WA1802 SOA for Architects Using WebSphere ESB

Student Labs

Web Age Solutions Inc.

Table of Contents

Lab 1 - Basic Mediation Flow Development

In this lab, we will develop a very simple mediation flow. A mediation flow can be used to implement most ESB patterns. We will not get into pattern implementation just yet. Our focus will be to learn the basics of development and testing using WID. We will also get to see a very simple mediation flow in action.

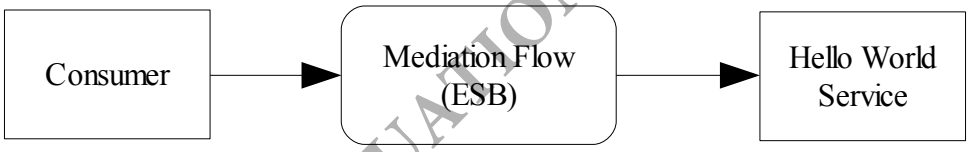
A mediation flow always acts as an intermediary between an actual service provider and consumer. To the consumer, the mediation flow appears as the service provider. To the provider, the mediation flow is the consumer. The true consumer is decoupled from the true provider.

In this lab, we will use a Java class as the service provider. Java class based service providers are quicker to develop compared to web services. They are great for testing.

We will not develop any consumer application. Instead, we will use the universal test tool to invoke methods of the mediation flow.

The business logic for the service will be very simple. It will have a method called sayHello. It will accept as input a string. It will return a greeting message as string. From a Java point of view, the method will look like this:

String sayHello(String name)

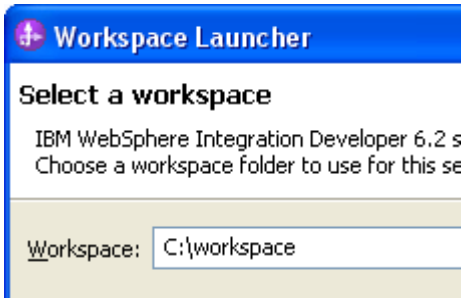


The business logic is kept intentionally simple so you can focus on the mechanisms of software development and testing using WID. Later labs will assume that you know these skills and avoid giving detail instructions.

Part 1 - Launch WID

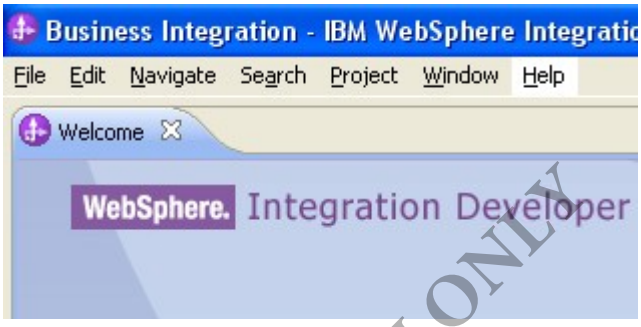
__1. From the Windows **Start** menu select **All Programs > IBM WebSphere Integration Developer > IBM WebSphere Integration Developer V6.2 > WebSphere Integration Developer V6.2**.

__2. When prompted to select a workspace, enter **C:\workspace**



__3. Click **OK**.

WID may show the welcome screen.



__4. Close the welcome screen if opens.

By default, WID puts you in the **Business Integration** perspective. (You can always see the name of the perspective in the title bar). Most SCA based development can be done from this perspective. For J2EE and other types of development, we will switch the perspective to something else.

Part 2 - Create a Mediation Module

SCA development can happen in two types of projects – Module and Mediation Module. A Mediation Module only allows you to create artifacts, such as mediation flow, that can be deployed in WebSphere ESB. A Module, on the other hand, allows you to create all artifacts that can be deployed in WebSphere Process Server. For example, a BPEL business process can only be created in a Module. A mediation flow can be created in either of the two projects.

Tip: If your target server runtime is WebSphere ESB, you should always use the Mediation Module project. This will prevent you from creating an artifact that can not be deployed in WebSphere ESB.

__1. From the menubar, select **File > New > Mediation Module**.

__2. Enter the following values:

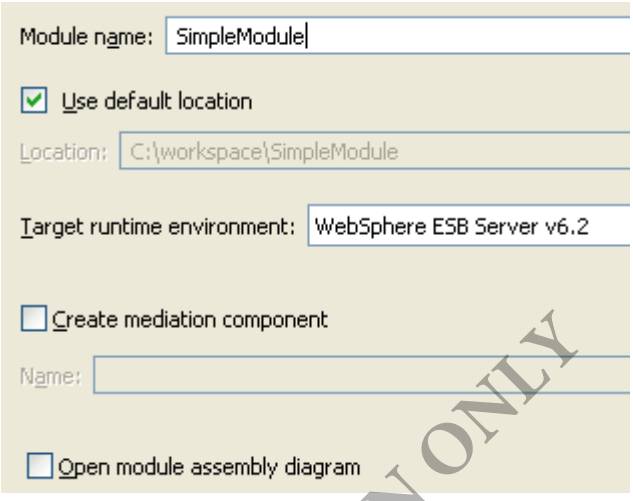
Module name: SimpleModule

Target runtime server: WebSphere ESB Server v6.2. **Important:** By default

WebSphere Process Server is selected. Make sure you change it to ESB.

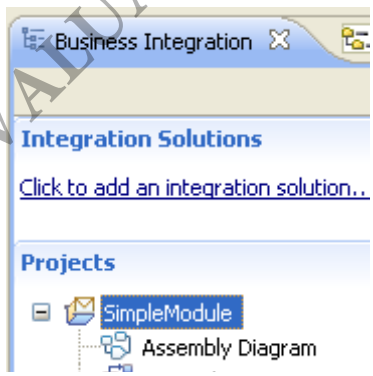
Create mediation component: Uncheck. This adds a dummy component to the assembly diagram without any implementation. This is more of an inconvenience than helpful. So we uncheck it.

Open module assembly diagram: Uncheck. We will not work with the assembly diagram yet. So, uncheck this option.



__3. Click **Finish**.

__4. WID will create the mediation module. In the **Business Integration** view, under the **Projects** section, verify that SimpleModule is shown.



Part 3 - Study the Project Structure

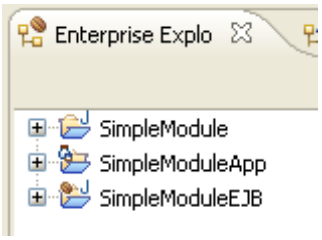
We will now observe how exactly a mediation module project is represented on the file system.

__1. From the menubar, select **Window > Open Perspective > Other**.

__2. Select **Java EE**.

__3. Click **OK**.

__4. Notice, three projects have been created:



These projects are as follows.

Project	**Remarks**
SimpleModule	This is a plain Java project that represents the actual mediation module. All SCA artifacts, like mediation flow and data transformation maps, are created here.
SimpleModuleApp	This is a J2EE enterprise application project. This is what gets deployed to the server as an EAR file. The EAR file includes the mediation module Java project as a JAR file.
SimpleModuleEJB	This is a EJB module project. For the SCA artifact, various EJBs are generated here. By default, WID generates this code every time you save a change to an artifact's file. You should not develop any code here since this project can be overwritten by WID at any time. This module is included in the application EAR as a JAR file.

__5. Close the Java EE perspective by selecting **Window > Close Perspective**.

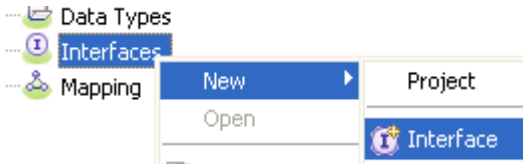
Part 4 - Create a Service Interface

Every SCA service must support one or more interfaces. A service interface is a collection of operations. SCA uses WSDL portType to define interfaces. You could also use Java interface for that purpose. But, in SOA, vendor and language neutral WSDL is always recommended.

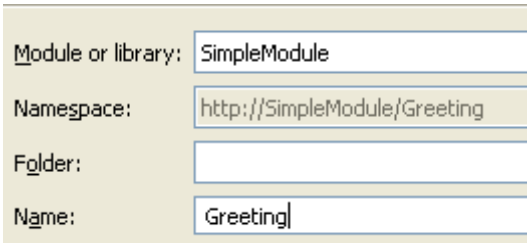
We will now create the interface for our very simple mediation flow. recall, it has only one operation that looks like this:

String sayHello(String name)

__1. Right click **Interfaces** and select **New > Interface**.



__2. Enter **Greeting** as the name of the interface.



Note that the namespace defaults to http://SimpleModule/Greeting. We will leave it that way. In real life, your organization needs to strictly control the namespace. In that case, you need to specify the correct namespace.

__3. Click **Finish**.

WID will create the interface and open it in the editor.

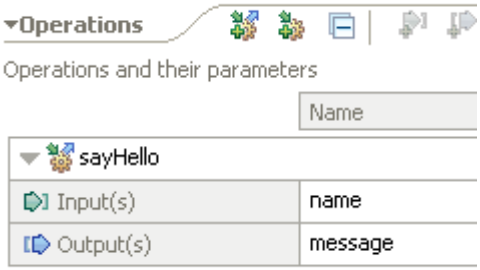
__4. In the editor, click the **Add Request Response Operation** toolbar icon to add a new operation.

Make the following changes:

__5. Change the name of the operation to **sayHello**

__6. Change the name of the input parameter to **name**

__7. Change the name of the output parameter to **message**



__8. Save and close the editor.

Part 5 - Create the Mediation Flow

Our mediation flow will conform to the service interface we have defined in the previous part.

__1. Right click **Integration Logic** and select **New > Mediation Flow**.

__2. Enter **GreeterFlow** as the name.

__3. Click **Next**.

As the source interface, you will add the Greeting interface we have defined previously. The source interface is what the mediation flow offers to its consumers.

__4. Click the **Add** button for Sources interfaces.

__5. The Greeting interface should be selected, just click **OK**.

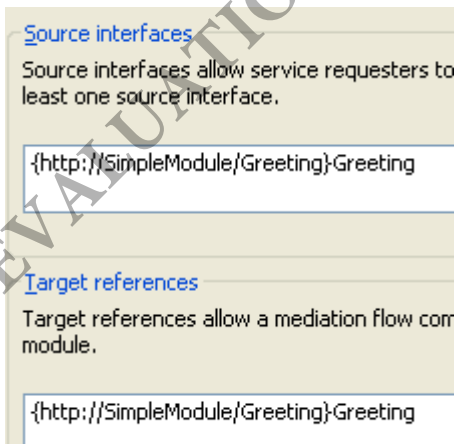
The sources interfaces will be updated.

