

# INFO3067 Week 3 Class 1

## Review

- Register
- AspNet Identity

## Session Theory

The good explanation on Session objects can be found in the following article:  
<http://www.codeproject.com/Articles/32545/Exploring-Session-in-ASP-Net>

For this course, read the article down to the heading of: **StateServer Session Mode**

data is easily available.

- There is not requirement of serialization to store data in InProc session mode.
- Implementation is very easy, similar to using the ViewState.

### Disadvantages:

Although InProc session is the fastest, common, and default mechanism, it has a lot of limitations:

- If the worker process or application domain is recycled, all session data will be lost.
- Though it is the fastest, more session data and more users can affect performance, because of memory usage.
- We can't use it in web garden scenarios.
- This session mode is not suitable for web farm scenarios.

As per the above discussion, we can conclude that InProc is a very fast session storing mechanism but suitable only for small web applications. InProc session data will get lost if we restart the server or recycle the application. It is also not suitable for Web Farm and Web Garden scenarios.

Now we will have a look the other options available to overcome these problems.

StateServer Session Mode

### Overview of StateServer session mode

Midterm will cover material up to here:

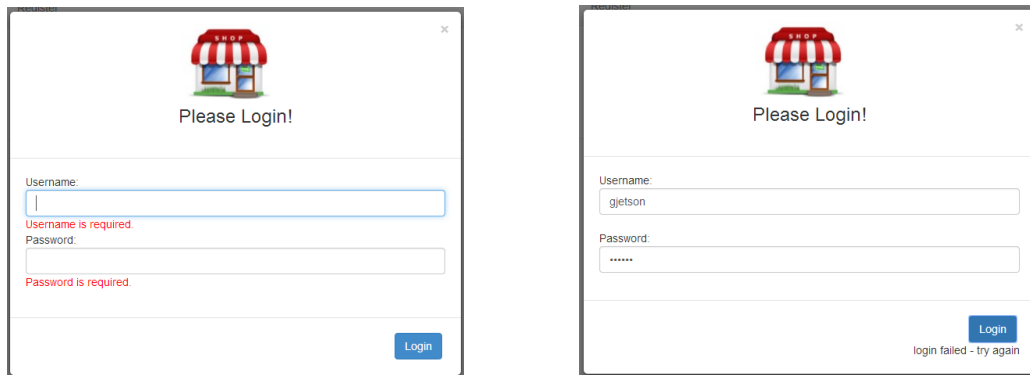
## Lab Part – A

Then as part of the lab answer the following 5 questions on the Session Class

1. What is used to distinguish one session from another?
2. Which Session State provider is used by default?
3. What is the main advantage to using a Session object?
4. What is the main disadvantage?
5. In which file do you code the 2 event handlers used by Session objects?

## Case Study cont'd - Login

With the Register process complete, we can now direct our attention to the **Login** process. Most sites have content that they only want members to view, so members are asked to login before they are allowed to proceed. Fortunately with our current infrastructure this is fairly easy to set up. Starting with your Home View, we can add another modal to the Home View to handle the login process:



The image displays two side-by-side screenshots of a login modal window. Both windows have a title bar with a close button (X) and a small icon of a shop with a red and white striped awning. The text 'Please Login!' is centered at the top of each window.

The left window shows the initial login form. It has two input fields: 'Username:' and 'Password:'. The 'Username' field is empty, and the 'Password' field is also empty. Below the 'Username' field, there is a red error message: 'Username is required.' Below the 'Password' field, there is a red error message: 'Password is required.' At the bottom right of the window, there is a blue 'Login' button.

The right window shows the login form after a failed attempt. The 'Username' field is now filled with the text 'gjetson'. The 'Password' field is filled with dots (masked). Below the 'Password' field, there is a red error message: 'login failed - try again'. At the bottom right of the window, there is a blue 'Login' button.

The controller code to handle the login is fairly trivial as the actual method is provided by the identity infrastructure (from last week) and the code is similar to the register process. We'll also store some information **in Session variables** so that we can use it on subsequent pages. A session variable is stored by the system on the server and makes the data available to all parts of the "Session" but it disallows other users (or sessions) access to the variables.

