

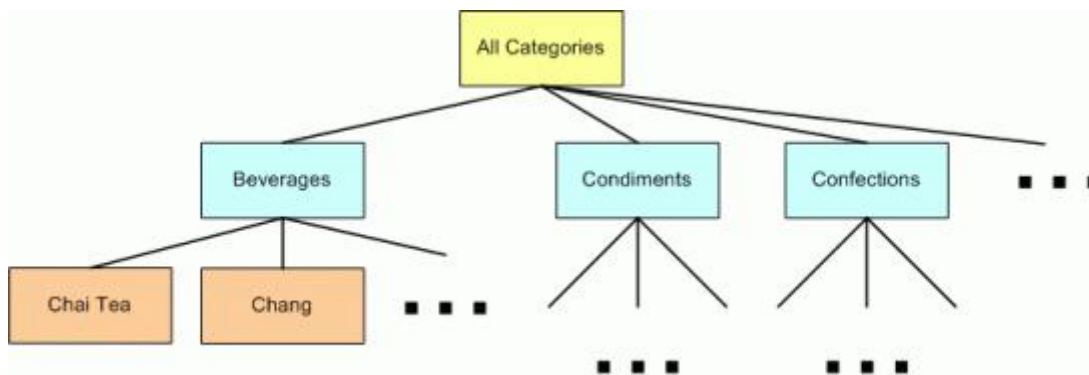
This tutorial is part of a set. Find out more about data access with ASP.NET in the Working with Data in ASP.NET 2.0 section of the ASP.NET site at <http://www.asp.net/learn/dataaccess/default.aspx>.

# Working with Data in ASP.NET 2.0 :: Building a Custom Database-Driven Site Map Provider

## Introduction

ASP.NET 2.0's site map feature enables a page developer to define a web application's site map in some persistent medium, such as in an XML file. Once defined, the site map data can be accessed programmatically through the [SiteMap class](#) in the [System.Web namespace](#) or through a variety of navigation Web controls, such as the SiteMapPath, Menu, and TreeView controls. The site map system uses the [provider model](#) so that different site map serialization implementations can be created and plugged into a web application. The default site map provider that ships with ASP.NET 2.0 persists site map structure in an XML file. Back in the [Master Pages and Site Navigation](#) tutorial we created a file named `Web.sitemap` that contained this structure and have been updating its XML with each new tutorial section.

The default XML-based site map provider works well if the site map's structure is fairly static, such as for these tutorials. In many scenarios, however, a more dynamic site map is needed. Consider the site map shown in Figure 1, where each category and product appear as sections in the website's structure. With this site map, visiting the web page corresponding to the root node might list all of the categories, whereas visiting a particular category's web page would list that category's products and viewing a particular product's web page would show that product's details.



**Figure 1: The Categories and Products Makeup the Site Map's Structure**

While this category- and product-based structure could be hard-coded into the `Web.sitemap` file, the file would need to be updated each time a category or product was added, removed, or renamed. Consequently, the site map maintenance would be greatly simplified if its structure was retrieved from the database or, ideally, from the Business Logic Layer of the application's architecture. That way, as products and categories were added, renamed, or deleted, the site map would automatically update to reflect these changes.

Since ASP.NET 2.0's site map serialization is built atop the provider model, we can create our own custom site map provider that grabs its data from an alternate data store, such as the database or architecture. In this tutorial we'll build a custom provider that retrieves its data from the BLL. Let's get started!

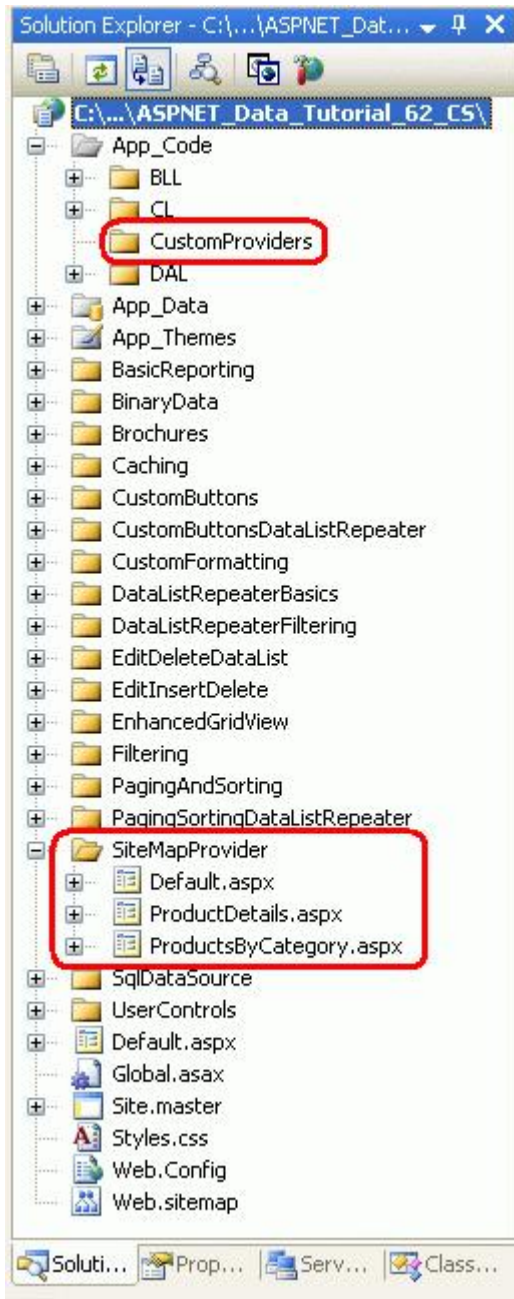
**Note:** The custom site map provider created in this tutorial is tightly coupled to the application's architecture and data model. Jeff Prosis's [Storing Site Maps in SQL Server](#) and [The SQL Site Map Provider You've Been Waiting For](#) articles examine a generalized approach to storing site map data in SQL Server.

## Step 1: Creating the Custom Site Map Provider Web Pages

Before we start creating a custom site map provider, let's first add the ASP.NET pages we'll need for this tutorial. Start by adding a new folder named `SiteMapProvider`. Next, add the following ASP.NET pages to that folder, making sure to associate each page with the `Site.master` master page:

- `Default.aspx`
- `ProductsByCategory.aspx`
- `ProductDetails.aspx`

Also add a `CustomProviders` subfolder to the `App_Code` folder.



**Figure 2: Add the ASP.NET Pages for the Site Map Provider-Related Tutorials**

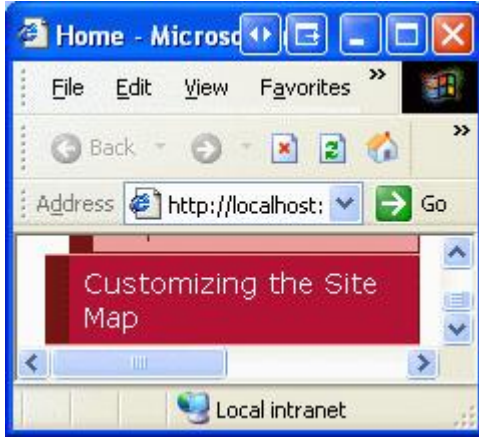
Since there is only one tutorial for this section, we don't need `Default.aspx` to list the section's tutorials. Instead, `Default.aspx` will display the categories in a `GridView` control. We'll tackle this in Step 2.

Next, update `Web.sitemap` to include a reference to the `Default.aspx` page. Specifically, add the following markup after the "Caching" `<siteMapNode>`:

```
<siteMapNode
  title="Customizing the Site Map" url="~/SiteMapProvider/Default.aspx"
  description="Learn how to create a custom provider that retrieves the site map
  from the Northwind database." />
```

After updating `Web.sitemap`, take a moment to view the tutorials website through a browser. The menu on the left

now includes an item for the sole site map provider tutorial.



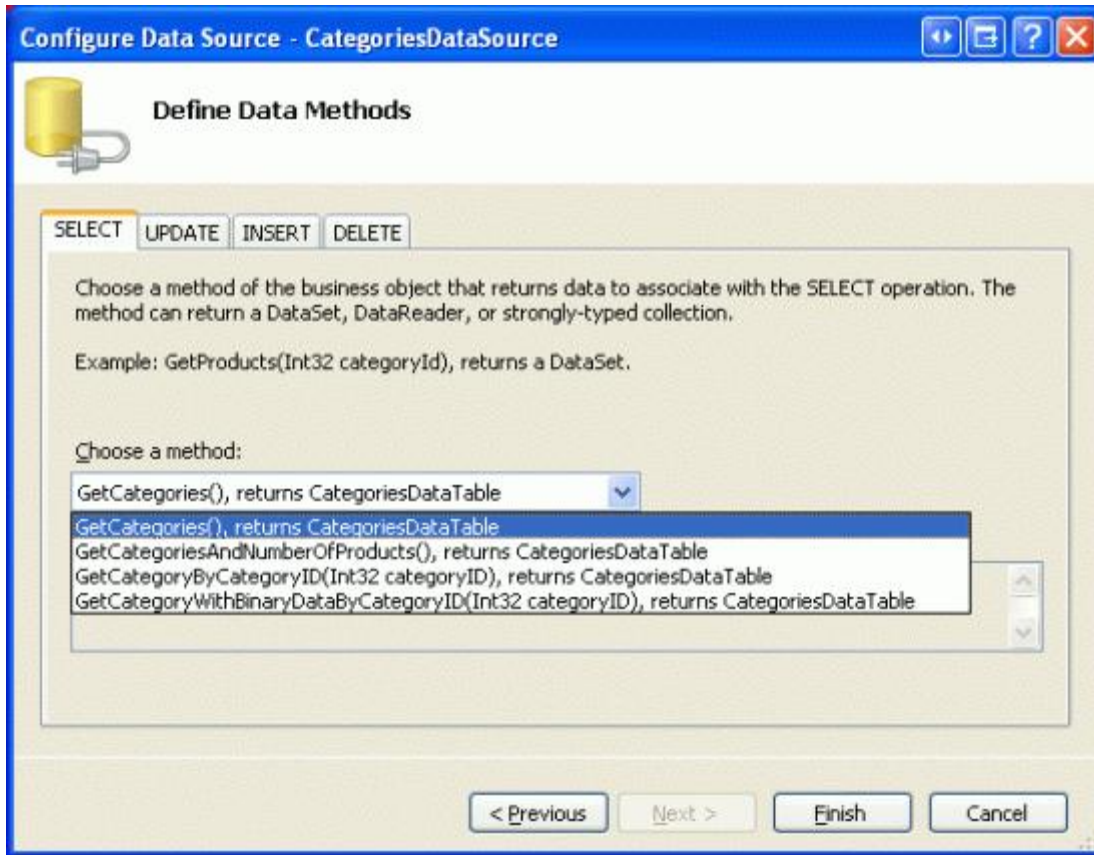
**Figure 3: The Site Map Now Includes an Entry for the Site Map Provider Tutorial**

This tutorial's main focus is to illustrate creating a custom site map provider and configuring a web application to use that provider. In particular, we'll build a provider that returns a site map that includes a root node along with a node for each category and product, as depicted in Figure 1. In general, each node in the site map may specify a URL. For our site map, the root node's URL will be `~/SiteMapProvider/Default.aspx`, which will list all of the categories in the database. Each category node in the site map will have a URL that points to `~/SiteMapProvider/ProductsByCategory.aspx?CategoryID=categoryID`, which will list all of the products in the specified *categoryID*. Finally, each product site map node will point to `~/SiteMapProvider/ProductDetails.aspx?ProductID=productID`, which will display the specific product's details.

