Getting Started with SubSonic

By Scott Kuhl (http://www.geekswithblogs.net/scottkuhl)

SubSonic is an open-source toolset, created by Rob Conery, as an attempt to put the fun back into programming and just get the job done. Inspired by Ruby on Rails, SubSonic takes a minimalist approach to coding and emphasizes convention over configuration. While it takes its inspiration from Ruby on Rails, it is not a port of it. (Check out MonoRail it that's what you're looking for.) Instead, SubSonic takes the best ideas of Ruby on Rails and adapts them into the already existing ASP.NET framework. Currently SubSonic, version 1.05, implements two core features:

ActiveRecord

This design pattern in its simplest form is one class per database table, one object per database row. SubSonic includes a build-time code generator to implement this pattern that acts as an object-relational mapper eliminating the need to write SQL. It also includes a dynamic query tool and simple store procedure support to extend the model when needed.

Scaffolding

Admin pages are a chore that scaffolding helps remove. Simply point a scaffold control at a table and you get the standard grid view and detail view to find and update data in the underlying table. While not meant to ever be shown to users, it makes a nice, quick and easy developer tool.

Requirements

SubSonic will work fine with Visual Web Developer 2005 Express Edition and SQL Server 2005 Express Edition, so you can get started without dropping a dime. You can also use MySQL or any database that can be accessed through Enterprise Library for .NET Framework 2.0, but SQL Server is probably the most likely setup.

Note: The sample web site included with the SubSonic source code includes an SQL script to create the Northwind database. This article will use that database when examples are needed.

Setup

Setup is easy, just download SubSonic from CodePlex and reference SubSonic.dll found in the bin directory. Alternatively, you can open the solution and compile a release build yourself.

(You will need Visual Studio 2005 Standard Edition to open the solution because it also includes a sample web site or you can use Visual C# 2005 Express Edition to open just the project.)

Configuration

SubSonic requires a minimal amount of configuration to get going.

SubSonic Configuration Section

Start by adding a SubSonic configuration section inside the configuration tag in the web.config file. This default configuration should work for most projects.

```
<configSections>
    <section name="SubSonicService" type="SubSonic.SubSonicSection, SubSonic"
    allowDefinition="MachineToApplication" restartOnExternalChanges="true"
    requirePermission="false"/>
</configSections>
```

Data Provider

Second, you will need to setup a data provider. Three are currently supported by SubSonic: SQL Server, MySQL and Enterprise Library. The following are sample configurations for each of these. This information is also added inside the configuration tag.

```
<SubSonicService defaultProvider="SqlDataProvider" spClassName="SPs"
fixPluralClassNames="true">
    <providers>
        <add name="SqlDataProvider" type="SubSonic.SqlDataProvider, SubSonic"
        connectionStringName="NorthwindConnection"/>
        <add name="ELib2DataProvider" type="ActionPack.ELib2DataProvider,
        ActionPack" connectionStringName="NorthwindSQL"/>
        <add name="MySqlDataProvider" type="ActionPack.MySqlDataProvider,
        ActionPack" connectionStringName="NorthwindMySQLConnection"/>
```

```
        </providers>
    </SubSonicService>
```

There are five values that can be set in the SubSonicService tag.

- **defaultProvider** - Multiple providers can be setup in the configuration. This value indicates which provider to use.
- **fixPluralClassNames** - SubSonic can remove the plural characters from the end of table names to make class names more consistent. For example, the Products table would produce a Product class.
- **generatedNamespace** - By default all classes generated will be part of the project's global namespace. This value overrides that behavior and includes all classes in the given namespace. For example, by setting this to Northwind you would get Northwind.Product.
- **spClassName** - Each stored procedure will generate a method of the same name. The value will be the class these methods are included under. For example, by setting this to SPs the CustOrderHist stored procedure would be SPs.CustOrderHist. Using the above namespace example in conjunction with this value would produce Northwind.SPs.CustOrderHist.
- **templateDirectory** - It is possible to override the code generated by SubSonic. This directory would contain the code templates to override the default templates supplied. This will be covered in greater detail later when discussing Code Generation Templates.
- **useSPs** - If you do not want a class generated for stored procedures, set this value to false.

