

# ChainBuilder ESB

*Visual Enterprise Integration™*

Version 1.1

## Custom Component Guide



©Copyright 2007  
Bostech Corporation  
2800 Corporate Exchange Drive  
Suite 260  
Columbus, OH 43231

## **Acknowledgements**

This document contains proprietary information that is the property of Bostech Corporation. Any reproduction, disclosure, or transfer of this document or the information contained herein without the express written consent of Bostech Corporation is strictly prohibited.

The use of the information contained in this document and the implementation of any of its techniques are the sole responsibility of the client and depend on the client's ability to evaluate the information and implement it into the client's operational environment.

Except for any express written warranties made by it, Bostech Corporation makes no warranties or representations with respect to any information contained herein, whether express, implied, statutory, or otherwise, in fact or in law, including without limitation, any implied warranties of merchantability or fitness for a particular purpose; and in no event shall Bostech Corporation be liable for any special, consequential, indirect, punitive, or exemplary damages in connection with the use of the information contained herein. The information contained in this document is subject to change at any time without notice.

## **Trademarks**

The following trademarks and acknowledgments apply to the information presented in this manual:

- ChainBuilder is a registered trademark of Bostech Corporation.
- Adobe and Acrobat Reader are registered trademarks of Adobe, Inc.
- Java is a registered trademark of Sun Microsystems, Inc.
- Windows (NT, 2000, XP, and Server 2003), .NET Framework, Internet Information Services (IIS) are registered trademarks of Microsoft Corporation.

## **Credits**

The following third-party products are used within the ChainBuilder product, and acknowledgments apply to the information presented in this manual:

- Acrobat Reader is created and licensed by Adobe, Inc.
- This product includes software developed by the Apache Software Foundation (<http://www.apache.org/>)
- This product includes software developed by Eclipse (<http://www.eclipse.org/>)

## Table of Contents

1. Introduction.....	1
2. Creating a New Component .....	1
2.1. New Custom Component Wizard.....	1
2.1.1. Custom Component Properties Page.....	1
2.1.2. Consumer/Provider Property Pages .....	1
2.1.3. Java Build Properties.....	1
2.2. Custom Component Project Layout.....	1
2.2.1. Base Package .....	1
2.2.2. Processors Sub-Package .....	1
2.2.3. UI Sub-Package.....	1
2.2.4. WSDL Sub-Package .....	1
2.2.5. Ant Build Scripts.....	1
2.3. Example.....	1
2.3.1. HelloWorld_SEWSDL1Deployer.....	1
2.3.2. HelloWorld_SEPProviderProcessor.....	1
2.3.3. HelloWorld_SECustomComponent .....	1
2.3.4. Building the Component.....	1
3. Using a Custom Component .....	1
3.1. Custom Component Wizard .....	1
3.2. Example.....	1
4. ChainBuilder ESB Community.....	1

## 1. Introduction

The ChainBuilder ESB IDE provides the ability to create and use custom Binding Components and Service Engines.

First, a new project wizard may be used to built out the framework for a new component. The wizard gathers information about the components setup and generates a new Java project for the component. Most of the "plumbing" for the component is automatically generated, so the developer can concentrate on the core logic. The generated project also creates a configuration jar file that allows the end user to include the new component in the Component Flow Editor.

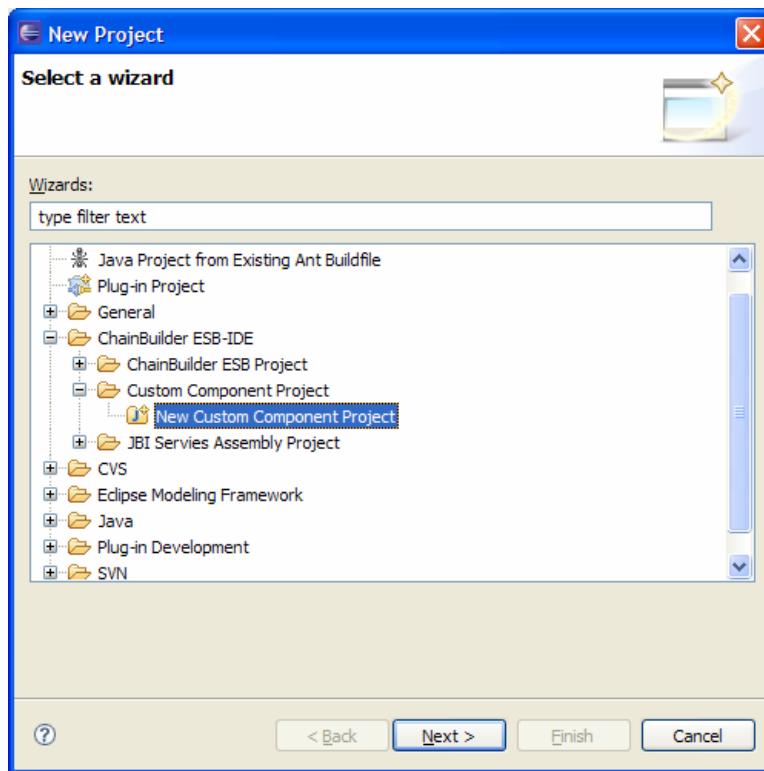
In order to create a useful component, it is important to understand the concepts in the JBI specification. It is not the intent of this document to provide a full explanation of what is involved in developing JBI components, but rather to give an overview of how the ChainBuilder ESB Custom Component Wizard can be used to generate the skeleton of a component. This skeleton does provide much of the JBI "plumbing", but the developer must still have an understanding of Message Exchanges, Normalized Messages, etc in order to create a useful component.

## 2. Creating a New Component

This section describes the process to create a new JBI component using the Custom Component Project Wizard.