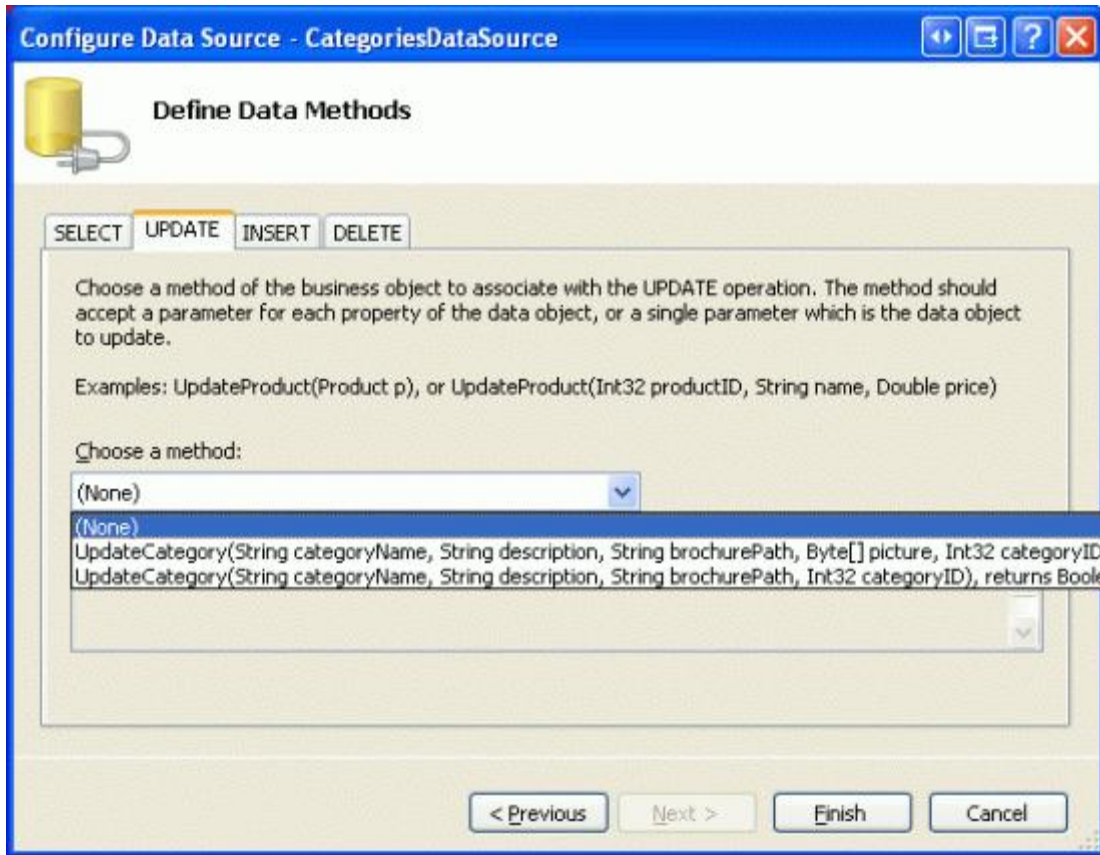
To start we need to create the `Default.aspx`, `ProductsByCategory.aspx`, and `ProductDetails.aspx` pages. These pages are completed in Steps 2, 3, and 4, respectively. Since the thrust of this tutorial is on site map providers, and since past tutorials have covered creating these sorts of multi-page master/detail reports, we will hurry through Steps 2 through 4. If you need a refresher on creating master/detail reports that span multiple pages, refer back to the [Master/Detail Filtering Across Two Pages](#) tutorial.

## Step 2: Displaying a List of Categories

Open the `Default.aspx` page in the `SiteMapProvider` folder and drag a `GridView` from the `Toolbox` onto the `Designer`, setting its `ID` to `Categories`. From the `GridView`'s smart tag, bind it to a new `ObjectDataSource` named `CategoriesDataSource` and configure it so that it retrieves its data using the `CategoriesBLL` class's `GetCategories` method. Since this `GridView` just displays the categories and does not provide data modification capabilities, set the drop-down lists in the `UPDATE`, `INSERT`, and `DELETE` tabs to "(None)".



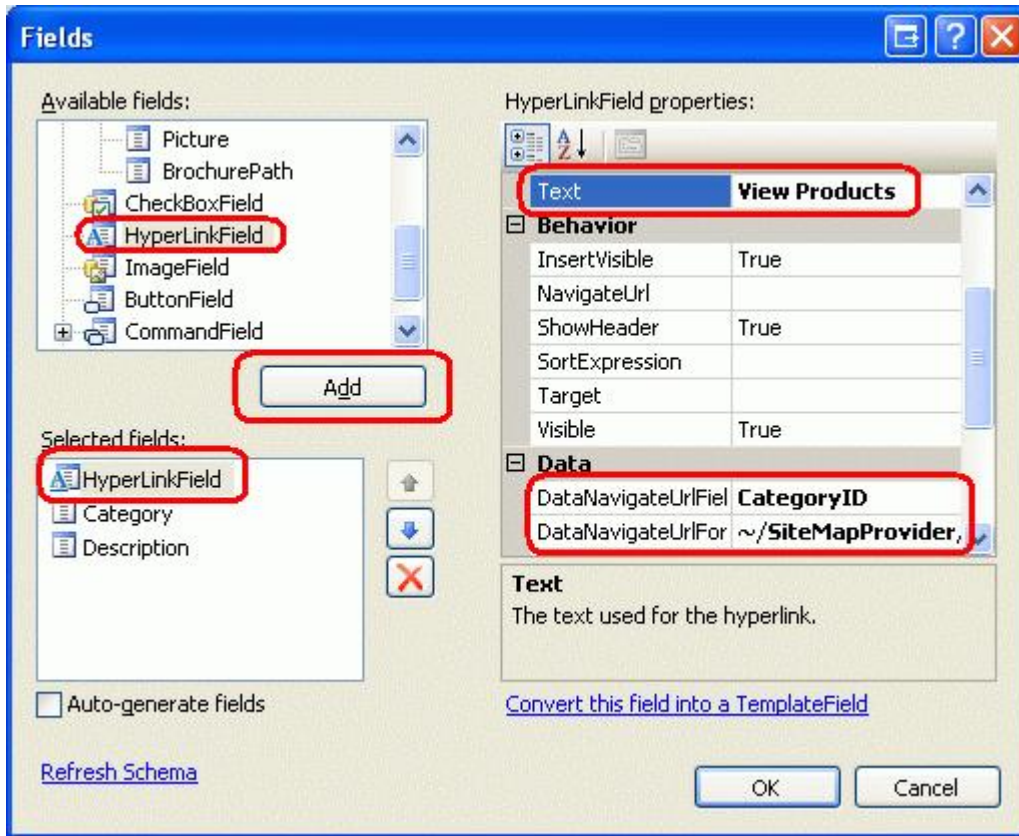
**Figure 4: Configure the ObjectDataSource to Return Categories Using the GetCategories Method**



**Figure 5: Set the Drop-Down Lists in the UPDATE, INSERT, and DELETE Tabs to “(None)”**

After completing the Configure Data Source wizard, Visual Studio will add a BoundField for `CategoryID`, `CategoryName`, `Description`, `NumberOfProducts`, and `BrochurePath`. Edit the GridView so that it only contains the `CategoryName` and `Description` BoundFields and update the `CategoryName` BoundField's `HeaderText` property to “Category”.

Next, add a HyperLinkField and position it so that it's the left-most field. Set the `DataNavigateUrlFields` property to `CategoryID` and the `DataNavigateUrlFormatString` property to `~/SiteMapProvider/ProductsByCategory.aspx?CategoryID={0}`. Set the `Text` property to “View Products”.



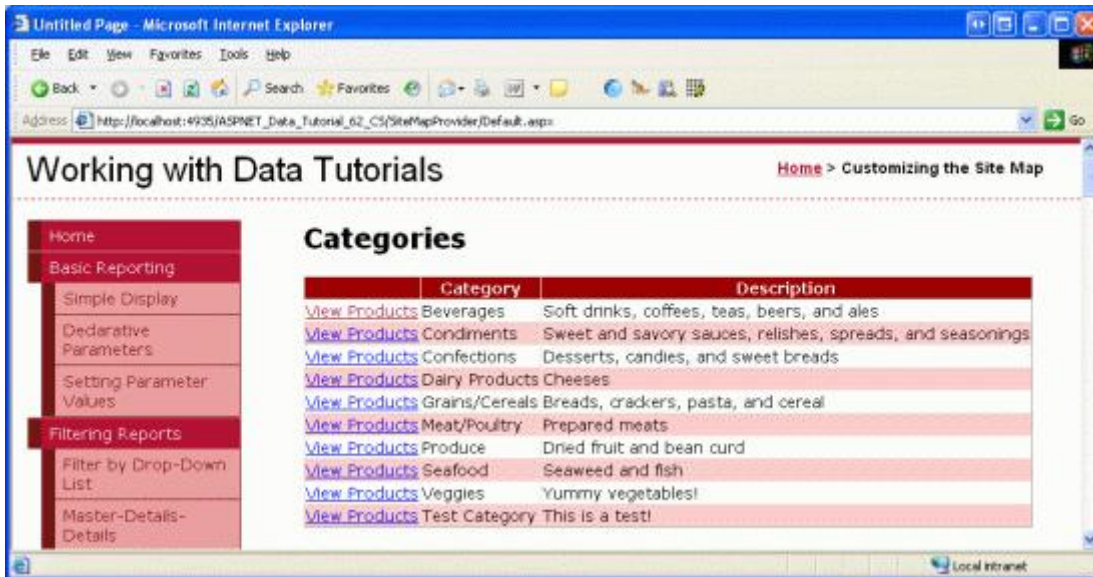
**Figure 6: Add a HyperLinkField to the Categories GridView**

After creating the ObjectDataSource and customizing the GridView's fields, the two controls' declarative markup will look like the following:

```
<asp:GridView ID="Categories" runat="server" AutoGenerateColumns="False"
  DataKeyNames="CategoryID" DataSourceID="CategoriesDataSource"
  EnableViewState="False">
  <Columns>
    <asp:HyperLinkField DataNavigateUrlFields="CategoryID"
      DataNavigateUrlFormatString=
        "~/SiteMapProvider/ProductsByCategory.aspx?CategoryID={0}"
      Text="View Products" />
    <asp:BoundField DataField="CategoryName" HeaderText="Category"
      SortExpression="CategoryName" />
    <asp:BoundField DataField="Description" HeaderText="Description"
      SortExpression="Description" />
  </Columns>
</asp:GridView>

<asp:ObjectDataSource ID="CategoriesDataSource" runat="server"
  OldValuesParameterFormatString="original_{0}" SelectMethod="GetCategories"
  TypeName="CategoriesBLL"></asp:ObjectDataSource>
```

Figure 7 shows `Default.aspx` when viewed through a browser. Clicking a category's "View Products" link takes you to `ProductsByCategory.aspx?CategoryID=categoryID`, which we will build in Step 3.