We'll start using Session variables in our Home Controller's Index method. Here we will initialize a variable to display if the user is not logged in yet, and another that we can use on the Home view:

```
U references
public ActionResult Index()
{
    if (Session["Message"] == null)
    {
        Session["Message"] = "Please Login First";
    }

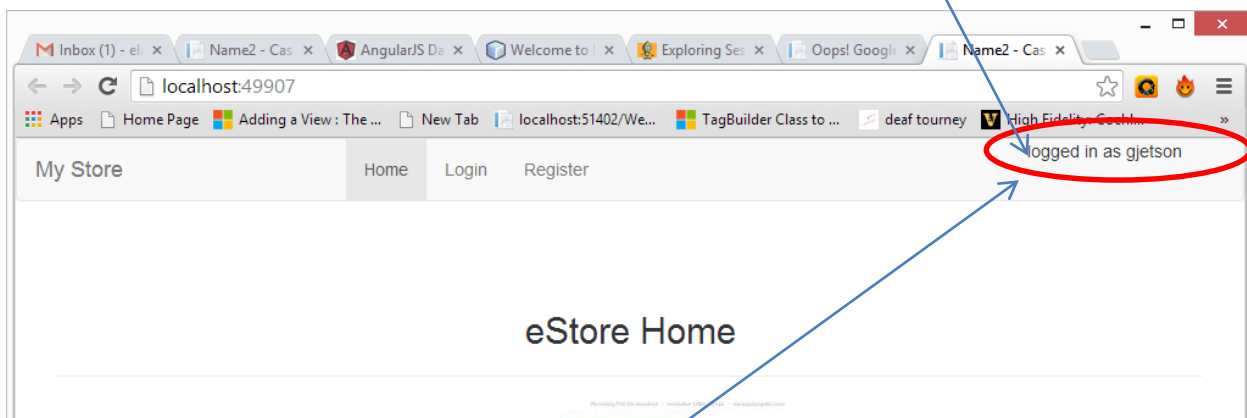
    if (Session["LoginStatus"] == null)
    {
        Session["LoginStatus"] = "not logged in";
    }

    return View();
}
```

Next we code a **Login** method:

```
[HttpPost]
[AllowAnonymous]
[ValidateAntiForgeryToken]
public async Task<ActionResult> Login(CustomerViewModel model, string returnUrl)
{
    var user = await UserManager.FindAsync(model.Username, model.Password);
    if (user != null)
    {
        await SignInAsync(user, false);
        model.GetCurrentProfile();
        Session["CustomerID"] = model.CustomerID;
        Session["Message"] = "Welcome " + model.Username;
        Session["LoginStatus"] = "logged in as " + model.Username;
        return Json(new { url = Url.Action("") }, JsonRequestBehavior.AllowGet);
    }
    else
    {
        ViewBag.Message = "login failed - try again";
        return PartialView("PopupMessage");
    }
}
```

Notice that the code still uses an instance of the CustomerViewModel, and it retrieves the current profile if the login was successful (you will need to write this method and the model method to retrieve a customer row based on username). If it was successful we return still return an ActionResult, but this time it is a JSON result. We're basically telling the page to return to itself (because the Url.Action is empty. This is used to close the popup instead of placing a message in the popup. We'll also like to indicate to the user whether or not they are logged on the menu line:



This is accomplished by placing the following markup in the master layout:

```
<ul class="nav pull-right">
    <li>@Session["LoginStatus"]</li>
</ul>
```

Again, if they are logged in successfully we return a JSON result with an empty URL., remove the popup and go back to the regular home page. For this to work we need to

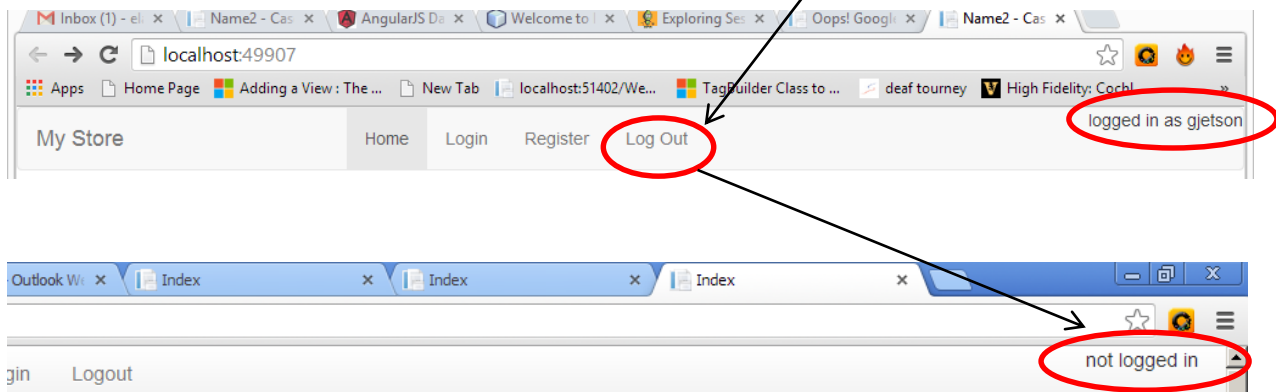
add a parameter (**OnSuccess**) to the Ajax form and have it execute a little Javascript in our login popup (you can add it to the Register popup as well):

```
@using (
    Ajax.BeginForm("Login", "Home", new AjaxOptions
    {
        InsertionMode = InsertionMode.Replace,
        HttpMethod = "POST",
        LoadingElementId = "ajaxSplash",
        UpdateTargetId = "messg",
        OnSuccess = "onSuccess"
    })
{
    <div id="register_popup" class="modal">...</div>
    <div class="modal" id="login_popup">
    <script type="text/javascript">
        var onSuccess = function (result) {
            if (result.url) {
                // if the server returned a JSON object containing an url
                // property we redirect the browser to that url
                window.location.href = result.url;
            }
        }
    </script>
}
```