Database Connection String

Third, you need to define a database connection string.

```
    <connectionStrings>
        <add name="NorthwindConnection" connectionString="Data
        Source=localhost\SQLExpress; Database=Northwind; Integrated Security=true;"/>
    </connectionStrings>
```

Build Provider Configuration

Fourth, you need to setup a build provider to create the auto generated classes. This needs to be added to the compilation tag.

```
    <buildProviders>
        <add extension=".abp" type="SubSonic.BuildProvider, SubSonic"/>
    </buildProviders>
```

Build Provider Definition

Last, you need to create an .abp file for this build provider to use. You do this by adding a text file named Builder.abp to the App_Code folder. Inside this file you indicate which database tables should have auto generate classes. If you want all tables, just enter *, otherwise, list the tables one per line.

Summary

In summary, these are the items you need to configure.

- SubSonic Configuration Section
- Data Provider
- Database Connection String
- Build Provider Configuration
- Build Provider Definition

Here is a sample web.config with all the SubSonic required values defined. You can also use the web.config included in the sample web site downloaded along with the SubSonic source code as a starting point, which is where these sample values are derived from.

```
<?xml version="1.0"?>

<configuration>

    <configSections>

        <section name="SubSonicService" type="SubSonic.SubSonicSection, SubSonic"
        allowDefinition="MachineToApplication" restartOnExternalChanges="true"
        requirePermission="false"/>

    </configSections>

    <appSettings/>

    <connectionStrings>

        <add name="NorthwindConnection" connectionString="Data
        Source=localhost\SQLExpress; Database=Northwind; Integrated
        Security=true;"/>

    </connectionStrings>

    <SubSonicService defaultProvider="SqlDataProvider" spClassName="SPs"
    fixPluralClassNames="true">

        <providers>

            <add name="SqlDataProvider" type="SubSonic.SqlDataProvider,
            SubSonic" connectionStringName="NorthwindConnection"/>

        </providers>

    </SubSonicService>

    <system.web>

        <compilation debug="true" defaultLanguage="C#">

            <buildProviders>

                <add extension=".abp" type="SubSonic.BuildProvider,
                SubSonic"/>

            </buildProviders>

            <assemblies>
```

```
                <add assembly="System.Management, Version=2.0.0.0,
                Culture=neutral, PublicKeyToken=B03F5F7F11D50A3A"/>

                <add assembly="System.Data.OracleClient, Version=2.0.0.0,
                Culture=neutral, PublicKeyToken=B77A5C561934E089"/>

                <add assembly="System.Configuration.Install, Version=2.0.0.0,
                Culture=neutral, PublicKeyToken=B03F5F7F11D50A3A"/>

                <add assembly="MySql.Data, Version=1.0.7.30072,
                Culture=neutral, PublicKeyToken=C5687FC88969C44D"/>
            </assemblies>
        </compilation>
    <authentication mode="Windows"/>
    </system.web>
</configuration>
```

To make sure everything is working, build the web site, then open up the Class View window and you should see class names that match your tables. For example, using the Northwind database, you should now see classes named Category, Customer and Employee.

Trust Level

Build providers will not work in a medium trust environment, they need full trust. This could be a problem if you plan on having someone else host your web application. Most hosting providers are not set up to run in a full trust environment. As a work around, there are two class generators provided in the SubSonic Starter Kit, which will be covered in detail later, which output code in text form that can be included and compiled directly into your application.

Two current exceptions to this hosting rule are Ultima Hosts and Discount ASP.net. It's a good idea to check with your hosting provider before beginning a SubSonic project.

Classes

At its heart, the most useful feature of SubSonic is the auto generated classes. As stated earlier, there will be one class for each table in the database. The next several sections will take an in-depth look at the functionality provided. The examples in the section will use the Products table from the Northwind database as their foundation.