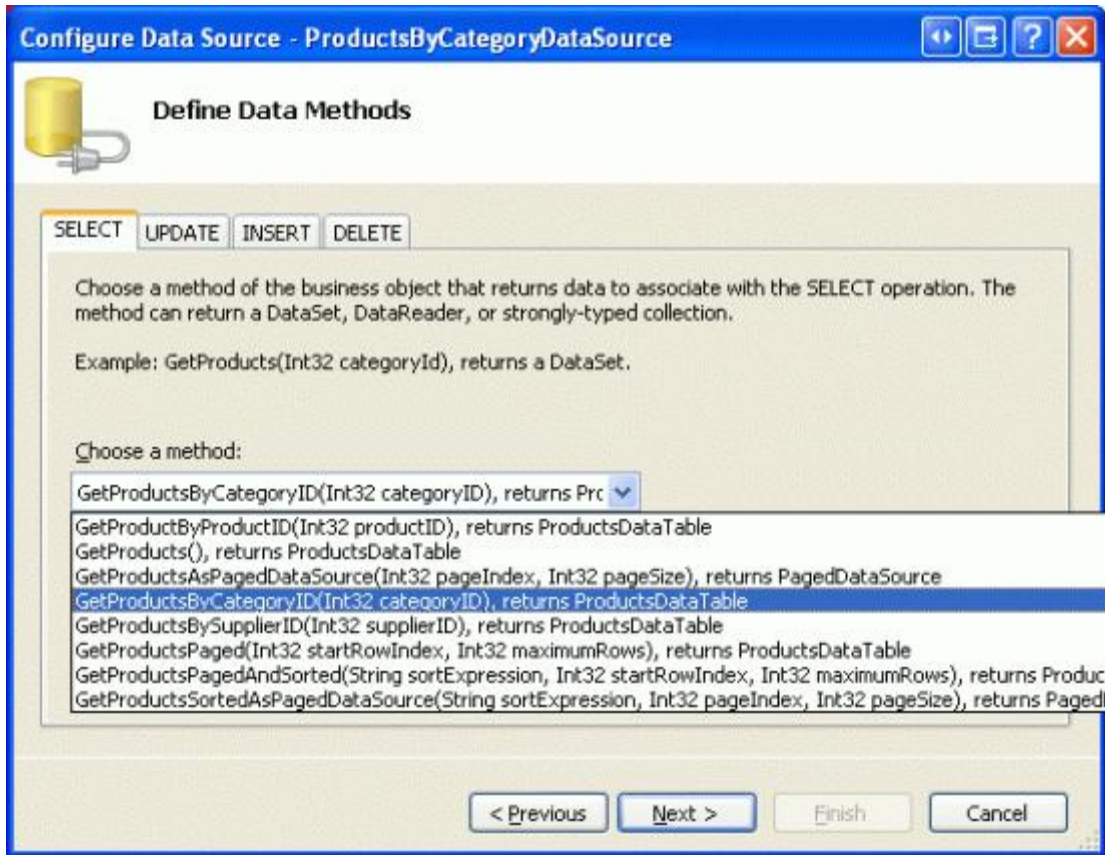


**Figure 7: Each Category is Listed Along with a “View Products” Link**

### Step 3: Listing the Selected Category’s Products

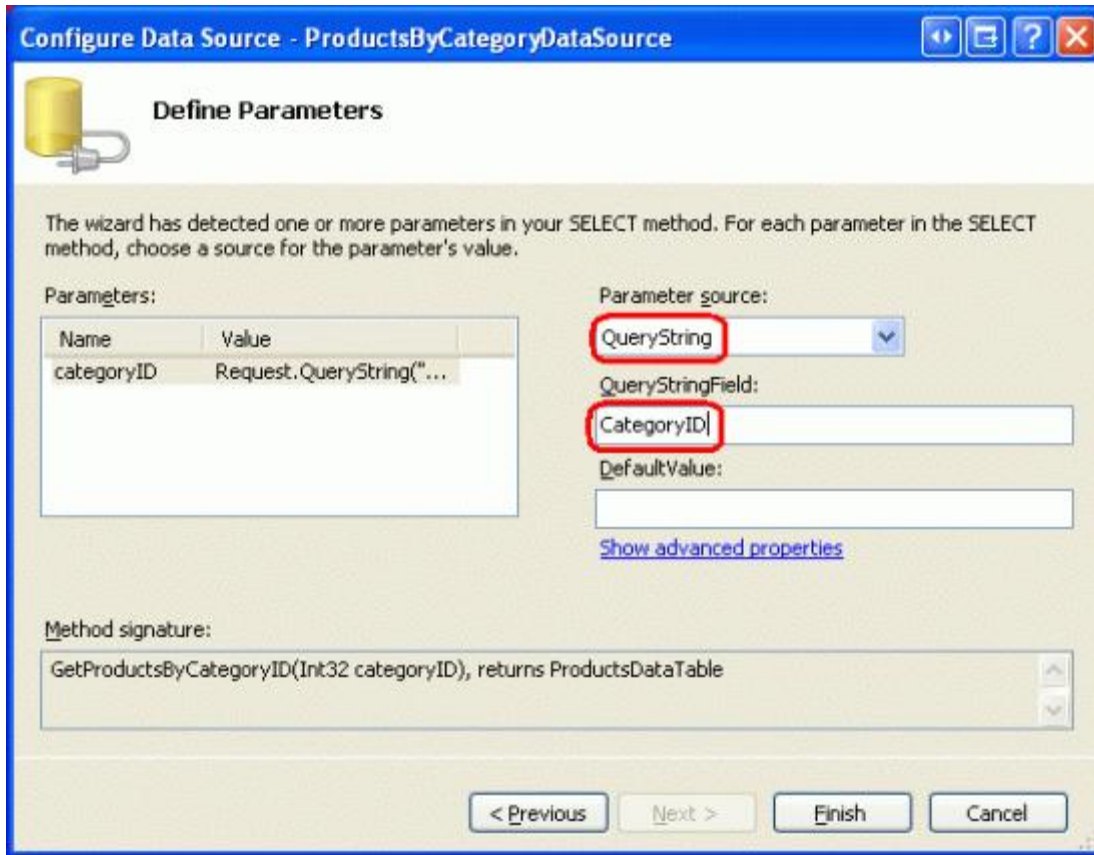
Open the `ProductsByCategory.aspx` page and add a `GridView`, naming it `ProductsByCategory`. From its smart tag, bind the `GridView` to a new `ObjectDataSource` named `ProductsByCategoryDataSource`. Configure the `ObjectDataSource` to use the `ProductsBLL` class’s `GetProductsByCategoryID(categoryID)` method and set the drop-down lists to “(None)” in the `UPDATE`, `INSERT`, and `DELETE` tabs.





**Figure 8: Use the ProductsBLL Class’s GetProductsByCategoryID (categoryId) Method**

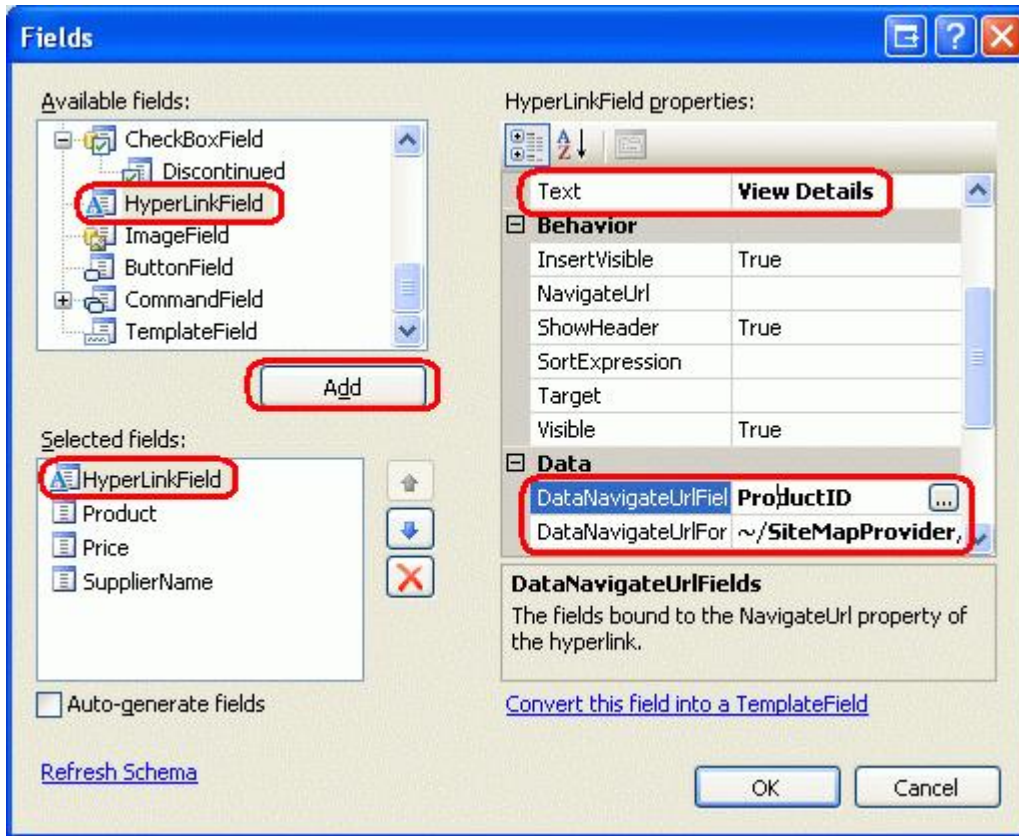
The final step in the Configure Data Source wizard prompts for a parameter source for *categoryId*. Since this information is passed through the querystring field `categoryId`, select `QueryString` from the drop-down list and enter “`categoryId`” in the `QueryStringField` textbox as shown in Figure 9. Click `Finish` to complete the wizard.



**Figure 9: Use the CategoryID Querystring Field for the categoryID Parameter**

After completing the wizard, Visual Studio will add corresponding BoundFields and a CheckBoxField to the GridView for the product data fields. Remove all but the `ProductName`, `UnitPrice`, and `SupplierName` BoundFields. Customize these three BoundFields' `HeaderText` properties to read "Product", "Price", and "Supplier", respectively. Format the `UnitPrice` BoundField as a currency.