As the target interface, also add the Greeting interface. The actual service provider supports this interface.

__6. Click the **Add** button for Target references.

__7. The Greeting interface should be selected, just click **OK**.

The target references will be updated.



The source and target interface does not have to be the same. In fact, the data translation pattern of ESB requires that they be different. In that case, the actual service consumer and provider support different service interfaces. The mediation flow acts as the bridge. We will see the implementation of this pattern in a lab later.

__8. Click **Finish**. WID will create the mediation flow and open it in the editor.

Part 6 - Design the Mediation Flow

Our mediation flow will be the simplest kind. There will be no mediation primitive in the

message flow. We will simply pass the input message straight to the service provider. Similarly, the response from the service provider will be sent back to the consumer.

__1. At the top of the editor, you will see two interfaces.

On the left hand side, we see the source interface. On the right hand side, we see the partner reference. The partner reference is a place holder for the actual service provider.

When a consumer calls the sayHello operation of the source interface, we will like the mediation flow to call the sayHello operation of the service provider. As a result, we need to connect to two operations.

__2. Move the mouse pointer over the sayHello operation of the source interface. A yellow lollipop will be shown.

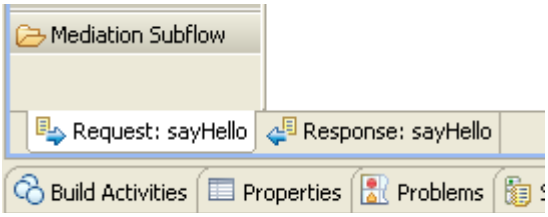
__3. Drag the lollipop and drop it on the sayHello operation of the target interface. This will connect the two.

Note: Two connected operations can have different names and completely different input and output data types. In that case, data and interface translation is applied within the mediation flow.

We will now design the request and response message flows for the sayHello source operation.

__4. Select the **sayHello** operation in the **source** interface.

__5. At the bottom pane, you will notice two tabs – **Request: sayHello** and **Response: sayHello**.

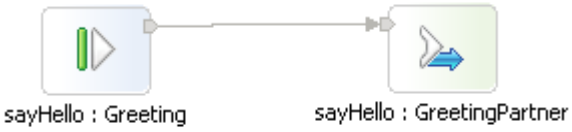


These tabs are used to switch the editor between request and response flow design.

Note: For one-way operations, there is no response message. In that case, the response flow tab is not shown.

__6. Click the **Request: sayHello** tab to make sure that the request flow design is open.

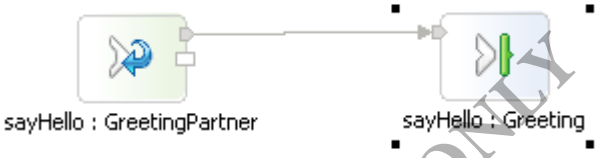
__7. Connect the output terminal of **sayHello : Greeting** (the source interface) to the input terminal of **sayHello : GreetingPartner**.



This creates a very simple passthrough message flow. Later, we will learn how to add mediation primitives in that flow to solve complex problems.

__8. Click the **Response: sayHello** tab to open the response flow design.

__9. Connect the output terminal of **sayHello : GreetingPartner** (the target interface) to the input terminal of **sayHello : Greeting**. Make sure you select the pointed output terminal of the GreetingPartner and not the rectangular fail terminal.

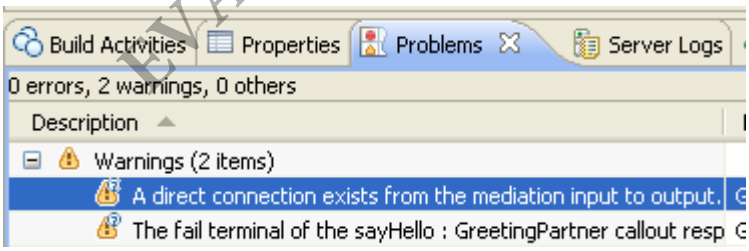


That means, the output from the service provider is passed as is to the consumer.

__10. Save and close the editor.

__11. Open the **Problems** view and expand **Warnings**. You should notice a warning about "A direct connection exists from the mediation input to output." This is fine for our very simple mediation flow.

There should be no error messages.



Part 7 - Create Service Provider

We will now implement the actual service provider. According to our design, it conforms to the Greeting service interface.

__1. Double click **Assembly Diagram** to open it.

__2. Right click the white area of the editor and select **Add > Java**.

__3. Change the name of the Java component to **GreeterBean**

__4. Right click the component and select **Add > Interface**.

__5. Select the **Greeting** interface and click **OK**.

__6. Right click the component again and select **Generate Implementation**.

__7. Click **New Package**.

__8. Enter **com.webage.greet** as the package name.

__9. Click **OK** to create the package.

__10. Click **OK** to select the new package for the implementation.

WID will create a class called com.webage.greet.GreaterBeanImpl and open it in the editor.

__11. Change the implementation of the **sayHello** method as follows.

```
public String sayHello(String name) {
          System.out.println("Saying hello to: " + name);

          return "Hello " + name;
}
```

__12. Save all files (Control+Shift+S).

Note: In this case, our Java class is implementing a service interface which is defined in a WSDL file. This is a unique feature of SCA. SCA uses a strict pattern where a service implementation, no matter how it is implemented, must support a WSDL based interface. In this case, the implementation is a Java class.

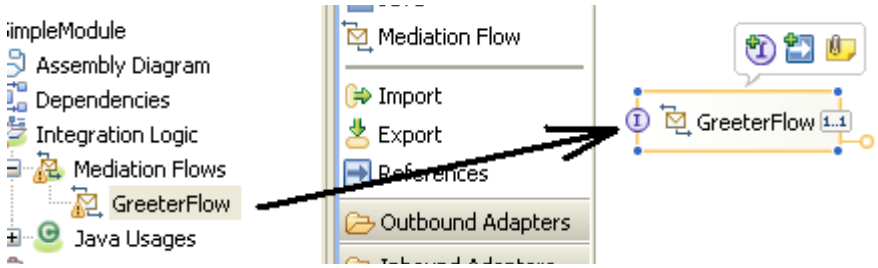
__13. Close the Java editor.

Part 8 - Wire the Components

Our mediation flow has a reference partner which is a placeholder for the actual service provider. We must wire that reference partner to the Java based service provider we have created above.

__1. Make sure that the assembly diagram is still open. If not open it.

__2. Drag the **GreeterFlow** mediation flow on to the diagram.



__3. Hover the mouse over the reference partner of the mediation flow and a lollipop will open up.

__4. Drag the lollipop and drop it over the interface of the Java implementation. This will connect the two as shown below.

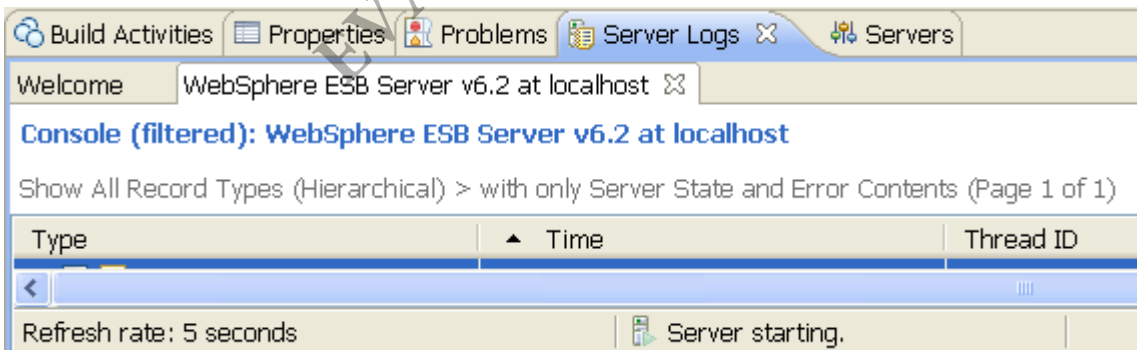
__5. Save changes.

Part 9 - Deploy the Application to Server

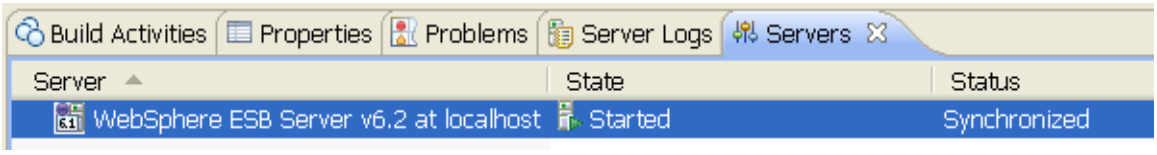
__1. Open the **Servers** view.

__2. Right click **WebSphere ESB Server v6.2 at localhost** and select **Start**.

The **Server Logs** view will open showing some messages.



Finally the Server changes to **Started**.



__3. Right click the server again and select **Add Remove Projects...**

__4. Click **Add All**.

__5. Click **Finish**.

__6. WID will start publishing the application files. Wait for the status of the server to change to **Synchronized**.

Part 10 - Test the Application

We will test the application by invoking the sayHello operation exposed by the mediation flow. We will use the built-in test client for that.

__1. Make sure that the server status is **Synchronized** before doing any test.

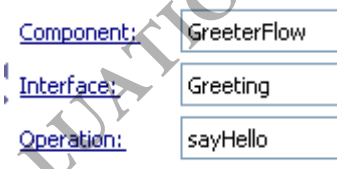
__2. Open the assembly diagram if it is not open.

__3. Right click any white area of the assembly diagram and select **Test Module**.

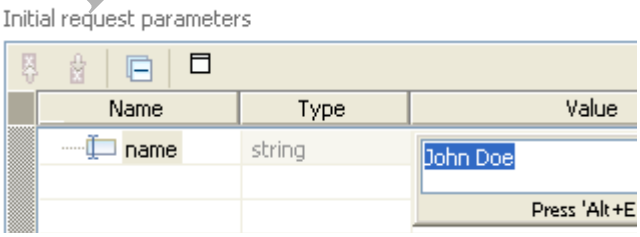
WID will open the test client page.