Extending the Model

The creator of SubSonic anticipated that not any one model would accommodate everyone's needs. Beyond providing the source code for you to change, there is another way you can adapt and extend the generated model without altering the core code base. SubSonic creates partial classes, a feature introduced in .NET 2.0 to make code generation tools more accessible. Partial classes allows you to add additional properties and methods without altering generated code and worrying about the generation process overwriting your changes.

Constructors

Creating a new object is straight forward.

```
Product product = new Product();
```

There are also two overloaded constructors that are used to load an existing object from the database. These are covered below when discussing the many ways SubSonic has of retrieving data.

Properties

Each object will have one property for each column in the table it is derived from. So a Product object would contain the following properties: CategoryID, Discontinued, ProductID, ProductName, QuantityPerUnit, ReorderLevel, SupplierID, UnitPrice, UnitsInStock and UnitsOnOrder.

Rules Enforcement

None of the properties enforce any authorization, validation or business rules. That is left up to you. How to implement these rules effectively is discussed later under Business Rules.

Object Key

Each object will have a key property, in the case of Product that key is ProductID. SubSonic uses this property as the object ID because ProductID is defined as the primary key in the database. (SubSonic will not work with tables that do not have a primary key.) Generally tables will have one of three different types of primary keys: GUID, an auto incrementing integer or a natural key. Any key that can be represented as a GUID, integer or string is supported, so all of these are covered. The one thing that is missing is support for multiple column primary keys.

State

There are three properties that help you manage the current state of an object:

- IsDirty - Does the object have changes that have not yet been saved to the database?
- IsLoaded - Was the object loaded from the database?
- IsNew - Was the object created in memory? In other words, this object does not exist in the database.

Columns

Sometimes you will need to pass in the name of a column in string format as a parameter to a method, as you will see later in the FetchByParameter methods. Instead of using a string literal, you use the static Columns collection found on each class.

```
Product.Columns.ProductName
```

Its also possible to get the column value by calling the GetColumnValue method found on each object, passing in the column name.

```
product.GetColumnValue(Product.Columns.ProductName);
```

Retrieving a Single Object

It is possible to load data at the object level using one of three overloaded **Load** methods by passing in either a DataRow, DataTable or IDataReader. But you are more likely to use the static FetchByID method or an overloaded constructor.

FetchByID

FetchByID takes has one parameter, the primary key value of the object in GUID, integer or string format, and returns a populated object.

```
Product product = Product.FetchByID(id);
```

If the object is not found, SubSonic will not throw an error. Instead you should check to see if the key property matches the passed in ID to determine if the object was found.

```
int id = 1;
Product product = Product.FetchByID(id);
if (product.ProductID == id)
{
    // Success
}
else
{
    // Not Found
}
```

Constructor

To do the same thing using the overloaded constructor method that takes the primary key looks like this.

```
Product product = new Product(id);
```

By using another overloaded constructor that takes a column name and value as the parameters, it is possible to load the object based on another uniquely identifying column instead of the primary key.

```
Product product = new Product(Product.Columns.ProductName, "Chai");
```

Be careful when using the last form. If you use a column that does not have a unique constraint and more than one record is returned by the database, your object will be populated with the first one.

Loading and Saving State

There is one other way to load an object. Each object has a method called NewFromXML which creates an object from an XML string. This method is meant to be used with the ToXML method to write the object's state to a temporary location such as Session state.

```
Session["Product"] = product.ToXML();
product = (Product)product.NewFromXML(Session["Product"].ToString());
```

Retrieving Multiple Objects

FetchAll

The easiest way to return a list of objects is the FetchAll static method. It does just what the name says, returning a list of every object of that type in the database in IDataReader format, making it easily bindable to data controls like the GridView.

```
GridView1.DataSource = Product.FetchAll();
GridView1.DataBind();
```