Next, add a `HyperLinkField` and move it to the left-most position. Set its `Text` property to "View Details", its `DataNavigateUrlFields` property to `ProductID`, and its `DataNavigateUrlFormatString` property to `~/SiteMapProvider/ProductDetails.aspx?ProductID={0}`.



**Figure 10: Add a “View Details” HyperLinkField that Points to ProductDetails.aspx**

After making these customizations, the GridView and ObjectDataSource’s declarative markup should resemble the following:

```
<asp:GridView ID="ProductsByCategory" runat="server" AutoGenerateColumns="False"
  DataKeyNames="ProductID" DataSourceID="ProductsByCategoryDataSource"
  EnableViewState="False">
  <Columns>
    <asp:HyperLinkField DataNavigateUrlFields="ProductID"
      DataNavigateUrlFormatString=
        "~/SiteMapProvider/ProductDetails.aspx?ProductID={0}"
      Text="View Details" />
    <asp:BoundField DataField="ProductName" HeaderText="Product"
      SortExpression="ProductName" />
    <asp:BoundField DataField="UnitPrice" DataFormatString="{0:c}"
      HeaderText="Price" HtmlEncode="False"
      SortExpression="UnitPrice" />
    <asp:BoundField DataField="SupplierName" HeaderText="Supplier"
      ReadOnly="True" SortExpression="SupplierName" />
  </Columns>
</asp:GridView>

<asp:ObjectDataSource ID="ProductsByCategoryDataSource" runat="server"
  OldValuesParameterFormatString="original_{0}"
  SelectMethod="GetProductsByCategoryID" TypeName="ProductsBLL">
  <SelectParameters>
    <asp:QueryStringParameter Name="categoryID"
```

```
QueryStringField="CategoryID" Type="Int32" />
</SelectParameters>
</asp:ObjectDataSource>
```

Return to viewing `Default.aspx` through a browser and click on the “View Products” link for Beverages. This will take you to `ProductsByCategory.aspx?CategoryID=1`, displaying the names, prices, and suppliers of the products in the Northwind database that belong to the Beverages category (see Figure 11). Feel free to further enhance this page to include a link to return users to the category listing page (`Default.aspx`) and a `DetailsView` or `FormView` control that displays the selected category’s name and description.

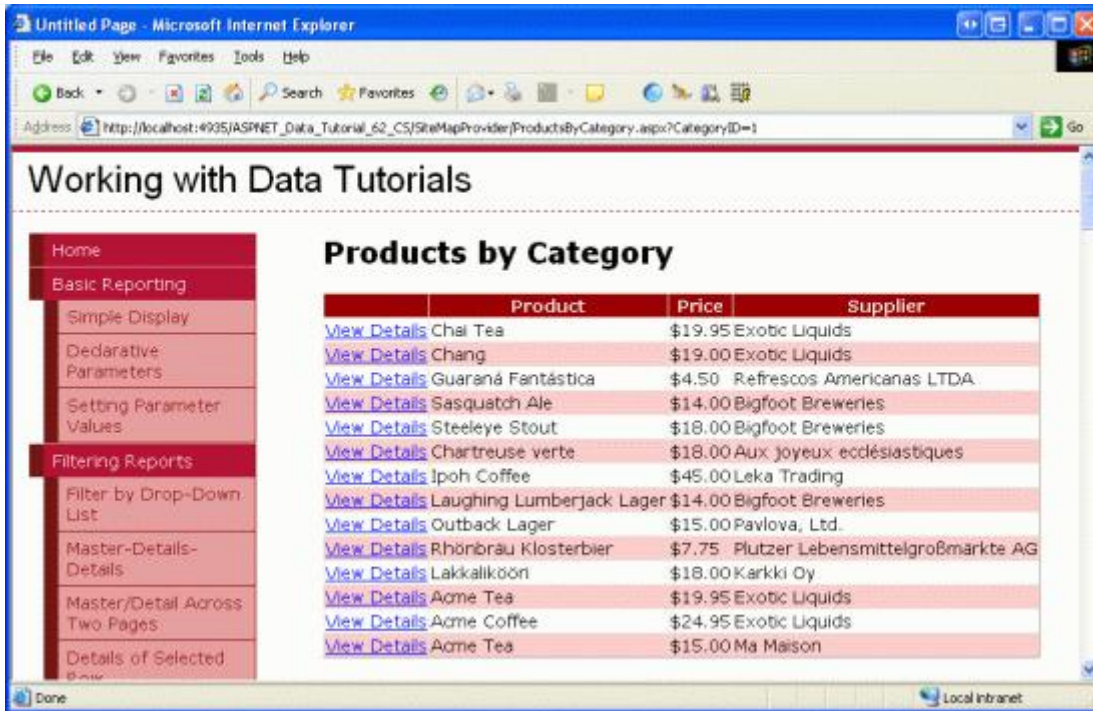
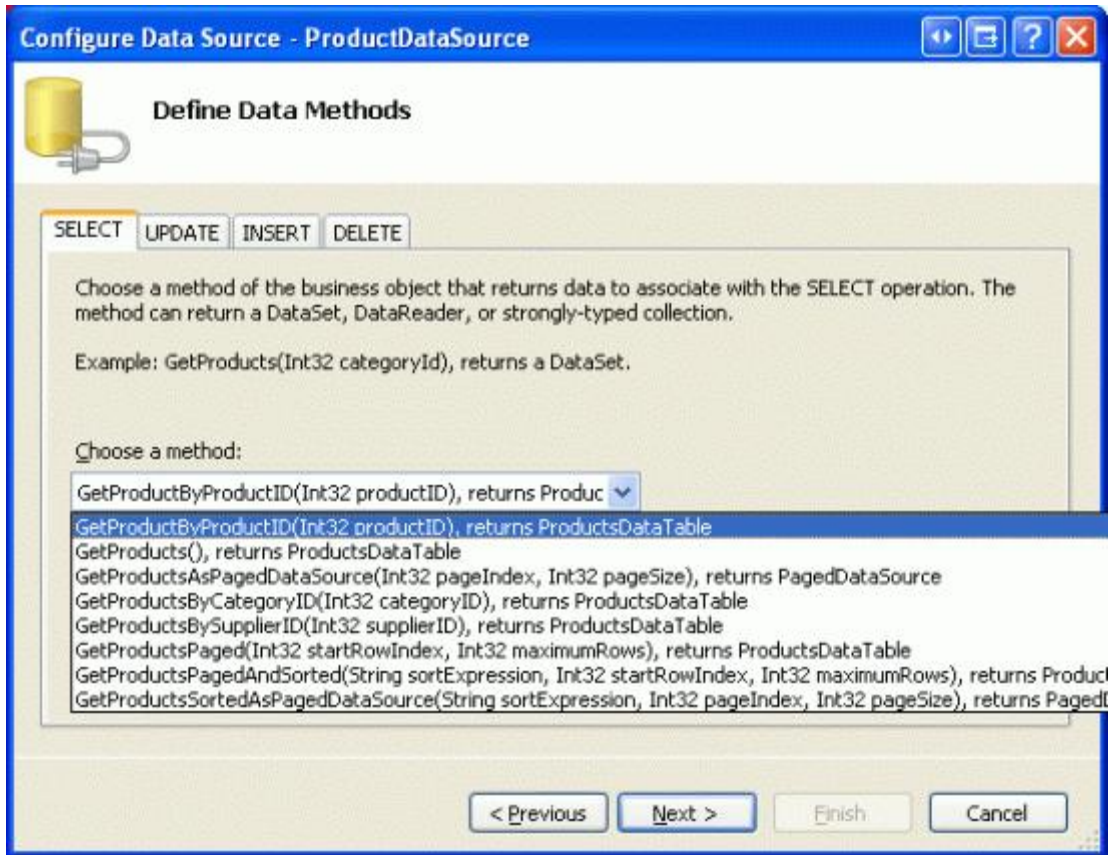


Figure 11: The Beverages’ Names, Prices, and Suppliers are Displayed

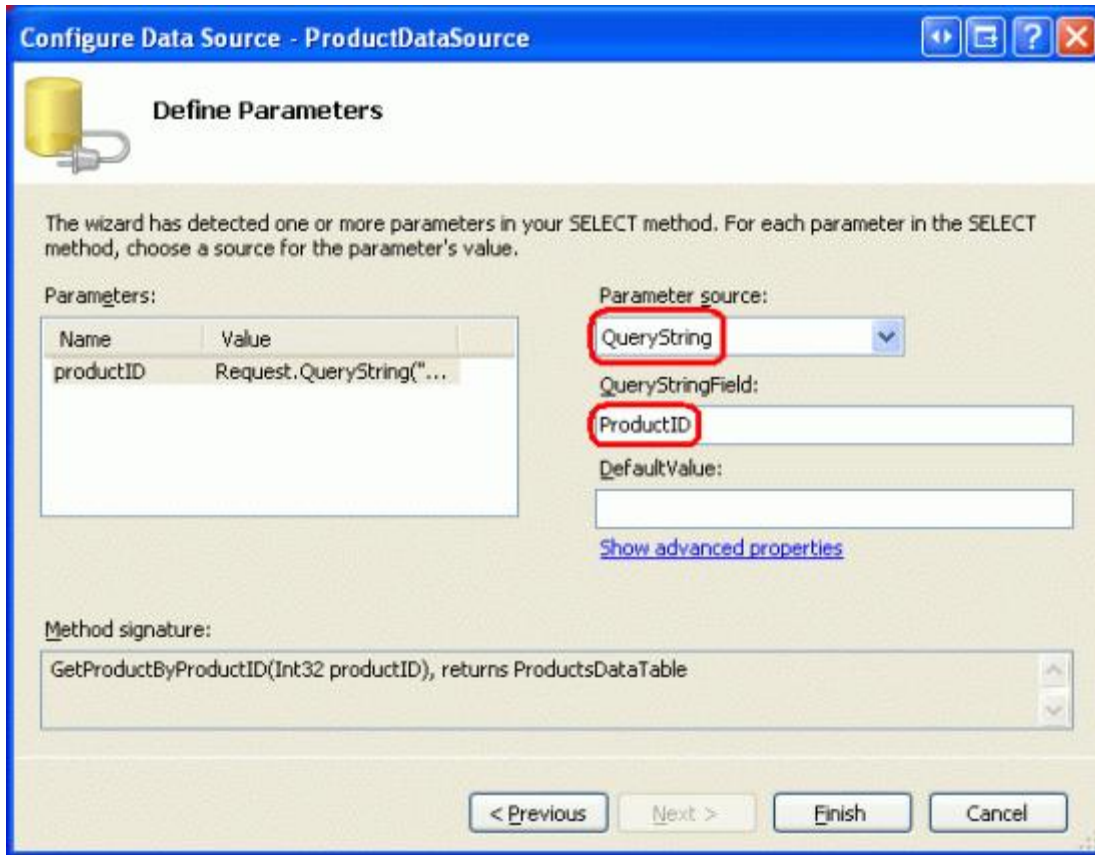
## Step 4: Showing a Product’s Details

The final page, `ProductDetails.aspx`, displays the selected products details. Open `ProductDetails.aspx` and drag a `DetailsView` from the Toolbox onto the Designer. Set the `DetailsView`’s `ID` property to `ProductInfo` and clear out its `Height` and `Width` property values. From its smart tag, bind the `DetailsView` to a new `ObjectDataSource` named `ProductDataSource`, configuring the `ObjectDataSource` to pull its data from the `ProductsBLL` class’s `GetProductByProductID(productID)` method. As with the previous web pages created in Steps 2 and 3, set the drop-down lists in the `UPDATE`, `INSERT`, and `DELETE` tabs to “(None)”.



