

Lesson 26: Northwind AJAX POST

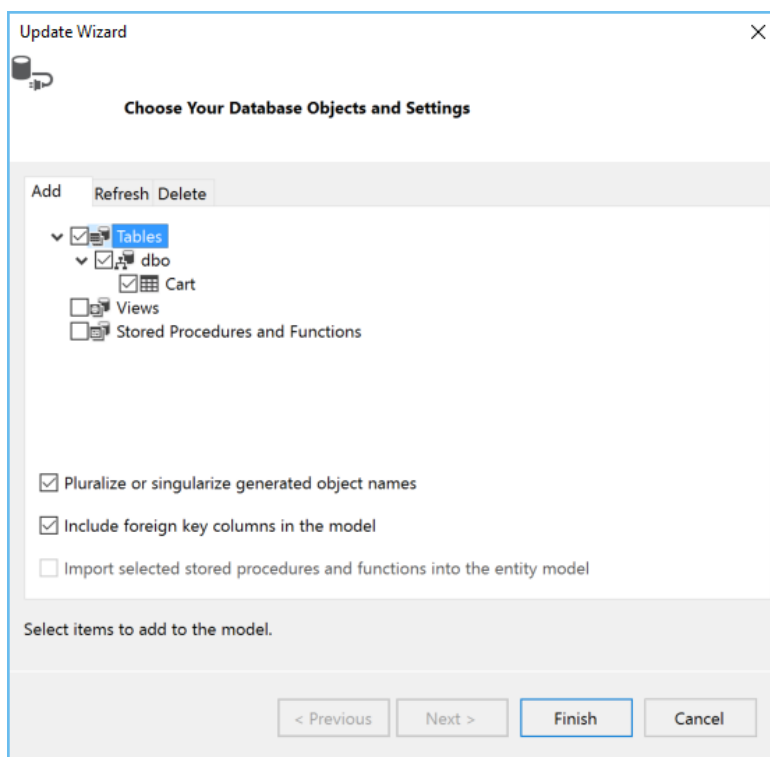
In this lesson, you will continue the development of the Northwind Store application by implementing an AJAX POST request. AJAX is short for Asynchronous Javascript And XML. AJAX is used to process GET or POST requests without the usual page refresh. In other words, we use Javascript to send GET or POST requests. Today we will use AJAX to add products to a customer's shopping cart without the need for a page refresh. The customer's shopping cart data will be stored in the database.

Agenda

1. Demonstration
2. Lab/Homework

Demonstration

1. Start by opening the **Northwind** project in Visual Studio.
2. We need to make some schema changes to our database for this example. We are going to add a new table to store the customer's cart items.
3. Using the Server Explorer, open the **Create_Cart.sql** file and execute it on the Northwnd database.
4. Refresh the Tables folder and notice the new table. The cart table is simple. It contains 4 fields:
 - a. **CartId** – (int primary key)
 - b. **ProductID** – (int foreign key) product in cart
 - c. **CustomerID** – (int foreign key) customer who the cart belongs to
 - d. **Quantity** – (int) product quantity in cart
5. Since we have modified our database schema, we need to update our data model.
6. Open the **Models\NorthwindModel.edmx**.
7. Right-click the canvas and select **Update Model from Database**.
8. In the Update Wizard, choose **Add – Tables - dbo - Cart**. Click Finish.



9. Save the Model. This is a good time to build and test your project.

10. Open the **Views\Product\Product** view.
11. Add a Bootstrap modal window to the view. The modal window markup can be placed anywhere. I put mine right after the sticky footer.

```
<!-- Modal window -->
<div id="myModal" class="modal fade" tabindex="-1" role="dialog">
  <div class="modal-dialog">
    <div class="modal-content">
      <div class="modal-header">
        <button type="button" class="close" data-dismiss="modal" aria-label="Close">
          <span aria-hidden="true">&times;</span>
        </button>
        <h4 class="modal-title">Add to Cart?</h4>
      </div>
      <div class="modal-body">
        Modal Body
      </div>
      <div class="modal-footer">
        <button type="button" class="btn btn-danger" data-dismiss="modal">
          Cancel
        </button>
        <button type="button" class="btn btn-primary" id="AddToCart">
          Continue
        </button>
      </div>
    </div><!-- /.modal-content -->
  </div><!-- /.modal-dialog -->
</div><!-- /.modal -->
```

