Lab 8: Gathering Schematized Data

**Lab Time: 60 Minutes**

**Lab Directory: ECM401.SalesProposalSchema**

**Lab Overview:**

In this lab, you will develop a reusable object model and API for managing sales proposals. Your object model will be based on an XML schema that you will use to validate the metadata that will be associated with every sales proposal document. You will also use the schema to develop an InfoPath form for gathering the metadata and storing it in a SharePoint form library.

# Exercise 1: Create a Sales Proposal Schema

1. Start by creating a new **Class Library** DLL project in Visual Studio. Give it the name **ECM401.SalesProposal** and delete the auto-generated **Class1.cs** file.
2. Right-click the **ECM401.SalesProposal** project node and select **Add -> New Item...** from the context menu. Choose the **XML Schema** item template and give it the name **SalesProposal.xsd** .

Note: You will be using the built-in schema editor within Visual Studio. This means you will develop the schema using the "raw" xml-based schema definition language.

1. Add the following code **inside** the **xs:schema** element.

Note: Be careful not to overwrite the generated **targetNamespace** and **xmlns** attributes when you insert the code snippet.

1. First, you will define the root **Sales Proposal** object. This object will contain all the other parts of the sales proposal data definition.

XML Snippet: 'Sales Proposal Schema - Root'

<!-- Root Object -->  
 <xs:element name="SalesProposal">  
 <xs:complexType>  
 <xs:sequence>  
 <xs:element name="CompanyInfo" type="CompanyInfo" minOccurs="1" maxOccurs="1"/>  
 <xs:element name="ClientInfo" type="ClientInfo" minOccurs="1" maxOccurs="1"/>  
 <xs:element name="ProposalInfo" type="ProposalInfo" minOccurs="1" maxOccurs="1"/>  
 </xs:sequence>  
 <xs:attribute name="DateCreated" type="xs:date" use="optional"/>  
 </xs:complexType>  
 </xs:element>

1. Next, you will define an object to describe the company offering the proposal, as well as the people involved in developing the proposal. This s a separate type that will correspond to a C# class when the schema is transformed. This object includes a sub-component that holds the company address information.

XML Snippet: 'Sales Proposal Schema - Company Info'

<!-- Company Information -->  
 <xs:complexType name="CompanyInfo">  
 <xs:sequence>  
 <xs:element name="Name" type="xs:string" minOccurs="1" maxOccurs="1"/>  
 <xs:element name="Author" type="xs:string" minOccurs="1" maxOccurs="1"/>  
 <xs:element name="Manager" type="xs:string" minOccurs="1" maxOccurs="1"/>  
 <xs:element name="SignedBy" type="xs:string" minOccurs="1" maxOccurs="1"/>  
 <xs:element name="SignedByTitle" type="xs:string" minOccurs="1" maxOccurs="1"/>  
 </xs:sequence>  
 </xs:complexType>  
  
 <!-- Address Block -->  
 <xs:complexType name="AddressBlock">  
 <xs:sequence>  
 <xs:element name="Street" type="xs:string" minOccurs="1" maxOccurs="1"/>  
 <xs:element name="City" type="xs:string" minOccurs="1" maxOccurs="1"/>  
 <xs:element name="State" type="xs:string" minOccurs="1" maxOccurs="1"/>  
 <xs:element name="Zip" type="xs:string" minOccurs="1" maxOccurs="1"/>  
 </xs:sequence>  
 </xs:complexType>

1. The next component describes the client who receives the proposal.

Note: The **CompanyInfo** and **ClientInfo** components both share the **AddressBlock** definition.

XML Snippet: 'Sales Proposal Schema - Client Info'

<!-- Company Information -->  
 <!-- Client Information -->  
 <xs:complexType name="ClientInfo">  
 <xs:sequence>  
 <xs:element name="Name" type="xs:string" minOccurs="1" maxOccurs="1"/>  
 <xs:element name="Description" type="xs:string" minOccurs="1" maxOccurs="1"/>  
 <xs:element name="Address" type="AddressBlock" minOccurs="1" maxOccurs="1"/>  
 <xs:element name="Contacts" type="ContactList" minOccurs="1" maxOccurs="1"/>  
 </xs:sequence>  
 </xs:complexType>

1. To faciliate communicating with the client, you will need a list of contacts. For maximum flexibility when dealing with collections of objects, you will declare a **Contact** object as well as a **ContactList** object.

