MIS 333K Homework 6: Order Entry Site

You are going to expand the food truck website that you created for Homeworks 2 & 3. We are going to keep selling food products and add suppliers and orders so we can track more information about our products and customers.

A parallel example with classes and schedules is available for you to use as an example. See Canvas for links to handouts and the Github repo with all of the sample code.

# Part One: Familiarize Yourself with the Requirements

1. ~~Go to the working example site for this homework:~~ [~~http://sp20homework6example.azurewebsites.net~~](http://sp20homework6example.azurewebsites.net/)
2. ~~Clone the relational data demo repository from GitHub so that you can see the code behind the example. The repo is located at~~ [~~https://github.com/mis333k-spr20/sp20RelationalDataDemo.git~~](https://github.com/mis333k-spr20/sp20RelationalDataDemo.git) ~~(You will need to copy and paste this link into your browser. You cannot open it directly from Word because you have to be logged in to GitHub to see the link.)~~
3. ~~Make sure you understand what the site is supposed to do BEFORE you begin any of the coding work~~

# Part Two: Database and Model Setup

1. ~~You will need to use the database attached to the connection string provided to you by the TAs via email.~~ **~~If you used this database for Homework 4 have not received your Homework 4 grade, you cannot start on this assignment.~~**
2. ~~You will need to create a new Visual Studio project. Name your project LastName\_FirstName\_HW6. Set up all of the folders, common project files, and client-side libraries that you usually add to your projects. Remember, there are resources for this on Canvas if you need a memory refresher.~~
3. ~~Add a DAL folder to your project and copy in the AppDbContext file from the Common Project Files page on Canvas.~~ 
   1. ~~Update the using and namespace statements to match your project name.~~
   2. ~~The AppDbContext file will have some errors until you finish adding everything that you need. This is okay for now~~
4. Update the Startup.cs file
   1. ~~Update the using and namespace statements to match your project name.~~
   2. ~~Uncomment all the lines of code that are actually code. You should leave the comments that are explanatory notes for the humans as comments.~~
   3. ~~If you get any errors that say you are missing a using statement, click on the light bulb to access the quick fix that will add in the using statement that you need.~~
   4. ~~Add the connection string to the new database you created above. Please do this right away.~~
   5. ~~In your connection string, change MultipleActiveResultSets = True.~~
5. Add an AppUser class to your models user for the user. This class MUST be called AppUser.
   1. ~~This class MUST be called AppUser.~~
   2. ~~This class MUST inherit from IdentityUser. This will cause an error because you are missing a using statement. Click on the lightbulb for the quick fix that will add the correct using statement.~~
   3. ~~The users table should hold the following information:~~
      1. ~~Id – use the built-in Identity field for the unique identifier for the user. This is the primary key for this model class.~~ **~~YOU SHOULD NOT ADD AN ADDITIONAL AppUserID.~~**
      2. ~~First name – the user’s first name. This field is required. Identity does not automatically create this field in the user table; you will need to add it to the AppUser class.~~
      3. ~~Last name – the user’s last name. This field is required. Identity does not automatically create this field in the user table; you will need to add it to the AppUser class.~~
      4. ~~Email – the user’s email. This field is required. Use the built-in Identity field for this. Email will also be the user’s username.~~
      5. ~~PhoneNumber – the user’s phone number. This field is required. Use the built-in Identity field for this.~~
      6. ~~All other fields (passwords, roles, etc.) will be inherited from the IdentityUser class. See~~ [~~https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.identity.entityframeworkcore.identityuser~~](https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.identity.entityframeworkcore.identityuser) ~~for more information about this base class.~~
6. ~~Add a class called Product to your Models folder. The product model should hold the following information:~~
   1. ~~ProductID – a unique identifier for each product. This should be an auto-generated number.~~
   2. ~~Name – the product’s name. This field is required.~~
   3. ~~Description – a brief description of the product. This field is not required~~
   4. ~~Price – the price of the product. This field is required and should be displayed with a $ and 2 decimal places.~~
7. ~~Add a class called Supplier to your Models folder. A supplier has the following scalar properties:~~
   1. ~~SupplierID (Primary key)~~
   2. ~~SupplierName (String)~~
   3. ~~Email (String)~~
   4. ~~PhoneNumber (String)~~

Note that a supplier is a company, and not a person. Thus, the supplier has no direct relationship to AppUser.

