Lab 4: Creating a Service

# Purpose

Estimated time to complete this lab: **25 minutes**.

In this lab you will learn what a Composite UI Application Block service is, how to create and register one, and how to consume it from your module using the dependency injection pattern. You will also be introduced to entity translation and unit testing.

After completing this lab, you will be able to:

* + Create, register, and consume a service
  + Translate entities using the Entity Translator Service
  + Write simple unit tests

# Preparation

Before proceeding with this lab, you must install and configure the prerequisite software. For more information, see the topic Start Here.

Open the solution for the previous lab (either the one that you created or the end solution for that lab.)

# Background

## Services

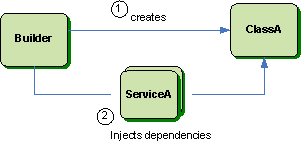
A Composite UI Application Block service is the name for an object that provides functionality to other components (the components can be in the same module or in another module) in a loosely coupled way; for example, services providing authorization, logging, or hardware communication functionality. It does not mean a Web service. The Composite UI Application Block and the initial solution you create with the Smart Client Software Factory include a set of basic services that you can use in your applications. You can also build your own services that provide infrastructure capabilities and business logic specific to your applications.

The development of services can be thought of a three-stage process:

* 1. **Creating the service**. In this stage you create a class and write the business logic for the service.
  2. **Registering the service**. In this stage you register the service with a Work Item. By registering it, you make it available to one or more modules of your solution depending on the Work Item you register the service with.
  3. **Consuming the service**. In this stage you obtain a reference to a valid instance of the service. Typically you obtain a reference to a service instance using the dependency injection pattern, which decouples your component from the service and facilitates testing of your component in isolation.

## Dependency Injection pattern

The Dependency Injection pattern helps to decouple your classes from its dependencies so the dependencies can be replaced or updated requiring minimal or no changes to your classes’ source code. This is especially useful if you want to be able to test your classes in isolation (without using the service dependencies). With the Dependency Injection pattern, you do not instantiate the dependencies explicitly in your class. Instead, you express dependencies declaratively in your class definition and use a separate object to obtain valid instances of your object’s dependencies and pass them to your object during the object creation and/or initialization. The Composite UI Application Block includes a dependency injection framework based on ObjectBuilder which you can use to implement the Dependency Injection Pattern in your application; this framework is used intensively across the factory.

* 1. 
  2. Figure 1
  3. Conceptual view of the dependency injection pattern

In this lab, you will use the dependency injection pattern to obtain an instance of a service. By using this pattern, you can test the consumer of the service (the presenter, in this case) in isolation.

Exercise 1: Creating the Sales Order Service

In this exercise, you will create a sales order service. This service will provide functionality to retrieve and ship sales orders. The service will interact with a back-end system through a Web service.

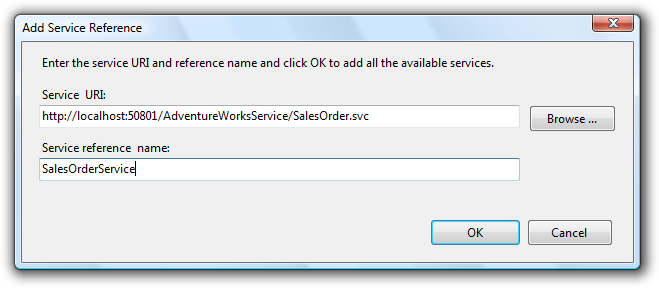
# Task 1. Add the AdventureWorksService Web site to the Solution

The Hands on Labs package includes a Windows Communication Foundation Web service named AdventureWorksService that interacts with a database to manage sales orders. The service that you will create in this exercise will interact with this back-end service. In this task, you will add the AdventureWorksService Web site to your solution.

* 1. In Solution Explorer, right-click the solution root node, point to **Add** and select **Existing Web Site**.
  2. In the **Add Existing Web Site** dialog, browse to the **Assets** folder and select the **AdventureWorksService** folder.
  3. Click **Open**. Your solution should look like the following figure.
     1. 
     2. Figure 2
     3. AdventureWorksService Web site
  4. Open the Properties window for the AdventureWorksService Web site.
  5. Set the property **Use dynamic ports** to **False**.

# Task 2. Create a Service Reference

In this task you will add a Service reference to the AdventureWorksService service in the ShippingModule project.