**Figure 12: Configure the ObjectDataSource to Use the `GetProductByProductID (productID)` Method**

The last step of the Configure Data Source wizard prompts for the source of the *productID* parameter. Since this data comes through the querystring field `ProductID`, set the drop-down list to `QueryString` and the `QueryStringField` textbox to “`ProductID`”. Finally, click the `Finish` button to complete the wizard.



**Figure 13: Configure the *productID* Parameter to Pull its Value from the `ProductID` Querystring Field**

After completing the Configure Data Source wizard, Visual Studio will create corresponding BoundFields and a CheckBoxField in the DetailsView for the product data fields. Remove the `ProductID`, `SupplierID`, and `CategoryID` BoundFields and configure the remaining fields as you see fit. After a handful of aesthetic configurations, my DetailsView and ObjectDataSource's declarative markup looked like the following:

```
<asp:DetailsView ID="ProductInfo" runat="server" AutoGenerateRows="False"
    DataKeyNames="ProductID" DataSourceID="ProductDataSource"
    EnableViewState="False">
    <Fields>
        <asp:BoundField DataField="ProductName" HeaderText="Product"
            SortExpression="ProductName" />
        <asp:BoundField DataField="CategoryName" HeaderText="Category"
            ReadOnly="True" SortExpression="CategoryName" />
        <asp:BoundField DataField="SupplierName" HeaderText="Supplier"
            ReadOnly="True" SortExpression="SupplierName" />
        <asp:BoundField DataField="QuantityPerUnit" HeaderText="Qty/Unit"
            SortExpression="QuantityPerUnit" />
        <asp:BoundField DataField="UnitPrice" DataFormatString="{0:c}"
            HeaderText="Price" HtmlEncode="False"
            SortExpression="UnitPrice" />
        <asp:BoundField DataField="UnitsInStock" HeaderText="Units In Stock"
            SortExpression="UnitsInStock" />
        <asp:BoundField DataField="UnitsOnOrder" HeaderText="Units On Order"
            SortExpression="UnitsOnOrder" />
        <asp:BoundField DataField="ReorderLevel" HeaderText="Reorder Level"
```

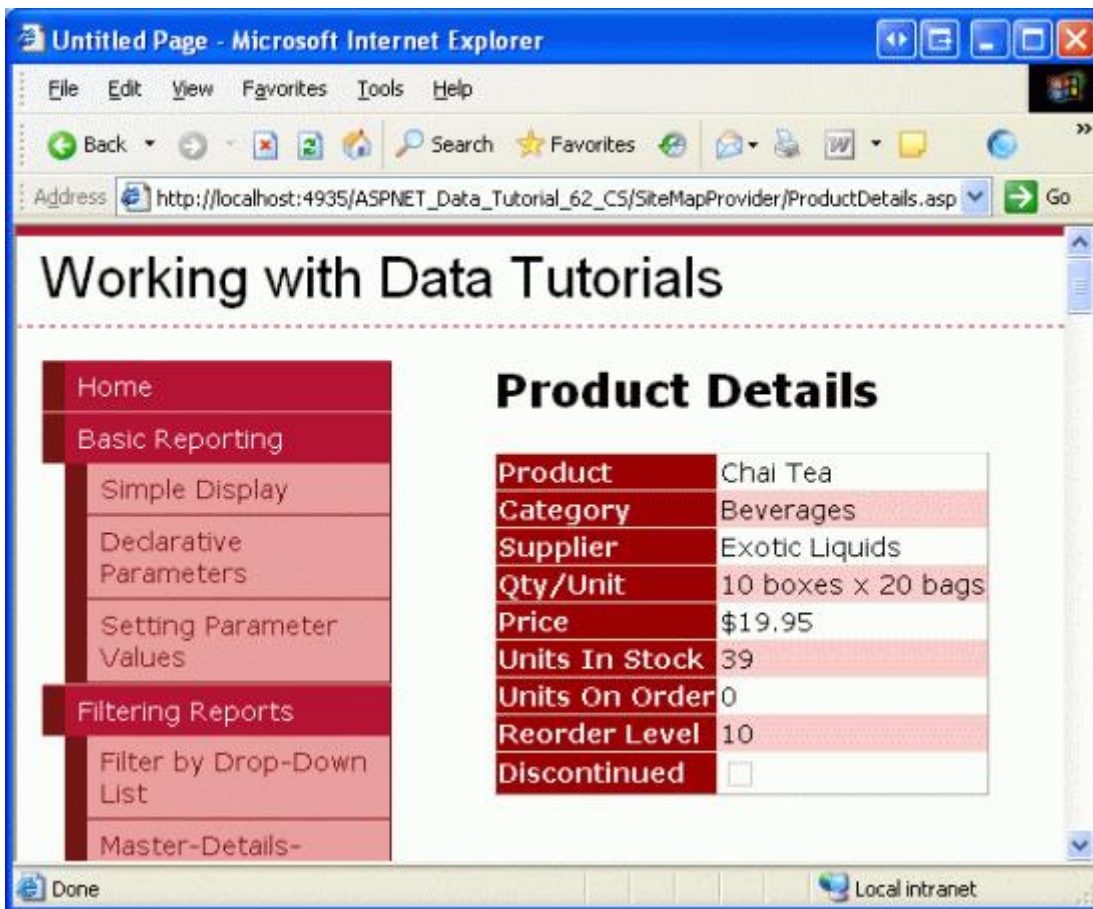
```

        SortExpression="ReorderLevel" />
        <asp:CheckBoxField DataField="Discontinued" HeaderText="Discontinued"
            SortExpression="Discontinued" />
    </Fields>
</asp:DetailsView>

<asp:ObjectDataSource ID="ProductDataSource" runat="server"
    OldValuesParameterFormatString="original_{0}"
    SelectMethod="GetProductByProductID" TypeName="ProductsBLL">
    <SelectParameters>
        <asp:QueryStringParameter Name="productID"
            QueryStringField="ProductID" Type="Int32" />
    </SelectParameters>
</asp:ObjectDataSource>

```

To test this page, return to `Default.aspx` and click on “View Products” for the Beverages category. From the listing of beverage products, click on the “View Details” link for Chai Tea. This will take you to `ProductDetails.aspx?ProductID=1`, which shows a Chai Tea’s details (see Figure 14).



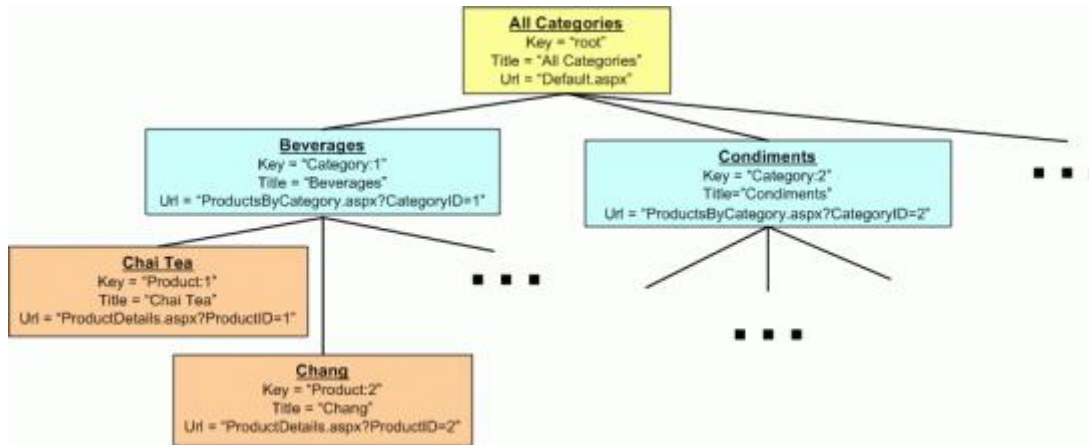
**Figure 14: Chai Tea’s Supplier, Category, Price, and Other Information is Displayed**

## Step 5: Understanding the Inner Workings of a Site Map Provider

The site map is represented in the web server’s memory as a collection of `SiteMapNode` instances that form a hierarchy. There must be exactly one root, all non-root nodes must have exactly one parent node, and all nodes

may have an arbitrary number of children. Each `SiteMapNode` object represents a section in the website's structure; these sections commonly have a corresponding web page. Consequently, the [SiteMapNode class](#) has properties like `Title`, `Url`, and `Description`, which provide information for the section the `SiteMapNode` represents. There is also a `Key` property that uniquely identifies each `SiteMapNode` in the hierarchy, as well as properties used to establish this hierarchy – `ChildNodes`, `ParentNode`, `NextSibling`, `PreviousSibling`, and so forth.

Figure 15 shows the general site map structure from Figure 1, but with the implementation details sketched out in finer detail.



**Figure 15: Each `SiteMapNode` has Properties Like `Title`, `Url`, `Key`, and So On**

The site map is accessible through the [SiteMap class](#) in the [System.Web namespace](#). This class's `RootNode` property returns the site map's root `SiteMapNode` instance; `CurrentNode` returns the `SiteMapNode` whose `Url` property matches the URL of the currently requested page. This class is used internally by ASP.NET 2.0's navigation Web controls.

When the `SiteMap` class's properties are accessed, it must serialize the site map structure from some persistent medium into memory. However, the site map serialization logic is not hard coded into the `SiteMap` class. Instead, at runtime the `SiteMap` class determines which site map *provider* to use for serialization. By default, the [XmlSiteMapProvider class](#) is used, which reads the site map's structure from a properly-formatted XML file. However, with a little bit of work we can create our own custom site map provider.

All site map providers must be derived from the [SiteMapProvider class](#), which includes the essential methods and properties needed for site map providers, but omits many of the implementation details. A second class, [StaticSiteMapProvider](#), extends the `SiteMapProvider` class and contains a more robust implementation of the needed functionality. Internally, the `StaticSiteMapProvider` stores the `SiteMapNode` instances of the site map in a `Hashtable` and provides methods like `AddNode(child, parent)`, `RemoveNode(siteMapNode)`, and `Clear()` that add and remove `SiteMapNodes` to the internal `Hashtable`. `XmlSiteMapProvider` is derived from `StaticSiteMapProvider`.