You can also pass the FetchAll method a SubSonic OrderBy.

```
GridView1.DataSource =
Product.FetchAll(SubSonic.OrderBy.Desc(Product.Columns.ProductName));
GridView1.DataSource =
Product.FetchAll(SubSonic.OrderBy.Asc(Product.Columns.ProductName));
```

FetchByParameter

If you want to list a subset of data rather than the entire list you can use FetchByParameter instead. At a minimum you only need to supply a column name and value to match.

```
GridView1.DataSource =
Product.FetchByParameter(Product.Columns.SupplierID, 1);
```

Just like the FetchAll, you can apply an OrderBy.

```
GridView1.DataSource =
Product.FetchByParameter(Product.Columns.SupplierID, 1,
SubSonic.OrderBy.Desc(Product.Columns.ProductName));
```

If you would rather find a range of objects, you can use the overloaded FetchByParameter that takes a SubSonic Comparison. Again you could also choose to append an OrderBy.

```
GridView1.DataSource =
Product.FetchByParameter(Product.Columns.SupplierID,
SubSonic.Comparison.GreaterThan, 1);
```

FetchByQuery

There is also a FetchByQuery method that takes a SubSonic Query.

```
GridView1.DataSource = Product.FetchByQuery(query);
```

Queries will be discussed later along with more information on comparisons.

Find

The Find static method allows you to retrieve matching records based on an existing object. For example, if you want to find all Products with SupplierID of 1 and CategoryID of 1 you can create a new Product object with those values and pass it into the Find method.

```
Product product = new Product();
product.SupplierID = 1;
```

```
product.CategoryID = 1;
GridView1.DataSource = Product.Find(product);
```

The Find method also has the option of passing an OrderBy.

Querying

If none of the Fetch or Find methods will work for a particular situation, you can still fall back on SubSonic querying or stored procedures which are discusses later.

Updating the Database

Saving your changes to the database is as easy as calling the Save method on the object.

```
product.Save();
```

But it's important to know what is happening when you call this method. The above example does not pass the current user information. As you'll see later when discussing conventions, SubSonic has the ability to keep some basic history information. If your tables are setup to record this information, you need to pass either the User ID in integer or GUID format, or the User Name in string format.

```
product.Save(User.Identity.Name);
```

If the primary key is a GUID or auto incrementing integer and the object is new, the save process will update the key property in the object. The save process will also set the IsNew and IsDirty properties to false.

Insert and Update

It is also possible to insert or update data through static class methods.

```
Product.Insert("Product Name", 1, 1, "10", 10.00, 10, 10, 1, false);
Product.Update("Product Name", 1, 1, "10", 10.00, 10, 10, 1, false);
```

These methods do all the work of instantiating the object, setting the properties and calling the Save method. The downside is that neither method returns an auto generated primary key value.

Deleting

There are two types of deletes: logical and permanent. A logical delete does not remove the record from the database, rather a column is used to mark the record as deleted. SubSonic will treat a column named "Deleted" or "IsDeleted" as this flag. To perform a logical delete, your table must have one of those columns defined as a BIT, then the IsDeleted property will show up on the object that can be set to true. When the Save method is called, the object will be marked as deleted. There is also a static method called Delete that takes an object key value and does the same thing.

```
Product.Delete(product.ProductID);
```

Note that this example will not work with the default Northwind database since the Products table does not have either type of delete column. You need to add a "Deleted" column to make it work.

Unfortunately, SubSonic treats this purely as convention when retrieving data. You will need to filter out deleted records, for example, by using a FetchByParameter method.

Permanent deletes are easier. Just call the static Destroy method.

```
Product.Destroy(product.ProductID);
```

Business Rules

There are two places that business rules can be injected into the Save process. A PreUpdate method is called at the beginning of the Save process and a PostUpdate method is called at the end. Both of these methods are virtual methods defined in each class that take no parameters and have no return value. To define these methods, create a partial class file with the same name as the class.

```
public partial class Product
{
    protected override void PreUpdate()
    {
        // Do something
    }

    protected override void PostUpdate()
    {
        // Do something
    }
}
```

This is an easy way to add simple authorization, validation and business rules. If you need more flexibility, check out the Controller pattern implementation discussed later in Conventions.