* 1. In Solution Explorer, right-click the AdventureWorksService Web site and select **View in Browser**. The Visual Studio Development Web Server will start and a browser window will appear.
  2. In the browser, click the **SalesOrder.svc** hyperlink.
  3. Copy to the clipboard the address that appears in the address bar. You need this address to add the service reference in the ShippingModule project.
  4. Close the browser and switch back to Visual Studio.
  5. In Solution Explorer, right-click the ShippingModule node and select **Add Service Reference**.
     1. 
     2. Figure 3
     3. Add Service Reference dialog
  6. Paste the service address in the **Service URI** field.
  7. Set the service reference name to **SalesOrderService** and click **OK**. Visual Studio will generate a client proxy for the service.
  8. Open the file app.config located in the root of the ShippingModule project. This file was automatically created by Visual Studio to store the endpoint configuration information for the client proxy.
  9. Move the configuration section **<system.serviceModel>** to the App.config file in the Shell project. To do this, copy the **<system.serviceModel>** configuration node, open the App.config file in the Shell project, and paste it below the **<configSections>** node.
  10. Delete the app.config file of the ShippingModule project.

In this exercise, you will use the client proxy generated by Visual Studio as a Composite UI Application Block service. Please note that a service does not need to be related to a Web service. For example, the software factory includes a service that is in charge of translating business entities, the Entity Translator Service. This service does not interact with a Web service; instead, it contains translation logic that is executed locally.

# Task 3. Register the Service

To be able to consume the sales order service using the dependency injection pattern, you need to register it with a WorkItem. Depending on the WorkItem you register the service with, the service will be available or not to different modules and components in the application.

* + If you register the service with the root WorkItem, it will be accessible by all the modules in the application.
  + If you register the service with a WorkItem other than the root WorkItem, it will be available only for the components created with that WorkItem or a descendent WorkItem. For example, if you register a service with a module WorkItem, only components in the module will be able to access it.

Internally, the availability of a service is determined by the WorkItem hierarchy of your application. When using dependency injection to create an object, ObjectBuilder attempts to locate the service in the WorkItem’s collection of services. If it cannot locate the service type requested, it asks the WorkItem’s parent WorkItem to locate the service. This continues until the service is found or the root WorkItem is reached. For more information about how services are located, see [Service Locator](ms-help://MS.VSIPCC.v80/ms.practices.scsf.2007may/SCSF/html/01-01-000-Introducing_the_Smart_Client_Software_Factory.htm) in the Smart Client Software Factory help.

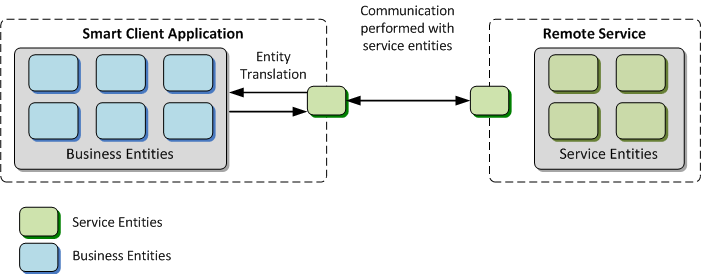
In this task, you will register the sales order service with the module WorkItem. Thus, it will not be available for other modules.

* 1. Open the file ModuleController.cs located in the root of the ShippingModule project.
  2. Add the following **using** statement at the top of the file. You will use it to refer to the service proxy created by Visual Studio.
     1. C#
     2. using AdventureWorks.ShippingModule.SalesOrderService;
  3. Implement the AddServices method in the ModuleController class to register the service. To do this, add the following highlighted code to the method’s body.
     1. C#
     2. private void AddServices()
     3. {
     4. **WorkItem.Services.AddNew<SalesOrderServiceClient, ISalesOrderService>();**
     5. }
     6. You use the **AddNew** method of the services collection to have ObjectBuilder create a new instance of the service and add it to the services collection of the WorkItem. The **AddNew** method allows you to specify the type that is used to register the service (this is the second generic parameter in the line of code you added in step 3). This is the type developers use to obtain a reference to the service when using the Dependency Injection pattern. By specifying the service’s interface as the registration type, you enable developers to write generic code that is not aware of the concrete service implementation being used. This is especially useful when testing your classes because you can easily replace dependencies with mock implementations and thus, test your classes in isolation.