When creating a custom site map provider that extends `StaticSiteMapProvider`, there are two abstract methods that must be overridden: [BuildSiteMap](#) and [GetRootNodeCore](#). `BuildSiteMap`, as its name implies, is responsible for loading the site map structure from persistent storage and constructing it in memory. `GetRootNodeCore` returns the root node in the site map.

Before a web application can use a site map provider it must be registered in the application's configuration. By default, the `XmlSiteMapProvider` class is registered using the name `AspNetXmlSiteMapProvider`. To register



additional site map providers, add the following markup to `Web.config`:

```
<configuration>
  <system.web>
    ...

    <siteMap defaultProvider="defaultProviderName">
      <providers>
        <add name="name" type="type" />
      </providers>
    </siteMap>
  </system.web>
</configuration>
```

The *name* value assigns a human-readable name to the provider while *type* specifies the fully-qualified type name of the site map provider. We'll explore concrete values for the *name* and *type* values in Step 7, after we've created our custom site map provider.

The site map provider class is instantiated the first time it is accessed from the `SiteMap` class and remains in memory for the lifetime of the web application. Since there is only one instance of the site map provider that may be invoked from multiple, concurrent web site visitors, it is imperative that the provider's methods be *thread-safe*.

For performance and scalability reasons, it's important that we cache the in-memory site map structure and return this cached structure rather than recreating it every time the `BuildSiteMap` method is invoked. `BuildSiteMap` may be called several times per page request per user, depending on the navigation controls in use on the page and the depth of the site map structure. In any case, if we do not cache the site map structure in `BuildSiteMap` then each time it is invoked we would need to re-retrieve the product and category information from the architecture (which would result in a query to the database). As we discussed in the previous caching tutorials, cached data can become stale. To combat this, we can use either time- or SQL cache dependency-based expiries.

**Note:** A site map provider may optionally override the [Initialize method](#). `Initialize` is invoked when the site map provider is first instantiated and is passed any custom attributes assigned to the provider in `Web.config` in the `<add>` element like: `<add name="name" type="type" customAttribute="value" />`. It is useful if you want to allow a page developer to specify various site map provider-related settings without having to modify the provider's code. For example, if we were reading the category and products data directly from the database as opposed to through the architecture, we'd likely want to let the page developer specify the database connection string through `Web.config` rather than using a hard coded value in the provider's code. The custom site map provider we'll build in Step 6 does not override this `Initialize` method. For an example of using the `Initialize` method, refer to [Jeff Prosise's Storing Site Maps in SQL Server](#) article.

## Step 6: Creating the Custom Site Map Provider

To create a custom site map provider that builds the site map from the categories and products in the Northwind database, we need to create a class that extends `StaticSiteMapProvider`. In Step 1 I asked you to add a `CustomProviders` folder in the `App_Code` folder - add a new class to this folder named `NorthwindSiteMapProvider`. Add the following code to the `NorthwindSiteMapProvider` class:

```
using System;
using System.Data;
using System.Configuration;
using System.Web;
using System.Web.Security;
using System.Web.UI;
```

```

using System.Web.UI.WebControls;
using System.Web.UI.WebControls.WebParts;
using System.Web.UI.HtmlControls;
using System.Web.Caching;

public class NorthwindSiteMapProvider : StaticSiteMapProvider
{
    private readonly object siteMapLock = new object();
    private SiteMapNode root = null;
    public const string CacheDependencyKey =
        "NorthwindSiteMapProviderCacheDependency";

    public override SiteMapNode BuildSiteMap()
    {
        // Use a lock to make this method thread-safe
        lock (siteMapLock)
        {
            // First, see if we already have constructed the
            // rootNode. If so, return it...
            if (root != null)
                return root;

            // We need to build the site map!

            // Clear out the current site map structure
            base.Clear();

            // Get the categories and products information from the database
            ProductsBLL productsAPI = new ProductsBLL();
            Northwind.ProductsDataTable products = productsAPI.GetProducts();

            // Create the root SiteMapNode
            root = new SiteMapNode(
                this, "root", "~/SiteMapProvider/Default.aspx", "All Categories");
            AddNode(root);

            // Create SiteMapNodes for the categories and products
            foreach (Northwind.ProductsRow product in products)
            {
                // Add a new category SiteMapNode, if needed
                string categoryKey, categoryName;
                bool createUrlForCategoryNode = true;
                if (product.IsCategoryIDNull())
                {
                    categoryKey = "Category:None";
                    categoryName = "None";
                    createUrlForCategoryNode = false;
                }
                else
                {
                    categoryKey = string.Concat("Category:", product.CategoryID);
                    categoryName = product.CategoryName;
                }

                SiteMapNode categoryNode = FindSiteMapNodeFromKey(categoryKey);

                // Add the category SiteMapNode if it does not exist
            }
        }
    }
}

```

```

        if (categoryNode == null)
        {
            string productsByCategoryUrl = string.Empty;
            if (createUrlForCategoryNode)
                productsByCategoryUrl =
                    "~/SiteMapProvider/ProductsByCategory.aspx?CategoryID="
                    + product.CategoryID;

            categoryNode = new SiteMapNode(
                this, categoryKey, productsByCategoryUrl, categoryName);
            AddNode(categoryNode, root);
        }

        // Add the product SiteMapNode
        string productUrl =
            "~/SiteMapProvider/ProductDetails.aspx?ProductID="
            + product.ProductID;
        SiteMapNode productNode = new SiteMapNode(
            this, string.Concat("Product:", product.ProductID),
            productUrl, product.ProductName);
        AddNode(productNode, categoryNode);
    }

    // Add a "dummy" item to the cache using a SqlCacheDependency
    // on the Products and Categories tables
    System.Web.Caching.SqlCacheDependency productsTableDependency =
        new System.Web.Caching.SqlCacheDependency("NorthwindDB", "Products");
    System.Web.Caching.SqlCacheDependency categoriesTableDependency =
        new System.Web.Caching.SqlCacheDependency("NorthwindDB", "Categories");

    // Create an AggregateCacheDependency
    System.Web.Caching.AggregateCacheDependency aggregateDependencies =
        new System.Web.Caching.AggregateCacheDependency();
    aggregateDependencies.Add(productsTableDependency, categoriesTableDependency)

    // Add the item to the cache specifying a callback function
    HttpRuntime.Cache.Insert(
        CacheDependencyKey, DateTime.Now, aggregateDependencies,
        Cache.NoAbsoluteExpiration, Cache.NoSlidingExpiration,
        CacheItemPriority.Normal,
        new CacheItemRemovedCallback(OnSiteMapChanged));

    // Finally, return the root node
    return root;
}
}

protected override SiteMapNode GetRootNodeCore()
{
    return BuildSiteMap();
}

protected void OnSiteMapChanged(string key, object value, CacheItemRemovedReason reason)
{
    lock (siteMapLock)
    {

```

```

        if (string.Compare(key, CacheDependencyKey) == 0)
        {
            // Refresh the site map
            root = null;
        }
    }
}

public DateTime? CachedDate
{
    get
    {
        return HttpRuntime.Cache[CacheDependencyKey] as DateTime?;
    }
}
}

```

Let's start with exploring this class's `BuildSiteMap` method, which starts with a [lock statement](#). The `lock` statement only allows one thread at a time to enter, thereby serializing access to its code and preventing two concurrent threads from stepping on one another's toes.

The class-level `SiteMapNode` variable `root` is used to cache the site map structure. When the site map is constructed for the first time, or for the first time after the underlying data has been modified, `root` will be `null` and the site map structure will be constructed. The site map's root node is assigned to `root` during the construction process so that the next time this method is called, `root` will not be `null`. Consequently, so long as `root` is not `null` the site map structure will be returned to the caller without having to recreate it.

If `root` is `null`, the site map structure is created from the product and category information. The site map is built by creating the `SiteMapNode` instances and then forming the hierarchy through calls to the `StaticSiteMapProvider` class's `AddNode` method. `AddNode` performs the internal bookkeeping, storing the assorted `SiteMapNode` instances in a `Hashtable`. Before we start constructing the hierarchy, we start by calling the `Clear` method, which clears out the elements from the internal `Hashtable`. Next, the `ProductsBLL` class's `GetProducts` method and the resulting `ProductsDataTable` are stored in local variables.

The site map's construction begins by creating the root node and assigning it to `root`. The overload of the [SiteMapNode's constructor](#) used here and throughout this `BuildSiteMap` is passed the following information:

- A reference to the site map provider (`this`).
- The `SiteMapNode`'s `Key`. This required value must be unique for each `SiteMapNode`.
- The `SiteMapNode`'s `Url`. `Url` is optional, but if provided, each `SiteMapNode`'s `Url` value must be unique.
- The `SiteMapNode`'s `Title`, which is required.

The `AddNode(root)` method call adds the `SiteMapNode` `root` to the site map as the root. Next, each `ProductRow` in the `ProductsDataTable` is enumerated. If there already exists a `SiteMapNode` for the current product's category, it is referenced. Otherwise, a new `SiteMapNode` for the category is created and added as a child of the `SiteMapNode` `root` through the `AddNode(categoryNode, root)` method call. After the appropriate category `SiteMapNode` node has been found or created, a `SiteMapNode` is created for the current product and added as a child of the category `SiteMapNode` via `AddNode(productNode, categoryNode)`. Note that the category `SiteMapNode`'s `Url` property value is `~/SiteMapProvider/ProductsByCategory.aspx?CategoryID=categoryID` while the product `SiteMapNode`'s `Url` property is assigned `~/SiteMapNode/ProductDetails.aspx?ProductID=productID`.

**Note:** Those products that have a database `NULL` value for their `CategoryID` are grouped under a category `SiteMapNode` whose `Title` property is set to "None" and whose `Url` property is set to an empty string. I

decided to set `Url` to an empty string since the `ProductBLL` class's `GetProductsByCategory(categoryID)` method currently lacks the capability to return just those products with a `NULL` `CategoryID` value. Also, I wanted to demonstrate how the navigation controls render a `SiteMapNode` that lacks a value for its `Url` property. I encourage you to extend this tutorial so that the “None” `SiteMapNode`'s `Url` property points to `ProductsByCategory.aspx`, yet only displays the products with `NULL` `CategoryID` values.

After constructing the site map, an arbitrary object is added to the data cache using a SQL cache dependency on the `Categories` and `Products` tables through an `AggregateCacheDependency` object. We explored using SQL cache dependencies in the preceding tutorial, *Using SQL Cache Dependencies*. The custom site map provider, however, uses an overload of the data cache's `Insert` method that we've yet to explore. This overload accepts as its final input parameter a delegate that is called when the object is removed from the cache. Specifically, we pass in a new [CacheItemRemovedCallback delegate](#) that points to the `OnSiteMapChanged` method defined further down in the `NorthwindSiteMapProvider` class.