1. ~~Add a class called Order to your Models folder. An order has the following scalar properties:~~
   1. ~~OrderID (Primary key)~~
   2. ~~OrderNumber - Order numbers start with 10001 and go up by one from there (10002, 10003, etc.)~~
   3. ~~OrderDate~~
   4. ~~OrderNotes~~
   5. ~~There should also be a named constant for the TAX\_RATE, which is 0.0825.~~
2. ~~There is a many-to-many relationship between products and orders. This relationship has several payload fields, so you will need a model class called OrderDetail. Order details have the following scalar properties:~~
   1. ~~Quantity~~
   2. ~~Product price (the price of the product at the time of order)~~
   3. ~~Extended price (quantity \* product price at the time of the order)~~
3. Add navigational properties to your model classes to reflect the relationships among the classes. The relationships are as follows:
   1. ~~Products are provided by multiple suppliers; suppliers provide multiple products. There is no need to store any data about a particular supplier/supplier combination (i.e. there is no payload here). A good name for the model class for the bridge table is ProductSupplier.~~
   2. ~~An order will consist of many order details, but a single order detail will only be associated with one order.~~
   3. ~~A product will be a part of many order details, but a single order detail will only be associated with one product.~~
   4. ~~Each order belongs to a single customer. A customer will have many orders. Admins may not place orders.~~
4. ~~Update your DbContext class in the DAL folder. You will need to have DbSets for suppliers, products, orders, and order details. The DbSet for Users is inherited from IdentityDbContext. You will also need a DbSet for the bridge table between Products and Suppliers.~~

~~Remember, you do NOT need to add a DbSet for AppUsers. Because your AppDbContext inherits from IdentityDbContext, you get this DbSet from the base class. The DbSet for AppUsers is called Users.~~ *~~If you add a second DbSet for AppUsers, you will get an error.~~*

1. Add the code to seed your first user into the database. This code will be run when you start the project. It will add a single admin user to your database.
   1. ~~Add a new folder to your project called Seeding.~~
   2. ~~Add the SeedIdentity.cs file from the IdentityTemplate GitHub repo to the Seeding folder~~
   3. ~~In the SeedIdentity.cs file, update the namespaces and using statements to match your project.~~
   4. ~~Look for the TODO around line 46 or 47 in the SeedIdentity.cs file. You will need to add code here to make sure that you add data to any fields you added to your AppUser model class in previous steps. (Hint: For this homework, it is probably just LastName, since I’ve given you FirstName as an example).~~
   5. ~~Save your changes to SeedIdentity.cs~~
2. ~~Update Startup.cs to ask the program to run SeedIdentity when your program starts.~~

~~Your Startup.cs class should have a method called Configure. The complete code for the Configure method is listed below. The parts to change are bolded.~~

~~public void Configure(IApplicationBuilder app, IHostingEnvironment env,~~ ***~~IServiceProvider service~~***~~)~~

~~{~~

~~//These lines allow you to see more detailed error messages~~

~~app.UseDeveloperExceptionPage();~~

~~app.UseStatusCodePages();~~

~~//TODO: Once you have added Identity into your project, you will need to uncomment this line~~

~~app.UseAuthentication();~~

~~//This line allows you to use static pages like style sheets and images~~

~~app.UseStaticFiles();~~

~~//This line configures the routing for MVC~~

~~app.UseMvc(routes => {~~

~~routes.MapRoute(~~

~~name: "default",~~

~~template: "{controller}/{action}/{id?}",~~

~~defaults: new { controller = "Home", action = "Index" });~~

~~});~~

***~~Seeding.SeedIdentity.AddAdmin(service).Wait();~~***

~~}~~

**~~CHECKPOINT: Make sure your project builds. Fix any errors until your project builds. Do not attempt to go past this step until you have fixed your errors.~~**

1. ~~Add a migration to initialize the database. Make sure to execute the migration as well. Remember the commands are:~~

~~dotnet ef migrations add Setup~~

~~dotnet ef database update –v~~

1. ~~Run the project. This will add your admin user to your database. After this runs successfully, you can comment out the last line of your Configure method in Startup so that the SeedIdentity code doesn’t run every time you start the project.~~

**~~CHECKPOINT: Go to Azure and make sure you have all of your tables in your database. Do not attempt to go beyond this point if your Tables and Fields do not exist in Azure.~~**

