the category GridView and filter the products. Model binding will set the parameter value using the selected row in the **categoriesGrid**. Since the argument name and control name do not match, you should specify the name of the control in the Control value provider.
     1. (Code Snippet – *Web Forms Lab - Ex01 – GetProducts*)
     2. C#
     3. **public IEnumerable<WebFormsLab.Model.Product> GetProducts([Control("categoriesGrid")]int? categoryId)**
     4. **{**
     5. **return this.db.Products.Where(p => p.CategoryId == categoryId);**
     6. **}**
     7. **Note:** This approach makes it easier to unit test these methods. On a unit test context, where Web Forms is not executing, the [Control] attribute will not perform any specific action.
  4. Open the **Products.aspx** page and locate the products GridView. Update the products GridView to show a link for editing the selected product.
     1. (Code Snippet – *Web Forms Lab - Ex01 – Edit Product Link*)
     2. HTML
     3. <h3>Products</h3>
     4. <asp:GridView ID="productsGrid" runat="server"
     5. CellPadding="4"
     6. AutoGenerateColumns="false"
     7. ItemType="WebFormsLab.Model.Product"
     8. DataKeyNames="ProductId"
     9. SelectMethod="GetProducts">
     10. <Columns>
     11. **<asp:TemplateField>**
     12. **<ItemTemplate><a href="ProductDetails.aspx?productId=<%#: Item.ProductId %>">View</a></ItemTemplate>**
     13. **</asp:TemplateField>**
     14. <asp:BoundField DataField="ProductId" HeaderText="ID" />
     15. <asp:BoundField DataField="ProductName" HeaderText="Name" />
     16. <asp:BoundField DataField="Description" HeaderText="Description" HtmlEncode="false" />
     17. <asp:BoundField DataField="UnitPrice" HeaderText="Price" />
     18. </Columns>
     19. <EmptyDataTemplate>
     20. Select a category above to see its products
     21. </EmptyDataTemplate>
     22. </asp:GridView>
  5. Open the **ProductDetails.aspx** page code-behind and add the **SelectProduct** method as shown below.
     1. Notice that the **[QueryString]** attribute is used to fill the method parameter from a productId parameter in the query string.
     2. (Code Snippet – *Web Forms Lab - Ex01 – SelectProduct Method*)
     3. C#
     4. **public Product SelectProduct([QueryString]int? productId)**
     5. **{**
     6. **return this.db.Products.Find(productId);**
     7. **}**
  6. Press **F5** to start debugging the site and go to the Products page. Select any category from the categories GridView and notice that the products GridView is updated.
     1. 
     2. Figure 6
     3. Showing products from the selected category
  7. Click the **View** link on a product to open the ProductDetails.aspx page.
     1. Notice that the page is retrieving the product with the SelectMethod using the productId parameter from the query string.
     2. 
     3. Figure 7
     4. Viewing the product details
     5. **Note:** The ability to type an HTML description will be implemented in the next exercise.

Task 5 – Using Model Binding for Update Operations

In the previous task, you have used model binding mainly for selecting data, in this task you will learn how to use model binding in update operations.

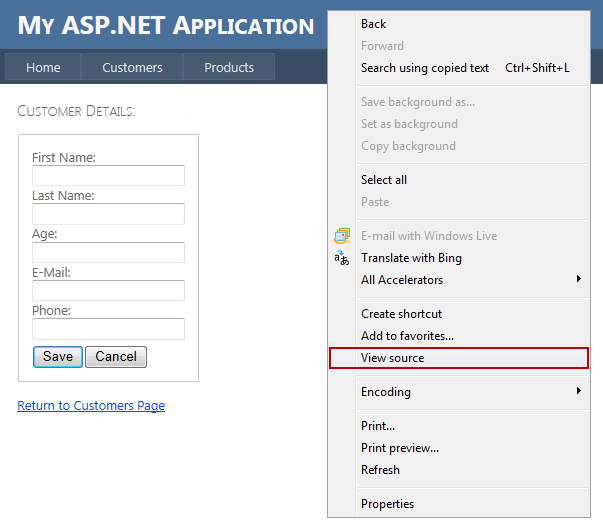
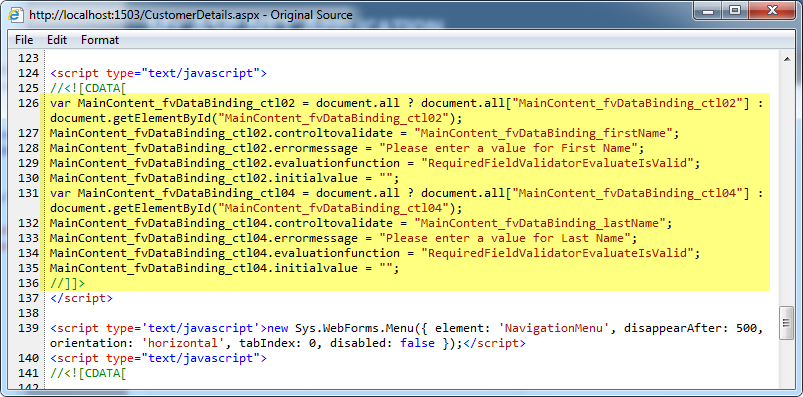
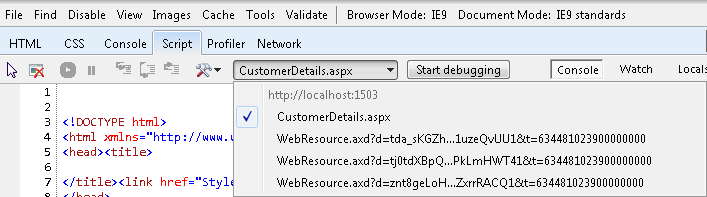
You will update the categories GridView to let the user update categories.