**Note:** The in-memory representation of the site map is cached through the class-level variable `root`. Since there is only one instance of the custom site map provider class and since that instance is shared among all threads in the web application, this class variable serves as a cache. The `BuildSiteMap` method also uses the data cache, but only as a means to receive notification when the underlying database data in the `Categories` or `Products` tables changes. Note that the value put into the data cache is just the current date and time. The actual site map data is *not* put in the data cache.

The `BuildSiteMap` method completes by returning the root node of the site map.

The remaining methods are fairly straightforward. `GetRootNodeCore` is responsible for returning the root node. Since `BuildSiteMap` returns the root, `GetRootNodeCore` simply returns `BuildSiteMap`'s return value. The `OnSiteMapChanged` method sets `root` back to `null` when the cache item is removed. With `root` set back to `null`, the next time `BuildSiteMap` is invoked, the site map structure will be rebuilt. Lastly, the `CachedDate` property returns the date and time value stored in the data cache, if such a value exists. This property can be used by a page developer to determine when the site map data was last cached.

## Step 7: Registering the `NorthwindSiteMapProvider`

In order for our web application to use the `NorthwindSiteMapProvider` site map provider created in Step 6, we need to register it in the `<siteMap>` section of `Web.config`. Specifically, add the following markup within the `<system.web>` element in `Web.config`:

```
<siteMap defaultProvider="AspNetXmlSiteMapProvider">
  <providers>
    <add name="Northwind" type="NorthwindSiteMapProvider" />
  </providers>
</siteMap>
```

This markup does two things: first, it indicates that the built-in `AspNetXmlSiteMapProvider` is the default site map provider; second, it registers the custom site map provider created in Step 6 with the human-friendly name “Northwind”.

**Note:** For site map providers located in the application's `App_Code` folder, the value of the `type` attribute is simply the class name. Alternatively, the custom site map provider could have been created in a separate Class Library project with the compiled assembly placed in the web application's `/Bin` directory. In that case, the `type` attribute value would be “*Namespace.ClassName, AssemblyName*”.

After updating `Web.config`, take a moment to view any page from the tutorials in a browser. Note that the

navigation interface on the left still shows the sections and tutorials defined in `Web.sitemap`. This is because we left `AspNetXmlSiteMapProvider` as the default provider. In order to create a navigation user interface element that uses the `NorthwindSiteMapProvider`, we'll need to explicitly specify that the "Northwind" site map provider should be used. We'll see how to accomplish this in Step 8.

## Step 8: Displaying Site Map Information Using the Custom Site Map Provider

With the custom site map provider created and registered in `Web.config`, we're ready to add navigation controls to the `Default.aspx`, `ProductsByCategory.aspx`, and `ProductDetails.aspx` pages in the `SiteMapProvider` folder. Start by opening the `Default.aspx` page and drag a `SiteMapPath` from the Toolbox onto the Designer. The `SiteMapPath` control is located in the Navigation section of the Toolbox.

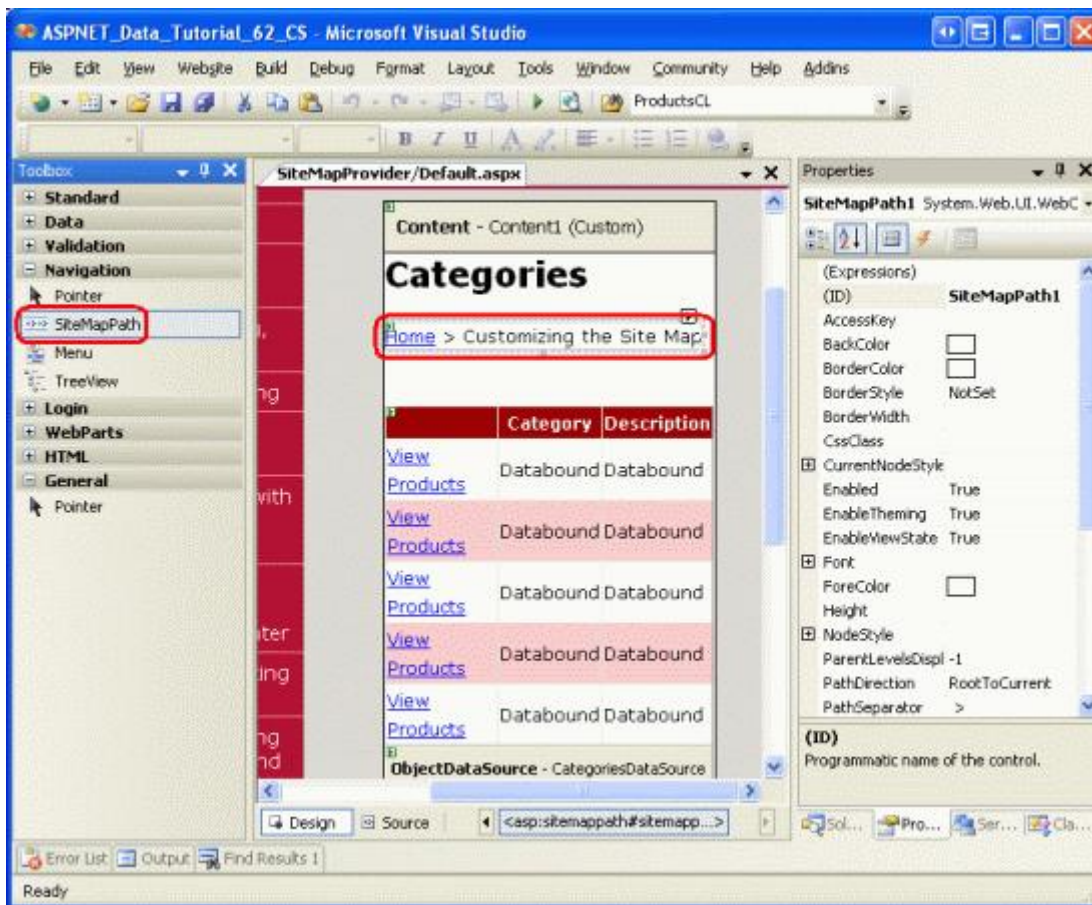
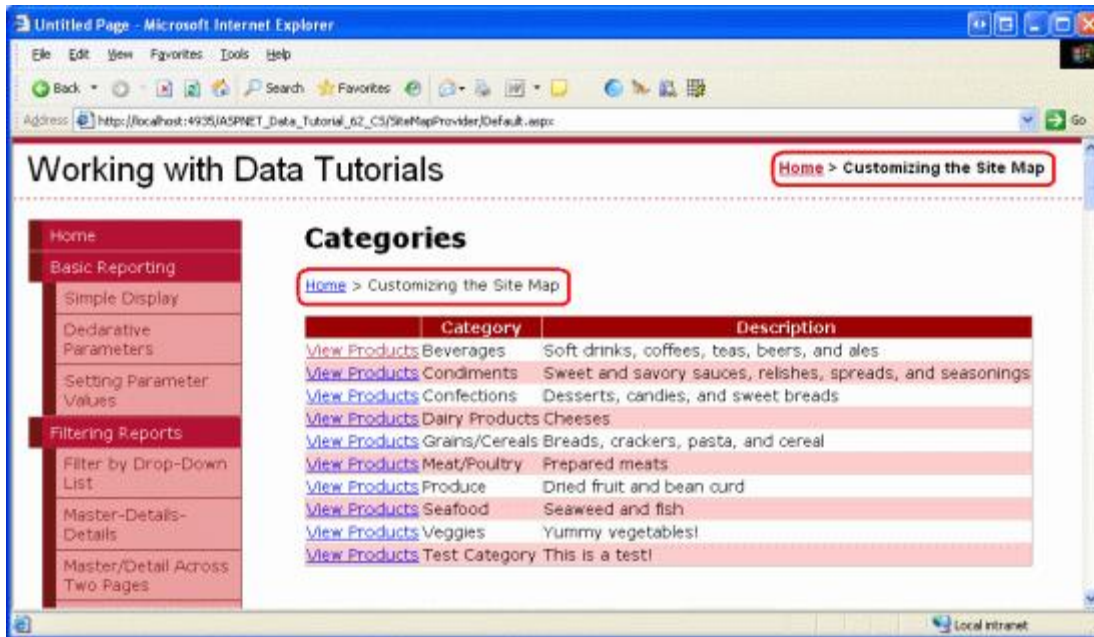


Figure 16: Add a `SiteMapPath` to `Default.aspx`

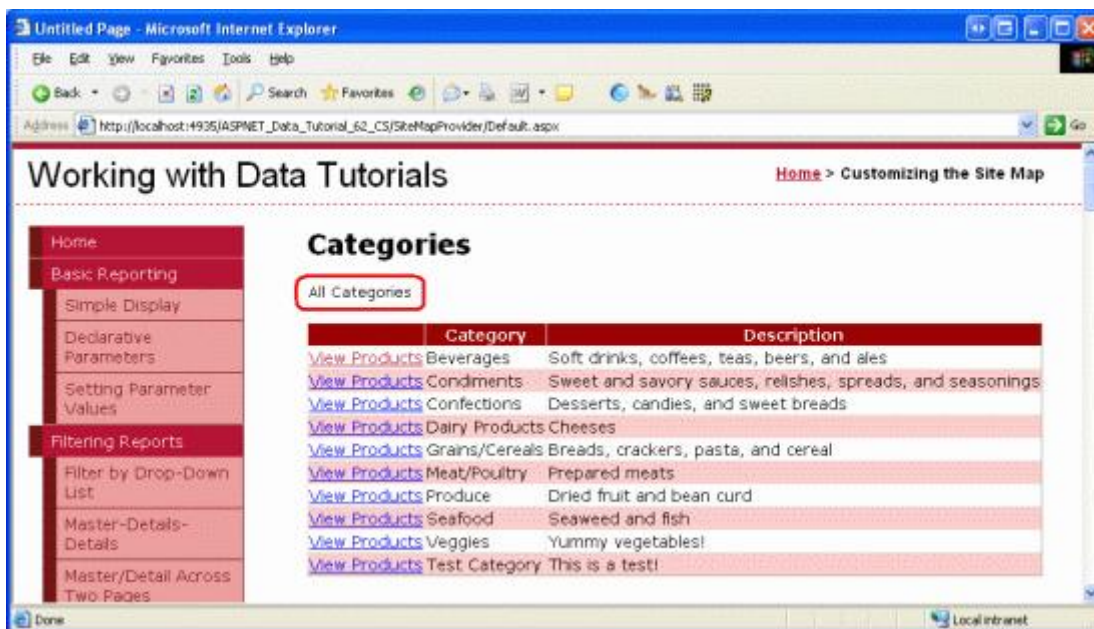
The `SiteMapPath` control displays a breadcrumb, indicating the current page's location within the site map. We added a `SiteMapPath` to the top of the master page back in the *Master Pages and Site Navigation* tutorial.

Take a moment to view this page through a browser. The `SiteMapPath` added in Figure 16 uses the default site map provider, pulling its data from `Web.sitemap`. Therefore, the breadcrumb shows "Home > Customizing the Site Map", just like the breadcrumb in the upper-right corner.