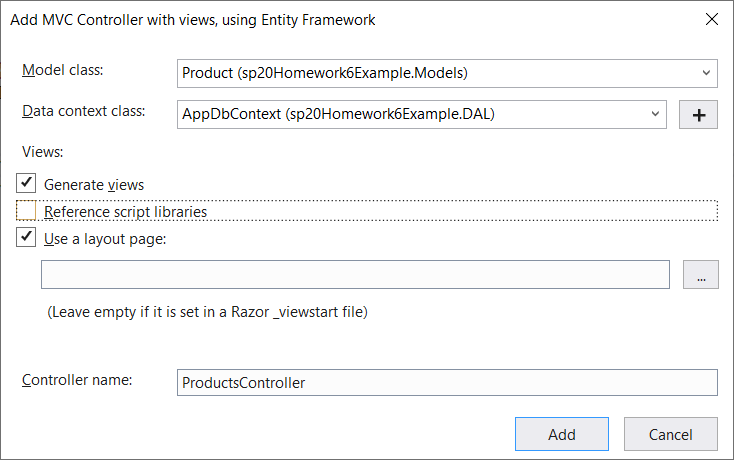
1. Add the files you need to manage users and roles in your database. These are all contained in a separate GitHub repository called sp20IdentityTemplate. The GitHub repo can be found here: <https://github.com/mis333k-spr20/sp20IdentityTemplate.git> (You will need to copy and paste this link into your browser – you will get an error if you just click on it from inside of Word because you have to be logged in to GitHub to be able to access the link.)

The repo contains a document called “How to Use the Identity Template.” **Go to that document and follow the instructions there before continuing in this document.** The first few steps in that document are repeats from things you did here, but it’s always a good idea to double-check that you have completed everything. There are a LOT of details!

The “How to Use the Identity Template” file also contains instructions for how to modify your project to restrict actions to certain roles. You can hold off on this for now. We will re-visit these instructions later in this document

**CHECKPOINT: Add the files from the IdentityTemplate. REALLY. Please don’t skip this step and then wonder why nothing works. ADD. THE. FILES. NOW.**

1. Scaffold controllers with views for Product, Supplier, Order, and OrderDetail.
   1. On Windows, right-click on the Controllers folder and select Add 🡪 Controller… Select “MVC Controller with views, using Entity Framework.”



* 1. On a Mac, you will need to follow the more detailed instructions for scaffolding that are posted on Canvas.

You will be scaffolding a total of 4 controllers: ProductsController, SuppliersController, OrdersControllers, and OrderDetailsController. You should NOT scaffold a controller for AppUser or for the bridge table class for Suppliers/Products.

1. If you haven’t already, import the corrected \_Layout.cshtml file into your Views/Shared folder so that your navbar will work correctly and your pages will be formatted nicely.
2. Add a HomeController to the Controllers folder. This should be an empty controller (do not scaffold). You should have imported the Home/Index view from the IdentityTemplate.

**CHECKPOINT: Start your project and make sure your homepage is working. Don’t go any farther until the above steps are working. You should be able to Register a new user and see all of the pages for Products, Suppliers, Orders.**

# Part Three: Add Authorize Tags and Clean Up Unused Actions/Views

In this homework, we have two types of users: admins and customers. Each role has access to certain parts of the application. We are going to set that up now.

1. Only admins can create and edit products. Add the correct [Authorize] tags on your ProductsController so that only admins may create and edit products.
2. Only show the “Create” and “Edit” links on Products/Index.cshtml to admins. Reference the code in the Layout view to see how to do this.
3. Only show the “Edit” link on Products/Details.cshtml to admins. Reference the code in the Layout view to see how to do this.
4. No one (not even admins) should be able to delete products because we could end up with orphan data. (If you delete a product that has been ordered, it will cause problems.)
   1. In the ProductsController, delete the action methods for deleting products.
   2. Delete the Views/Products/Delete view from your project.
   3. Edit the Views/Products/Index view to remove the links to delete products
5. Only admins should be able to see any information related to Suppliers. Add the correct [Authorize] tag to the Suppliers controller so that only Admins can see the pages associated with Suppliers.
6. We don’t want any users (even admins) to be able to delete suppliers because deleting a supplier associated with a product could result in orphan data.
   * Remove all code from SuppliersController that allows the user to delete a supplier.
   * Delete the Suppliers/Delete view.
   * Edit the Suppliers/Index view to remove the links to delete suppliers.