__4. In the test client page, under **Detailed Properties**, set the **Component** to **GreeterFlow.** That means, the test client will invoke the operations of the mediation flow.

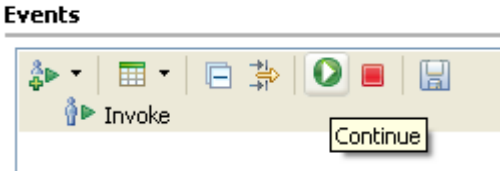
__5. Verify that the **sayHello** operation is selected since that is the only operation available from the interface.



__6. In the **Initial request parameters** table, set the value of the name parameter to something.



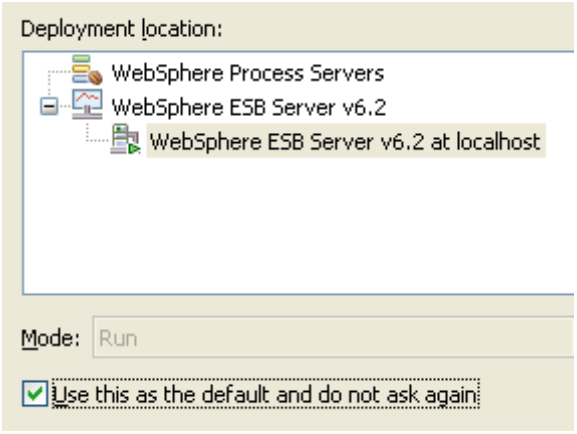
__7. Under **Events** click the **Continue** toolbar button to invoke the operation.



__8. In the **Deployment Location** dialog, expand **WebSphere ESB Server v6.2** and

select **WebSphere ESB Server v6.2 at localhost**.

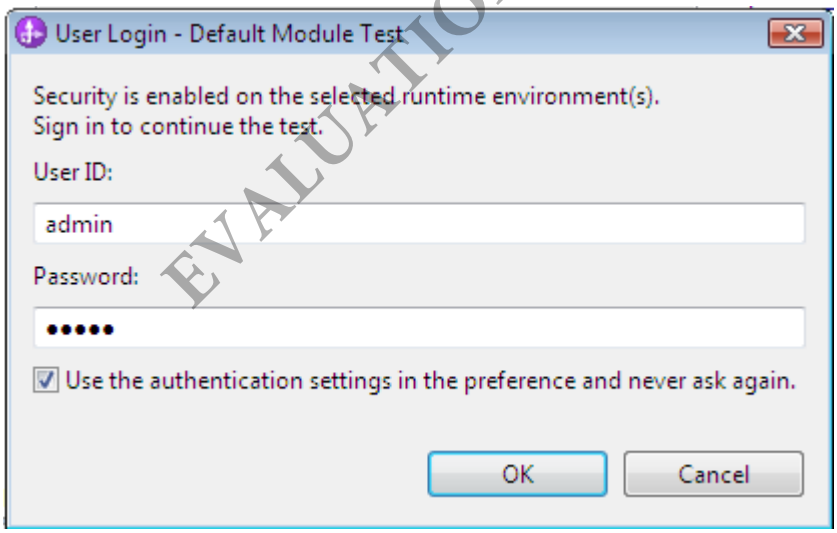
__9. Also, check **Use this as the default and do not ask again**.



Note: Here, WID is trying to deploy a special application (TestController62) that is used by the test client to carry out testing.

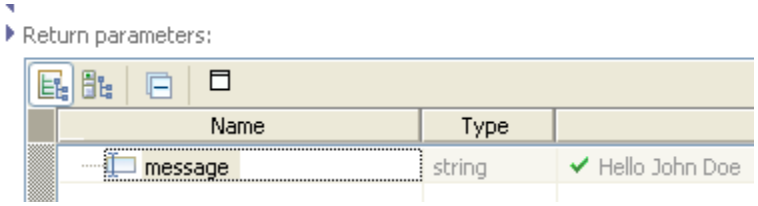
__10. Click **Finish**.

__11. Check the checkbox to always use the same authentication settings and click **OK** in the **User Login** dialog.

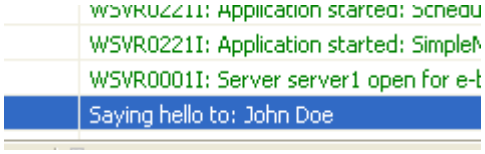


Security is enabled for the server. This is where the administrative user ID and password is entered. Without a valid password, publishing will fail.

__12. After the test ends, the **Return parameters** table will show the reply.



__13. Open the **Server Logs** view. At the very bottom, the output from the Java component will be shown.



__14. Close the test page without saving.

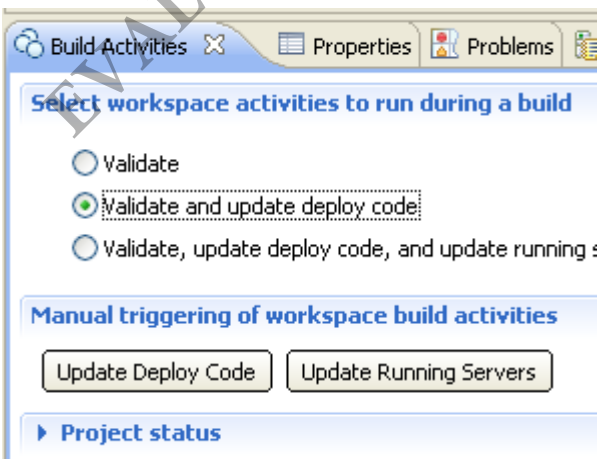
Part 11 - Configure Build Activities

Every time you change an SCA artifact file or assembly diagram, WID performs a build. As a part of the build, three things have to happen.

1. Validate the changes made.
2. Generate new code and compile them.
3. Publish updated files to the server.

You can configure how many of these three steps are performed automatically.

__1. Open the **Build Activities** view.



Note that by default **Validate and update deploy code** is selected. Here, "update deployed code" is referring to code generation.

That means, updated files are not published to the server by default after a build. You can click on the **Update Running Servers** button to do that. Many of the changes will be picked up by the server after files are published. In a few rare cases, we have to restart the

server.

__2. Close all open files. (Ctrl+Shift+W)

Part 12 - Review

In this lab, we created a very simple mediation flow. During testing, we invoked the sayHello operation of the mediation flow. The flow, in turn, invoked the sayHello operation of the reference partner. The partner was wired to a Java implementation. As a result, the sayHello method of the Java class got invoked.

The response from the service provider Java class was returned back to the test client.

A few things to remember from this lab:

1. Every mediation flow has a source interface and one or more target interface.
2. For each target interface, a reference partner is defined in the mediation flow.
3. A method from the source interface is connected to one or more methods of the target interfaces.
4. For each source method, you need to design the request and response flow.
5. Every reference partner needs to be wired to an actual service implementation (or an import if the implementation exists outside the module).

Lab 2 - Using Mediation Primitives

In the previous lab we have not used any mediation primitive in the request or response flow. In this lab, we will add the message logger primitive. This primitive can save a message in a file or database.

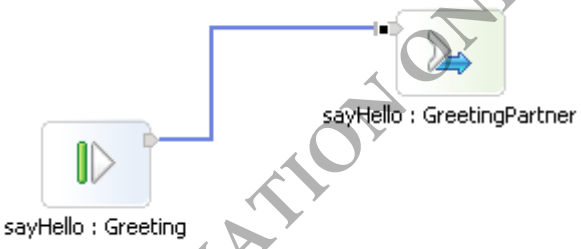
Now, for the first time, our mediation flow will begin to offer some real value. In this case, we are essentially implementing the audit logging pattern. A part of the ESB pattern, audit logging pattern saves request and response messages for latter retrieval.

Part 1 - Add Primitive to Request Flow

__1. Open **GreeterFlow** mediation flow in the editor by double clicking on it.

__2. Select **sayHello** operation in the source interface. This will open the message flow in the editor.

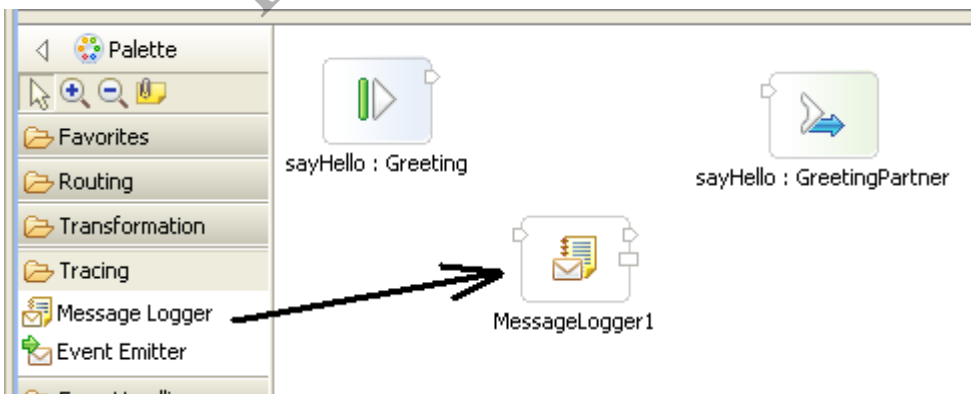
__3. In the bottom pane, for the request message flow, select the connection between **sayHello : Greeting** and **sayHello : GreetingPartner**.



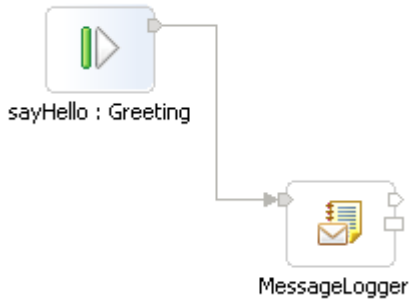
__4. Press the delete key to remove the connection.