XML Snippet: 'Sales Proposal Schema - Contacts'

<!-- Contact List -->  
 <xs:complexType name="ContactList">  
 <xs:sequence>  
 <xs:element name="Contact" type="Contact" minOccurs="0" maxOccurs="unbounded"/>  
 </xs:sequence>  
 </xs:complexType>  
  
 <!-- Contact (describes an individual contact) -->  
 <xs:complexType name="Contact">  
 <xs:sequence>  
 <xs:element name="Name" type="xs:string" minOccurs="1" maxOccurs="1"/>  
 <xs:element name="Title" type="xs:string" minOccurs="1" maxOccurs="1"/>  
 <xs:element name="Email" type="xs:string" minOccurs="1" maxOccurs="1"/>  
 <xs:element name="Phone" type="xs:string" minOccurs="1" maxOccurs="1"/>  
 <xs:element name="Fax" type="xs:string" minOccurs="1" maxOccurs="1"/>  
 </xs:sequence>  
 </xs:complexType>

1. The **ProposalInfo** component describes the actual proposal, including title and description, starting and ending dates and a list of deliverables.

XML Snippet: 'Sales Proposal Schema - ProposalInfo'

<!-- Proposal Info -->  
 <xs:complexType name="ProposalInfo">  
 <xs:sequence>  
 <xs:element name="Title" type="xs:string" minOccurs="1" maxOccurs="1"/>  
 <xs:element name="Description" type="xs:string" minOccurs="1" maxOccurs="1"/>  
 <xs:element name="StartingDate" type="xs:date" minOccurs="1" maxOccurs="1"/>  
 <xs:element name="EndingDate" type="xs:date" minOccurs="1" maxOccurs="1"/>  
 <xs:element name="Deliverables" type="DeliverablesList" minOccurs="1" maxOccurs="1"/>  
 </xs:sequence>  
 </xs:complexType>  
  
 <!-- Deliverables List -->  
 <xs:complexType name="DeliverablesList">  
 <xs:sequence>  
 <xs:element name="Deliverable" type="Deliverable" minOccurs="0" maxOccurs="unbounded"/>  
 </xs:sequence>  
 </xs:complexType>

1. Finally, each **Deliverable** includes an identifier, an amount, and a **ServiceItem** which is defined as an enumeration of the kinds of services that the company provides.

XML Snippet: 'Sales Proposal Schema - Deliverables'

<!-- Deliverable (describes an individual deliverable) -->  
 <xs:complexType name="Deliverable">  
 <xs:simpleContent>  
 <xs:extension base="xs:string">  
 <xs:attribute name="Name" type="xs:string" use="optional"/>  
 <xs:attribute name="Type" type="ServiceItem" use="optional"/>  
 <xs:attribute name="Amount" type="xs:decimal" use="optional"/>  
 </xs:extension>  
 </xs:simpleContent>  
 </xs:complexType>  
  
 <!-- Service Item Enumeration -->  
 <xs:simpleType name="ServiceItem">  
 <xs:restriction base="xs:string">  
 <xs:enumeration value="Design"/>  
 <xs:enumeration value="Consulting"/>  
 <xs:enumeration value="Mentoring"/>  
 <xs:enumeration value="Development"/>  
 <xs:enumeration value="Installation"/>  
 <xs:enumeration value="SystemAdministration"/>  
 <xs:enumeration value="SecurityAssessment"/>  
 <xs:enumeration value="Programming"/>  
 <xs:enumeration value="ProjectManagement"/>  
 <xs:enumeration value="BusinessDevelopment"/>  
 <xs:enumeration value="Marketing"/>  
 <xs:enumeration value="Sales"/>  
 <xs:enumeration value="Support"/>  
 </xs:restriction>  
 </xs:simpleType>

Note: The **Deliverable** object is declared using the **xs:simpleContent** declaration so that the description can be conveniently placed inside the **<Deliverable> tags.**

# Exercise 2: Generate Serialization Classes

1. In this exercise, you will generate wrapper classes for each of the components defined above. To do this, you will use the **XsdClassGenerator** custom tool which is located in the **Student\Resources\Tools** directory.