**CHECKPOINT: Add a few suppliers to see if this part is working. Edit the supplier to see if your changes persist. You won’t be able to see the associated products because these suppliers won’t have any products associated with them yet. (That’s the next step.) Don’t go any farther until everything above here is working.**

# Part Four: Allow the User to Edit the Supplier/Product Relationship

1. On the Products/Create view, add a listbox to display all the suppliers. Users should be able to select one or more suppliers for the product.
2. On the Products/Create GET method, add the code to populate the ViewBag with the Multiselect list for the listbox.
3. On the Products/Create POST method, add a parameter for list of selected suppliers and add the code to store the selected suppliers on the database. See the “Relational Data” handout and the relational data demo repository for help with this.
4. On the Products/Edit view and actions, make similar changes to allow the user to Edit the suppliers associated with a product. Again, see the “Relational Data” handout and the relational data demo repository for help with this.
5. The user does NOT need to be able to add/edit the products associated with a supplier on Supplier/Create or Supplier/Edit. If the user wants to change the supplier/product relationship, then he or she can do so by editing through products.
6. On the Suppliers/Index view: No changes needed. Only admins will see this page, so it is okay to leave the Create link.
7. On the Suppliers/Create view: No changes needed. Just allow the user to create the scalar properties for Suppliers.
8. On the Suppliers/Edit view: No changes needed. Just allow the user to edit the scalar properties for Suppliers.
9. On the Suppliers/Details view, show all the products that are provided by this supplier. Show the product name. You will need to modify the controller code to have the appropriate Include and ThenInclude statements.
10. On the Products/Details view, show ADMINS only all of the suppliers for this product. Show supplier name. You will need to modify the Product/Details action to add an Include/ThenInclude statement to the query that pulls the product data.

**CHECKPOINT: Add a few products to see if this part is working. Edit the product (especially the related suppliers) to see if your changes persist. You should now also be able to see the associated products on the Supplier/Details views. Don’t go any farther until everything above here is working.**

# Part Five: Allow the User to Create an Order and Add Products

1. Allow the user to create the order object. Use the scaffolded code for the Order/Create action as a starting point. You do not need to change the existing GET method for Order/Create. Make the following changes to the POST method for Order/Create
   1. Modify the OrdersController so that only authorized users can access the action methods.
   2. Only customers may create orders. Modify the OrdersController so that only Customers can create orders.
   3. Modify Views/Orders/Index so that only Customers see the create link.
   4. Modify the OrdersController/Index method to display all the orders when the user is an admin. When the user is a customer, they should only see their own orders.
   5. In Views/Order/Create.cshtml, remove the code for the controls for the fields that the user can’t modify (OrderNumber, OrderDate)
   6. Modify the OrdersController/Create POST action method to meet our business rules:
      1. The system should automatically generate the next order number. The user should not be able to enter or edit the order number.
      2. The system should automatically record the date of the order. The user should not be able to enter or edit the order date. (Hint: DateTime.Now will “ask” the computer for the current date – you can use this to set the OrderDate property.)
      3. The system should automatically associate the order with the logged in user. You will need to modify the controller to get an instance of the User Manager class to help you find the user. See the RegistrationsController in the RelationalDataDemo to help you with this.
      4. Instead of re-directing the user to the Index view after they have created the order, direct them to a view that will allow them to add a product to the order (OrderDetails/Create)
2. Modify the OrderDetailsController/Create GET action method to create a new OrderDetail for the order
   1. This action method will need a parameter for the OrderID from the order you just created.
   2. Create a new instance of the order detail class
   3. Find the order associated with the OrderID that was passed in as a parameter.
   4. Set the new order detail’s order equal to the order you just found.
   5. Populate the ViewBag with a list of the existing products.
   6. Pass the OrderDetails/Create view the newly created order detail object
3. Modify the OrderDetails/Create view to allow the user to select the product and quantity for the order detail.
   1. Remove the code for properties that are not to be edited by the user.
   2. Add the code for the drop-down for products
   3. Add a hidden element for the Order ID

@Html.HiddenFor(model => model.Order.OrderID)