12. Modify the **.product-row** event listener. Open the modal window when a row is clicked.

```
// product-row is clicked
$('#products').on('click', '.product-row', function () {
  //alert(this.id);
  $('#myModal').modal();
});
```

13. Test in browser. The modal window appears when a product-row is clicked.
14. What we really want displayed in the modal window is a form that allows the customer to select the quantity (see below).

Add to Cart?

Product	Price	Quantity	Total
Flotemysost	\$21.50	<input type="text" value="1"/>	\$21.50

Cancel

Continue

15. Modify the modal-body.

```
<div class="modal-body">
  <div class="row">
    <div class="col-xs-5"><strong>Product</strong></div>
    <div class="col-xs-2"><strong>Price</strong></div>
    <div class="col-xs-2"><strong>Quantity</strong></div>
    <div class="col-xs-3"><strong>Total</strong></div>
  </div>
  <div class="row">
    <div class="col-xs-5" id="ProductName">Cool Product</div>
    <div class="col-xs-2">
      $<span id="UnitPrice">15.00</span>
    </div>
    <div class="col-xs-2">
      <input type="number" min="1" value="1" id="Quantity" class="form-control" />
    </div>
    <div class="col-xs-3">
      $<span id="Total">15.00</span>
    </div>
  </div>
</div>
```

16. Test in browser. It doesn't look quite right. Let's enhance with some CSS.

17. Add the **.pad-top** class selector to the **Content\Site.css**.

```
.pad-top{
  padding-top:6px;
}
```

18. Apply the pad-top and text-right CSS classes where needed.

```
<div class="modal-body">
  <div class="row">
    <div class="col-xs-5"><strong>Product</strong></div>
    <div class="col-xs-2"><strong>Price</strong></div>
    <div class="col-xs-2"><strong>Quantity</strong></div>
    <div class="col-xs-3 text-right"><strong>Total</strong></div>
  </div>
  <div class="row">
    <div class="col-xs-5 pad-top" id="ProductName">Cool Product</div>
    <div class="col-xs-2 pad-top">
      $<span id="UnitPrice">15.00</span>
    </div>
    <div class="col-xs-2">
      <input type="number" min="1" value="1" id="Quantity" class="form-control" />
    </div>
    <div class="col-xs-3 pad-top text-right">
      $<span id="Total">15.00</span>
    </div>
  </div>
</div>
```

19. Let's add the Javascript to calculate and display the total price (product price & quantity). Add the Quantity event listener.

```
// update total when cart quantity is changed
$('#Quantity').change(function () {
    var total = parseInt($(this).val()) * parseFloat($('#UnitPrice').html());
    $('#Total').html(total.toFixed(2));
});
```

20. Test in browser. It's pretty good, notice the total is display to decimals. It would be nice to format the string with commas for thousands.
21. Add this function that regex to format a number with commas.

```
// function to display commas in number
function numberWithCommas(x) {
    return x.toString().replace(/\B(?=(\d{3})+(?!\d))/g, ",");
}
```

22. Update the total.

```
$('#Total').html(numberWithCommas(total.toFixed(2)));
```

23. Test in browser. Enter quantity > 66.
24. Now we need to display the selected product's actual name and price in the modal window.
25. Modify the table's tbody.

```
<tbody id="products">
    @foreach (Product p in Model)
    {
        <tr class="product-row" id="@p.ProductID">
            <td><span id="name_@p.ProductID">@p.ProductName</span> (@p.QuantityPerUnit)</td>
            <td class="text-right">
                $<span id="price_@p.ProductID">@string.Format("{0:n2}", p.UnitPrice)</span>
            </td>
            <td class="text-right">@p.UnitsInStock</td>
        </tr>
    }
</tbody>
```