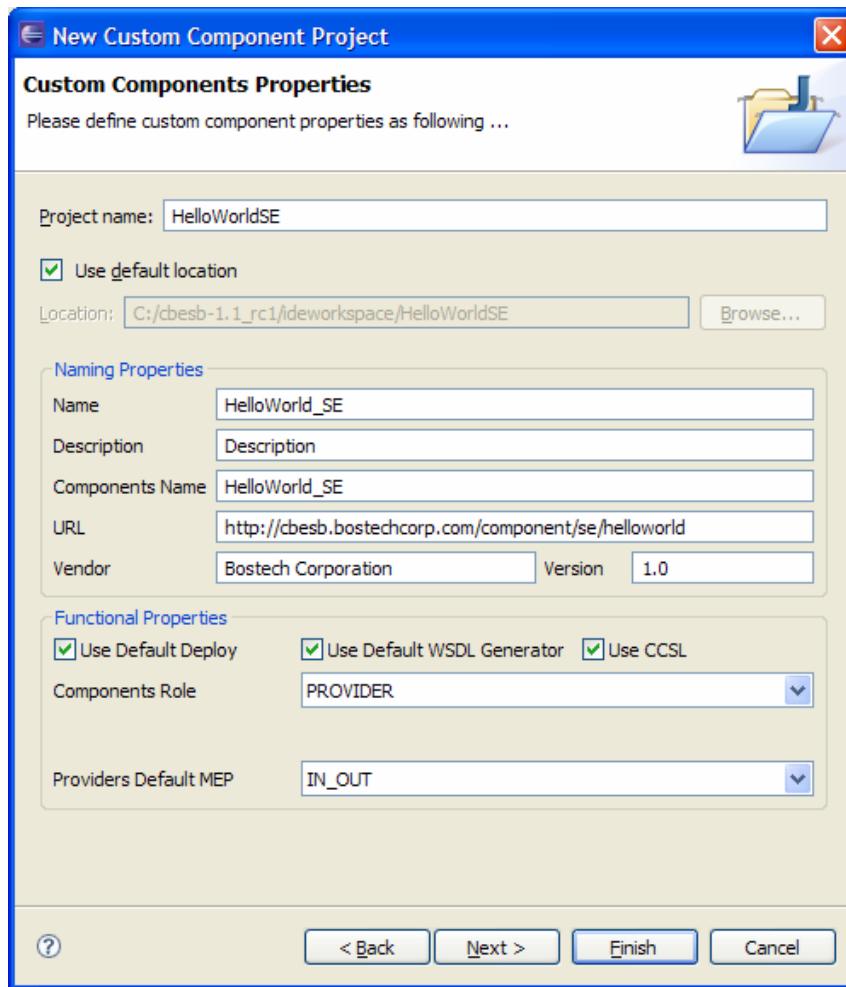
### 2.1. New Custom Component Wizard

To create a new component, right click in the Package Explorer view and select New → Project from the menu. In the wizard, select "New Custom Component Project" from list of project types.



#### 2.1.1. Custom Component Properties Page

The Custom Components Properties page contains the settings used to define the identification of the component as well as how it will function.



The information in the Naming Properties area is used to identify the component.

The information in the Functional Properties area is used to generate the appropriate code for the component.

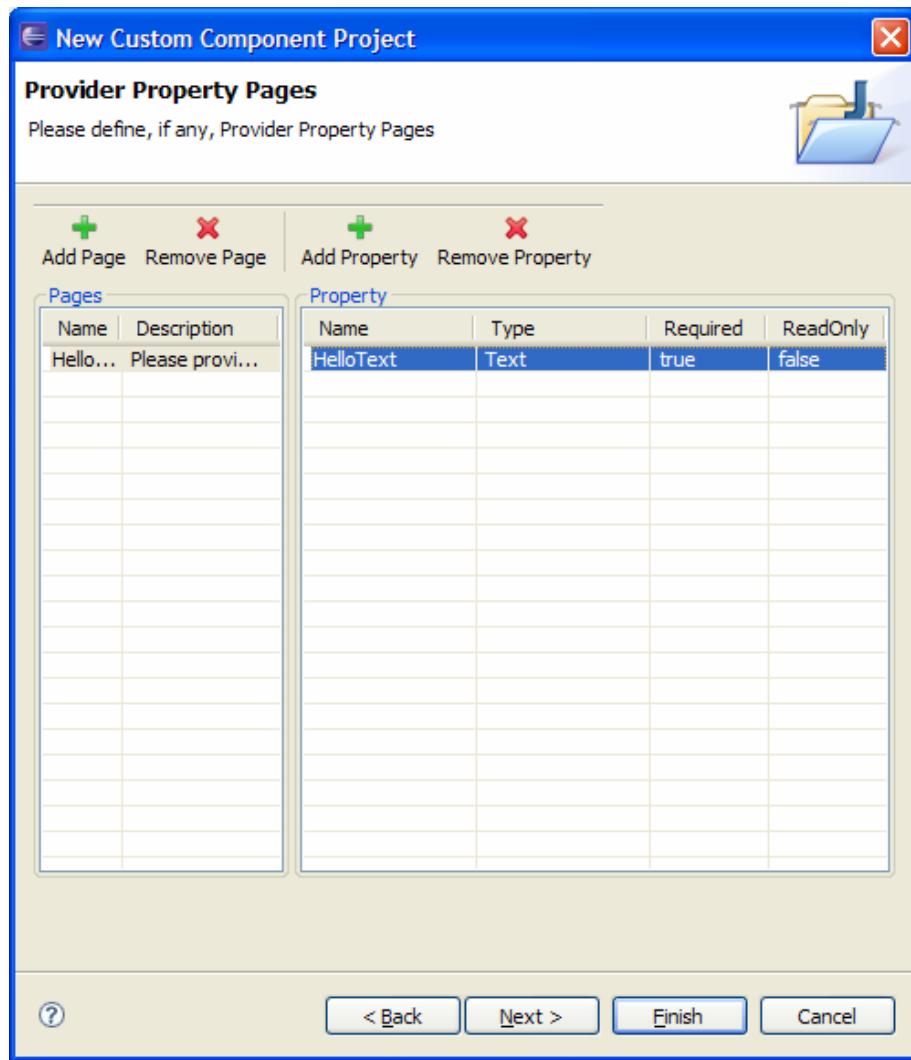
- Use Default Deploy - This specifies that the component will use the CBESB default deployer classes. If this is not used, then the developer must create their own method for deploying a Service Unit.
- Use Default WSDL Generator - This specifies that the Component Flow Custom Component plug-in will use the default WSDL generator to create the deployment WSDL. This consists of a "config" extensibility element that contains an attribute for each property defined for the component.
- Use CCSL - This determines whether the component will use the ChainBuilder Common Services Layer. By using checking this, Service Units deployed to the component will be able to take advantage of the CCSL functionality and the Component Flow wizard for the component will include panels to configure the CCSL settings.
- Components Role - The component may be configured as a Consumer, Provider or Both. If developing a Binding Component, then usually Both is an appropriate

setting. If developing a Service Engine, then Provider would be the appropriate setting.

- Consumer/Provider Default MEP - For each role, the developer should select the default Message Exchange Pattern used by the component.

### 2.1.2. Consumer/Provider Property Pages

For each Role that a component functions as, one or more property pages can be defined. These property pages are used in the Component Flow Editor when configuring the component.