# Task 4. Create and Register Entity Translators

Remote services that your smart client application interacts with may use certain entities for the return value and/or the parameters of the methods they expose. These entities belong to the service domain, and typically they are created and maintained by a team different to that developing the smart client application. These entities are referred to as **service entities**. In contrast, entities that belong to the domain model of your smart client application and which you are in control of are referred to as **business entities**.

Changes in service entities may impact your application, requiring several changes in your code. To reduce the impact in your application, it is a recommended practice to:

* + Use business entities throughout your application, and limit the use of service entities only to the points in which interaction with remote services is required
  + Translate service entities into business entities and vice versa when interacting with remote services
  1. 
  2. Figure 4
  3. Entity translation

To facilitate the entity translation process, the Smart Client Software Factory includes a service you can use to translate your entities, the entity translator service. In this task, you will use this service to translate your entities when communicating with the sales order service.

To use the entity translator service, you have to build entity translators for your entities. Entity translators are objects used by the entity translator service to translate between specific entity types. In the following steps you will create entity translators for the **Order** and the **OrderLineItem** entities.

* 1. **Note:** For more detailed information about the entity translator service and entity translators, see [How to: Translate Between Business Entities and Service Entities](ms-help://MS.VSIPCC.v80/ms.practices.scsf.2007may/SCSF/html/03-01-090-How_to_Translate_Between_Business_Entities_and_Service_Entities.htm) in the Smart Client Software Factory help.
  2. Create a folder named **EntityTranslators** in the ShippingModule project.
  3. Add a new class file named OrderLineItemTranslator.cs to the EntityTranslators folder. The OrderLineItemTranslator class will translate **OrderLineItem** objects (business entities) to **SalesOrderLineItemElement** objects (service entities) and vice versa.
  4. Add the following **using** statements at the top of the file.
     1. C#

using AdventureWorks.Infrastructure.Library.EntityTranslators;

* + 1. using AdventureWorks.ShippingModule.BusinessEntities;
    2. using AdventureWorks.ShippingModule.SalesOrderService;
    3. using AdventureWorks.Infrastructure.Interface.Services;
  1. Make your class extend the EntityMapperTranslator base class and implement the **BusinessToService** and **ServiceToBusiness** methods as shown in the code below.
     1. C#
     2. public class OrderLineItemTranslator : EntityMapperTranslator<OrderLineItem, SalesOrderLineItemElement>
     3. {
     4. protected override SalesOrderLineItemElement BusinessToService(IEntityTranslatorService service, OrderLineItem value)
     5. {
     6. SalesOrderLineItemElement result = new SalesOrderLineItemElement();
     7. result.ProductName = value.Description;
     8. result.Location = value.Location;
     9. result.SalesOrderDetailID = value.ProductId;
     10. result.OrderQty = value.Quantity;
     11. result.CarrierTrackingNumber = String.Format("{0}#{1}", value.Location, value.BoxNumber);
     12. return result;
     13. }
     14. protected override OrderLineItem ServiceToBusiness(IEntityTranslatorService service, SalesOrderLineItemElement value)
     15. {
     16. OrderLineItem result = new OrderLineItem();
     17. result.Description = value.ProductName;
     18. result.Location = value.Location;
     19. result.ProductId = value.SalesOrderDetailID;
     20. result.Quantity = value.OrderQty;
     21. return result;
     22. }
     23. }
  2. Add a new class file named OrderTranslator.cs to the EntityTranslators folder. The OrderTranslator class will translate **Order** objects (business entities) to **SalesOrderElement** objects (service entities) and vice versa.
  3. Add the following **using** statements at the top of the file.
     1. C#

using AdventureWorks.Infrastructure.Library.EntityTranslators;