* 1. Open the **Products.aspx** page and update the categories GridView to auto-generate the Edit button and use the new **UpdateMethod** attribute to specify an **UpdateCategory** method to update the selected item.
     1. (Code Snippet – *Web Forms Lab - Ex01 – Enable Categories Update*)
     2. HTML
     3. <asp:GridView ID="categoriesGrid" runat="server"
     4. AutoGenerateColumns="false"
     5. CellPadding="4"
     6. ItemType="WebFormsLab.Model.Category" DataKeyNames="CategoryId"
     7. SelectMethod="GetCategories"
     8. AutoGenerateSelectButton="true"
     9. **AutoGenerateEditButton="true"**
     10. **UpdateMethod="UpdateCategory">**
     11. The DataKeyNames attribute in the GridView define which are the members that uniquely identify the model-bound object and therefore, which are the parameters the update method should at least receive.
  2. Open the **Products.aspx.cs** code-behind file and implement the **UpdateCategory** method. The method should receive the category ID to load the current category, populate the values from the GridView and then update the category.
     1. (Code Snippet – *Web Forms Lab - Ex01 – UpdateCategory*)
     2. C#
     3. **public void UpdateCategory(int categoryId)**
     4. **{**
     5. **var category = this.db.Categories.Find(categoryId);**
     7. **TryUpdateModel(category);**
     8. **if (ModelState.IsValid)**
     9. **{**
     10. **this.db.SaveChanges();**
     11. **}**
     12. **}**
     13. The new **TryUpdateModel** method in the Page class is responsible of populating the model object using the values from the controls in the page. In this case, it will replace the updated values from the current GridView row being edited into the **category** object.
     14. **Note:** The next exercise will explain the usage of the ModelState.IsValid for validating the data entered by the user when editing the object.
  3. Run the site and go to the Products page. Edit a category. Type a new name and then click **Update** to persist the changes.
     1. 
     2. Figure 8
     3. Editing categories

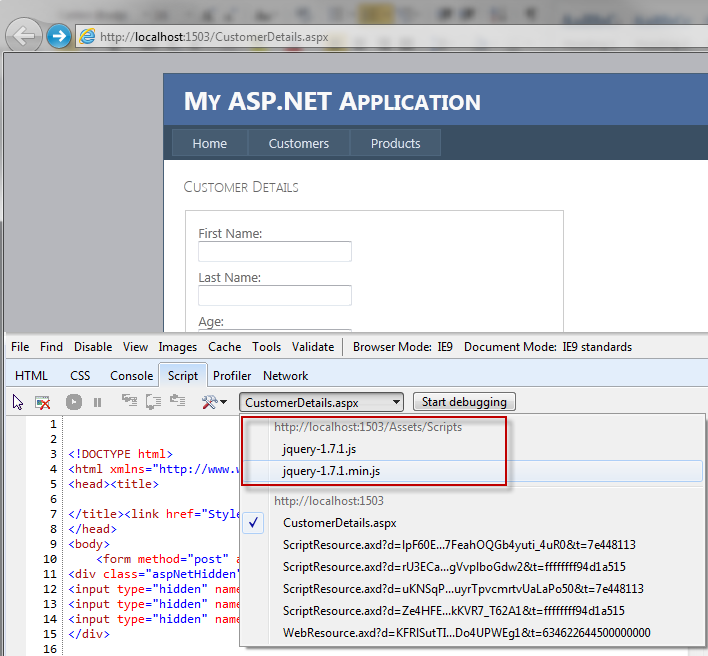
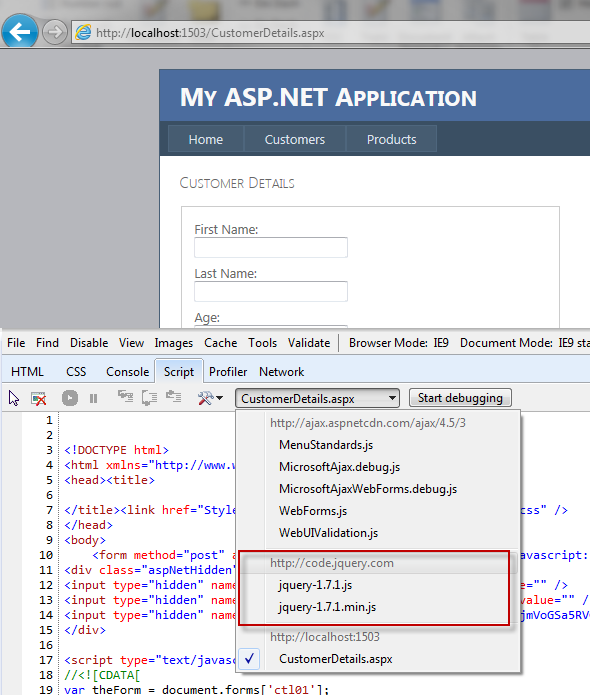
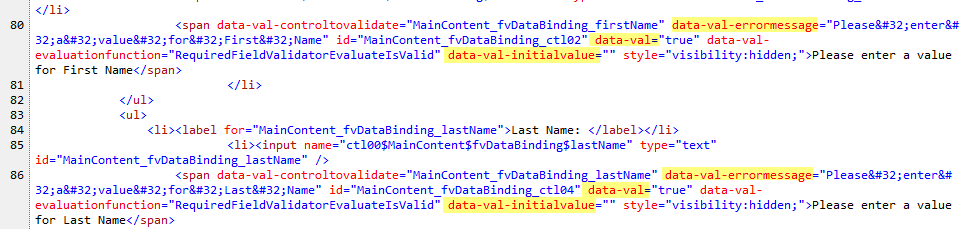
Exercise 2: Data Validation

* 1. In this exercise, you will learn about the new data validation features in ASP.NET 4.5. You will check out the new unobtrusive validation features in Web Forms. You will use data annotations in the application model classes for user input validation, and finally, you will learn how to turn on or off request validation to individual controls in a page.