__5. On the left hand side of the editor, expand the **Tracing** section.

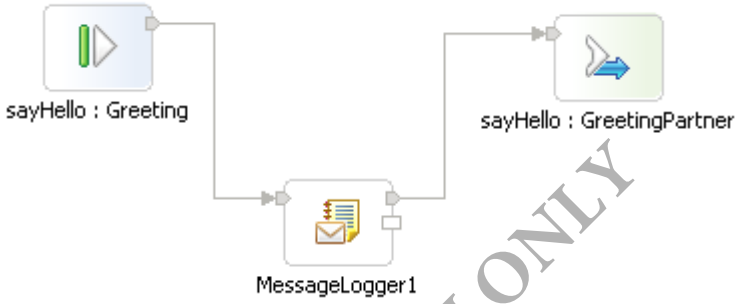
__6. Drag and drop the **Message Logger** primitive on the editor.



__7. Connect the output terminal of **sayHello : Greeting** to the input terminal of **MessageLogger1** as shown below.



__8. Connect the output terminal of **MessageLogger1** to the input terminal of **sayHello : GreetingPartner**.



This means, the request message will first go inside the logger primitive. This primitive does not alter the message in any way and outputs the same message that came in as input. The output of the logger is then fed into the reference partner as input.

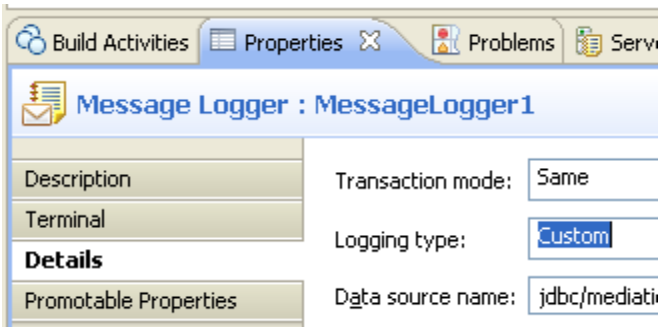
Now, we will configure the logger.

__9. Select **MessageLogger1**.

__10. Open the **Properties** view.

__11. Select the **Details** tab.

__12. Change the **Logging type** from Database to **Custom**.



Note: By default, a logger saves messages in a database. The database is pointed to by the **Data source name** property. The default data source name is jdbc/mediation/messageLog. This points to a cloudscape (Derby) database where the necessary schema is already defined. Alternatively, you can point message logger to a

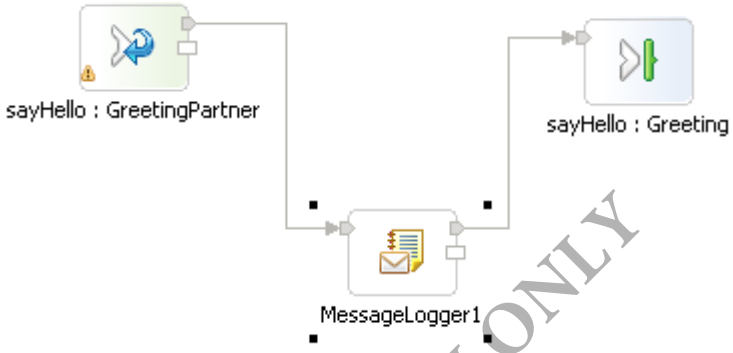
DB2 or Oracle database.

If you set the logging type to custom, then the logger saves the messages in a file.

Part 2 - Add Primitive to Response Flow

__1. At the bottom of the editor, click the **Response: sayHello** tab to switch to the response flow.

__2. Follow the steps in the previous part to insert a logger primitive in the flow.



__3. Change the logging type of the logger to **custom**.

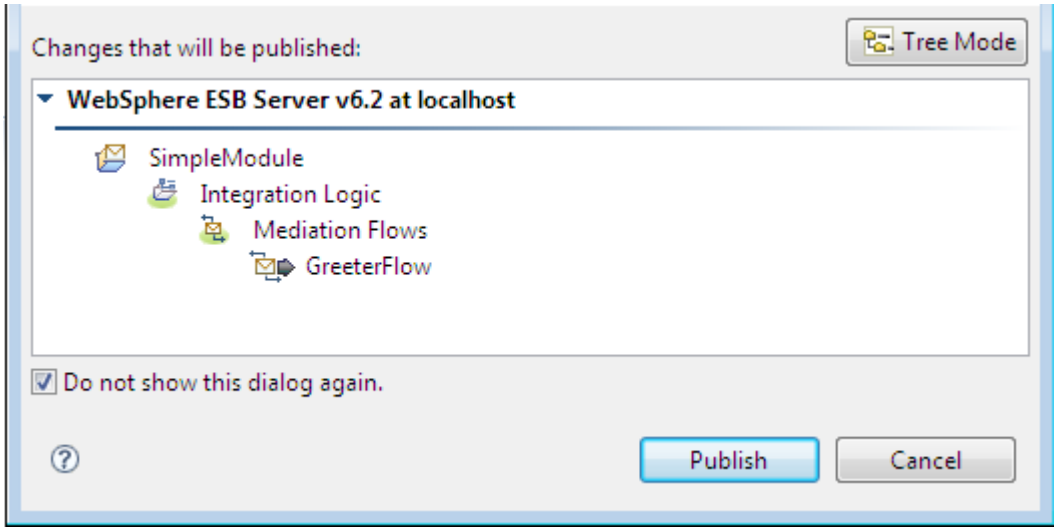
__4. Save the mediation flow.

Part 3 - Publish Changes

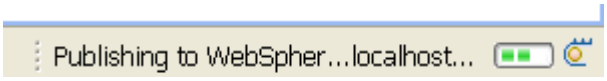
__1. Open the **Build Activities** tab.

__2. Click **Update Running Servers**.

__3. Check the '**Do not show this dialog again**' box and press the **Publish** button.



The server will start publishing, wait until is done.



Part 4 - Test

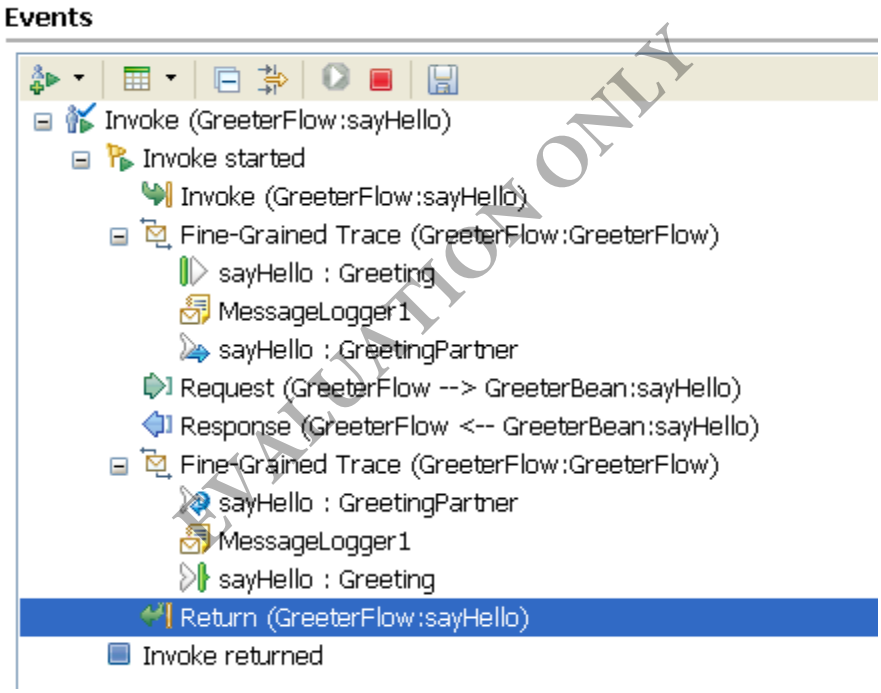
__1. Open the **assembly diagram**.

__2. Right click the white area and select **Test Module**.

__3. Set the component to **GreeterFlow**. It may default to GreeterBean but this would not test the whole flow.

__4. Enter a name in the input table.

__5. Click the **Continue** toolbar button under **Events**.



__6. Close the test without saving it.

We will now inspect the messages saved in the log files. Messages are saved in the folder pointed to by the TEMP environment variable (both in Windows and UNIX).

__7. Open a command prompt window.

__8. Enter the following command:

```
cd  /d  %TEMP%
```

__9. List all message log files:

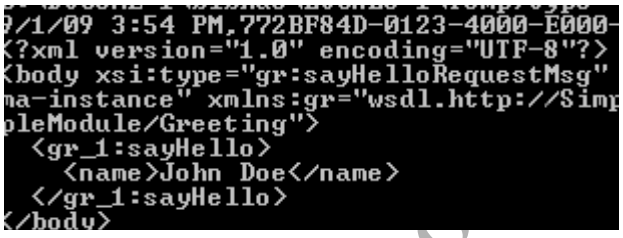
```
dir  *.log
```

__10. You should notice these two log files.

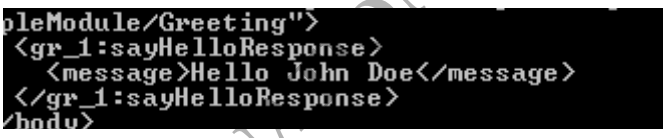
- MessageLog0.log – Contains request messages.
- MessageLog1.log – Contains response messages.

__11. View the request message as follows.

```
type  MessageLog0.log
```



__12. Similarly, view the response message in MessageLog1.log.



__13. Close the Command prompt window.

__14. Back in WID, close all open files.

Part 5 - Review

In this lab, we implemented the audit logging pattern. Note, how the pattern can be used to audit log a service without changing the service implementation. Now, you are beginning to see some of the advantages of using an ESB as an intermediary.