**Figure 17: The Breadcrumb Uses the Default Site Map Provider**

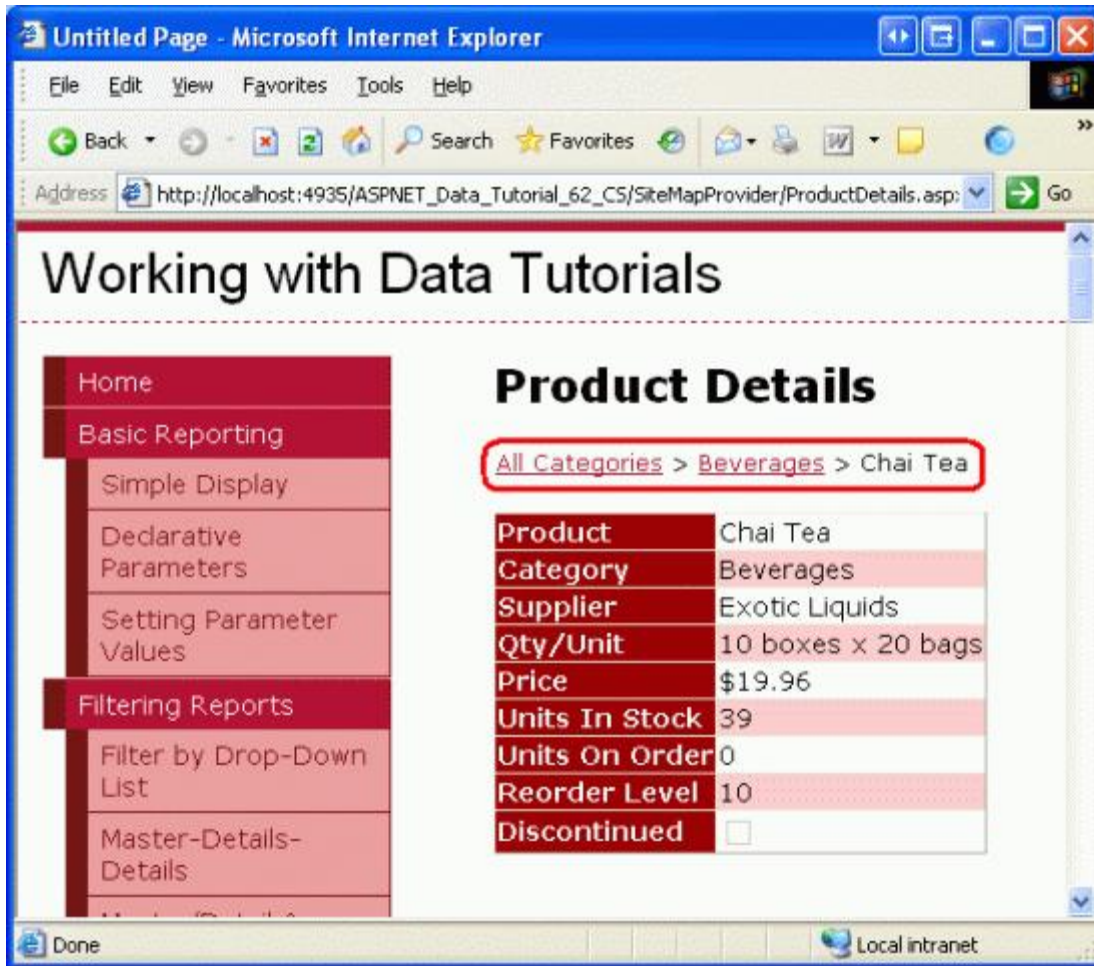
To have the SiteMapPath added in Figure 16 use the custom site map provider we created in Step 6, set its [SiteMapProvider](#) property to “Northwind”, the name we assigned to the NorthwindSiteMapProvider in Web.config. Unfortunately, the Designer continues to use the default site map provider, but if you visit the page through a browser after making this property change you’ll see that the breadcrumb now uses the custom site map provider.



**Figure 18: The Breadcrumb Now Uses the Custom Site Map Provider NorthwindSiteMapProvider**

The SiteMapPath control displays a more functional user interface in the `ProductsByCategory.aspx` and

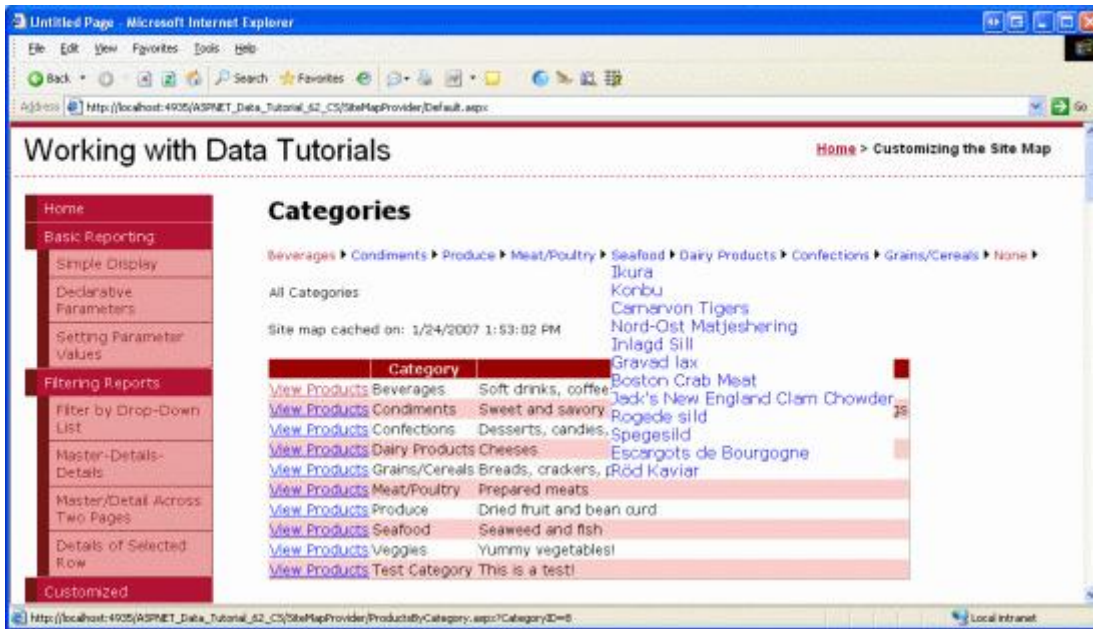
ProductDetails.aspx pages. Add a SiteMapPath to these pages, setting the SiteMapProvider property in both to “Northwind”. From Default.aspx click on the “View Products” link for Beverages, and then on the “View Details” link for Chai Tea. As Figure 19 shows, the breadcrumb includes the current site map section (“Chai Tea”) and its ancestors: “Beverages” and “All Categories”.



**Figure 19: The Breadcrumb Now Uses the Custom Site Map Provider NorthwindSiteMapProvider**

Other navigation user interface elements can be used in addition to the SiteMapPath, such as the Menu and TreeView controls. The Default.aspx, ProductsByCategory.aspx, and ProductDetails.aspx pages in the download for this tutorial, for example, all include Menu controls (see Figure 20). See [Examining ASP.NET 2.0's Site Navigation Features](#) and the [Using Site Navigation Controls](#) section of the [ASP.NET 2.0 QuickStarts](#) for a more in-depth look at the navigation controls and site map system in ASP.NET 2.0.





**Figure 20: The Menu Control Lists Each of the Categories and Products**

As mentioned earlier in this tutorial, the site map structure can be accessed programmatically through the `SiteMap` class. The following code returns the root `SiteMapNode` of the default provider:

```
SiteMapNode root = SiteMap.RootNode;
```

Since the `AspNetXmlSiteMapProvider` is the default provider for our application, the above code would return the root node defined in `Web.sitemap`. To reference a site map provider other than the default, use the `SiteMap` class's [Providers property](#) like so:

```
SiteMapNode root = SiteMap.Providers["name"].RootNode;
```

Where *name* is the name of the custom site map provider (“Northwind”, for our web application).

To access a member specific to a site map provider, use `SiteMap.Providers["name"]` to retrieve the provider instance and then cast it to the appropriate type. For example, to display the `NorthwindSiteMapProvider`'s `CachedDate` property in an ASP.NET page, use the following code:

```
NorthwindSiteMapProvider customProvider =
    SiteMap.Providers["Northwind"] as NorthwindSiteMapProvider;
if (customProvider != null)
{
    DateTime? lastCachedDate = customProvider.CachedDate;

    if (lastCachedDate != null)
        LabelID.Text = "Site map cached on: " + lastCachedDate.Value.ToString();
    else
        LabelID.Text = "The site map is being reconstructed!";
}
```

**Note:** Be sure to test out the SQL cache dependency feature. After visiting the `Default.aspx`,

`ProductsByCategory.aspx`, and `ProductDetails.aspx` pages, go to one of the tutorials in the Editing, Inserting, and Deleting section and edit the name of a category or product. Then return to one of the pages in the `SiteMapProvider` folder. Assuming enough time has passed for the polling mechanism to note the change to the underlying database, the site map should be updated to show the new product or category name.

## Summary

ASP.NET 2.0's site map features includes a `SiteMap` class, a number of built-in navigation Web controls, and a default site map provider that expects the site map information persisted to an XML file. In order to use site map information from some other source – such as from a database, the application's architecture, or a remote Web service – we need to create a custom site map provider. This involves creating a class that derives, directly or indirectly, from the `SiteMapProvider` class.

In this tutorial we saw how to create a custom site map provider that based the site map on the product and category information culled from the application architecture. Our provider extended the `StaticSiteMapProvider` class and entailed creating a `BuildSiteMap` method that retrieved the data, constructed the site map hierarchy, and cached the resulting structure in a class-level variable. We used a SQL cache dependency with a callback function to invalidate the cached structure when the underlying `Categories` or `Products` data is modified.

Happy Programming!

## Further Reading

For more information on the topics discussed in this tutorial, refer to the following resources:

- [Storing Site Maps in SQL Server](#) and [The SQL Site Map Provider You've Been Waiting For](#)
- [A Look at ASP.NET 2.0's Provider Model](#)
- [The Provider Toolkit](#)
- [Examining ASP.NET 2.0's Site Navigation Features](#)

## About the Author

[Scott Mitchell](#), author of seven ASP/ASP.NET books and founder of [4GuysFromRolla.com](#), has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at [mitchell@4GuysFromRolla.com](mailto:mitchell@4GuysFromRolla.com), or via his blog, which can be found at <http://ScottOnWriting.NET>.

## Special Thanks To...

This tutorial series was reviewed by many helpful reviewers. Lead reviewers for this tutorial were Dave Gardner, Zack Jones, Teresa Murphy, and Bernadette Leigh. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at [mitchell@4GuysFromRolla.com](mailto:mitchell@4GuysFromRolla.com).