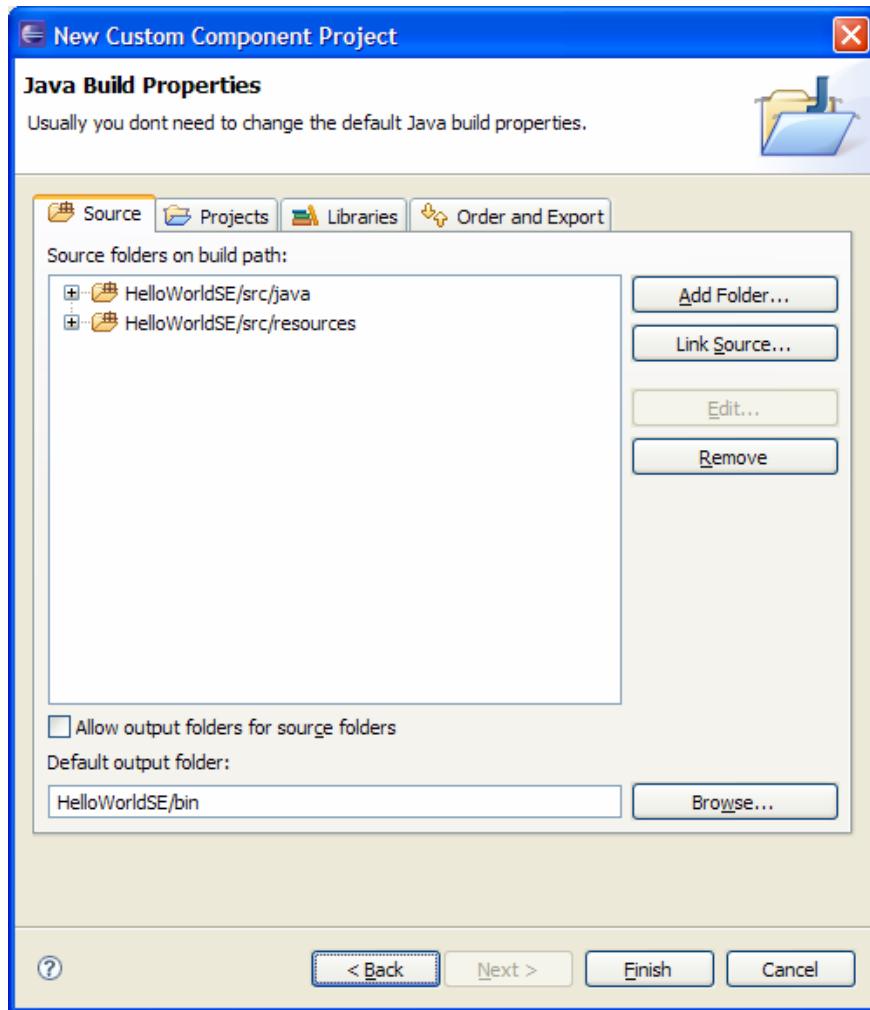
The Pages table on the left is used to identify the configuration pages that will be displayed in the Component Flow Editor. Most components will probably only need one page, but multiple may be defined. For each page added to the Pages table, a set of properties may be defined in the table on the right. These properties will be used to configure the component at runtime (through WSDL extensions) as well as define what is displayed on the wizard page in the Component Flow Editor.

**Property Settings:**

- Name - the name of the property must be a valid XML name, so it is best to stick to a combination of alphanumeric characters.
- Type - Defines the type of data for the property.
  - Text - String data
  - File - Allows for the selection of a file using a Browse button.
  - Enumeration - A fixed set of values.
  - Boolean - True/False.
  - Endpoint - Allows the selection of an endpoint defined within the SA project.
- Required - Determines whether the property is required or not.
- ReadOnly - Determines whether the property is read-only, or if the user may configure it.

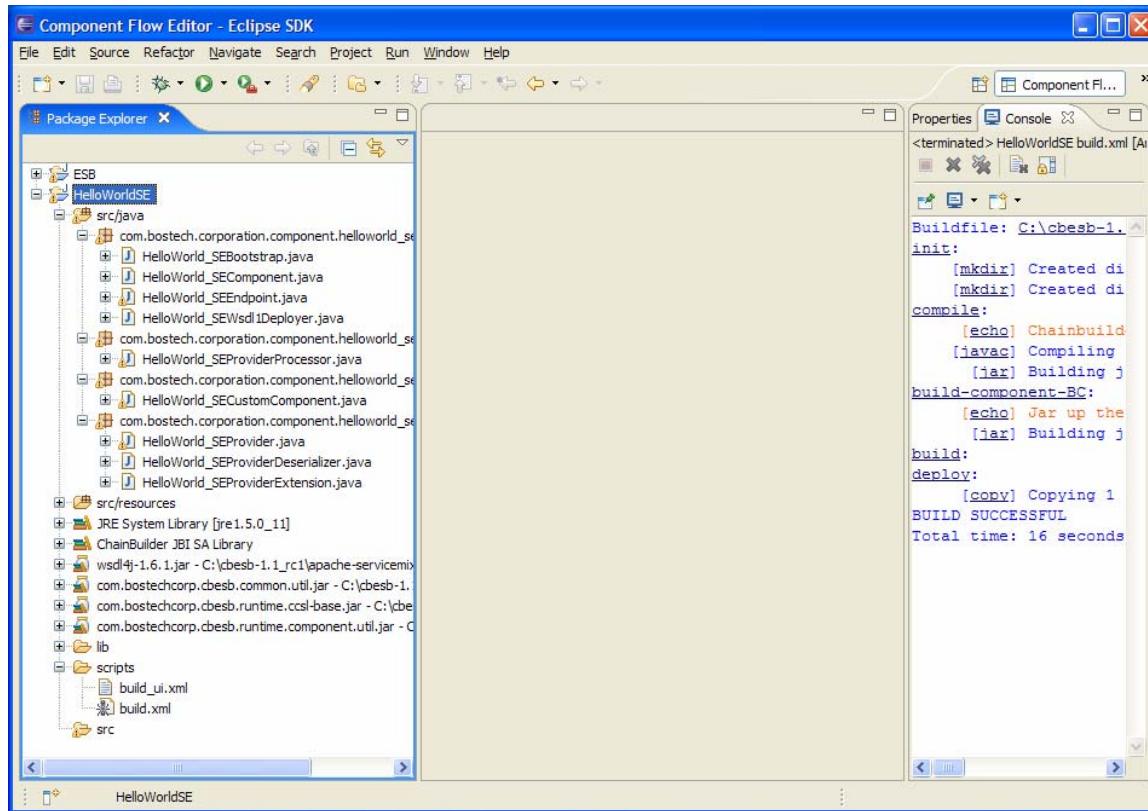
### 2.1.3. Java Build Properties

The Java Build Properties is the same as a normal Eclipse Java project. You may edit these settings to add dependent libraries/jar files or just click Finish. Any of these settings may be changed later if necessary.



## 2.2. Custom Component Project Layout

The generated project will contain four Java packages and two ant build scripts as shown in the example below.