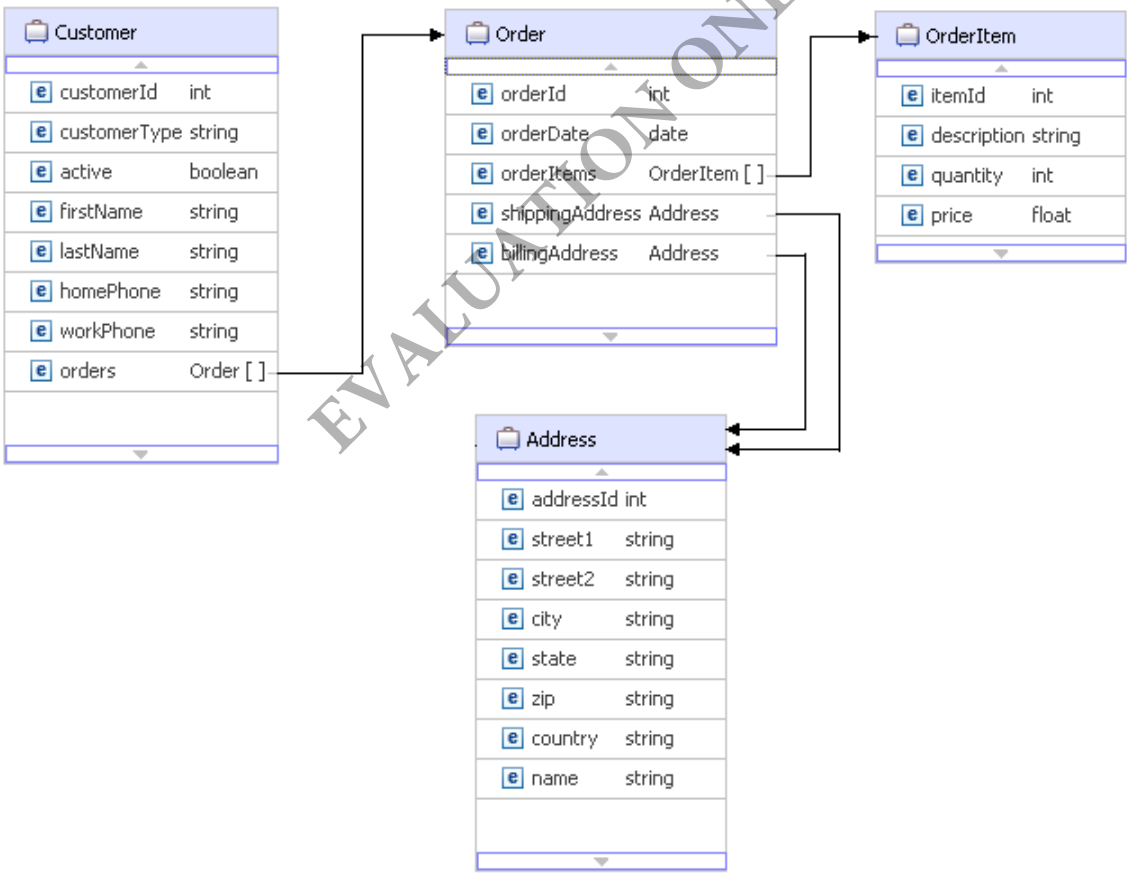
Lab 3 - Create Business Objects

Business objects represent the data that's passed between service components in your SCA modules. When you create interfaces (WSDL or Java) for your components, which you'll do in the next lab, the input and output parameters correspond with business objects. Business objects are defined using XML Schema Definition (XSD) files.

WebSphere ESB and Process Server uses the Service Data Object (SDO) API to work with business objects. Specifically, the **commonj.sdo.DataObject** interface represents a data object.

Business objects can store both simple attributes (e.g., string, int, double, boolean) and complex attributes (i.e., other business objects). An attribute can be multi-valued, which means it can be stored as an array.

In this lab, you will create several business objects. Specifically, you will create Customer, Address, Order, and OrderItem business objects which are inter-related. A Customer can have multiple Orders. An Order can have multiple OrderItems. An Order can also have a shipping address and billing address, both of which are of type Address.



Following this exercise, you should be able to:

- Create a default namespace.
- Create a business integration module.
- Create business objects using the Business Object Editor.
- View the physical files created by WID

Part 1 - Create a Default Namespace

Namespaces are used in XML to uniquely identify XML elements and attributes. They help prevent collisions between elements and attributes with the same names. Namespaces are analogous to Java packages which uniquely identify class names. When creating an XML schema, a target namespace is specified. All elements and attributes defined in the schema belong to that target namespace.

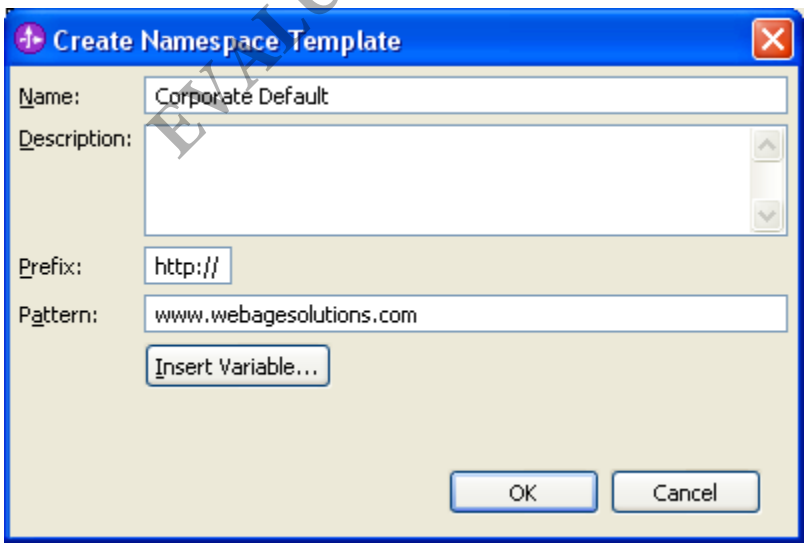
In this part, you will create a default namespace that's used when creating business objects and other kinds of XML artifacts. Namespaces typically correspond with a URI (usually a URL), since they must be unique and URIs are unique. You will set the default namespace to http://www.webagesolutions.com. Normally, you would set it to your company's URL.

__1. Select **Window->Preferences** from the menu bar.

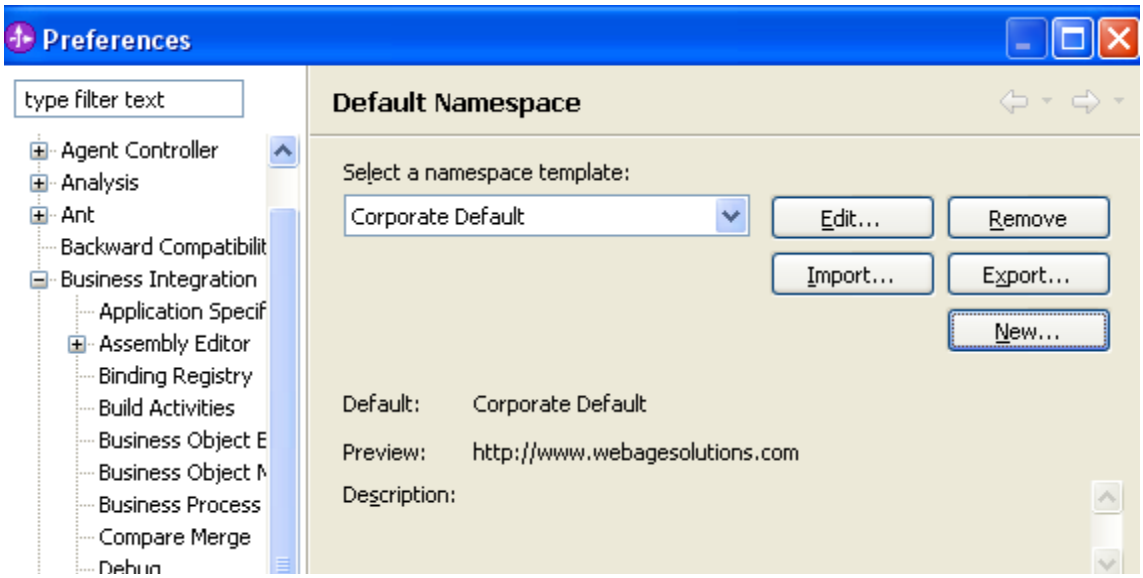
__2. In the left pane, expand **Business Integration** and select **Default Namespace**.

__3. In the right pane, click the **New** button to create a new default namespace.

__4. Type **Corporate Default** in the **Name** field and **www.webagesolutions.com** in the **Pattern** field.



__5. Click **OK** to create the namespace.



__6. Click **OK** to close the preferences.

Part 2 - Create a Business Integration Module

In this part, you will create a business integration module (module for short) for storing your business objects.

__1. Select **File > New > Mediation Module** from the menu bar.

__2. Enter **CustomerOrder** for the **Module Name**.

__3. Uncheck **Create mediation component**.

__4. Uncheck **Open module assembly diagram**.

__5. Click **Finish**. The **CustomerOrder** project should appear in the **Business Integration** view.

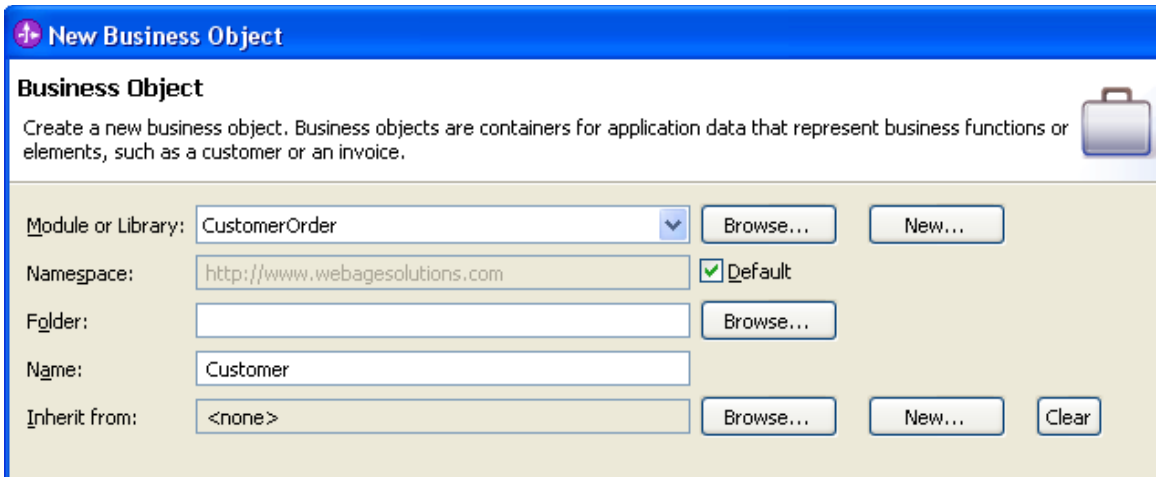
Part 3 - Create the Customer Business Object

In this part, you will create the Customer business object using the Business Object Editor. You will add simple fields to it, such string, boolean, and int. You will also see how you can specify metadata for the fields, including whether or not its required, has enumerated values and/or matches a pattern.