26. We've added span elements and id attributes using the ProductID. When a product-row is selected, we can use the ProductID to retrieve the ProductName and Price.
27. View in browser. Examine the source code. . Notice the id attribute of the product name and price span.

```
...
<tr class="product-row" id="1">
    <td><span id="name_1">Chai</span> (10 boxes x 20 bags)</td>
    <td class="text-right">$<span id="price_1">18.00</span></td>
    <td class="text-right">39</td>
</tr>
...
```

28. Create a global variable to store the selected ProductID. Then, modify the product-row event listener.

```
// global variable to store selected ProductID
var ProductID;
// product-row is clicked
```

```

$('#products').on('click', '.product-row', function () {
    ProductID = this.id;
    // display selected product's name & price in modal
    $('#ProductName').html($('#name_' + ProductID).html());
    $('#UnitPrice').html($('#price_' + ProductID).html());
    // set product quantity = 1
    $('#Quantity').val(1);
    // calculate and display total in modal
    $('#Quantity').change();
    // display modal
    $('#myModal').modal();
});

```

29. Test in browser. Try filtering the product list by price. Test with several products. What happened?
30. Remember that we are re-creating the tbody of our table with the results of our AJAX GET request. We need to modify that code so we can accurately retrieve the Product Name & Price.

```

for (i = 0; i < data.length; i++) {
    var id = data[i].ProductID;
    var str = "<tr class='product-row' id='" + id + "'>";
    str += "<td><span id='name_" + id + "'>" + data[i].ProductName + "</span> (" +
        data[i].QuantityPerUnit + "</td>";
    str += "<td class='text-right'>$<span id='price_" + id + "'>" +
        data[i].UnitPrice.toFixed(2) + "</span></td>";
    str += "<td class='text-right'>" + data[i].UnitsInStock + "</td>";
    str += "</tr>";
    $('#products').append(str);
}

```

31. Test in browser. It should be working now.
32. That's it for the view for now.
33. When we submit our AJAX POST request, we need to send the ProductID, CustomerID, and quantity with the request. We will send this data as a Json object.
34. Create a new model class, CartDTO in the Models folder.

```

public class CartDTO
{
    public int ProductID { get; set; }
    public int CustomerID { get; set; }
    public int Quantity { get; set; }
}

```

35. Create a new controller named **Cart**.
36. Import the **Northwind.Models** namespace.

```
using Northwind.Models;
```

37. Create the **AddToCart** controller method.

```

// POST: Cart/AddToCart
[HttpPost]
public JsonResult AddToCart(CartDTO cartDTO)
{
    if (!ModelState.IsValid)
    {
        Response.StatusCode = 400;
    }
}