Task 1 – Unobtrusive Validation

* 1. Forms with complex data including validators tend to generate too much JavaScript code in the page, which can represent about 60% of the code. With unobtrusive validation enabled, your HTML code will look cleaner and tidier.
  2. In this section, you will enable unobtrusive validation in ASP.NET to compare the HTML code generated by both configurations.
  3. Open **Visual Studio 11** and open the **WebFormsLab-Ex2-Begin.sln** solution located in the **Source\Ex2-Validation\Begin** folder of this lab. Alternatively, you can continue working on your existing solution from the previous exercise.
  4. Press **F5** to start the web application. Go to the Customers page and click the **Add a New Customer** link.
  5. Right-click on the browser page, and select **View Source** option to open the HTML code generated by the application.
     1. 
     2. Figure 9
     3. Showing the page HTML code
  6. Scroll through the page source code and notice that ASP.NET has injected JavaScript code in the page to perform the validations and show the error list.
     1. 
     2. Figure 10
     3. Validation JavaScript code in CustomerDetails page
  7. Press **F12** to open the developer tools.
  8. Select the **Script** tab and expand the “CustomerDetails.aspx” combo to see all the referenced scripts.
     1. Notice that there are no references to the jQuery library. The validations are performed using the Microsoft Ajax Libraries together with JavaScript code injected within the page.
     2. 
     3. Figure 11
     4. No jQuery used for validations
  9. Close the browser and go back to Visual Studio.
  10. Now you will enable unobtrusive validation. Open **Web.Config** and locate **ValidationSettings:UnobtrusiveValidationMode** key in the **AppSettings** section**.** Set the key value to **WebForms**.
      1. Web.Config
      2. <configuration>
      3. ...
      4. <appSettings>
      5. <add key="aspnet:uselegacysynchronizationcontext" value="false" />
      6. **<add key="ValidationSettings:UnobtrusiveValidationMode" value="WebForms"/>**
      7. **Note:** You can also set this property in the “**Page\_Load**” event in case you want to enable Unobtrusive Validation only for some pages.
  11. Open the **Global.asax.cs** file and add the following using statement (shown in bold)
      1. (Code Snippet *– Web Forms Lab - Ex02 – Using statements)*
      2. C#
      3. **using System.Web.SessionState;**
      4. **using WebFormsLab.Model;**
      5. **using System.Web.ModelBinding;**
  12. Within the **Application\_Start** method of the **Global.asax.cs** file, add the following code
      1. (Code Snippet *– Web Forms Lab - Ex02 – ScriptREsourceDefinition)*
      2. C#
      3. **ScriptResourceDefinition myScriptResDef = new ScriptResourceDefinition();**
      4. **myScriptResDef.Path = "~/Assets/Scripts/jquery-1.7.1.min.js";**
      5. **myScriptResDef.DebugPath = "~/Assets/Scripts/jquery-1.7.1.js";**
      6. **myScriptResDef.CdnPath = "http://code.jquery.com/jquery-1.7.1.min.js";**
      7. **myScriptResDef.CdnDebugPath = "http://code.jquery.com/jquery-1.7.1.js";**
      8. **ScriptManager.ScriptResourceMapping.AddDefinition("jquery", null, myScriptResDef);**
  13. Open **Site.Master.** Add the code below to include a **ScriptManager** on the page to include a script reference to the jQuery client library.
      1. (Code Snippet – *Web Forms Lab - Ex01 – Script Manager*)
      2. HTML
      3. <form runat="server">
      4. **<asp:ScriptManager ID="uxScriptManagerMasterPage" runat="server" EnableCdn="False">**
      5. **<Scripts>**
      6. **<asp:ScriptReference Name="jquery" />**
      7. **</Scripts>**
      8. **</asp:ScriptManager>**

...

* 1. Open **CustomerDetails.aspx** and press **F5** to start the Web application.
  2. Press the F12 key to open the IE developer tools. Once the developer tools is open, select the script tab. Select **CustomerDetails.aspx** from the menu and take note that the scripts required to run jQuery on the page have been loaded into the browser from the local site.
     1. 
     2. Figure 12
     3. Loading the jQuery JavaScript files directly from the local IIS server
  3. Close the browser to return to Visual Studio. Open the **Site.Master** file again and locate the **ScriptManager** that you just added to the page. Change the value of the **EnableCdn** property to be **True**. This will force jQuery to be loaded from the online URL, not from the local site’s URL.
  4. Open **CustomerDetails.aspx** in Visual Studio. Press the F5 key to run the site. Once Internet Explorer opens, press the F12 key to open the developer tools. Select the **Script** tab, and then take a look at the drop-down list. Note the jQuery JavaScript files are no longer being loaded from the local site, but rather from the online jQuery CDN.
     1. 
     2. Figure 13
     3. Loading the jQuery JavaScript files from the jQuery CDN
  5. Open the HTML page source code again using the View source option in the browser. Notice that by enabling the unobtrusive validation ASP.NET has replaced the injected JavaScript code with data- \*attributes.
     1. 
     2. Figure 14
     3. Unobtrusive validation code
     4. **Note:** In this example, you saw how a validation summary with Data annotations was simplified to only a few HTML and JavaScript lines. Previously, without unobtrusive validation, the more validation controls you add, the bigger your JavaScript validation code will grow.

Task 2 – Validating the Model with Data Annotations

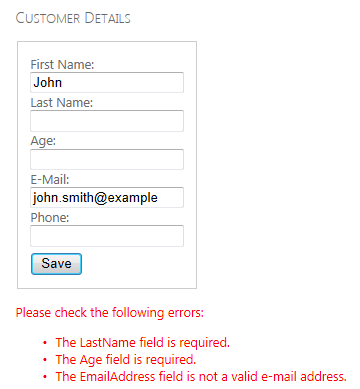
* 1. ASP.NET 4.5 introduces data annotations validation for Web Forms. Instead of having a validation control on each input, you can now define constraints in your model classes and use them across all your web application. In this section, you will learn how to use data annotations for validating a new/edit customer form.
  2. Open **CustomerDetail.aspx** page**.** Notice that the customer first name and second name in the EditItemTemplate and InsertItemTemplate sections are validated using a RequiredFieldValidator controls. Each validator is associated to a particular condition, so you need to include as many validators as conditions to check.
  3. Add data annotations to validate the Customer model class. Open **Customer.cs** class in the **Model** folder and *decorate* each property using data annotation attributes.
     1. (Code Snippet – *Web Forms Lab - Ex02 – Data Annotation*s)
     2. C#
     3. namespace WebFormsLab.Model
     4. {
     5. using System.Collections.Generic;
     6. using System.ComponentModel.DataAnnotations;
     7. public class Customer
     8. {
     9. **[Key]**
     10. **public int Id { get; set; }**
     11. **[Required]**
     12. **public string FirstName { get; set; }**
     13. **[Required]**
     14. **public string LastName { get; set; }**
     15. **[Range(0, 130)]**
     16. **public int Age { get; set; }**
     17. **public Address Address { get; set; }**
     18. **[Phone]**
     19. **public string DaytimePhone { get; set; }**
     20. **[EmailAddress, StringLength(256)]**
     21. **public string EmailAddress { get; set; }**
     22. }
     23. }
     24. **Note:** .NET Framework 4.5 has extended the existing data annotation collection. These are some of the data annotations you can use: [CreditCard], [Phone], [EmailAddress], [Range], [Compare], [Url], [FileExtensions], [Required], [Key], [RegularExpression].