__1. In the **Business Integration** view, under the **CustomerOrder** module, right click **Data Types** and select **New > Business Object**.

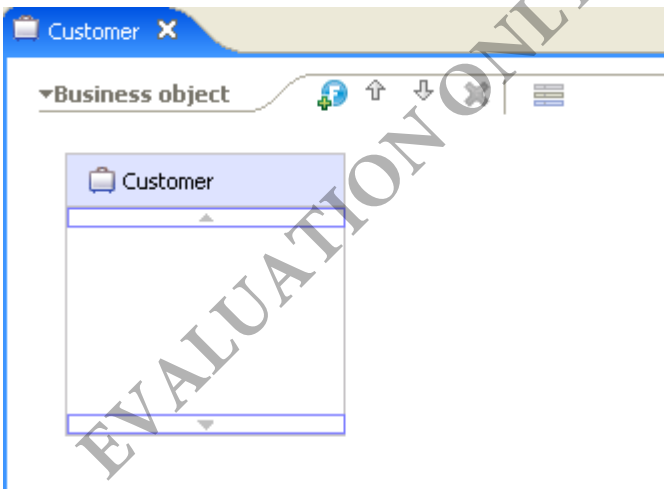
__2. Notice that the **Namespace** field is pre-populated with the default you set earlier: **http://www.webagesolutions.com**.

__3. Enter **Customer** in the **Name** field.

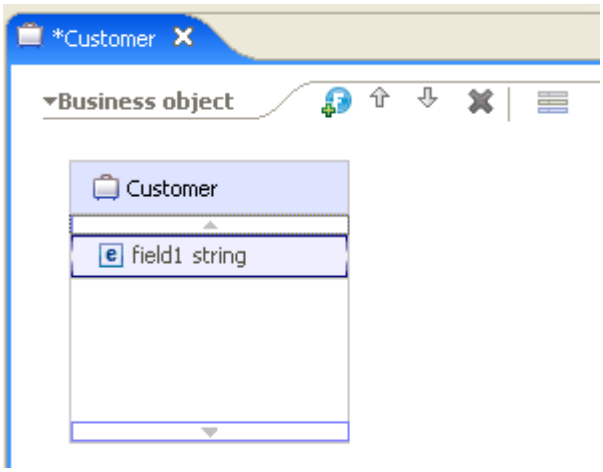


__4. Click **Finish**.

The **Customer** business object will be created and should automatically be opened inside the Business Object Editor.



__5. Now let's define our Customer business object. Select the **Customer** object. Then click the field icon (F in a blue circle) to add a field to the business object. A field called **field1** should be added to the Customer object of type **string**.



__6. Click on **field1** and rename it **customerId**. You will need to hit <ENTER> to accept the new name before you can make other changes.

__7. Click on **string** and select **int** from the drop-down.

__8. Using the procedure above, add the following fields:

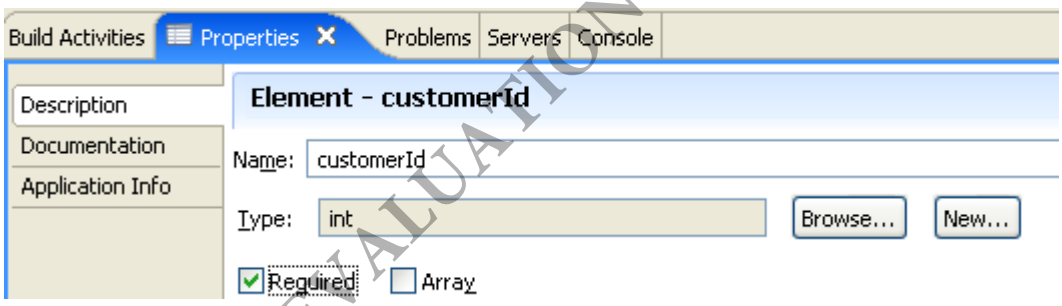
Name	**Type**
customerType	string
active	boolean
firstName	string
lastName	string
homePhone	string
workPhone	string

Your business object should look as follows:

__9. Save your changes.

__10. Since the customerId uniquely identifies a customer, it should be required. To specify this, right click on **customerId** and select **Show in Properties** from the menu.

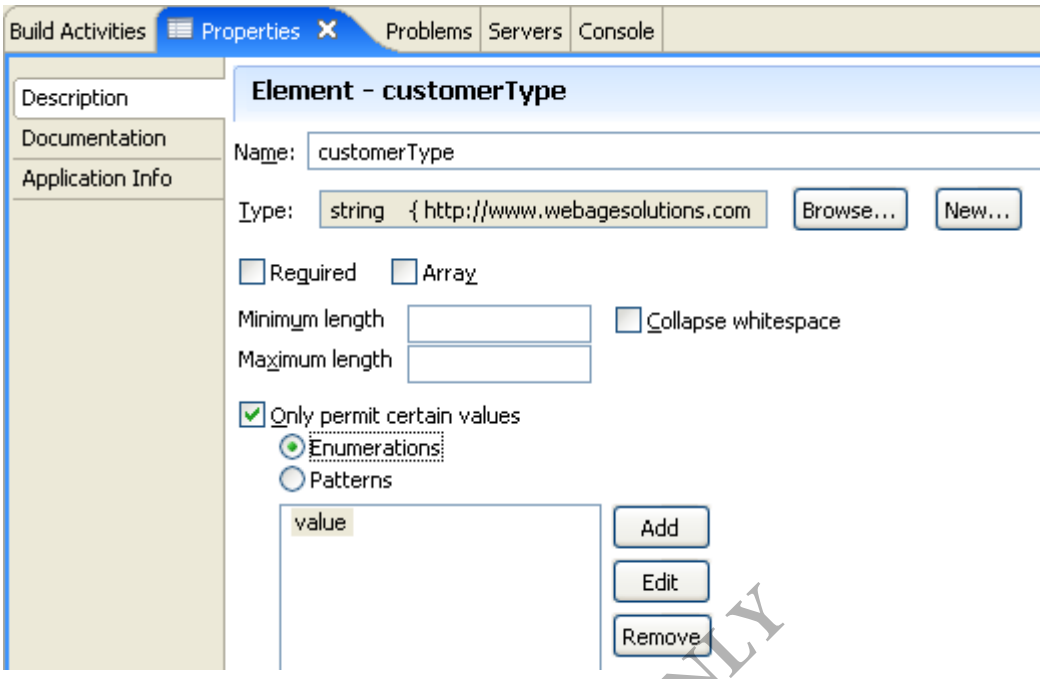
__11. The **Properties** view should appear at the bottom of WID. Click the **Required** checkbox.



__12. It's also possible to specify that a field can only contain certain values using the enumerations feature. For example, the customerType can only be set to the values silver, gold, or platinum. Click on the **customerType** field in the Business Object Editor.

__13. In the **Properties** view, check the **Only permit certain values** checkbox.

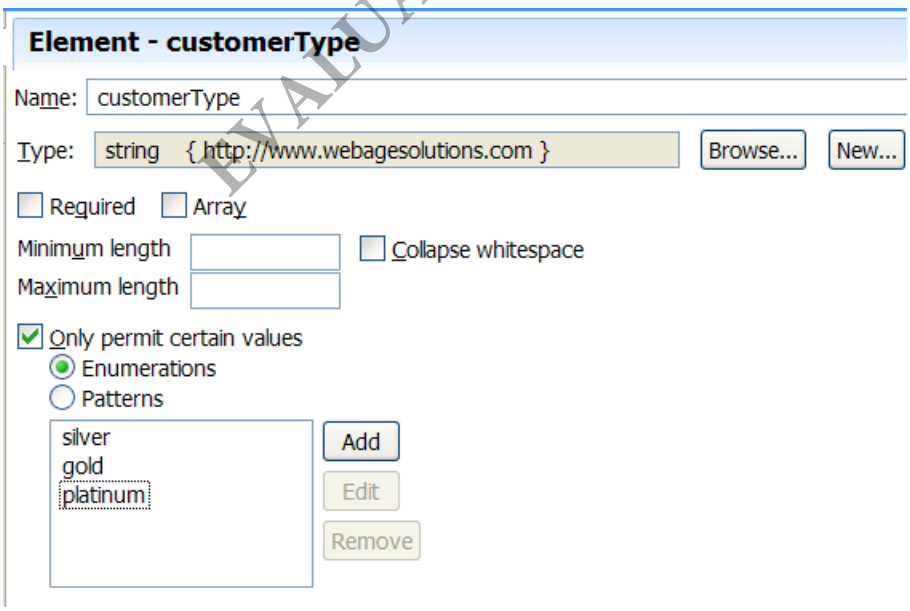
__14. Then, click the **Enumerations** radio button.



__15. By default, a single entry named **value** appears in the list box. Click the **Edit** button and rename it as **silver**. If the value is already highlighted you may not need to click the Edit button.

__16. Click the **Add** button and rename the **value** entry as **gold**

__17. Likewise, add the entry **platinum**



__18. You can also specify a regular expression pattern that a field must match when being input. The homePhone and workPhone fields should be entered in the format: xxx-xxx-xxxx. To accomplish this, click on the **homePhone** field in the Business Object

Editor.