```

```

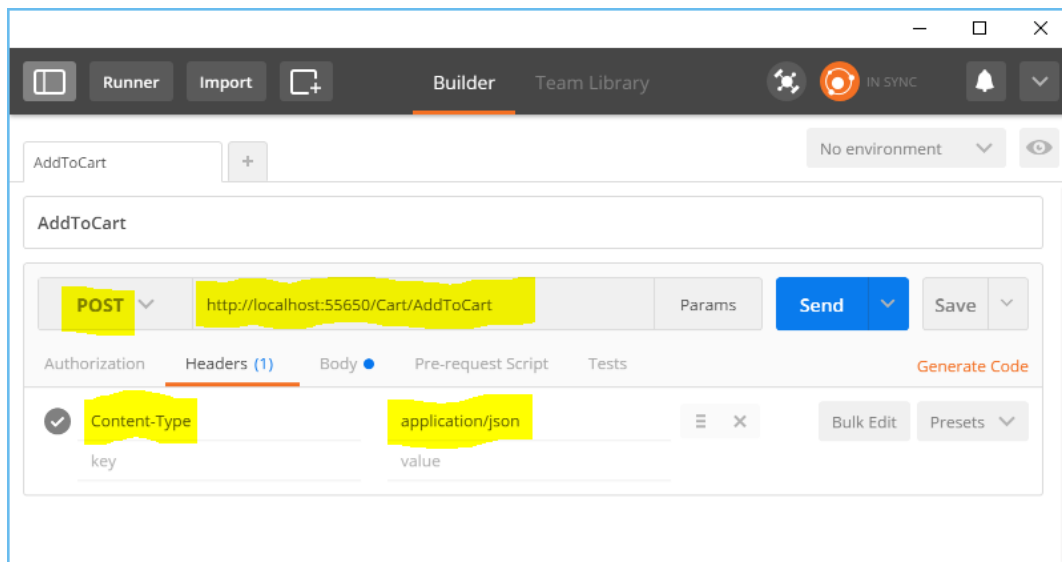
    return Json(new { }, JsonRequestBehavior.AllowGet);
}

// create cart item from Json object
Cart sc = new Cart();
sc.ProductID = cartDTO.ProductID;
sc.CustomerID = cartDTO.CustomerID;
sc.Quantity = cartDTO.Quantity;

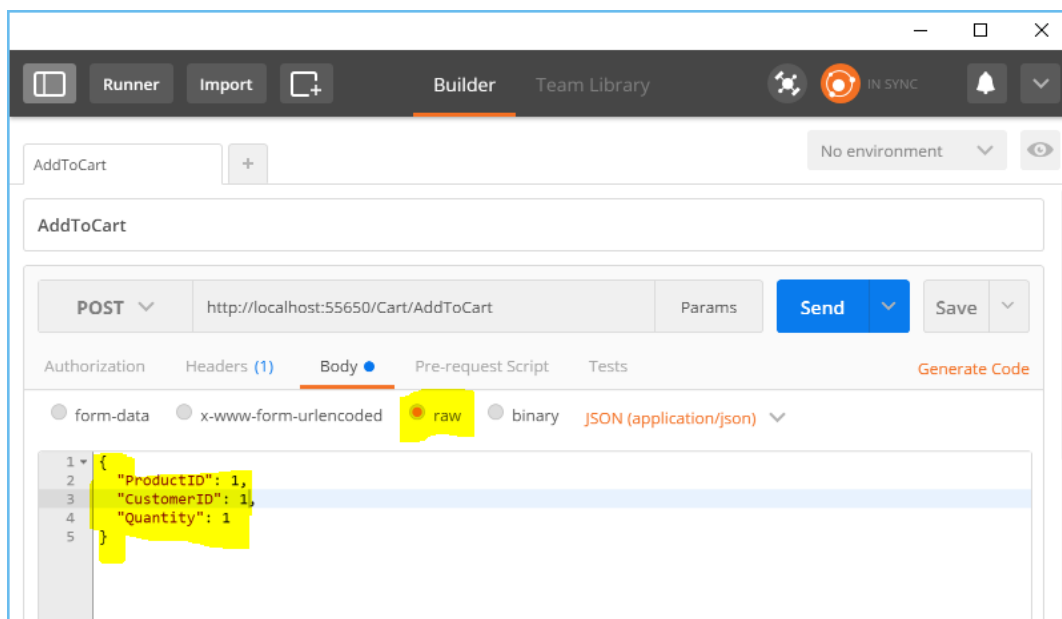
return Json(sc, JsonRequestBehavior.AllowGet);
}