Some usage examples:

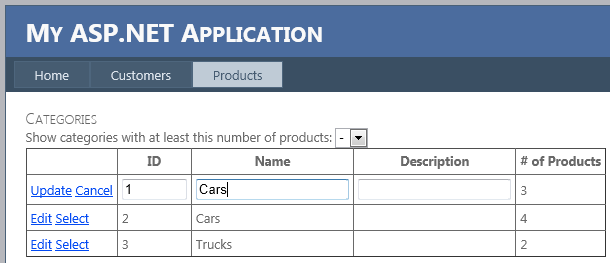
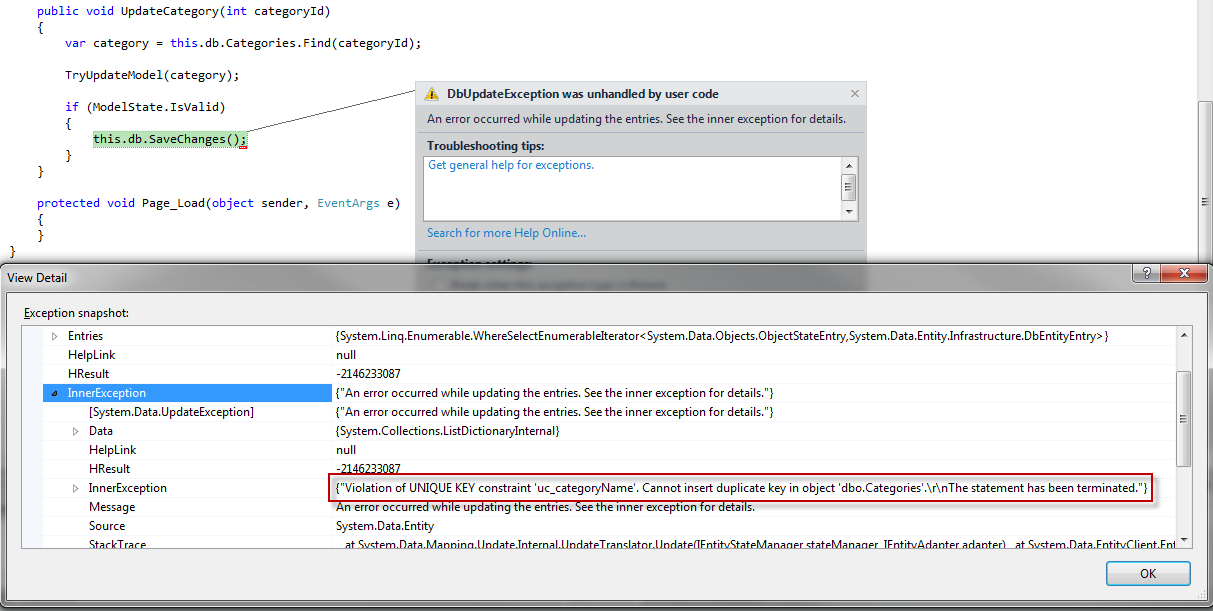
[Key]: Specifies that an attribute is the unique identifier

* + 1. [Range(0.4, 0.5, ErrorMessage="{Write an error message}"]: Double range
    2. [EmailAddress(ErrorMessage="Invalid Email"), MaxLength(56)]: Two annotations in the same line.
    3. You can also define your own error messages within each attribute.
  1. Open **CustomerDetails.aspx** and remove all the RequiredFieldvalidators for the first and last name fields in the in EditItemTemplate and InsertItemTemplate sections of the FormView control.
     1. HTML
     2. <EditItemTemplate>
     3. <fieldset>
     4. <ul>
     5. <li><asp:Label runat="server" AssociatedControlID="firstName">First Name: </asp:Label></li>
     6. <li><asp:TextBox runat="server" ID="firstName" Text='<%#: BindItem.FirstName %>' />
     7. <!-- Remove the following line-->
     8. ~~&nbsp;<asp:RequiredFieldValidator runat="server" ControlToValidate="firstName" ErrorMessage="Please enter a value for First Name" ForeColor="Red" />~~
     9. </li>
     10. </ul>
     11. <ul>
     12. <li><asp:Label runat="server" AssociatedControlID="lastName">Last Name: </asp:Label></li>
     13. <li><asp:TextBox runat="server" ID="lastName" Text='<%#: BindItem.LastName %>' />
     14. <!-- Remove the following line-->
     15. ~~&nbsp;<asp:RequiredFieldValidator runat="server" ControlToValidate="lastName" ErrorMessage="Please enter a value for Last Name" ForeColor="Red" />~~
     16. </li>
     17. </ul>
     18. <ul>
     19. ...
     20. <InsertItemTemplate>
     21. <fieldset>
     22. <ul>
     23. <li><asp:Label runat="server" AssociatedControlID="firstName">First Name: </asp:Label></li>
     24. <li><asp:TextBox runat="server" ID="firstName" Text='<%# BindItem.FirstName %>' />
     25. <!-- Remove the following line-->
     26. ~~&nbsp;<asp:RequiredFieldValidator runat="server" ControlToValidate="firstName" ErrorMessage="Please enter a value for First Name" ForeColor="Red" />~~
     27. </li>
     28. </ul>
     29. <ul>
     30. <li><asp:Label runat="server" AssociatedControlID="lastName">Last Name: </asp:Label></li>
     31. <li><asp:TextBox runat="server" ID="lastName" Text='<%#: BindItem.LastName %>' />
     32. <!-- Remove the following line-->
     33. ~~&nbsp;<asp:RequiredFieldValidator runat="server" ControlToValidate="lastName" ErrorMessage="Please enter a value for Last Name" ForeColor="Red" />~~
     34. </li>
     35. </ul>
     36. <ul>
     37. **Note:** One advantage of using data annotations is that validation logic is not duplicated in your application pages. You define it once in the model, and use it across all the application pages that manipulate data.
  2. Open **CustomerDetails.aspx** code-behind and locate the SaveCustomer method. This method is called when inserting a new customer and receives the Customer parameter from the FormView control values. When the mapping between the page controls and the parameter object occurrs, ASP.NET will execute the model validation against all the data annotation attributes and fill the ModelState dictionary with the errors encountered, if any.
     1. The ModelState.IsValid will only return true if all the fields on your model are valid after performing the validation.
     2. C#
     3. public void SaveCustomer(Customer customer)
     4. {
     5. **if (ModelState.IsValid)**
     6. {