Note: If you have not done so already, you will need to run the **XsdClassGeneratorSetup** batch file in order to register the assembly as a **custom tool** within Visual Studio.

1. Make sure the **Properties** window is open in Visual Studio, then in the **Solution Explorer** window, right-click the **SalesProposal.xsd** file. Enter **XsdClassGenerator** in the **Custom Tool** property and press the **Enter** key.
2. A new node appears beneath the **SalesProposal.xsd** file. Double-click the generated **SalesProposal.cs** file to open it for editing. Your file should resemble the following illustration.
3. In later exercises you will add code to extend the functionality of the generated classes. At this point, you can build the project to create a compiled assembly. This will ultimately become a custom **Sales Proposal API** that can be applied to different kinds of solutions.
4. In the next exercise you will use the schema to develop an InfoPath form for gathering Sales Proposal data.

# Exercise 3: Create an InfoPath Form for Gathering Data

In order to test the Sales Proposal API, you will need sample data. You have the option of entering the data manually, using the built-in schema validation provided by the Visual Studio XML editor or you can use InfoPath to create a richer user experience. In this exercise, you will use **Visual Studio Tools for Office (VSTO)** to create an InfoPath form to generate your sample data.

1. Start by creating a new **InfoPath Form Template** project in Visual Studio. Give it the name **ECM401.SalesProposalForm** as shown below.
2. In the next dialog, select **Form Template** based on **XML or Schema** . This allows you to automatically define the data source for the form based on the schema you have already created.
3. In the next dialog, browse to the **ECM401.SalesProposal** project folder and select the **SalesProposal.xsd** file you created above.
4. Press OK to generate the form template and open it in the Visual Studio **InfoPath Form Designer** .

Note: The InfoPath form designer is integrated into the Visual Studio IDE.

1. Open the **Design Tasks** window to begin designing your form.
2. Next, click the **Data Source** link to examine the **SalesProposal** data source definition that InfoPath has created. Notice that the data source is **locked** to prevent inadvertent modifications of the schema.

Note: By locking data sources based on external schemas, InfoPath ensures that the schema will not be changed when controls are dragged onto the form designer surface.

1. At this point, you can start dragging data elements onto the form. Use the following illustration as a guide as you drag and drop your controls. The most important point is to ensure you have included all of the data elements. One way to do that is to simply drag the entire data source onto the design surface and then rearrange the controls. Another approach (used here) is to create the basic layout using **layout tables** and then drag individual data elements or element groups into the layout table cells.
2. When you finish designing your form, you are ready to test it. To do this, right-click the **manifest.xsf** file in the **Solution Explorer** and select **Open With...** from the context menu. Choose **Microsoft Office InfoPath 2007** and press **OK** . This will open a new instance of **InfoPath** with your form ready for editing. Enter some data as shown below.
3. After entering your sample data, save the file by browsing to the **ECM401.SalesProposal** project directory. Name the file **Sample Data** . InfoPath will save the input data to a new file named **Sample Data.xml** . Double-click the file in the **ECM401.SalesProposal** project to view it within Visual Studio.
4. Now you have schematized data that can be serialized and deserialized using a custom API and that can also be created easily using a rich user interface that ensures the data will conform to the schema. In the next exercise, you will publish the form to SharePoint and use it to gather sales proposal data.

# Exercise 4: Publish the Form to SharePoint

In this exercise, you will publish the **Sales Proposal** form to a SharePoint **Form Library** . This will form the basis for extending the functionality of your solution in later labs and exercises. Instead of publishing the form directly into SharePoint, you will package the form into a **Feature** so it can be deployed easily to many sites.