```

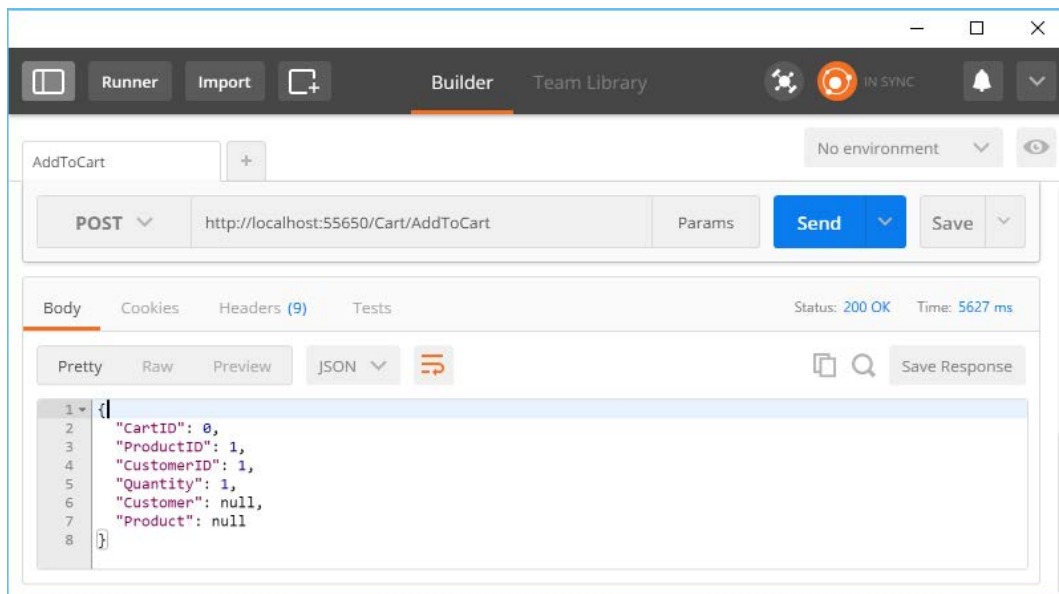
38. Let's test the POST request in **POSTMan**. Configure the **POST** request. Add the **Content-Type: application/json** header to the request.



39. Click the **Body** Tab. Choose the **Raw** option and add the raw Json to the request.



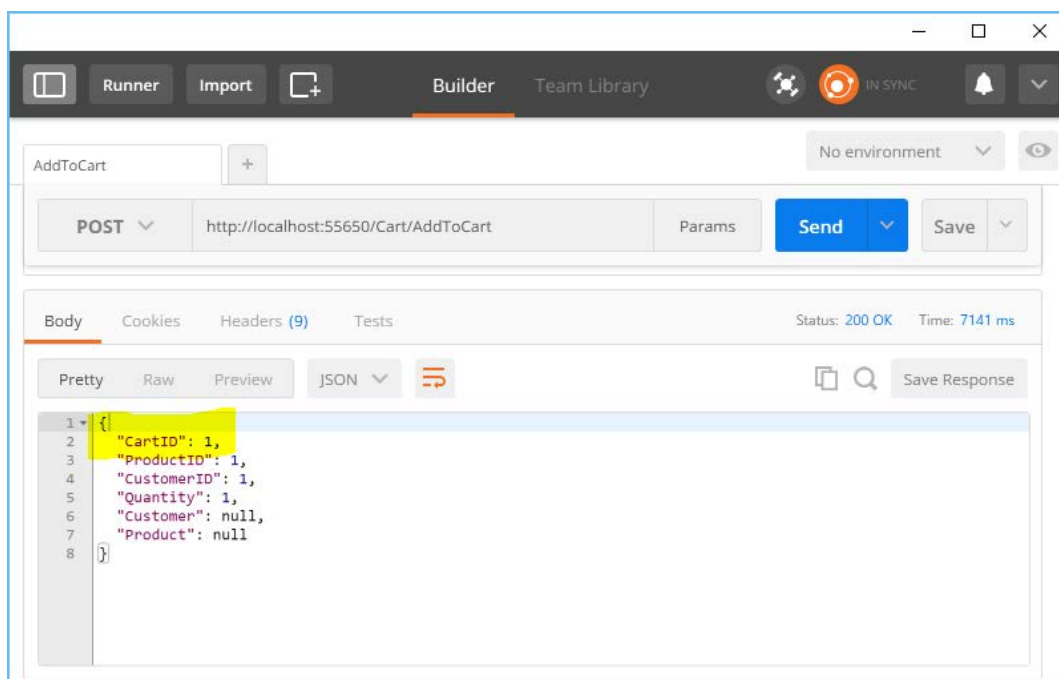
40. Test the request by clicking the Send button. We are hoping for response code 200.