### 2.2.1. Base Package

The base package for the component contains four classes. These classes (and their base classes) implement the interfaces defined by the JBI specification for a component.

**Bootstrap** - This class can perform initialization of the component if necessary. The generated class is empty, but methods from the base class may be over-ridden if the component needs some custom initialization.

**Component** - This is the main component class. This will rarely need to be modified.

**Endpoint** - The endpoint class contains the settings for a particular consumer or provider endpoint that has been deployed to the component. The generated class will contain all of the properties defined in the wizard as well as setter/getter methods.

**Wsdl1Deployer** - This class is responsible for configuring a new Endpoint from a WSDL document. This class will need to be modified to take the settings from the WSDL extension classes and set the appropriate properties in the Endpoint class.

## 2.2.2. Processors Sub-Package

The processors sub-package contains the consumer or provider (or both) processor class. This is where most of the component-specific logic will be added.

**ConsumerProcessor** - Handles the consumer role processing. This involves taking input from some external source and creating a new Message Exchange that is sent to the Normalized Message Router. If In-Out MEP is supported by the component, it will also process the response message when the Message Exchange is returned.

**ProviderProcessor** - Handles the provider role processing. This involves processing a Message Exchange that is received from the Normalized Message Router, doing component specific logic and optionally returning a reply.

## 2.2.3. UI Sub-Package

The ui package is used by the Component Flow Editor to display the configuration wizard for the component.

**CustomComponent** - This class implements the ICustComponent interface by extending the BaseCustComponent class. This class should be modified to set appropriate default values, set up custom Icons, etc.

## 2.2.4. WSDL Sub-Package

the wsdl package contains the WSDL extension classes needed to process the custom configurations settings in the deployment WSDL file. Each role defined for the component will have a set of classes to de-serialize the settings in the WSDL file. These settings can then be accessed by the Wsdl1Deployer to create a new Endpoint instance. In most cases these classes can be used without modification.

## 2.2.5. Ant Build Scripts

The Ant build scripts are located in the scripts directory in the project. To execute

build.xml - Compiles and packages the runtime component. When finished, the component jar file is placed in the components directory of the ChainBuilder ESB installation where it can be used at runtime.

build\_ui.xml - Compiles and packages the UI jar file. When finished, the custom component archive is placed in the customComp directory of the ChainBuilderESB installation where the Component Flow Editor will detect it and display it as a selection when Custom component is selected from the palette.

## 2.3. Example

The screen shots in the previous sections showed settings for a Service Engine named "HelloWorld\_SE". Continuing with this example, this section will show how to modify the generated class files to create a component that receives a message exchange and writes a message to the process log and if the exchange is In-Out, a message will be returned as the response.

### 2.3.1. HelloWorld\_SEWsdl1Deployer

The generated Wsdl1Deployer file needs some logic added to it to read the setting from the WSDL extension and set it in the Endpoint class.

First, the registerExtensions method should be modified to register the custom extension classes. The changes are shown highlighted below:

```
protected void registerExtensions(ExtensionRegistry registry) {
    super.registerExtensions(registry);
    //TODO: add code like this: FileInputExtension.register(registry);
    HelloWorld_SEProviderExtension.register(registry);
}
```

Second, the createEndpoint method should be modified to set the appropriate properties in the new endpoint instance.

```
protected CbEndpoint createEndpoint(ExtensibilityElement[] portElement,
                                    ExtensibilityElement[] bindingElement)
{
    logger.debug("createEndpoint portElement=" + portElement);

    HelloWorld_SEEndpoint endpoint = new HelloWorld_SEEndpoint();

    HelloWorld_SEProvider providerPortElement =
        (HelloWorld_SEProvider) portElement[0];

    endpoint.setHelloText(providerPortElement.getHelloText());
    // if JBI extension is used, its value (role, defaultMep,
    defaultOperation) will no longer be used

    //TODO: add property to endpoint
    //Code like:
    endpoint.setRole(Role.PROVIDER);
    endpoint.setDefaultMep(
        ((BaseCommonAttribute) portElement[0]).getDefaultMep()));

    return endpoint;
}
```

### 2.3.2. HelloWorld\_SEProviderProcessor

The generated ProviderProcessor needs to be modified to process a Message Exchange when it is received. This (and ConsumerProcessor when appropriate) is where most custom logic needs to be placed.

First, we will add a class variable to hold the endpoint and set it in the constructor. This will be used in the other methods to access endpoint properties. In our example, the property we will need to access is the text message that will be written to the log file and used as the reply message.

```
private HelloWorld_SEEndpoint endpoint;

public HelloWorld_SEProviderProcessor(HelloWorld_SEEndpoint endpoint) {
    super(endpoint);
    this.endpoint = endpoint;
}
```

Next, we will modify the processInMessage method. This method is called when an In-Only Message Exchange is received by the component. In our example, when an In-Only message exchange is received, we will just write a message in the process log.

```
public void processInMessage(QName service, QName operation,
    NormalizedMessage in, MessageExchange exchange)
    throws Exception {

    logger.info("Received In-Only Message Exchange.");
    logger.info(endpoint.getHelloText());
}
```

Finally, we will modify the processInOutMessage method. This is called when an In-Out Message Exchange is received by the component. In our example, when an In-Out message exchange is received, we will write a message to the log and set the contents of the out message to the message.

```
public boolean processInOutMessage(QName service, QName operation,
    NormalizedMessage in, NormalizedMessage out,
    boolean optionalOut, MessageExchange exchange)
    throws Exception {

    logger.info("Received In-Out Message Exchange.");
    logger.info(endpoint.getHelloText());

    //Set the hello text as the content of the out message
    NormalizedMessageHandler nmh = new NormalizedMessageHandler(out,
        getProviderSvcDescHandlerInstance());
    StringSource strSrc = new StringSource(endpoint.getHelloText());
    nmh.addRecord(strSrc);
    nmh.generateMessageContent();

    return true;
}
```

The NormalizedMessageHandler class is a convenience class that is used by all ChainBuilder ESB components. By using this class, it makes sure that the message contents are compatible with other CBESB components.

### 2.3.3. HelloWorld\_SECustomComponent

The CustomComponent class is used by the Component Flow Editor to display the component configuration wizard. By modifying this class, it is possible to control what is displayed in the wizard in the IDE.