## Case Study cont'd - Logout

We'll add another option to our menu to let the user logout:

```
<li id="lgOffMnuItm"><a href="#">Log Out</a></li>
</ul>
<ul class="nav pull-right loginStatus">
    <li>@Session["LoginStatus"]</li>
</ul>
@using (Html.BeginForm("LogOff", "Home", FormMethod.Post, new { id = "logoutForm" }))
{
    @Html.AntiForgeryToken()
}
```



We'll use the following code in the Home controller to logout. Notice that we use the AuthenticationManager class and we clear out any session information with the built in Session object's Abandon method:

```
//
// POST: /Home/LogOff
[HttpPost]
[ValidateAntiForgeryToken]
0 references
public ActionResult LogOff()
{
    AuthenticationManager.SignOut();
    Session.Abandon();
    return RedirectToAction("Index", "Home");
}
```

## Case Study cont'd - Shop

Once logged on we'll let the user do some shopping, we'll need to provide a list of products. You'll need to load a minimum of 12-15 products into your products table. Remember, the products table has the following layout from the SQL we ran in week 1:

|   | Column Name    | Data Type      | Allow Nulls                         |
|---|----------------|----------------|-------------------------------------|
| ? | ProductID      | nvarchar(15)   | <input type="checkbox"/>            |
|   | ProductName    | nvarchar(50)   | <input type="checkbox"/>            |
|   | GraphicName    | nvarchar(20)   | <input type="checkbox"/>            |
|   | CostPrice      | money          | <input type="checkbox"/>            |
|   | MSRP           | money          | <input type="checkbox"/>            |
|   | QtyOnHand      | int            | <input type="checkbox"/>            |
|   | QtyOnBackOrder | int            | <input type="checkbox"/>            |
|   | Description    | nvarchar(2000) | <input type="checkbox"/>            |
|   | Timer          | timestamp      | <input checked="" type="checkbox"/> |

Notice that there is a column called **GraphicName**. We won't store the graphic as a blob in this class, frankly it's just plain inefficient. Instead we'll store the name of the graphic as a string. Also, in the description please provide some text that **is at least a few sentences long**. We'll see where this will show up shortly.

Moving on to the models layer, we just need one method in a class called **ProductModel** that returns a list of entities, so the coding is fairly trivial here:

```

/// </summary>
/// <returns>List of Product classes defined from EF</returns>
public List<Product> GetAll()
{
    eStoreDBEntities dbContext;
    List<Product> allProducts = null;
    try
    {
        dbContext = new eStoreDBEntities();
        allProducts = dbContext.Products.ToList();
    }
    catch (Exception ex)
    {
        ErrorRoutine(ex, "ProductData", "GetAll");
    }
    return allProducts;
}

```

The **ProductViewModel** class in the next layer should contain the following properties:

```

// Auto Implemented Properties
public int Qty { get; set; }
public string ProdCode { get; set; }
public string ProdName { get; set; }
public string Description { get; set; }
public string Graphic { get; set; }
public decimal Msrp { get; set; }
public decimal CostPrice { get; set; }
public int Qob { get; set; }
public int Qoh { get; set; }

```

And a method that returns a list of ProductViewModels to the web site:

```

/// <returns>List of ProductWeb instances to Presentation layer</returns>
public List<ProductViewModel> GetAll()
{
    List<ProductViewModel> webProducts = new List<ProductViewModel>();
    try
    {
        ProductModel data = new ProductModel();
        List<Product> dataProducts = data.GetAll();

        // We return ProductViewModel instances as the Asp layer has no knowledge of EF
        foreach (Product prod in dataProducts)
        {
            ProductViewModel pvm = new ProductViewModel();
            pvm.ProdCode = prod.ProductID;
            pvm.ProdName = prod.ProductName;
            pvm.Graphic = prod.GraphicName;
            pvm.CostPrice = prod.CostPrice;
            pvm.Msrp = prod.MSRP;
            pvm.Qob = prod.QtyOnBackOrder;
            pvm.Qoh = prod.QtyOnHand;
            pvm.Description = prod.Description;
            webProducts.Add(pvm); // add to list
        }
    }
    catch (Exception ex)
    {
        ErrorRoutine(ex, "ProductViewModel", "GetAll");
    }
    return webProducts;
}

```

Notice, we've converted the list of ProductModels returned from the Models layer into a list of ProductViewModels that we return to the Web layer, with our N-Tier architecture we don't let the Web layer directly touch the Model layer.

With the list returned to the web site, we need to present the product information into a format that simulates a shopping experience, namely select from a catalogue. We'll do this next class

## LAB 5

- Read the Asp.net Session article
  - Answer the 5 questions
- Complete the Login/Logout process
- Create the ProductModel, ProductViewModel classes
- Submit 5 screen shots in the same Word doc
  - Login screen with validation errors
  - Login failed
  - Login worked make sure I can see the logged in status in the menu
  - The GetAll method from your ProductViewModel class
  - A – SELECT \* FROM products (min. of 15 products with real data) from management studio.