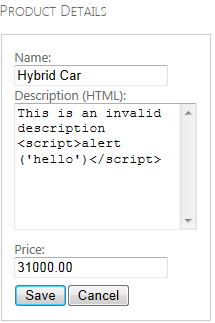
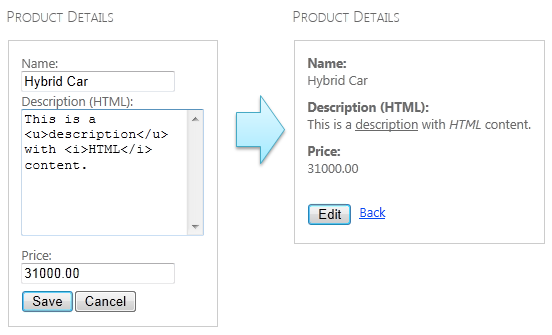
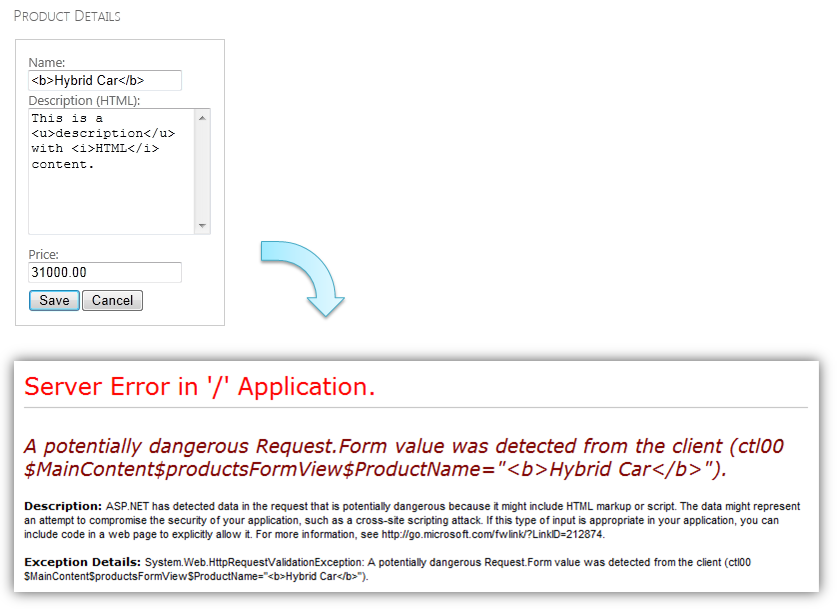
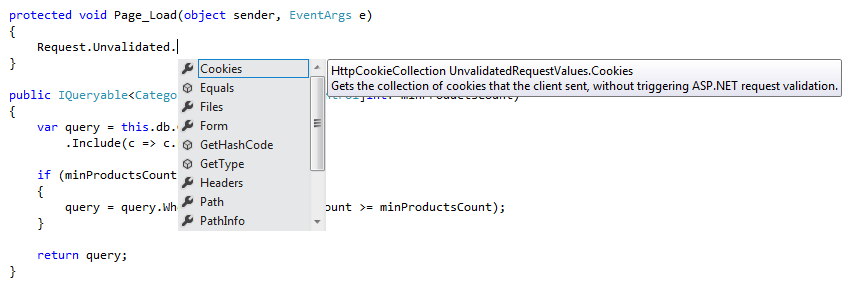
using (var db = new WebFormsLab.Model.ProductsContext())

* + 1. {
    2. ...
  1. Add a **ValidationSummary** control at the end of the CustomerDetails page to show the list of model errors.
     1. (Code Snippet – *Web Forms Lab - Ex02 – ValidationSummary*)
     2. HTML
     3. </fieldset>
     4. </InsertItemTemplate>
     5. </asp:FormView>
     6. **<asp:ValidationSummary runat="server" ShowModelStateErrors="true" ForeColor="Red" HeaderText="Please check the following errors:"/>**
     7. </asp:Content>
     8. The **ShowModelStateErrors** is a new property on the ValidationSummary control that when set to **true**, the control will show the errors from the ModelState dictionary. These errors come from the data annotations validation.
  2. Press **F5** to run the Web application. Complete the form with some erroneous values and click **Save** to execute validation. Notice the error summary at the bottom.
     1. 
     2. Figure 15
     3. Validation with Data Annotations

Task 3 – Handling Custom Database Errors with ModelState

* 1. In previous version of Web Forms, handling database errors such as a too long string or a unique key violation could involve throwing exceptions in your repository code and then handling the exceptions on your code-behind to display an error. A great amount of code is required to do something relatively simple.
  2. In Web Forms 4.5, the ModelState object can be used to display the errors on the page, either from your model or from the database, in a consistent manner.
  3. In this task, you will add code to properly handle database exceptions and show the appropriate message to the user using the ModelState object.
  4. While the application is still running, try to update the name of a category using a duplicated value.
     1. 
     2. Figure 16
     3. Updating a category with a duplicated name
     4. Notice that an exception is thrown due to the “unique” constraint of the **CategoryName** column.
     5. 
     6. Figure 17
     7. Exception for duplicated category names
  5. Stop debugging. In the **Products.aspx.cs** code-behind file, update the **UpdateCategory** method to handle the exceptions thrown by the db.SaveChanges() method call and add an error to the **ModelState** object.
     1. The new **TryUpdateModel** method updates the category object retrieved from the database using the form data provided by the user.
     2. (Code Snippet – *Web Forms Lab - Ex01 – UpdateCategory Handle Errors*)
     3. C#
     4. public void UpdateCategory(int categoryId)
     5. {
     6. var category = this.db.Categories.Find(categoryId);
     7. TryUpdateModel(category);
     8. if (ModelState.IsValid)
     9. {
     10. **try**
     11. **{**
     12. **this.db.SaveChanges();**
     13. **}**
     14. **catch (DbUpdateException)**
     15. **{**
     16. **ModelState.AddModelError("CategoryName", string.Format("A category with the name {0} already exists.", category.CategoryName));**
     17. **}**
     18. }
     19. }
     20. **Note:** Ideally, you would have to identify the cause of the DbUpdateException and check if the root cause is the violation of a unique key constraint.
  6. Open **Products.aspx** and add a **ValidationSummary** control below the categories GridView to show the list of model errors.
     1. (Code Snippet – *Web Forms Lab - Ex02 – Categories ValidationSummary*)
     2. HTML
     3. <asp:GridView ID="categoriesGrid" runat="server"
     4. ...
     5. </asp:GridView>
     7. **<asp:ValidationSummary ID="ValidationSummary1" runat="server" ShowModelStateErrors="true" />**
     8. <h3>Products</h3>
  7. Run the site and go to the Products page. Try to update the name of a category using an duplicated value.
     1. Notice that the exception was handled and the error message appears in the **ValidationSummary** control.
     2. 
     3. Figure 18
     4. Duplicated category error