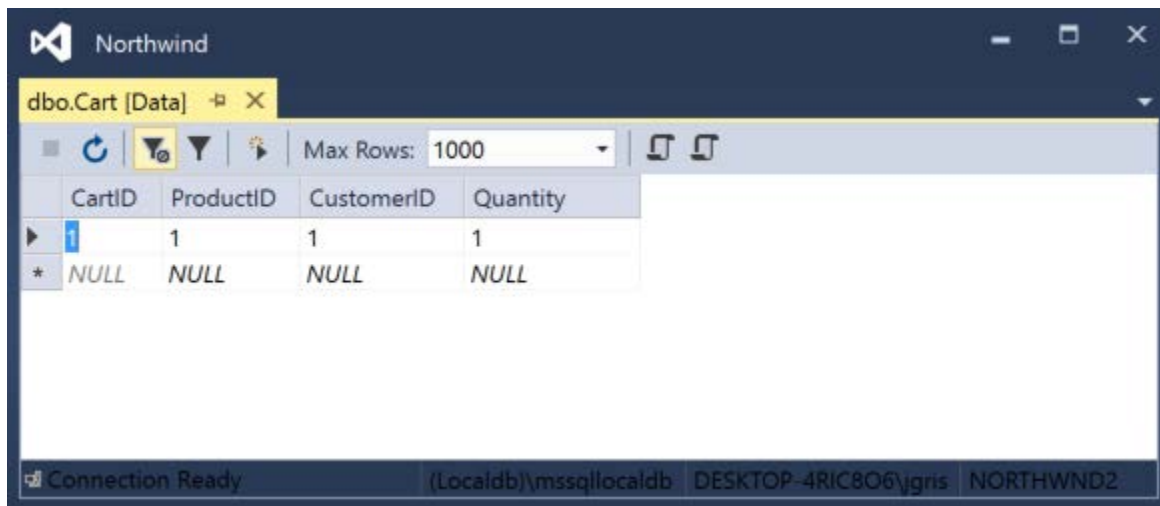
41. Adding the item to the database is fairly simple now. Modify the **AddToCart** method. Add the **using** statement right before the return statement.

```
using (NORTHWNDEntities db = new NORTHWNDEntities())
{
    // add the product to the customer's cart
    db.Carts.Add(sc);
    db.SaveChanges();
}
```

42. Test in **POSTMan** (ONE TEST ONLY!). Notice the **CartID** of the newly added record.



43. Using Visual Studio's **Server Explorer**, check the **Cart** table for the new record.



	CartID	ProductID	CustomerID	Quantity
		1	1	1
*	NULL	NULL	NULL	NULL

44. What do you think will happen if we format the exact same request? **DO NOT TEST IT!**

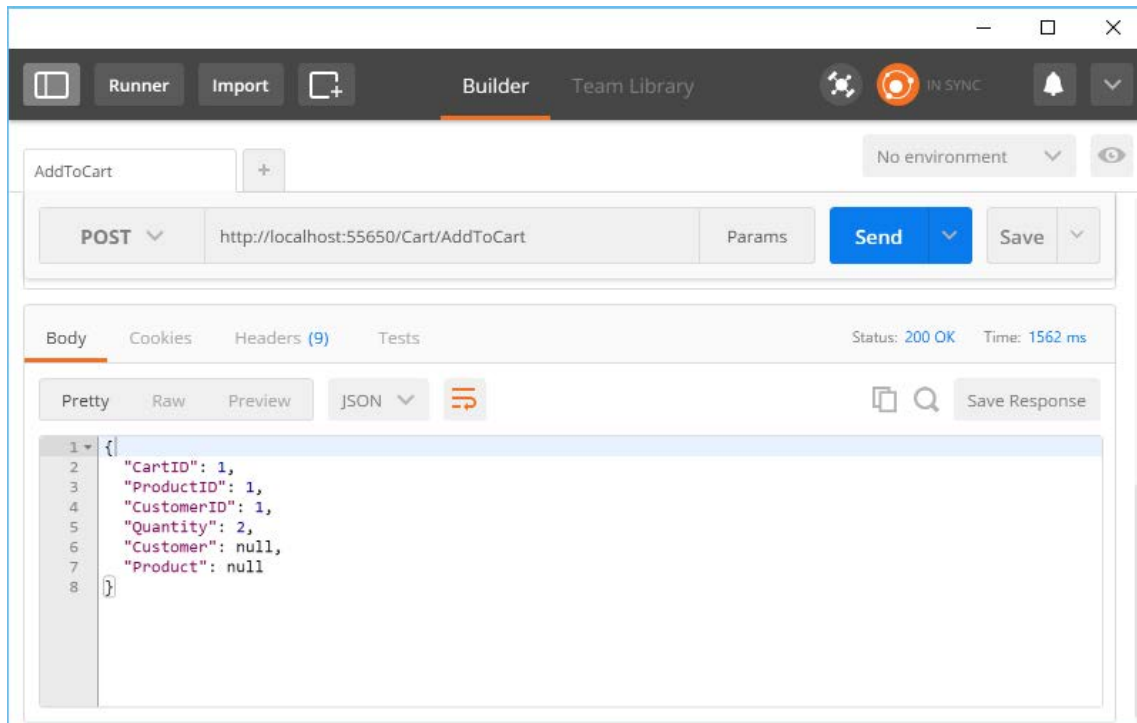
45. We don't want a new record created for the same Customer and Product in the cart. How can we fix this? All we need to do is update the quantity.