In our example, we will set a default value for the "HelloText" property. This is done by changing a value in the constructor.

```
public HelloWorld_SECustomComponent() {

    super();
    //TODO: set big icon name,like "Echo32.ico", and make sure the icon
    has copied to src/resources

    bigIconResourceLocation="";

    //TODO: set samll icon ,like "Echo16.ico",make sure the icon has
    copied to src/resources

    smallIconResourceLocation="";

    componentURI=
    "http://cbesb.bostechcorp.com/component/se/helloworldhelloworld_se/1.0"
    ;
    name="HelloWorld_SE";
    componentName="HelloWorld_SE";

    description="Description";
    vendor="Bostech Corporation";
    useDefaultDeploy=true;
    useDefaultWSDLGenerator=true;
    useCCSL=true;
    version="1.0";
    providerDefaultMep=DefaultMEP.IN_OUT;
    consumerDefaultMep=DefaultMEP.IN_OUT;
    role=Role.PROVIDER;

    //BaseCustWizard providerWizard1=new
    BaseCustWizard("EchoWizard","this is the provider wiazrd");
    //providerWizard1.addProperty(new
    FileProperty("EchoFile",false,false,"f://Echo",null,true,null));
    //this.addWizardPage(providerWizard1, Role.PROVIDER);

    /* Please Modify all the default values of the constructor's
    parameters

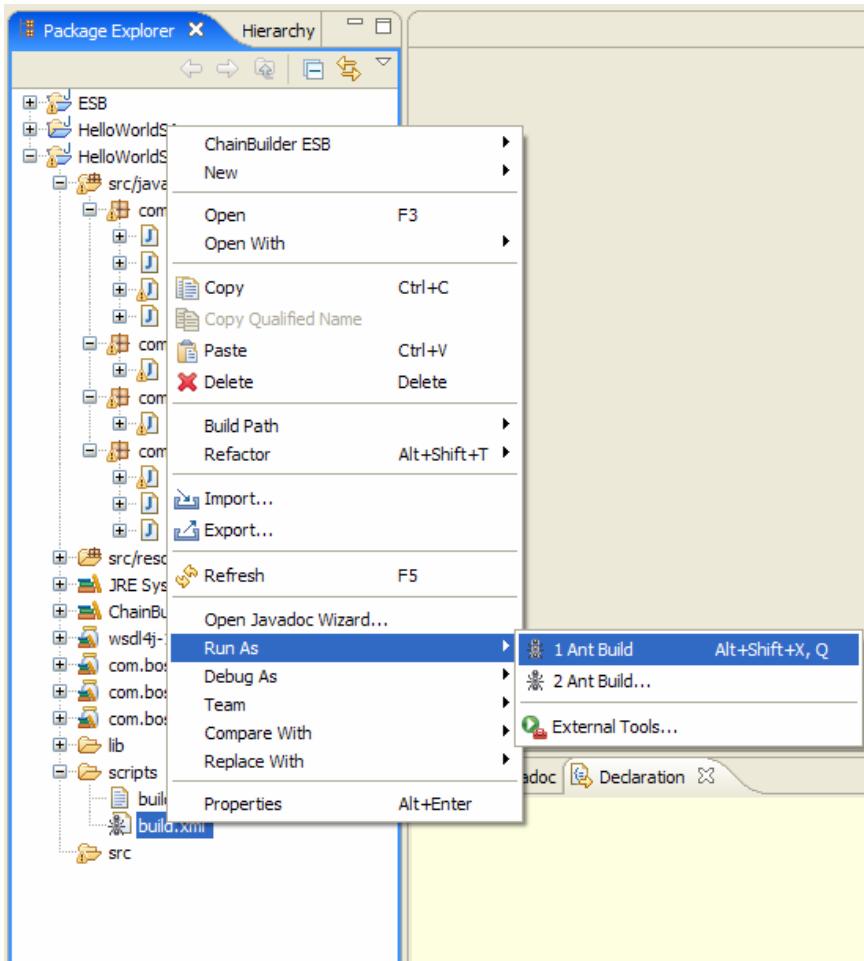
    public BooleanProperty(String name,
```

```
        boolean readOnly,
        boolean required,
        boolean defaultValue)
    public EnumProperty(String name,
        boolean readOnly,
        boolean required,
        String defaultValue,
        String[] values,
        boolean editable)
    public FileProperty(String name,
        boolean readOnly,
        boolean required,
        String baseFolderName,
        String fileName,
        boolean allowNewFile,
        FolderBrowseStyle folderBrowseStyle)
    public TextProperty(String name,
        boolean readOnly,
        boolean required,
        String defaultValue)
    EndPointProperty(String name,
        boolean readOnly,
        boolean required)
    */
    // ---Provider Pages
    BaseCustWizard providerWizard1= new BaseCustWizard("Hello World
    Providor", "Please provide the settings for the Hello World Provider
    endpoint.");
    providerWizard1.addProperty(new
    TextProperty("HelloText",false,true, "Hello!"));
    this.addWizardPage(providerWizard1, Role.PROVIDER);
}
```

### 2.3.4. Building the Component

After making the modifications, the project needs to be built. This is done by running the two ant scripts that were generated.

First, right-click on the build.xml file in the scripts directory and select Run As -> Ant Build.



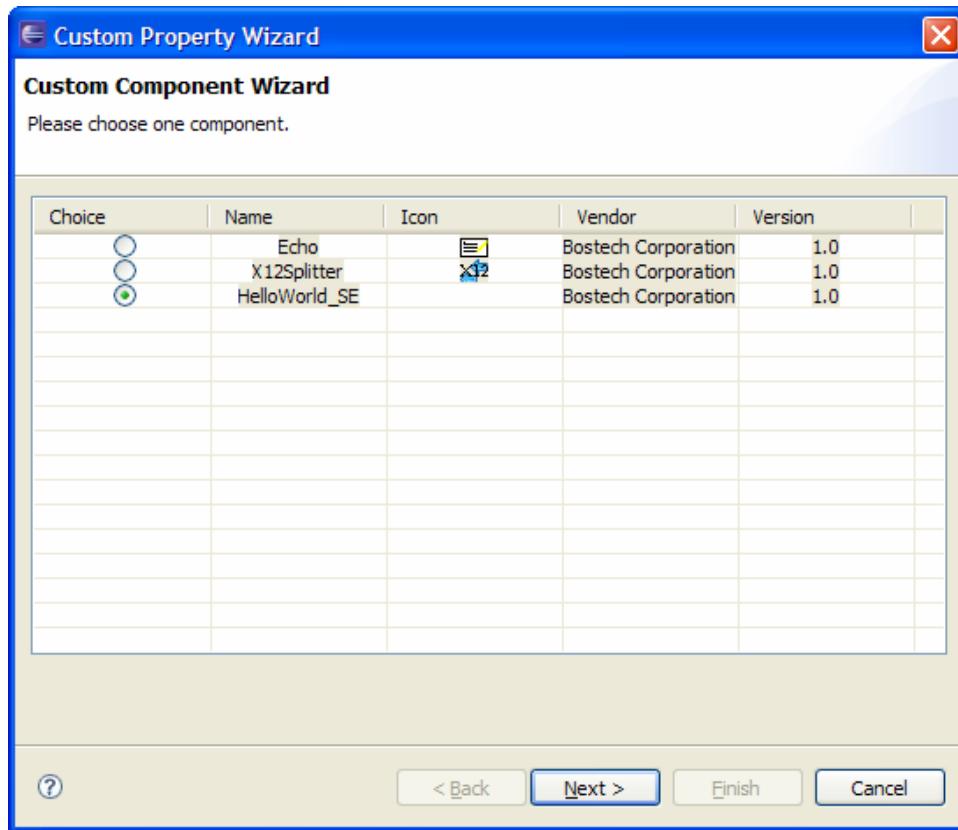
When this completes, do the same thing for the build\_ui.xml file.