Task 4 – Request Validation in ASP.NET Web Forms 4.5

* 1. The request validation feature in ASP.NET provides a certain level of default protection against cross-site scripting (XSS) attacks. In previous versions of ASP.NET, request validation was enabled by default and could only be disabled for an entire page. With the new version of ASP.NET Web Forms you can now disable the request validation for a single control, perform lazy request validation or access un-validated request data (be careful if you do so!).
  2. Press **Ctrl+F5** to start the site without debugging and go to the Products page. Select a category and then click the **Edit** link on any of the products.
  3. Type a description containing potentially dangerous content, for instance including HTML tags. Take notice of the exception thrown due to the request validation.
     1. 
     2. Figure 19
     3. Editing a product with potentially dangerous content
     4. 
     5. Figure 20
     6. Exception thrown due to request validation
  4. Close the page and, in Visual Studio, press **SHIFT+F5** to stop debugging.
  5. Open the **ProductDetails.aspx** page and locate the **Description** TextBox.
  6. Add the new **ValidateRequestMode** property to the TextBox and set its value to **Disabled**.
     1. The new **ValidateRequestMode** attribute allows you to disable the request validation granularly on each control. This is useful when you want to use an input that may receive HTML code, but want to keep the validation working for the rest of the page.
     2. HTML
     3. <li>
     4. <asp:TextBox runat="server" ID="Description" TextMode="MultiLine" Cols="60" Rows="8" Text='<%# BindItem.Description %>'
     5. **ValidateRequestMode="Disabled" />**
     6. </li>
  7. Press **F5** to run the web application. Open the edit product page again and complete a product description including HTML tags. Notice that you can now add HTML content to the description.
     1. 
     2. Figure 21
     3. Request validation disabled for the product description
     4. **Note:** In a production application, you should sanitize the HTML code entered by the user to make sure only safe HTML tags are entered (for example, there are no <script> tags). To do this, you can use [Microsoft Web Protection Library](http://wpl.codeplex.com/).
  8. Edit the product again. Type HTML code in the Name field and click **Save**. Notice that Request Validation is only disabled for the Description field and the rest of the fields re still validated against the potentially dangerous content.
     1. 
     2. Figure 22
     3. Request validation enabled in the rest of the fields
  9. Now open Web.config and check the **HttpRuntime** element, notice the new 4.5 validation mode.
     1. XML
     2. <system.web>
     3. ...
     4. **<httpRuntime requestValidationMode="4.5" encoderType="System.Web.Security.AntiXss.AntiXssEncoder, System.Web, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a" />**
     6. </system.web>
     7. ASP.NET Web Forms 4.5 includes a new request validation mode to perform request validation lazily. With the request validation mode set to **4.5**, if a piece of code accesses *Request.Form["key"]*, ASP.NET 4.5's request validation will only trigger request validation for that specific element in the form collection.
     8. Additionally, ASP.NET 4.5 now includes core encoding routines from the Microsoft Anti-XSS Library v4.0. The Anti-XSS encoding routines are implemented by the new *AntiXssEncoder* type found in the new *System.Web.Security.AntiXss* namespace. With the **encoderType** parameter configured to use *AntiXssEncoder*, all output encoding within ASP.NET automatically uses the new encoding routines.
  10. ASP.NET 4.5 request validation also supports un-validated access to request data. ASP.NET 4.5 adds a new collection property to the **HttpRequest** object called **Unvalidated**. When you navigate into **HttpRequest.Unvalidated** you have access to all of the common pieces of request data, including Forms, QueryStrings, Cookies, URLs, and so on.
      1. 
      2. Figure 23
      3. Request.Unvalidated object
      4. **Note:** **Please use the *HttpRequest.Unvalidated* property with caution!** Make sure you carefully perform custom validation on the raw request data to ensure that dangerous text is not round-tripped and rendered back to unsuspecting customers!

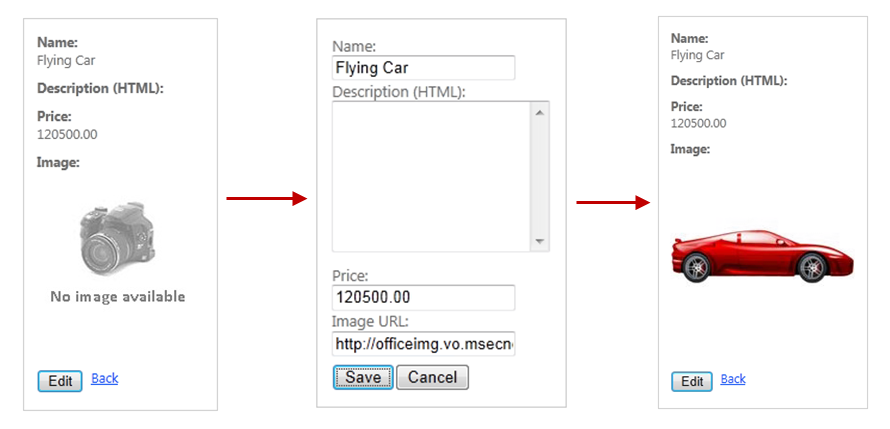
Exercise 3: Asynchronous Page Processing in ASP.NET Web Forms

* 1. In this exercise, you will be introduced to the new asynchronous page processing features in ASP.NET Web Forms.