46. Modify the **AddToCart using** statement.

```
using (NORTHWNDEntities db = new NORTHWNDEntities())
{
    // if there is a duplicate product id in cart, simply update the quantity
    if (db.Carts.Where(c => c.ProductID == sc.ProductID && c.CustomerID ==
sc.CustomerID).Any())
    {
        // this product is already in the customer's cart,
        // update the existing cart item's quantity
        Cart cart = db.Carts.Where(c => c.ProductID == sc.ProductID && c.CustomerID ==
sc.CustomerID).FirstOrDefault();
        cart.Quantity += sc.Quantity;
        sc = new Cart(){
            CartID = cart.CartID,
            ProductID = cart.ProductID,
            CustomerID = cart.CustomerID,
            Quantity = cart.Quantity
        };
    }
    else
    {
        // this product is not in the customer's cart, add the product
        db.Carts.Add(sc);
    }

    db.SaveChanges();
}
```


47. Test the same request in **POSTMan**. Notice the updated record (quantity).



48. Using Visual Studio's **Server Explorer**, check the **Cart** table for the updated record.

49. Now we can update our view to send the AJAX POST request to add the item to the customer's cart.

50. Open the **Views\Product\Product** view.

51. Attach an event listener to the modal window's **AddToCart** button.

```
// attach event listener to modal window's update button
$('#AddToCart').click(function(){
    alert(ProductID + "|" + $('#Quantity').val());
});
```

52. Test in browser.

53. Remember, we need the **ProductID**, **CustomerID**, and the **Quantity** to format our POST request. The **ProductID** is saved to a global javascript variable, the **Quantity** is stored in a form field, however, the **CustomerID** is a bit trickier.

54. Since we are allowing anonymous browsing, there is no guarantee that we even have a **CustomerID**.

55. We need to verify the **CustomerID**. Modify the Razor block near the top of the view. Create a variable that stores the **CustomerID** for authenticated requests, or -1 for unauthenticated requests.

```
@{
    ViewBag.Title = "Product";
    int CustomerID = Request.IsAuthenticated ? Convert.ToInt32(User.Identity.Name) : -1;
}
```

56. Modify the event listener that opens the modal window. Add a decision to verify the CustomerID.

```
// product-row is clicked
$('#products').on('click', '.product-row', function () {
    // handle error if customer is not authenticated
    if(@CustomerID == -1){
        // display error
        alert("No ID");
    } else {
        // show modal window
        ProductID = this.id;
        // display selected product's name & price in modal
        $('#ProductName').html($('#name_' + ProductID).html());
        $('#UnitPrice').html($('#price_' + ProductID).html());
        // set product quantity = 1
        $('#Quantity').val(1);
        // calculate and display total in modal
        $('#Quantity').change();
        // display modal
        $('#myModal').modal();
    }
});
```

57. Test in browser.

58. We now have all of the data we need to format our AJAX POST request, however, we can improve the error message displayed to the user – Javascript alerts are for amateurs!