The runtime component jar file will be placed in the %CBESB\_HOME%/components directory where it can then be used by the server. The UI custom component jar file will be placed in the %CBESB\_HOME%/customComp directory where the Component Flow Editor will be looking for it.

### 3. Using a Custom Component

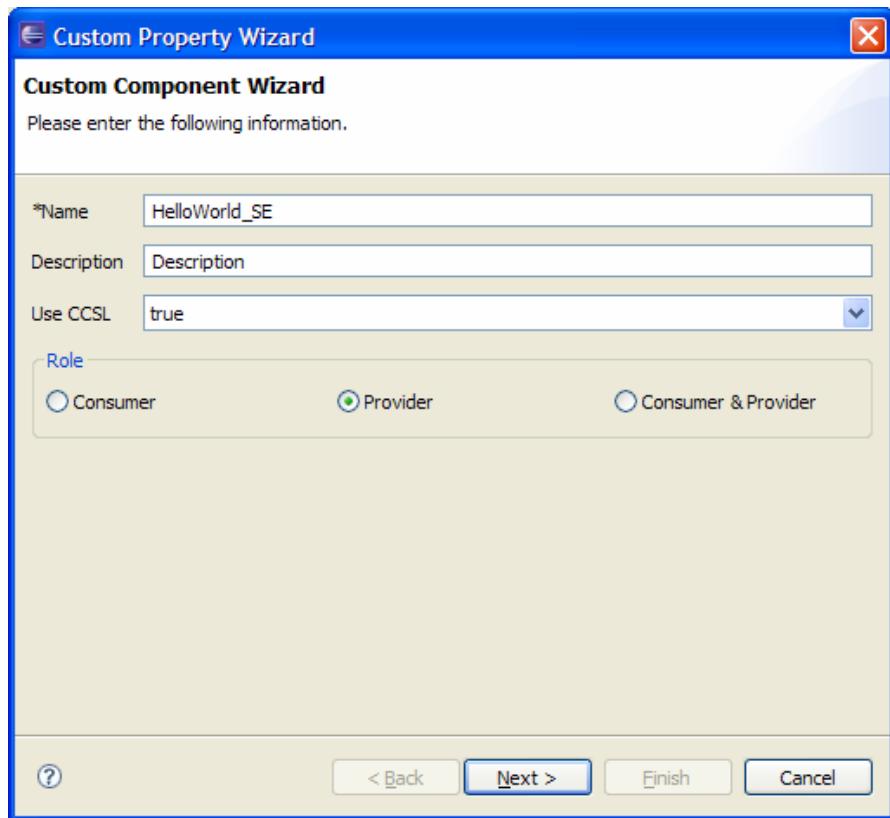
To configure a Service Assembly project to use a custom component, it is simply a matter of selecting the "Custom" component from the palette. This will start the wizard that allows you to select which custom component to use.

#### 3.1. Custom Component Wizard

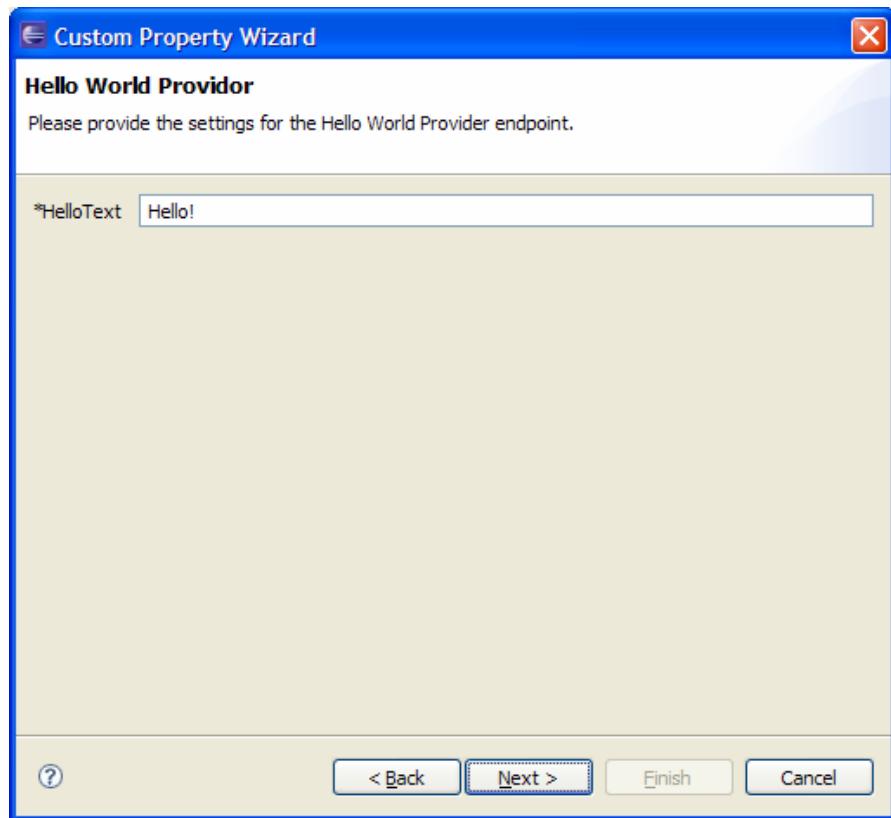


Select the radio button in the row containing the component you would like to use and click Next.

The rest of the panels in the wizard depend on the custom component's implementation. The first screen allows the selections for name and role similar to the CBESB built-in components.



Then each property page that was defined during the creation of the component is displayed.