* + 1. using AdventureWorks.ShippingModule.BusinessEntities;
    2. using AdventureWorks.ShippingModule.SalesOrderService;
    3. using AdventureWorks.Infrastructure.Interface.Services;
  1. Use the following code to define the translator.
     1. C#
     2. public class OrderTranslator : EntityMapperTranslator<Order, SalesOrderElement>
     3. {
     4. protected override SalesOrderElement BusinessToService(IEntityTranslatorService service, Order value)
     5. {
     6. SalesOrderElement result = new SalesOrderElement();
     7. result.CustomerName = value.CustomerName;
     8. result.SalesOrderID = value.OrderId;
     9. result.DetailItems = service.Translate<SalesOrderLineItemElement[]>(value.LineItems.ToArray());
     10. return result;
     11. }
     12. protected override Order ServiceToBusiness(IEntityTranslatorService service, SalesOrderElement value)
     13. {
     14. Order result = new Order();
     15. result.CustomerName = value.CustomerName;
     16. result.OrderId = value.SalesOrderID;
     17. OrderLineItem[] details = service.Translate<OrderLineItem[]>(value.DetailItems);
     18. result.LineItems = new List<OrderLineItem>(details);
     19. return result;
     20. }
     21. }

To be able to use the entity translators you created, you need to register them with the entity translator service. In the next steps you will register them.

* 1. Open the file ModuleController.cs located in the root of the ShippingModule project.
  2. Add the following **using** statements.
     1. C#
     2. using AdventureWorks.Infrastructure.Interface.Services;
     3. using AdventureWorks.ShippingModule.EntityTranslators;
  3. Create a private method named **RegisterEntityTranslators.** In this method, obtain a reference to the entity translator service (it is automatically registered by the Composite UI Application Block with the **IEntityTranslatorService** type) and register the OrderTranslator and OrderLineItemTranslator translators as demonstrated in the code below.
     1. C#
     2. private void RegisterEntityTranslators()
     3. {
     4. IEntityTranslatorService translator = WorkItem.Services.Get<IEntityTranslatorService>(true);
     5. translator.RegisterEntityTranslator(new OrderLineItemTranslator());
     6. translator.RegisterEntityTranslator(new OrderTranslator());
     7. }
  4. Update the **Run** method to call the **RegisterEntityTranslators** method in the first line.
     1. C#
     2. public override void Run()
     3. {
     4. **RegisterEntityTranslators**();
     5. AddServices();
     6. ExtendMenu();
     7. ExtendToolStrip();
     8. AddViews();
     9. }
  5. **Note:** You can register entity translators by writing code in the ModuleController class as described in this task, or by writing code in the Module class of the module, as instructed in the Smart Client Software Factory help; both approaches are valid. In this case, the entity translators were registered in the **Run** method of the ModuleController class to keep all the module start up logic encapsulated in this method.

# Task 5. Create Unit Tests for the Presenter (Optional)

In this task you will write a unit test for the OnViewReady method of the ShipNewOrderView’s presenter. The test will verify that the presenter retrieves the next order to be displayed to the user by calling the **GetNextOrder** method of the SalesOrderService and that it binds the view to that order.