59. Let's use the **jBox** plugin for jQuery - <http://plugins.jquery.com/jBox/>. We are going to use jBox notices.

60. Add the **jBox.min.js** file to the **Scripts** folder, and the **jBox.css** file to the **Content** folder.

61. Add the **link element** to your view. I added it right before the **scripts** section.

```
<link href="~/Content/jBox.css" rel="stylesheet">
```

62. Add the **script element** to the **scripts** section of your view.

```
<script src="~/Scripts/jBox.min.js"></script>
```

63. Adding a **notice** is simple. Replace the **Javascript alert** with a new **jBox notice**.

```
new jBox('Notice', {
    content: 'Sign in to access your cart'
});
```

64. Test in browser. You can close the notice by clicking it.

65. Of course, the **jBox** can be customized. Let's have it **autoClose** after 2 seconds and set the **color** to **red** (default color is black).

```
new jBox('Notice', {
    content: 'Sign in to access your cart',
    autoClose: 2000, // time in milliseconds
    color: 'red', // black, red, green, blue, yellow
});
```

66. Test in browser. Click a product-row multiple times quickly. The notices are stacked by default. Let's over-ride that behavior.

```
stack: false // true, false
```

67. Test in browser.
68. There is an option to close the Notice when the escape key is pressed. Let's add that option.

```
closeOnEsc: true
```

69. Test in browser.
70. That's looking pretty good. We can also change the **x,y attributes** of the notice. We can use precise pixel locations or named locations (**top, bottom, right, left**). The default is top, right. In addition, we can attach to an existing element and position relative to it.

```
target: $('#' + this.id),
position: {
  x: 'left',
  y: 65
},
```

71. Test in browser.
72. Let's move on to the AJAX POST request. When we send an AJAX request, we will need to wait for the response. The wait time depends on a number of variables that are generally outside of our control. There should be some feedback to the user that we are waiting... for a response.
73. Modify the **AddToCart** event listener. Let's display a **jBox** Notice. When the AJAX POST request is sent, we will show the notice. When the response is received, we will hide it.

```
$('#AddToCart').click(function(){
  // hide modal
  $('#myModal').modal('hide');
  // display loading notice
  var loadingNotice = new jBox('Notice', {
    content: 'Please wait...',
    autoClose: false,
    closeOnClick: false,
    color: 'blue',
    overlay: true,
    target: $('#' + ProductID),
    position: { x: 'left', y: 65 }
  });
  loadingNotice.open();
  // AJAX POST Request
});
```

74. Test in browser. You need to sign in as a customer and select a product-row. Notice the overlay.
75. It's time to format the AJAX POST request.

```
// AJAX POST Request
var URL = "@Url.Content("~/")Cart/AddToCart";
$.post( URL, { ProductID: ProductID, CustomerID: @CustomerID, Quantity:
$('#Quantity').val() })
.always(function( data, textStatus, statusObject ) {
  loadingNotice.close();
  if(statusObject.status == 200){
    // success
    console.log($('#name_' + ProductID).html() + ' added to cart')
  } else {
    // error
    console.log("Http response code: " + data.status);
    console.log("Http response: " + data.statusText);
  }
});
```

```
    }  
});
```

76. Before we test this in the browser, let's make a minor change to the **AddToCart** controller method. Open the **Controller\Cart** controller. Comment out the line that commits the database changes and simulate a network delay.

```
//db.SaveChanges();  
// simulate network delay  
System.Threading.Thread.Sleep(1500);
```

77. Build your solution and test in browser.

78. Let's add a **jBox notice** when the update has successfully completed.

```
// success  
new jBox('Notice', {  
    content: $('#name_' + ProductID).html() + ' added to cart',  
    autoClose: 2000,  
    color: 'green',  
    closeOnEsc: true,  
    target: $('#' + ProductID),  
    position: { x: 'left', y: 65 }  
});
```

79. Test in browser.

80. Open the **Controller\Cart** controller. Uncomment out the line that commits the database changes and remove the simulated network delay.

```
db.SaveChanges();  
// simulate network delay  
// System.Threading.Thread.Sleep(1500);
```

81. Rebuild the solution and test in browser.