Task 1 – Updating the Product Details Page to Upload and Show Images

* 1. In this task, you will update the product details page to allow the user to specify an image URL for the product and display it in the read-only view. You will create a local copy of the specified image by downloading it synchronously. In the next task, you will update this implementation to make it work asynchronously.
  2. Open **Visual Studio 11** and load the **WebFormsLab-Ex3-Begin.sln** solution located in **Source\Ex3-Async\Begin** from this lab’s folder. Alternatively, you can continue working on your existing solution from the previous exercises.
  3. Open the **ProductDetails.aspx** page source and add a field in the FormView’s ItemTemplate to show the product image.
     1. (Code Snippet – *Web Forms Lab - Ex03 – Show Product Image*)
     2. HTML
     3. <ItemTemplate>
     4. <fieldset>
     5. <ul>
     6. <li><b><asp:Label ID="Label2" runat="server" AssociatedControlID="itemProductName">Name:</asp:Label></b></li>
     7. <li><asp:Label runat="server" ID="itemProductName" Text='<%# Item.ProductName %>' /></li>
     8. <li><b><asp:Label ID="Label3" runat="server" AssociatedControlID="itemDescription">Description (HTML):</asp:Label></b></li>
     9. <li><asp:Label runat="server" ID="itemDescription" Text='<%# Item.Description %>' /></li>
     10. <li><b><asp:Label ID="Label4" runat="server" AssociatedControlID="itemUnitPrice">Price:</asp:Label></b></li>
     11. <li><asp:Label runat="server" ID="itemUnitPrice" Text='<%# Item.UnitPrice %>' /></li>
     12. **<li><b><asp:Label ID="Label5" runat="server" AssociatedControlID="itemUnitPrice">Image:</asp:Label></b></li>**
     13. **<li><img src="<%# string.IsNullOrEmpty(Item.ImagePath) ? "/Images/noimage.jpg" : Item.ImagePath %>" alt="Image" /></li>**
     14. </ul>
     15. <br />
     16. <ul>
     17. <li>
     18. <asp:Button ID="Button1" runat="server" CommandName="Edit" Text="Edit" />&nbsp;
     19. <asp:HyperLink NavigateUrl="~/Products.aspx" Text="Back" runat="server" />
     20. </li>
     21. </ul>
     22. </fieldset>
     23. </ItemTemplate>
  4. Add a field to specify the image URL in the FormView’s EditTemplate.
     1. (Code Snippet – *Web Forms Lab - Ex03 – Edit Image URL*)
     2. HTML
     3. <EditItemTemplate>
     4. <fieldset>
     5. <ul>
     6. <li><asp:Label ID="Label2" runat="server" AssociatedControlID="ProductName">Name:</asp:Label></li>
     7. <li><asp:TextBox runat="server" ID="ProductName" Text='<%# BindItem.ProductName %>' /></li>
     8. <li><asp:Label ID="Label3" runat="server" AssociatedControlID="Description">Description (HTML):</asp:Label></li>
     9. <li><asp:TextBox runat="server" ID="Description" TextMode="MultiLine" Cols="60" Rows="8" Text='<%# BindItem.Description %>' ValidateRequestMode="Disabled" />
     10. </li>
     11. <li><asp:Label ID="Label4" runat="server" AssociatedControlID="UnitPrice">Price:</asp:Label></li>
     12. <li><asp:TextBox runat="server" ID="UnitPrice" Text='<%# BindItem.UnitPrice %>' /></li>
     13. **<li><asp:Label ID="Label1" runat="server" AssociatedControlID="ImagePath">Image URL:</asp:Label></li>**
     14. **<li><asp:TextBox runat="server" ID="ImagePath" Text='<%#: BindItem.ImagePath %>' /></li>**
     15. </ul>
     16. <br />
     17. <ul>
     18. <li>
     19. <asp:Button runat="server" CommandName="Update" Text="Save" />
     20. <asp:Button runat="server" CommandName="Cancel" Text="Cancel" CausesValidation="false" />
     21. </li>
     22. </ul>
     23. </fieldset>
     24. </EditItemTemplate>
  5. Open the **ProductDetails.aspx.cs** code-behind file and create an **UpdateProductImage** method to store remote images in the local **Images** folder and update the product entity with the new image location value.
     1. (Code Snippet – *Web Forms Lab - Ex03 – UpdateProductImage*)
     2. C#
     3. **private void UpdateProductImage(Product product)**
     4. **{**
     5. **string imageUrl = product.ImagePath;**
     6. **if (!string.IsNullOrEmpty(imageUrl) && !VirtualPathUtility.IsAbsolute(imageUrl))**
     7. **{**
     8. **product.ImagePath = string.Format("/Images/{0}{1}", product.ProductId, Path.GetExtension(imageUrl));**
     9. **using (var wc = new WebClient())**
     10. **{**
     11. **wc.DownloadFile(imageUrl, Server.MapPath(product.ImagePath));**
     12. **}**
     13. **}**
     14. **}**
  6. Update the **UpdateProduct** method to call the **UpdateProductImage** method.
     1. (Code Snippet – *Web Forms Lab - Ex03 – UpdateProductImage Call*)
     2. C#
     3. public void UpdateProduct(int productId)
     4. {
     5. var product = this.db.Products.Find(productId);
     6. TryUpdateModel(product);
     7. **this.UpdateProductImage(product);**
     8. if (ModelState.IsValid)
     9. {
     10. this.db.SaveChanges();
     11. }
     12. }
  7. Add the following namespace directives.
     1. (Code Snippet – *Web Forms Lab - Ex03 – Namespaces*)
     2. C#
     3. **using System.Net;**

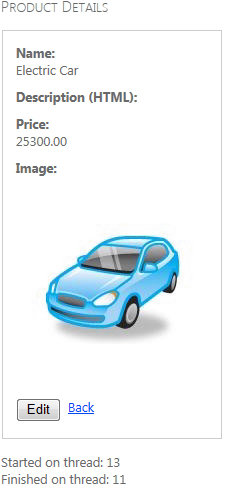
**using System.IO;**

* 1. Run the application and try to upload an image for a product. For example, you can use the following image URL from Office Clip Arts: <http://officeimg.vo.msecnd.net/en-us/images/MB900437099.jpg>
     1. 
     2. Figure 24
     3. Setting an image for a product

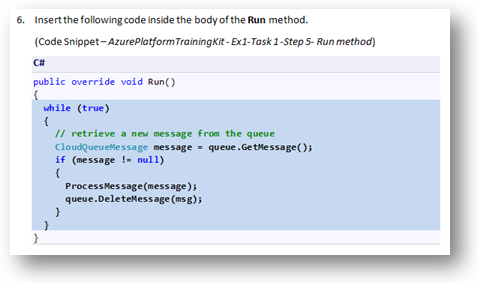
Task 2 – Adding Asynchronous Processing to the Product Details Page

* 1. In this task, you will update the product details page to make it work asynchronously. You will enhance a long running task – the image download process – by using ASP.NET 4.5 asynchronous page processing.
  2. Asynchronous methods in web applications can be used to optimize the way ASP.NET thread pools are used. In ASP.NET there are a limited number of threads in the thread pool for attending requests, thus, when all the threads are busy, ASP.NET starts to reject new requests, sends application error messages and makes your site unavailable.
  3. Time-consuming operations on your web site are great candidates for asynchronous programming because they occupy the assigned thread for a long time. This includes long running requests, pages with lots of different elements and pages that require offline operations, such querying a database or accessing an external web server. The advantage is that if you use asynchronous methods for these operations, while the page is processing, the thread is freed and returned to the thread pool and can be used to attend to a new page request. This means, the page will start processing in one thread from the thread pool and might complete processing in a different one, after the async processing completes.
  4. Open the **ProductDetails.aspx** page. Add the **Async** attribute in the **Page** element and set it to **true**. This attribute tells ASP.NET to implement the IHttpAsyncHandler interface.
     1. HTML
     2. <%@ Page Title="" Language="C#" MasterPageFile="~/Site.Master" AutoEventWireup="true"
     3. CodeBehind="ProductDetails.aspx.cs" Inherits="WebFormsLab.ProductDetails"
     4. **Async="true" %>**
  5. Add a Label at the bottom of the page to show the details of the threads running the page.
     1. (Code Snippet – *Web Forms Lab - Ex03 – Threads Message Label*)
     2. HTML
     3. <EmptyDataTemplate>Product not found</EmptyDataTemplate>
     4. </asp:FormView>
     5. **<asp:Label ID="threadsMessageLabel" runat="server" />**
     6. </asp:Content>
  6. Open up **ProductDetails.aspx.cs** and add the following namespace directive.
     1. (Code Snippet – *Web Forms Lab - Ex03 – Namespaces 2*)
     2. C#
     3. **using System.Threading;**
  7. Modify the **UpdateProductImage** method to download the image with an asynchronous task. You will replace the **WebClient** **DownloadFile** method with the **DownloadFileTaskAsync** method and include the **await** keyword.
     1. (Code Snippet – *Web Forms Lab - Ex03 – UpdateProductImage Async*)
     2. C#
     3. private void UpdateProductImage(Product product)
     4. {
     5. string imageUrl = product.ImagePath;
     6. if (!string.IsNullOrEmpty(imageUrl) && !VirtualPathUtility.IsAbsolute(imageUrl))
     7. {
     8. product.ImagePath = string.Format("/Images/{0}{1}", product.ProductId, Path.GetExtension(imageUrl));
     9. **RegisterAsyncTask(new PageAsyncTask(async(t) =>**
     10. **{**
     11. **using (var wc = new WebClient())**
     12. **{**
     13. **await wc.DownloadFileTaskAsync(imageUrl, Server.MapPath(product.ImagePath));**
     14. **}**
     15. **}));**
     16. }
     17. }
     18. The RegisterAsyncTask registers a new page asynchronous task to be executed in a different thread. It receives a lambda expression with the Task (t) to be executed.