__19. In the **Properties** view, check the **Only permit certain values** checkbox.

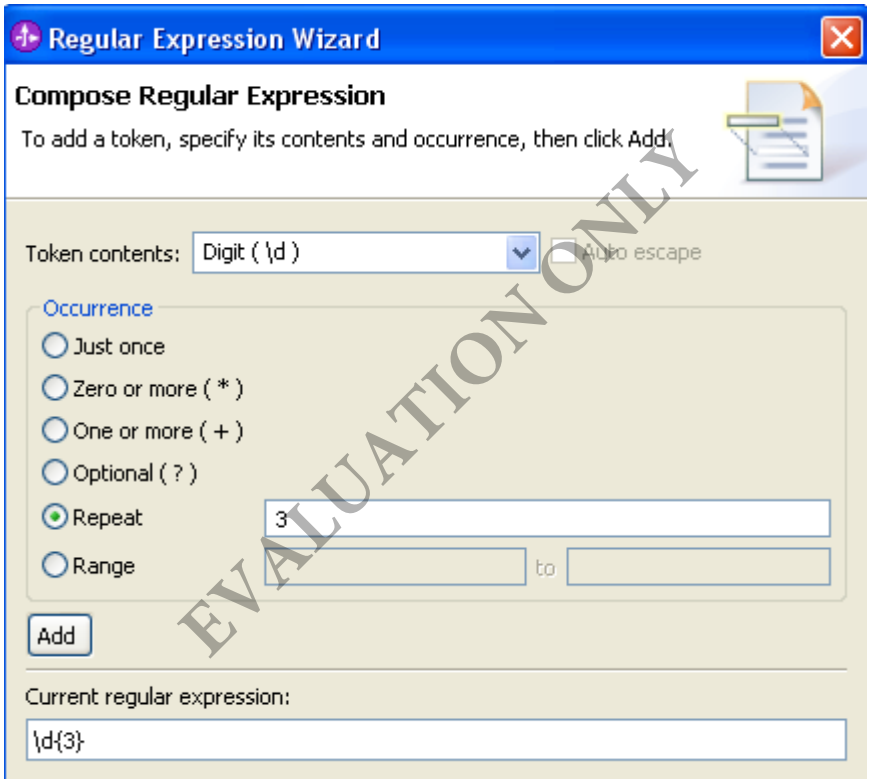
__20. Then, click the **Patterns** radio button.

__21. In the **Regular Expression Wizard** dialog, select **Digit (\d)** from the **Token contents** drop-down.

__22. Then, click the **Repeat** radio button and type 3 in the text field to the right of it.

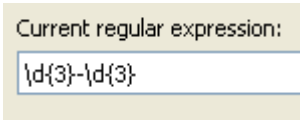
__23. Then click the **Add** button.

The **Current Regular Expression** field should now show **\d{3}** which means the user needs to type in 3 digits for the phone number. We're not done yet.



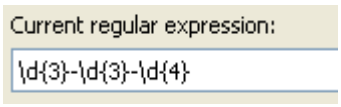
__24. Type a dash (-) next to **\d{3}** in the **Current regular expression field**.

__25. Click the **Add** button again to append **\d{3}** to the regular expression.



__26. Type another dash (-) next to **\d{3}-\d{3}** in the **Current regular expression field**.

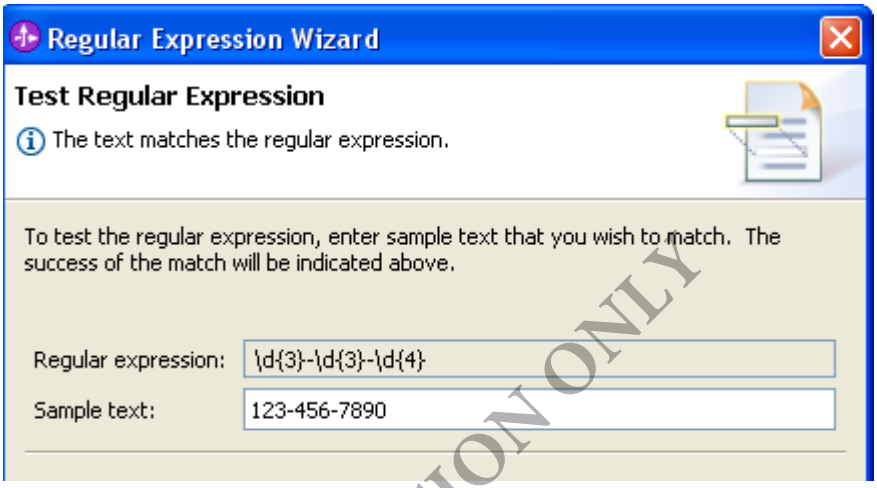
__27. Change the **Repeat** value from 3 to 4 and click **Add** again. Voila! Your regular expression is complete.



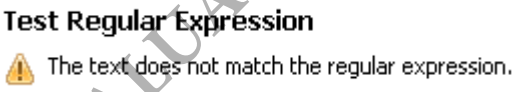
__28. WID also allows you test your regular expression to make sure you input it correctly. To do so, click the **Next** button.

__29. In the **Sample text** field, enter **123-456-7890**

__30. The message **The text matches the regular expression** should appear at the top of the dialog.



Try entering an wrongly formatted phone number.



__31. Click **Finish**.

__32. Repeat the process for the **workPhone**.

__33. Save your changes.

__34. Close the file.

Part 4 - Create the Address Business Object

Up until this point, you've created fields having only simple data types. It's also possible to create fields having complex data types. The complex data types are themselves other business objects. For example, an Order, which you will be creating shortly, has a shipping address and a billing address. Since an Order has multiple addresses rather than just one, it's recommended to create another business object called Address, instead of simply adding address fields to the Order business object. In this part, you'll create the Address business object.

__1. In the **Business Integration** view, right click on **Data Types** under **CustomerOrder**

and select **New > Business Object**.

__2. Type **Address** in the **Name** field.

__3. Click **Finish**.

The **Address** business object will be created and should automatically be opened inside the Business Object Editor.

__4. Enter the following fields:

Name	**Type**	**Required**
addressId	int	X
street1	string	
street2	string	
city	string	
state	string	
zip	string	
country	string	
name	string	

Your business object should look as follows:

Note
You added a **name** field to Address, since a customer may have an order shipped to an address where she doesn't live. The name would correspond with the name of the person who lives at that address.

__5. Save your changes.

__6. Close the file.

Part 5 - Create the Order and OrderItem Business Objects

In this part, you'll create the Order business object. You'll use the Address business object you created in the last part when declaring the shippingAddress and billingAddress fields. You'll also add an orderItems field to the Order business object. When declaring the orderItems field, you'll create an OrderItem business object on the fly, since an order can consist of multiple order items. Afterwards, you'll complete the Customer business object by adding a field called orders to it of type Order. Since both orderItems and orders are multi-valued, you'll declare them as arrays.

When you create a business object on the fly (e.g., OrderItem) when defining the parent business object, this is referred to as a top-down based approach to business object development. On the other hand, when you create the business object in advance (e.g., Address and Order), this is referred to as a bottom-up based approach to business object development. Both approaches are valid.

__1. In the **Business Integration** view, right click on **Data Types** under **CustomerOrder** and select **New > Business Object**.

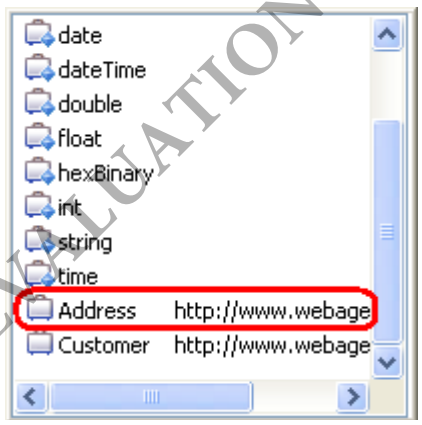
__2. Type **Order** in the **Name** field.

__3. Click **Finish**.

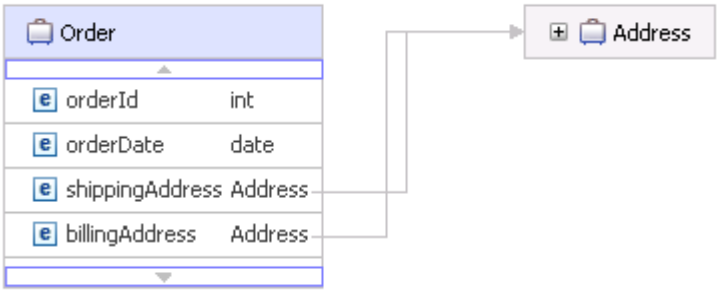
__4. The **Order** business object will be created and should automatically be opened inside the Business Object Editor. Enter the following fields:

Name	Type	Required
orderId	int	X
orderDate	date	
shippingAddress	Address	
billingAddress	Address	

__5. When specifying the **Address** type, you'll need to scroll to the bottom of the type drop-down, since that's where all business objects are located.



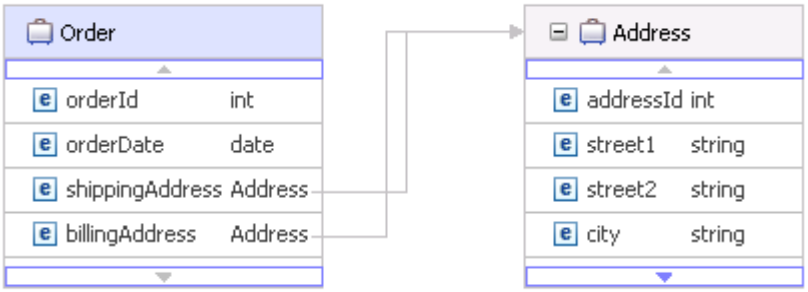
Your business object should look as follows:



Notice that **Address** was added to the diagram and two arrows were drawn between

Order and **Address** showing the parent-child relationship between the two business objects.

__6. Click on the plus (+) sign next to **Address** to expand it and see its fields. Use the bottom arrow to scroll through its fields.



Also, notice that you can't edit any of Address's fields. To edit Address, you can simply double click on **Address** and it will open up in the Business Object editor.

__7. Return to the business object editor showing the **Order** business object if you are not there already.

__8. Add another field named **orderItems**

__9. This time, when specifying the data type, click the **New...** option in the drop-down. It's located at the top of the drop-down.

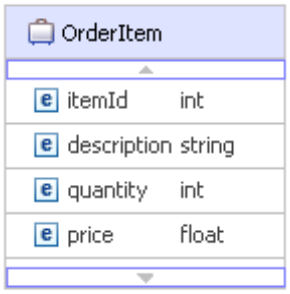
__10. In the **New Business Object** dialog, type **OrderItem** in the **Name** field and click **Finish**.

The **OrderItem** business object will be created and should automatically be opened inside the Business Object Editor.