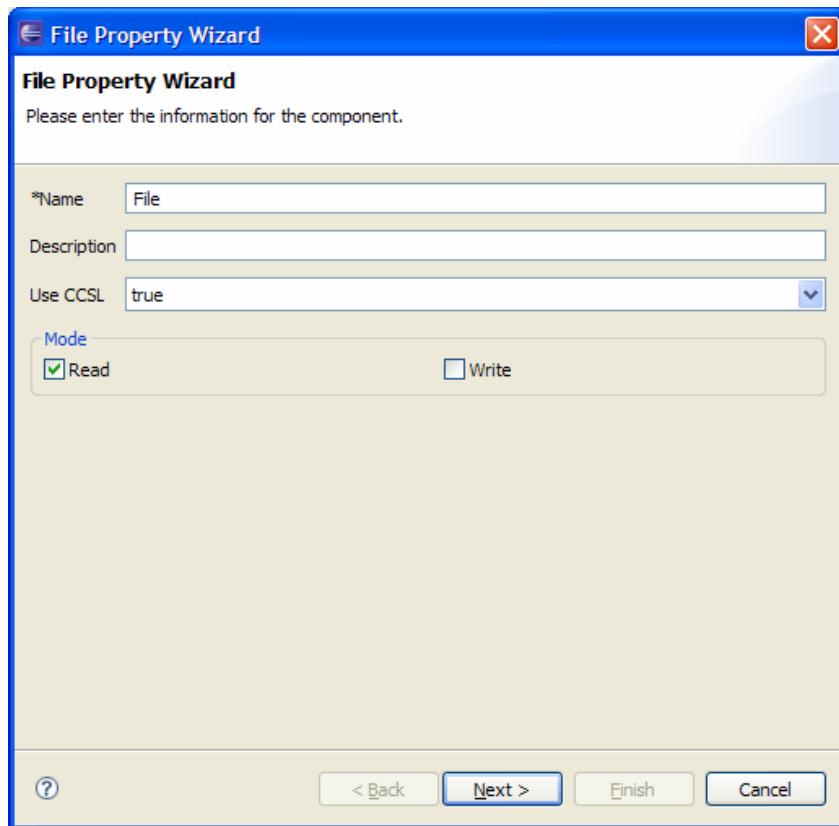


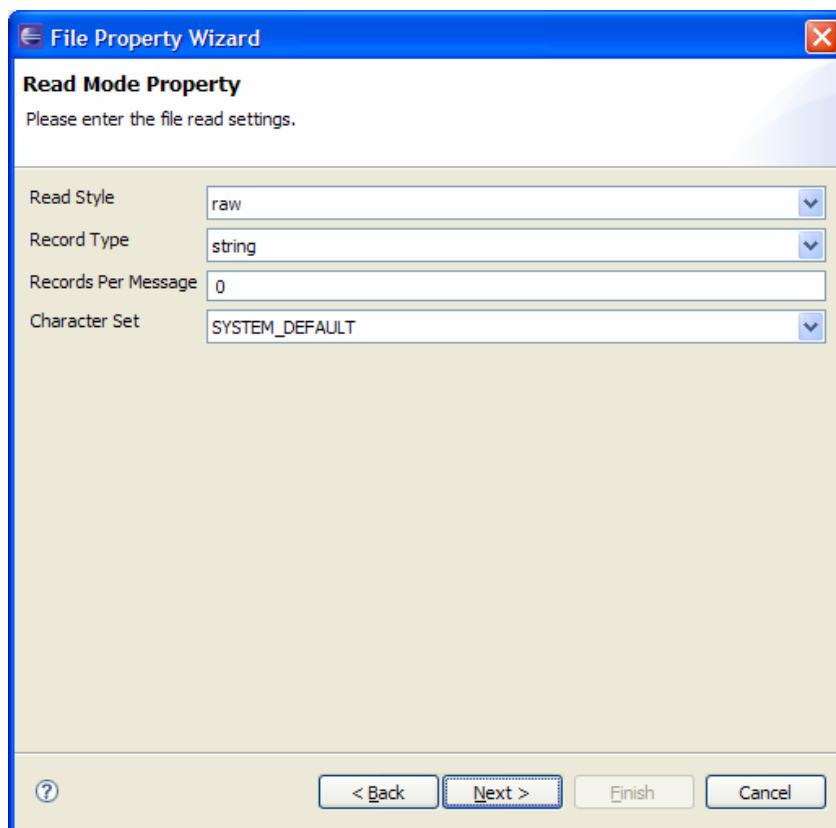
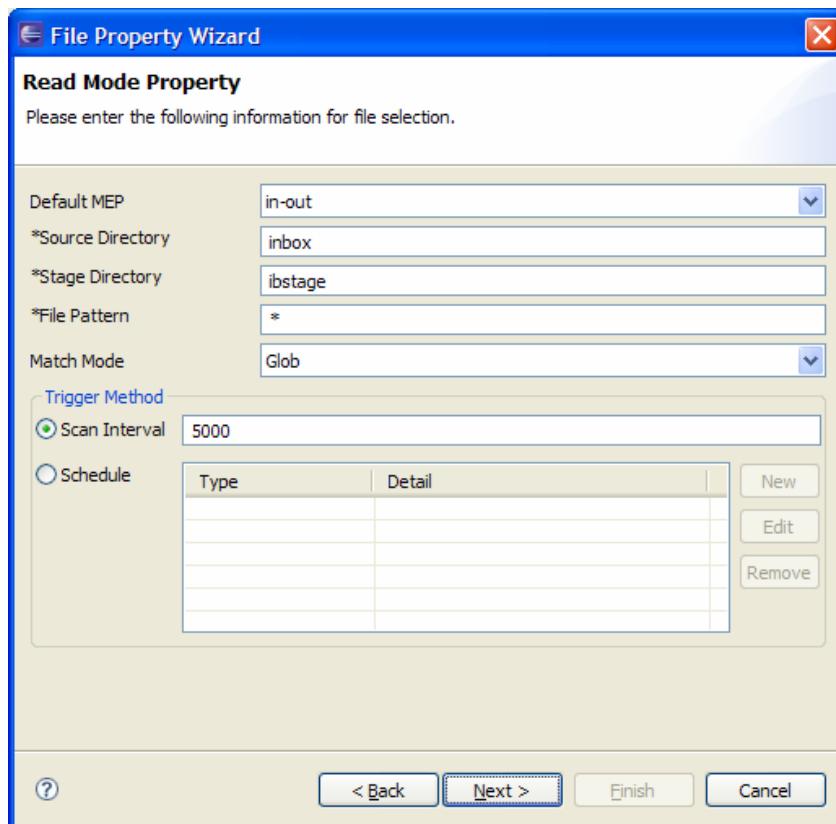
Our example component only has a single page with a setting for the message that will be written to the log or reply message.