Underlying Data

Auto generated classes may expose a simple object based interface, but below the surface the data is being held in ADO.NET DataTables. SubSonic exposes a few methods that allow you to get at the underlying structure.

The TableName property returns the name of the actual table associated with the object.

```
product.TableName;
```

The static class method GetTableSchema returns the underlying table.

```
SubSonic.TableSchema.Table tableSchema = Product.GetTableSchema();
```

Note: There is also a static property Schema that returns the same information. For the property to work, the class must be instantiated at least once. The GetTableSchema takes care of this for you. Both of these methods will be used later when examining Queries.

The Inspect method returns an HTML representation of the object using <table> markup tags. There is an overloaded version of the method that takes a single boolean parameter, useHtml, that can be set to false to return plain text.

```
string html = product.Inspect();
```

Collections

For a more traditional object-oriented approach to multiple objects you can use the auto-generated collections. Just like classes, SubSonic will also generate a collection for every table. (The naming convention for collections is ClassCollection.) These collections are implemented as Generic Lists that include some additional functionality.

Loading a Collection

The simplest way to load a collection is by calling the Load method with no parameters to retrieve every record.

```
ProductCollection products = new ProductCollection();
products.Load();
```

You can also pass the Load method an IDataReader, DataTable or SubSonic Query.

```
IDataReader reader = Product.FetchAll();
products.Load(reader);
reader.Close();
```

(Make sure you close the reader. The Load method will not do this for you.)

Ordering a Collection

You can order a collection based on a single column by calling OrderByAsc or OrderByDesc before calling the Load method.

```
ProductCollection products = new ProductCollection();
products.OrderByAsc(Product.Columns.ProductName);
products.Load();
```

Filtering a Collection

You can filter a collection by calling BetweenAnd, Where or WhereDatesBetween before calling the Load method.

```
OrderCollection orders = new OrderCollection();
orders.BetweenAnd(Order.Columns.OrderDate, new DateTime(1980, 1, 12),
DateTime.Now);
orders.Load();
```

The Where method works just like the SubSonic Query Where which is covered in the next section.

Queries

Beyond all the basic filtering available in Fetch methods and collection filtering, SubSonic also provides a way to dynamically build SQL queries. To get started, create a new Query object, using the CreateQuery method.

```
SubSonic.Query query = Product.CreateQuery();
```

or pass it the table name

```
SubSonic.Query query = new SubSonic.Query(product.TableName);
```

or pass it the table schema

```
SubSonic.Query query = new SubSonic.Query(Product.GetTableSchema());
```

The last option will prevent SubSonic from loading the table information from the database, eliminating an extra call.

Note: You can also get the table schema by calling the static method BuildTableSchema or the property Schema.

Running the Query

Executing the query follows the same rules as executing ADO.NET queries. The Execute method runs the query and returns nothing.

```
query.Execute();
```

The ExecuteDataSet method returns a DataSet.

```
GridView1.DataSource = query.ExecuteDataSet();
```

The ExecuteReader method returns a DataReader.

```
GridView1.DataSource = query.ExecuteReader();
```

And finally, the ExecuteScalar returns a single value.

```
Label1.Text = query.ExecuteScalar().ToString();
```

Ordering

You can order a query by setting the OrderBy property to either ascending or descending and providing a column name.

```
query.OrderBy = SubSonic.OrderBy.Asc(Product.Columns.ProductName);
```

or

```
query.OrderBy = SubSonic.OrderBy.Desc(Product.Columns.ProductName);
```

Unfortunately at this time there is no way to order by multiple columns.