1. Modify the OrderDetailsController/Create POST action method that will add the newly created OrderDetail to the database.
   1. The post action method should have parameters for the order detail object (OrderDetail type) and selected product (int).
   2. Modify (or remove) the bind list to include the Order property on the list of editable properties for the order detail.
   3. Find the product associated with the int selected product parameter. Set the OrderDetail’s product property equal to this newly found product.
   4. Find the order associated with the order detail object parameter. Set the OrderDetail’s order property equal to this newly found order.
   5. Set the OrderDetail’s price equal to the selected product’s price. This will save the price that the user paid, so that updates to the price on the product will not affect this order.
   6. Calculate the extended price for the order detail and store it in the OrderDetail’s ExtendedPrice property.
   7. If the model state is valid, add the order detail to the database and re-direct the user to the details view for the order.
   8. If the model state is not valid, return the user to the OrderDetails/Create view.
2. On the Orders/Details view:
   1. Show all the order details associated with this order. Show product name, price, quantity, and extended price. You will need to modify the Order/Details action method to add an include statement to the query that gets the order information.
   2. Show an order summary section that displays the following:
      1. Order subtotal (the sum of the extended prices of the order)
      2. Sales tax (subtotal \* 0.0825)
      3. Order total (subtotal + sales tax)

SUPER HELPFUL HINT: Add read-only properties to the order model class to do these summary calculations! Since these values will not be stored on the tables on the database, you do NOT need to add a migration after you have added these read-only properties.

1. On the Orders/Index view:
   1. Add a column to the table to display the order total. The order total should be formatted with a $ and 2 decimal places. You will need to update the Orders/Index action method to add an include statement to the query that retrieves all of the order information.
   2. Remove the links that allow the user to delete an order.
2. Update the Orders/Edit View to display OrderDetail information
   1. Users should not be able to edit the order number or order date. Modify the scaffolded code to display these values without allowing the user to edit them.
   2. Add a hidden element for OrderID. The code should be

<input type="hidden" asp-for="OrderID" />

* 1. Add a table that shows the existing OrderDetails for that order. Show Product Title, Price, Quantity, and Extended Price. (Hint: Copy this from your Order/Details view that you modified above.) You will need to modify the Orders/Edit action method to add an include statement to the query that retrieves the order information.
  2. Add a button to the Orders/Edit view to allow the user to add another item to the order. This button should direct the user to the OrderDetails/Create action method and include the order number as a route value.
  3. Add another button to the Orders/Edit view to allow the user to edit the existing items on the order. This button should direct the user to the OrderDetails/Index action method and include order number as a route value.

1. Modify the OrderDetailsController/Index GET action method that will allow the user to edit the OrderDetail(s) from the order (OrderDetailsController/Index)
   1. The action method should have an int parameter for OrderID
   2. Find the order details associated with this order using a where statement. Be sure to add an include statement to pull in the product information.
   3. Pass the list of order details associated with the order to the Orders/Index view
2. Modify the OrderDetails/Index view to display the list of order details associated with the order
   1. Add code to display the product name on the view
   2. Change the action links at the bottom of the view to give the user the opportunity to edit or delete the order detail. These links should take the user to the OrderDetails/Edit or OrderDetails/Delete action methods, respectively. Remove the links for viewing the details of the order details.
3. Modify the OrderDetailsController/Edit POST action method to re-calculate the order detail’s product price and extended price and store these values in the OrderDetail’s properties. Modify the return statement on so that you go back to Order/Details for that order when the user has finished editing the order detail.
4. You shouldn’t have to make any changes to the OrderDetails/Delete view. Modify the return statement on the OrderDetailsController/Delete POST action so that you go back to Order/Details for that order when the user has finished deleting the order detail.
5. We won’t be directly viewing the details of order details, so remove the Details action method from the OrderDetailsController. Delete the OrderDetails/Details view from the Views folder.

Your system should also support the following business rules:

1. There are two classifications of users: customers and administrators (admins). Users can register themselves as customers, but admins cannot register themselves as admins. An existing admin can add a customer to the admin role using the RoleAdmin pages (see the Identity Template).
2. Only admins can see the page to change the role of a customer (RoleAdmin). Anonymous users and registered customers should not see the link to Manage Users. If non-admins attempt to visit the page by typing in the URL, they should see either a request to log in or an error page.
3. Make sure that you have pre-loaded an admin user into your database so we can test the functionality. The user’s email should be admin@example.com and the password should be Abc123! (HINT: The Seeding/SeedIdentity method that you added above should do this for you!

Add some test data and play around with everything to make sure it works. You might want to just start with ADDING order details before you attempt editing and removing order details. For this homework, we aren’t concerned with what data is in your database, so feel free to experiment as much as you need to (but remember, I can see this data, so keep it business appropriate, please!)

Publish your website to a NEW web app on Azure. The app name should be sp20LastNameFirstNameHW6.azurewebsites.net. Submit the link to the Azure page and a zipped copy of your solution file on Canvas.