* 1. **Note:** This task is optional; you can skip it if you are not interested in implementing unit tests.
  2. Add a reference to the Infrastructure.Library project in the ShippingModule.Tests project. To do this, in Solution Explorer right-click the ShppingModule.Tests projects and select **Add Reference**. From the **Projects** tab, select Infrastructure.Library and click **OK**.
  3. Add a reference to the System.Runtime.Serialization assembly in the ShippingModule.Tests project. To do this, in Solution Explorer right-click the ShppingModule.Tests projects and select **Add Reference**. From the **.NET** tab, select **System.Runtime.Serialization** and click **OK**.
  4. Open the file Views\ShipNewOrderViewPresenterFixture.cs in the ShippinModule.Tests project. This file was automatically created when you ran the Add View recipe.
  5. Add the following **using** statements.
     1. C#
     2. using AdventureWorks.ShippingModule.SalesOrderService;
     3. using AdventureWorks.Infrastructure.Interface.Services;
     4. using AdventureWorks.Infrastructure.Library.Services;
     5. using AdventureWorks.ShippingModule.EntityTranslators;
     6. using AdventureWorks.ShippingModule.BusinessEntities;
  6. Implement a unit test that verifies the OnViewReady method. To do this, use a mock service and a mock view. By using the mocks, you can test the presenter without the need for the real service and real view when running the unit tests.
     1. C#
     2. [TestMethod]
     3. public void OnViewReadyBindsViewToOrder()
     4. {
     5. MockSalesOrderSerivce salesService = new MockSalesOrderSerivce();
     6. salesService.NextOrder = CreateSimpleOrder();
     7. IEntityTranslatorService translatorService = CreateTranslatorService();
     8. ShipNewOrderViewPresenter presenter = new ShipNewOrderViewPresenter(salesService, translatorService);
     9. MockShipNewOrderView view = new MockShipNewOrderView();
     10. presenter.View = view;
     11. presenter.OnViewReady();
     12. Assert.IsTrue(salesService.GetNextOrderCalled);
     13. Assert.IsTrue(view.BindToOrderCalled);
     14. Assert.AreEqual(salesService.NextOrder.SalesOrderID, view.Order.OrderId);
     15. }
     16. First, a mock sales order service is created and a simple order is set as the next order. This order will be returned by the mock service when the **GetNextOrder** method is invoked on it. Then, an entity translator service is set up and a new instance of the presenter is created. After that, a mock view is created and the **OnViewReady** method is called on the presenter. Finally, three assertions are made: the first one verifies that the **GetNextOrder** method was called for the presenter, the second one verifies that the **BindToOrder** method was invoked on the view, and the last one verifies that the order passed to the view is the one that the sales order service returned.
  7. Paste the following code to define the utility methods used in the unit test.
     1. C#
     2. private static SalesOrderElement CreateSimpleOrder()
     3. {
     4. SalesOrderElement order = new SalesOrderElement();
     5. order.DetailItems = new SalesOrderLineItemElement[] { new SalesOrderLineItemElement() };
     6. return order;
     7. }
     8. private static IEntityTranslatorService CreateTranslatorService()
     9. {
     10. EntityTranslatorService translator = new EntityTranslatorService();
     11. translator.RegisterEntityTranslator(new OrderTranslator());
     12. translator.RegisterEntityTranslator(new OrderLineItemTranslator());
     13. return translator;
     14. }
  8. The Add View recipe automatically generated the code for a mock view. Replace it with the following code.
     1. C#
     2. class MockShipNewOrderView : IShipNewOrderView
     3. {
     4. public bool BindToOrderCalled = false;
     5. public Order Order = null;
     6. #region IShipNewOrderView Members
     7. public void BindToOrder(Order order)
     8. {
     9. BindToOrderCalled = true;
     10. Order = order;
     11. }
     12. public void ShowMessage(string message)
     13. {
     14. throw new Exception("The method or operation is not implemented.");
     15. }
     16. #endregion
     17. }
     18. In the MockShipNewOrderView class, two public fields are declared: BindToOrderCalled and Order. BindToOrderCalled is a flag that indicates whether the BindToOrder method was called, and Order contains the last order passed to the BindToOrder method.
  9. Paste the following code to define the mock sales order service.
     1. C#
     2. class MockSalesOrderSerivce : ISalesOrderService
     3. {
     4. public SalesOrderElement NextOrder = null;
     5. public bool GetNextOrderCalled = false;
     6. #region ISalesOrderService Members
     7. public SalesOrderElement GetNextOrder()
     8. {
     9. GetNextOrderCalled = true;
     10. return NextOrder;
     11. }
     12. public void ShipOrder(SalesOrderElement order)
     13. {
     14. throw new Exception("The method or operation is not implemented.");
     15. }
     16. #endregion
     17. }
     18. The MockSalesOrderService class contains two public fields: NextOrder and GetNextOrderCalled. GetNextOrderCalled is a flag that indicates whether the GetNextOrderCalled method was called, and NextOrder contains the order that will be returned by the GetNextOrder method.
  10. If you run the unit tests now, you will get compilation errors. This happens because the ShipNewOrderViewPresenter does not have a constructor that receives an ISalesOrderService instance and an IEntityTranslatorService instance as parameters. To fix this, replace the signature of the ShipNewOrderViewPresenter class´s constructor with the following code. This class is implemented in the file Views\ShipNewOrderView\ShipNewOrderViewPresenter.cs in the ShippingModule project.
      1. C#
      2. **public ShipNewOrderViewPresenter(ISalesOrderService orderService, IEntityTranslatorService translator)**
      3. {
      4. \_order = CreateOrder();
      5. }
  11. Add the following **using** statements at the top of the file.
      1. C#
      2. using AdventureWorks.Infrastructure.Interface.Services;
      3. using AdventureWorks.ShippingModule.SalesOrderService;
  12. Run the unit test, it should fail. In the next task you will write code in the presenter to make it pass.
      1. **Note:** When you created the ShippingModule module, the Add Business Module recipe automatically generated a unit test named **OnLoadCreateModuleController** which verifies that a module controller is added to the root WorkItem when the module is initialized. This test fails because it needs to be updated to reflect the changes you did to the ModuleController class. Updating this unit test is out of the scope of this lab. If you want to disable it, add a **[Ignore]** attribute to the test.