1. Start by creating a new **SharePoint Feature Project** named **ECM401.SalesProposalFeature** . Set the **Feature Scope** to **Web** .
2. Next, re-open the **manifest.xsf** file in **design mode** by double-clicking in the **Solution Explorer** window. Select **Form Options...** from the **Tools** menu.
3. Select **Property Promotion** on the left side of the **Form Options** dialog. From here, you will select the data elements you wish to be promoted to SharePoint list columns. Starting with the **Company Name** field, navigate to the **Name** element inside the **Company Info** element and give it the name **Company** as shown below.
4. Continue adding the remaining fields shown below.
5. Select **Security and Trust** on the left side of the dialog. Specify **Domain** security for the form. This will allow it to be opened from within a SharePoint form library.
6. Click **OK** to save your changes and return to the design surface. From the **Design Tasks** window, click the **Publish Form Template...** link.

Note: "Publishing" is **NOT** the same as "saving" a form template. The form template you are working on is saved as part of the SalesProposalForm project. In this next step, you will publish the form into the **12\TEMPLATE\FEATURES\ECM401.SalesProposalFeature** folder so it can be packaged along with the feature.

1. When you click the **Publish Form Template...** link, Visual Studio responds by building your project. This is necessary because any managed code you might write would have to be compiled and packaged within the form template you are publishing. In this case, you are not writing any managed code, but the assembly is part of the form template anyway. After the project is built, you are presented with series of dialogs. Choose the options shown in the next series of illustrations.
2. Browse to the **ECM401.SalesProposalFeature** project folder. Within that, navigate down to the **12\TEMPLATE\FEATURES\ECM401.SalesProposalFeature** folder. Give the form the name **SalesProposal** .
3. Clear the default access path, and then click **OK** to continue in the popup dialog box.
4. Review your options and then click **Publish** to generate the published form.
5. Now that you have a form template in the feature folder, you can proceed to package it into your solution. Double-click the **elements.xml** file in the **ECM401.SalesProposalFeature** project and insert the following code inside the **<Elements>** tag.

XML Snippet: 'Sales Proposal Feature Elements'

<ListInstance  
 FeatureId="00BFEA71-1E1D-4562-B56A-F05371BB0115"  
 TemplateType="115"  
 Id="SalesProposals"  
 Title="Sales Proposals"  
 Description="Proposal data for clients."  
 Url="SalesProposals"  
 OnQuickLaunch="True"  
 />  
  
 <Module Name="FormTemplate" List="115" Url="SalesProposals/Forms">  
 <File Url="SalesProposalForm.xsn" Type="GhostableInLibrary"/>  
 </Module>

Note: These elements declare a form library with the name "Sales Proposals" and instruct SharePoint to load the form into a specific location within the form library. In the next step, you will write code that attaches the form to the library as a document template.

1. Double-click the **FeatureReceiver.cs** file to open it for editing. Add an override for the **OnActivated** virtual method. Be sure to choose the version that is appropriate for the feature scope. In this case, choose the method that accepts an **SPWeb** parameter. Replace the body of the method with the following code.

Code Snippet: 'Sales Proposal List'

try  
 {  
 // Setup the document template for the form library.  
 SPDocumentLibrary formLib = web.Lists["Sales Proposals"] as SPDocumentLibrary;  
 if (formLib != null)  
 {  
 formLib.DocumentTemplateUrl = "SalesProposals/Forms/SalesProposalForm.xsn";  
 formLib.Update();  
 }  
 }  
 catch  
 {  
 }

1. Build the project, open the web browser and navigate to the **localhost:401/bpm/part1** site. From the **Site Actions** menu, choose **Site Settings** and then choose **Site Features** . Scroll to the **ECM401.SalesProposalFeature** and activate it. Go back to the home page of the site and verify that the **Sales Proposal** form library was created.
2. Open the form library and click the **New** menu command to create a new form. Verify that InfoPath opens with your form.

**This concludes the lab exercises.**