__11. For the OrderItem business object, enter the following fields:

Name	**Type**	**Required**
itemId	int	X
description	string	
quantity	int	
price	float	

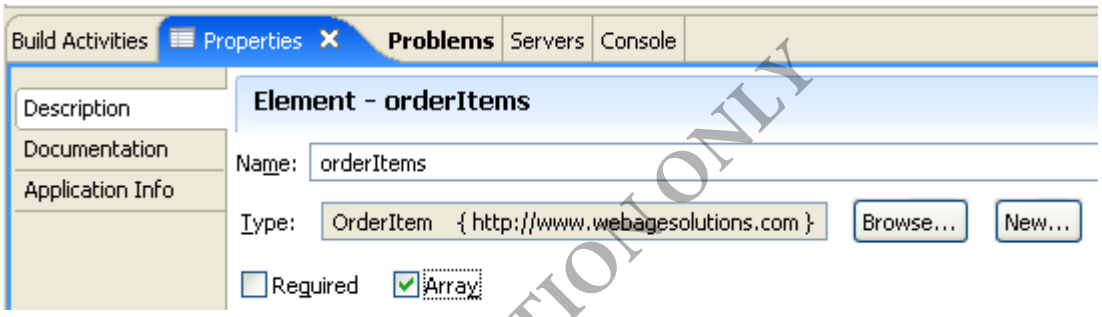
__12. Your business object should look as follows:



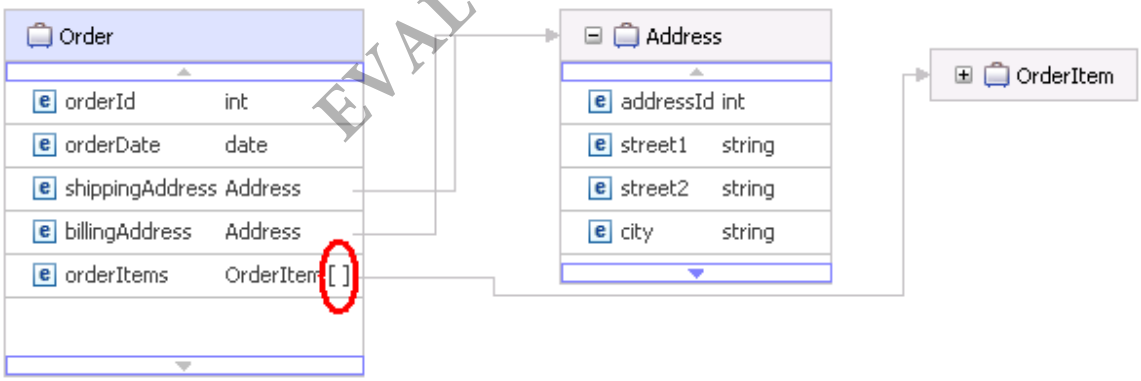
__13. Save your changes and close the **OrderItem** file.

__14. Back in the **Order** Business Object Editor, specify that an Order can have more than one OrderItem by clicking on the **orderItems** field.

__15. In the **Properties** view, check **Array**.



Notice that a set of square brackets ([ ]) now appears to the right of **orderItems** indicating that it contains an array of OrderItem business objects.

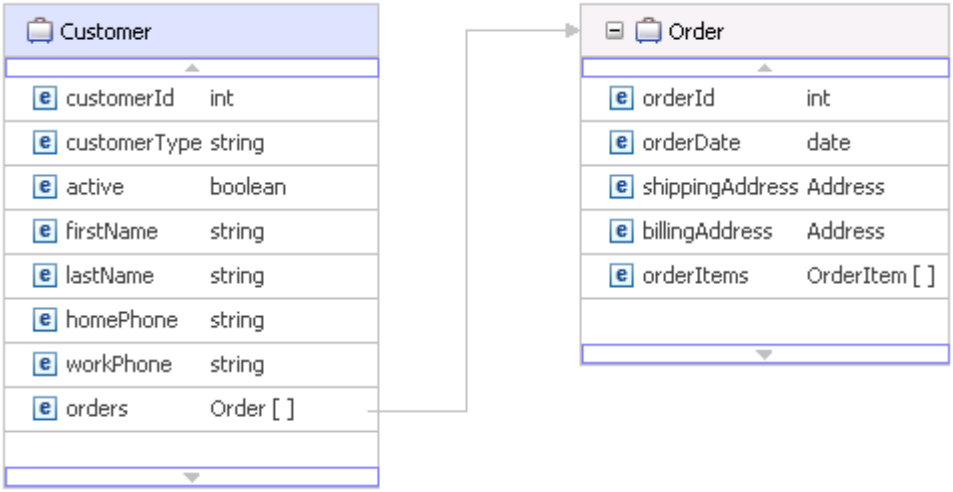


__16. Save your changes and close the file.

__17. Last but not least, open the **Customer** business object.

__18. Add a field to the Customer business object named **orders** of type **Order**. You may need to click the Customer business object in the editor to select it first.

__19. Declare the field as an array.



Note
WID allows you to view business objects only one level deep. Consequently, you can't view OrderItem in the Customer's Business Object Editor.

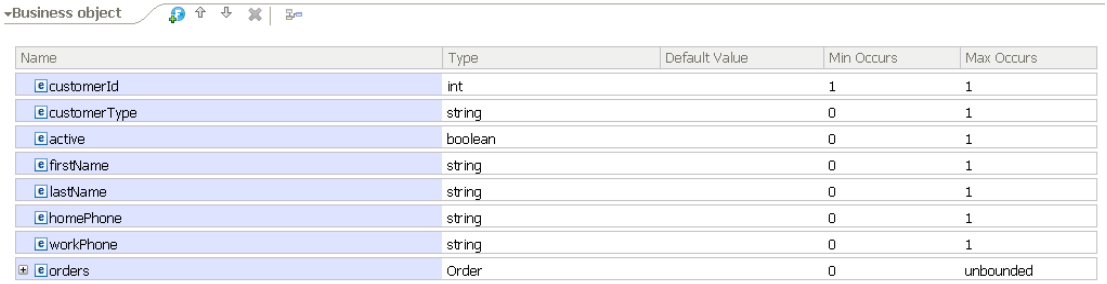
__20. Save your changes.

__21. It's also possible to display the Customer business object in a tabular format. To do so, click on the icon on the far right of the editor toolbar.

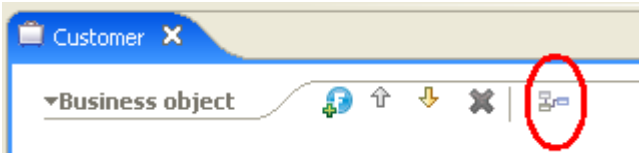
__22. Now you're in the table view. In addition to displaying the field names and data types for the business object and its children, the table view displays their default values (if any) and multiplicity (number of minimum occurrences and maximum occurrences).

Also observe the following:

- **customerId** has a **Min Occurs** and **Max Occurs** both set to **1**. This means that it must have one and only one value, since we declared it as a required field.
- **orders** has a **Min Occurs** set to **0** and a **Max Occurs** set to **unbounded**. This means that a Customer can have 0 or more orders. These values are used when specifying arrays. Likewise, **orderItems** has a **Min Occurs** set to **0** and a **Max Occurs** set to **unbounded**, since it's also an array.
- All of the other attributes have a **Min Occurs** of **0** and **Max Occurs** of **1**, which means they're optional fields (as opposed to required fields).



__23. Switch back to the diagram view by clicking on the icon on the far right of the editor toolbar. Notice the icon has changed.



__24. Close all open files. Save if there are any unsaved changes.

Congratulations! You've successfully created a hierarchy of business objects.

Part 6 - View the Physical Files

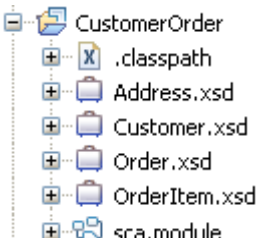
In this part, you'll see the files that get created behind the scenes when you create a business object. Specifically, you'll see that an XML schema definition (XSD) was created for each business object.

WID tries to keep things as "logical" as possible to make things easier on the *integration developer*, since he/she is only required to have basic programming skills (knows how to create loops, conditionals, and perform string manipulation). WID provides graphical editors wherever possible to simplify and abstract complex programming details.

__1. In the **Business Integration** view, right click on the **CustomerOrder** module and select **Show Files**.

__2. WID will open the **Physical Resources** view and stack it on top of the **Business Integration** view. Notice that the business objects you created are located directly underneath the **CustomerOrder** project and correspond with XSDs (can tell based on the file extension).

Note
In a future lab, you will begin to use folders when creating business objects and other kinds of SCA artifacts, so that the resulting files are better organized. This is especially important when creating larger projects.



__3. Double click on Customer.xsd. Notice that WID still opens it with the Business Object Editor.

__4. To change this behavior, close the file. Then right click on it and select **Open With->Text Editor** from the menu. WID will now open it with a text editor so you can view the actual XML.

Notice the following:

- The **targetNamespace** attribute is set to **http://www.webagesolutions.com**, which is the default namespace we set at the beginning of the lab.
- The XML schema file **Order.xsd** is included so the Customer XSD can reference the Order element.
- **Customer** is declared as a complex type, since it contains child elements (correspond with the fields we declared).
- The Max Occurs and Min Occurs columns you saw earlier in the Business Object Editor's table view correspond with the **maxOccurs** and **minOccurs** XSD attributes.


```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   targetNamespace="http://www.webagesolutions.com"
   xmlns:bons1="http://www.webagesolutions.com">
   <xsd:include schemaLocation="Order.xsd"></xsd:include>
   <xsd:complexType name="Customer">
      <xsd:sequence>
         <xsd:element minOccurs="1" name="customerId"
            type="xsd:int">
         </xsd:element>
         ...
      </xsd:sequence>
   </xsd:complexType>
</xsd:schema>
```

__5. Close the file.

Note
You could have also opened the file with the full XML Schema editor. We opened it with the text editor just to see the contents easily.

__6. The Customer.xsd file will automatically be opened with the text editor from this point forward unless you change it back to open with the Business Object Editor. To

change this, right click on the file and select **Open With > Business Object Editor**.

__7. Close the Business Object Editor.

__8. Switch back to the **Business Integration** view.

Part 7 - Review

In this lab, you set a default namespace within WID for any XML files you create. Next, you created four inter-related business objects: Customer, Address, Order, and OrderItem. Business objects are used to represent data in a SOA. The business objects get passed to and fro between service components when they communicate with one another by invoking each other's operations. You used the Business Object Editor, which is quite powerful, to create the business objects. You also switched to the Physical Resources view and saw that WID creates XML schema files behind the scenes to represent the business objects.