Filtering

Columns

By default, the query will return all columns in the table. This can be changed by supplying a comma delimited list of columns to the SelectList property.

```
query.SelectList = Product.Columns.ProductName + ", " +
Product.Columns.SupplierID;
```

Rows

The query will also return all rows in the table. This can be changed several ways, the easiest of which is the Top property, which can be set to return only the given number of rows.

```
query.Top = "10";
```

(I'm not sure the reasoning, but this value needs to be a string instead of an integer.)

Filtering by a date range is possible with the AddBetweenAnd method which takes a column name, start date, and end date. This method can be called multiple times to limit by more than one column.

```
query.AddBetweenAnd(Order.Columns.OrderDate, new DateTime(1980, 1, 1),
DateTime.Now);
```

Or you can do the same thing with non-date values using AddBetweenValues.

```
query.AddBetweenValues(Product.Columns.ProductName, "A", "F");
```

The final and most powerful method is AddWhere. Like the AddBetween methods it can be called multiple times to create a complete WHERE clause. AddWhere has several different constructors, the simplest of which takes a column name and matching value.

```
query.AddWhere(Product.Columns.SupplierID, 2);
```

You can also supply a Comparison instead of doing an exact match. (The complete comparison possibilities are Blank, Equals, GreaterOrEquals, GreaterThan, LessOrEquals, LessThan, Like, NotEquals, and NotLike.)

```
query.AddWhere(Product.Columns.SupplierID,
SubSonic.Comparison.GreaterThan, 2);
```

SubSonic also supports the concept of paging data by setting the PageSize and PageIndex properties.

```
query.PageIndex = 2;
query.PageSize = 5;
```

Updating and Deleting

It is possible to issue insert, update and delete commands through the query object by setting the QueryType property. (Your options are Delete, Insert, Select, Update).

```
query.QueryType = SubSonic.QueryType.Insert;
```

If you want to add a specific update setting, such as setting all records to a specific value, you can use the AddUpdateSetting method, which can be called multiple times.

```
query.AddUpdateSetting(Product.Columns.UnitsInStock, 100);
```

Aggregate Functions

Three aggregate functions are included as static methods. To get a column average call GetAverage with the table and column name, and an optional Where object.

```
SubSonic.Query.GetAverage(Product.Schema.Name,
Product.Columns.UnitPrice.ToString());
```

You can also do the same for GetCount and GetSum.

Commands

If the querying power of SubSonic falls short for your needs, you can still use the existing functionality and extend it by accessing the commands that are being built before execution. (These can also be very helpful when debugging.)

The query object has four commands: BuildCommand, BuildDeleteCommand, BuildSelectCommand and BuildUpdate that all return QueryCommand objects, as well as the GetSql method that returns the raw SQL.

```
string sql = query.GetSql();
```

Each auto-generated object also has the ability to return a QueryCommand by calling one of four methods: GetInsertCommand, GetSelectCommand, GetUpdateCommand and GetDeleteCommand.

```
SubSonic.QueryCommand cmd =
product.GetInsertCommand(User.Identity.Name);
```

Stored Procedures

Some tasks are just too complicated for dynamic query building and/or require a greater level of control. To handle this, SubSonic supports stored procedures. Each stored procedure will produce an equivalent static method in the class defined in the configuration file. By default this is SPs. Each method will have one parameter for each stored procedure parameter and return a StoredProcedure object.

```
SubSonic.StoredProcedure sp = SPs.CustOrderHist(customerID);
```

The stored procedure can then either call Execute, ExecuteScalar, GetDataSet or GetReader to execute and return the data.

```
GridView1.DataSource = sp.GetReader();
```

Or combining the two statements into one:

```
GridView1.DataSource = SPs.CustOrderHist(customerID).GetReader();
```

You can also work with the QueryCommand by referencing the Command property.