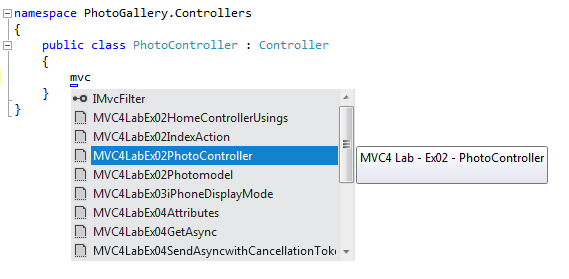
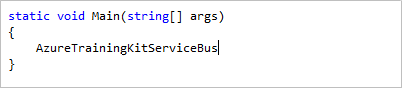
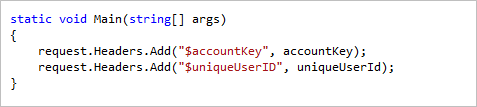
The **await** keyword in the **DownloadFileTaskAsync** method converts the remainder of the method into a callback that is invoked asynchronously after the **DownloadFileTaskAsync** method has completed. ASP.NET will resume the execution of the method by automatically maintaining all the HTTP request original values. The new asynchronous programming model in .NET 4.5 enables you to write asynchronous code that looks very much like synchronous code, and let the compiler handle the complications of callback functions or continuation code.

* + 1. **Note:** RegisterAsyncTask and PageAsyncTask were already available since .NET 2.0. The await keyword is new from the .NET 4.5 asynchronous programming model and can be used together with the new TaskAsync methods from the .NET WebClient object.
  1. Add code to display the threads on which the code started and finished executing.
     1. (Code Snippet – *Web Forms Lab - Ex03 – Show threads*)
     2. C#
     3. private void UpdateProductImage(Product product)
     4. {
     5. string imageUrl = product.ImagePath;
     6. if (!string.IsNullOrEmpty(imageUrl) && !VirtualPathUtility.IsAbsolute(imageUrl))
     7. {
     8. product.ImagePath = string.Format("/Images/{0}{1}", product.ProductId, Path.GetExtension(imageUrl));
     9. RegisterAsyncTask(new PageAsyncTask(async(t) =>
     10. {
     11. **var startThread = Thread.CurrentThread.ManagedThreadId;**
     12. **using (var wc = new WebClient())**
     13. **{**
     14. **await wc.DownloadFileTaskAsync(imageUrl, Server.MapPath(product.ImagePath));**
     15. **}**
     16. **var endThread = Thread.CurrentThread.ManagedThreadId;**
     17. **threadsMessageLabel.Text = string.Format("Started on thread: {0}<br /> Finished on thread: {1}", startThread, endThread);**
     18. }));
     19. }
     20. }
  2. Open the web site’s **web.config** file. Add the following appSetting variable.
     1. XML
     2. **<add key="aspnet:UseTaskFriendlySynchronizationContext" value="true"/>**
  3. Press **F5** to run the application and upload an image for the product. Notice the threads ID where the code started and finished may be different. This is because asynchronous tasks run on a separate thread from ASP.NET thread pool. When the task completes, ASP.NET puts the task back in the queue and assigns any of the available threads.
     1. 
     2. Figure 25
     3. Downloading an image asynchronously

Appendix: Using Code Snippets

* 1. With code snippets, you have all the code you need at your fingertips. The lab document will tell you exactly when you can use them, as shown in the following figure.
  2. 
  3. Figure 26
  4. Using Visual Studio code snippets to insert code into your project

#### To add a code snippet using the keyboard (C# only)

* 1. Place the cursor where you would like to insert the code.
  2. Start typing the snippet name (without spaces or hyphens).
  3. Watch as IntelliSense displays matching snippets' names.
  4. Select the correct snippet (or keep typing until the entire snippet's name is selected).
  5. Press the Tab key twice to insert the snippet at the cursor location.
  6. 
  7. Figure 27
  8. Start typing the snippet name
  9. 
  10. Figure 28
  11. Press Tab to select the highlighted snippet
  12. 
  13. Figure 29
  14. Press Tab again and the snippet will expand

#### To add a code snippet using the mouse (C#, Visual Basic and XML)

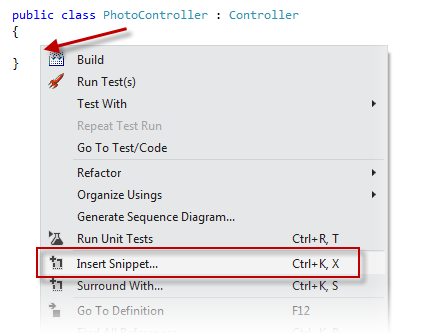
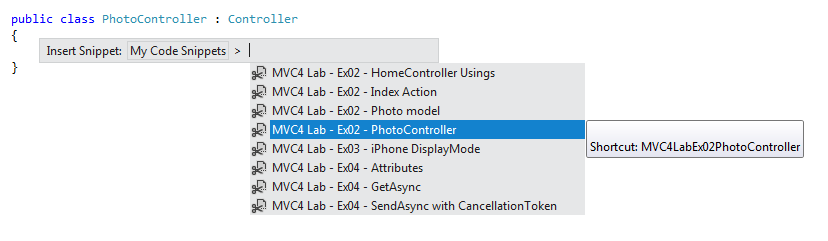
* 1. Right-click where you want to insert the code snippet.
  2. Select **Insert Snippet** followed by **My Code Snippets**.
  3. Pick the relevant snippet from the list, by clicking on it.
  4. 
  5. Figure 30
  6. Right-click where you want to insert the code snippet and select Insert Snippet
  7. 

Figure 31

* 1. Pick the relevant snippet from the list, by clicking on it