In this task you created a simple unit test to verify the behavior of the OnViewReady method of the presenter. To better understand how to write unit tests for your smart client application, see the test projects in the Bank Branch Client reference implementation.

# Task 6. Consume the Service from the Presenter

In this task, you will update the ShipNewOrderViewPresenter class to consume the sales order service.

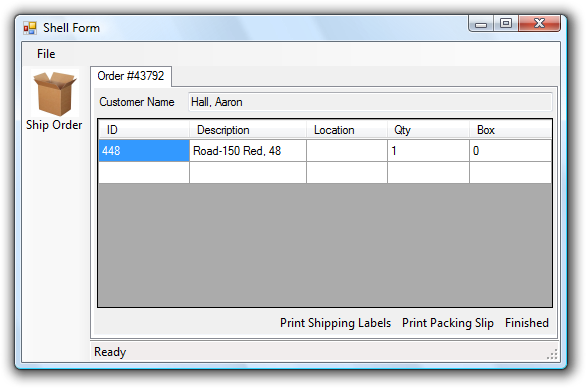
* 1. Open the file Views\ShipNewOrderView\ShipwNewOrderViewPresenter.cs located in the ShippingModule project.
  2. If you do not have the following **using** statements, add them.
     1. C#
     2. using AdventureWorks.Infrastructure.Interface.Services;
     3. using AdventureWorks.ShippingModule.SalesOrderService;
  3. Update the class constructor to receive instances of the ISalesOrderService and IEntityTranslatorService services using dependency injection, and to store the instances in member variables for further use. To do this, replace the existing constructor with the following code.
     1. C#
     2. **private ISalesOrderService \_orderService;**
     3. **private IEntityTranslatorService \_translator;**
     4. public ShipNewOrderViewPresenter
     5. (
     6. **[ServiceDependency] ISalesOrderService orderService,**
     7. **[ServiceDependency] IEntityTranslatorService translator**
     8. )
     9. {
     10. **\_translator = translator;**
     11. **\_orderService = orderService;**
     12. \_order = CreateOrder();
     13. }
  4. In the constructor, invoke the method **GetNextOrder** of the sales order service to retrieve the next order to show, instead of creating an order locally. Translate the service order entity into a business entity using the entity translator service and store it in the \_order member variable. To do this, replace the line:
     1. C#
     2. \_order = CreateOrder();

With the following code:

* + 1. C#
    2. \_order = translator.Translate<Order>(\_orderService.GetNextOrder());
  1. Remove the **CreateOrder** method and the **nextOrderNumber** variable definitions. You need them no longer because the logic for order retrieval is now encapsulated in the sales order service.
  2. If you created the unit test for the presenter, run it now. It should pass.
  3. In the **Submit** method, translate the \_order business entity into a service entity and call the **ShipOrder** method of the sales order service to ship the order, as shown in the highlighted code below.
     1. C#
     2. public void Submit()
     3. {
     4. if (!Validate())
     5. {
     6. View.ShowMessage("Please enter all the box numbers before performing this operation.");
     7. return;
     8. }
     9. **\_orderService.ShipOrder(\_translator.Translate<SalesOrderElement>(\_order));**
     10. OnCloseView();
     11. }

# Task 7. Compile and run the solution

In this task you will verify that ShipNewOrderView view consumes the sales order service correctly.

* 1. Build and run the application.
     1. The MDI interface will appear. Click the **Ship Order** button to open the **ShipNewOrderView** view displaying an order.
     2. 
     3. Figure 5
     4. ShipNewOrderView view displaying an order retrieved from the sales order Web service.
  2. Enter a numeric value for the Box column, and then click the **Finished** button to submit the order. The ShipNewOrderView view will close.
  3. Close the application.

To check the finished solution open the solution file **CS\Developer\04-CreatingService\AdventureWorksCycles.sln**.