Scaffolding

The scaffold control is used to quickly create developer level admin pages. By dropping a single control on the page, you get a GridView and update controls. This control should appear in your Toolbox under SubSonic Components. Just set the TableName property and you're ready to go.

```
<cc1:Scaffold ID="Scaffold1" runat="server"
TableName="Products"></cc1:Scaffold>
```

It is also possible to apply some visual formatting through the EditTableCSSClass, EditTableItemCSSClass, EditTableLabelCSSClass and GridViewSkinID properties.

And you can set the delete confirm message with the DeleteConfirm property.

Code Generation Templates

There is one more way you can tweak SubSonic without changing the code. SubSonic creates the auto-generated classes by using standard text. By adding alternate text files to the templateDirectory defined in the configuration file, you can change this behavior.

For a full sample list of these files, check out the CodeTemplates directory in the sample web site.

Conventions

Controllers

Instead of accessing the data functions directly, the Controller design pattern is recommended. To implement this, create a Controller class in the App_Code directory.

```
public class ProductController
```

Then add methods to retrieve the data using collections.

```
public ProductCollection GetAllProducts()
```

And to update the data.

```
public void UpdateProduct(Product product)
```

While there is nothing enforcing these rules, it will make it easier to insert business rules and re-use methods from a single location.

Database

These conventions are pulled directly from the SubSonic documentation.

- Every database table needs to have a primary key. You can't use SubSonic if this isn't the case.
- Integer-based keys are preferable, for performance. I personally use GUIDs only as identifiers and not keys. This isn't required.
- Every table should have some auditing ability built in, but this is not required. These fields are:
    - CreatedOn (datetime)
    - CreatedBy (nvarchar(50))
    - ModifiedOn (datetime)
    - ModifiedBy (nvarchar(50))
- If you want to use logical deletes, you can by adding a field called "Deleted" or "IsDeleted"

- Table names should be singular
- Column names should never contain reserved words (system, string, int, etc)
- Column names should not be the same as table names

Sample Web

The sample web that is downloaded along with the SubSonic source has been discussed periodically throughout this guide. The following is a list of what is provided.

- App_Code\Utility.cs - Miscellaneous functions commonly needed when building web applications.
- App_Themes - A default theme to get you started.
- Dev\CodeTemplates - Text file representations of the templates used to generate classes. Use these as a starting point if you need to create custom templates.
- Dev\DB - Sample database scripts for schema and data loading.
- Dev\BatchClassGenerator.aspx - If your environment does not allow running in Full Trust mode, run this page to generate classes that can be compiled into the build. Add these classes to your project's App_Code folder and remove the build provider settings from the configuration file.
- Dev\BatchScaffoldGenerator.aspx - Create scaffold pages automatically for each table.
- Dev\ClassGenerator.aspx - Similar to BatchClassGenerator, but only creates code for a single class that is meant to be copy and pasted into a class file.
- Scripts - SQL file for Northwind.

Starter Kits

SubSonic Starter Kit

From the CodePlex site, you can also download the SubSonic Starter Kit. To use this, from Visual Studio, select New -> Web Site and pick SubSonic Starter Site. This will give you a good starting template for your own site that includes the following.

- _Dev\ASPNET_Membership - Web pages for user and role membership editing.
- _Dev\Generators - The same generators found in the sample web site plus a MigrationGenerator page. Run this page and it will create SQL scripts containing your database schema, data or both.
- _Dev\SampleQueries.aspx - Some SubSonic code samples.
- App_Code\TestCondition.cs - Some simple data validation.
- App_Code\Utility.cs - Same as the sample web.
- App_Themes\Default - Same as the sample web.
- images - Some useful, generic sample images such as a loading item and processing spinner.
- js - JavaScript helper files from Scriptaculous.
- Modules\ResultMessage.ascx - Format result message control.

- Default.aspx - Three column CSS formatted starting page.
- Login.aspx - Sample login page.
- PasswordRecover.aspx - Sample password recovery page.
- Web.config - Pre-configured for SubSonic and Atlas (AJAX.NET).

Commerce Starter Kit

The Commerce Starter Kit 2.0 is now available which also uses SubSonic.