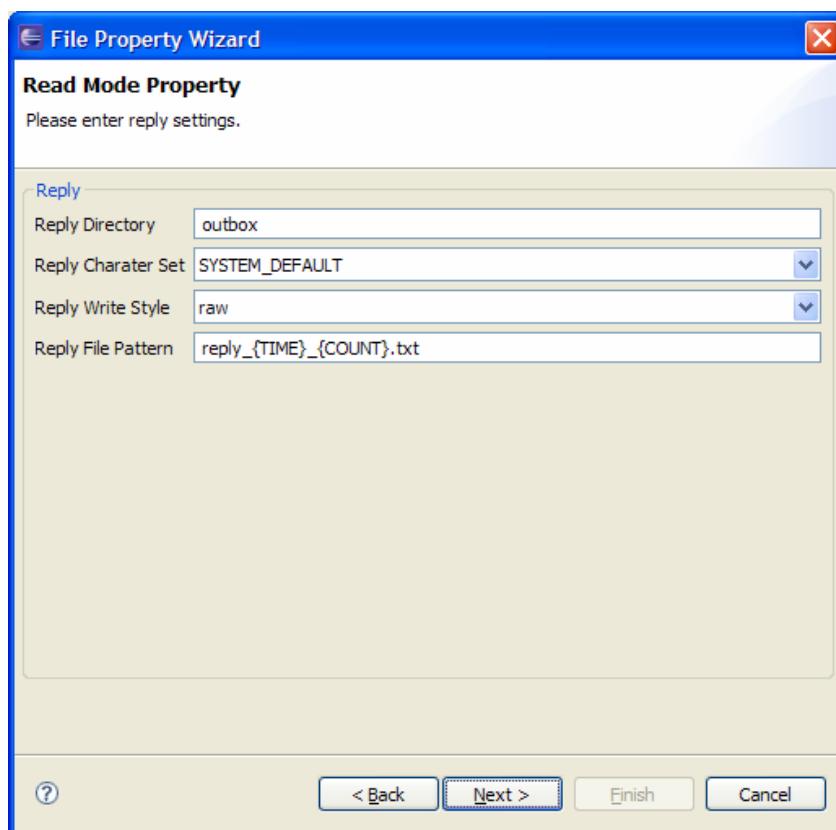
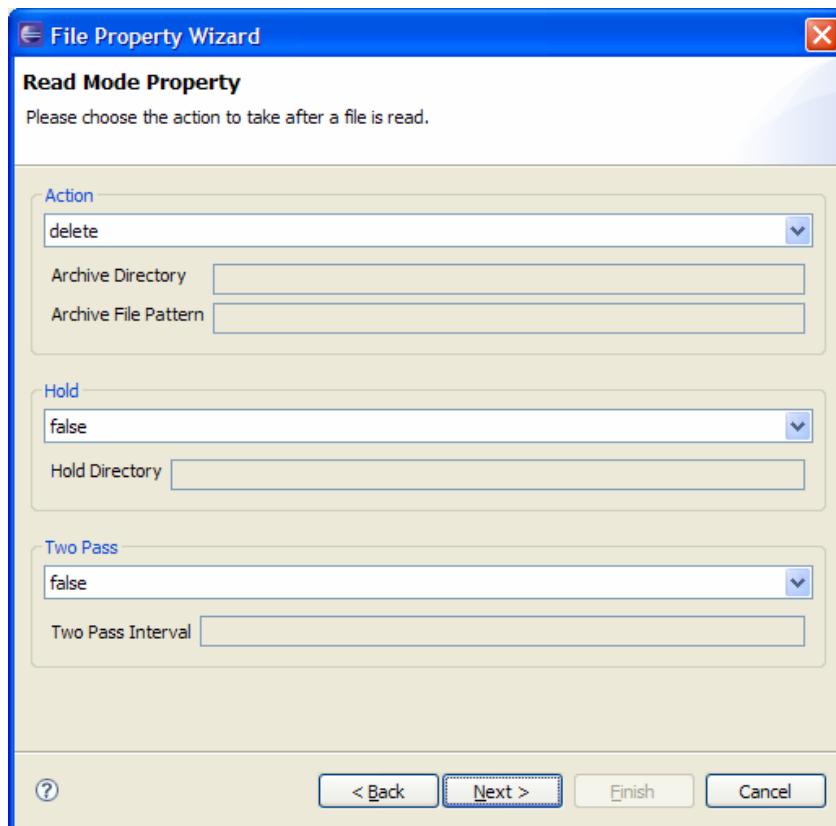
### 3.2. Example

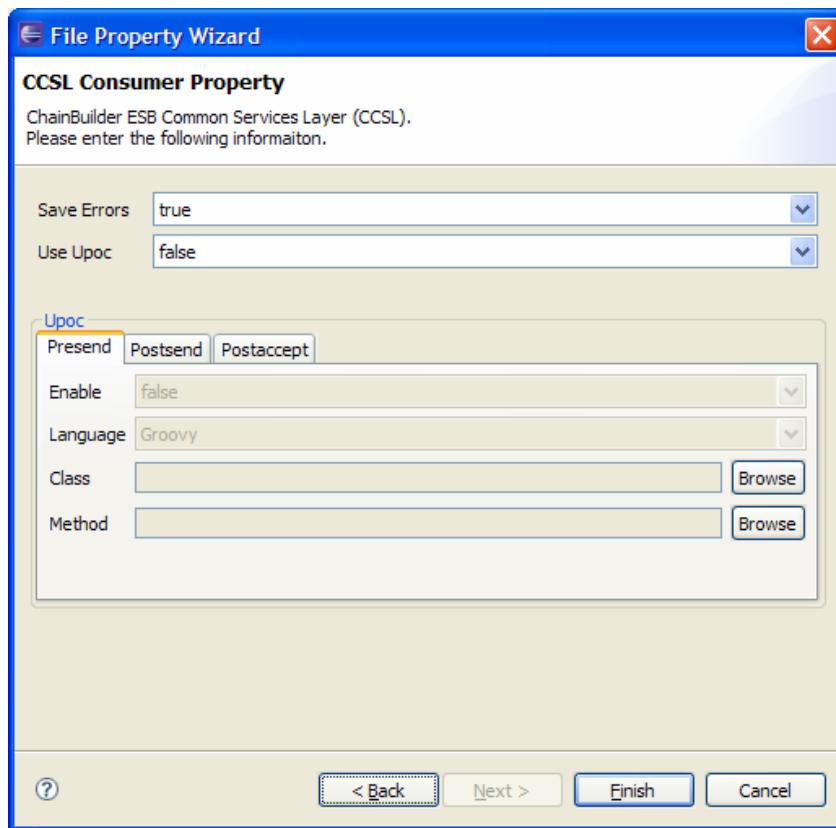
To test the HelloWorld\_SE example component, we will create a new Service Assembly project that will use a File component to read a file to generate a message exchange. The contents of the message isn't important. The message exchange will be sent to the HelloWorld\_SE component.

First add a File component to the component flow diagram. The file component should be configured as follows:

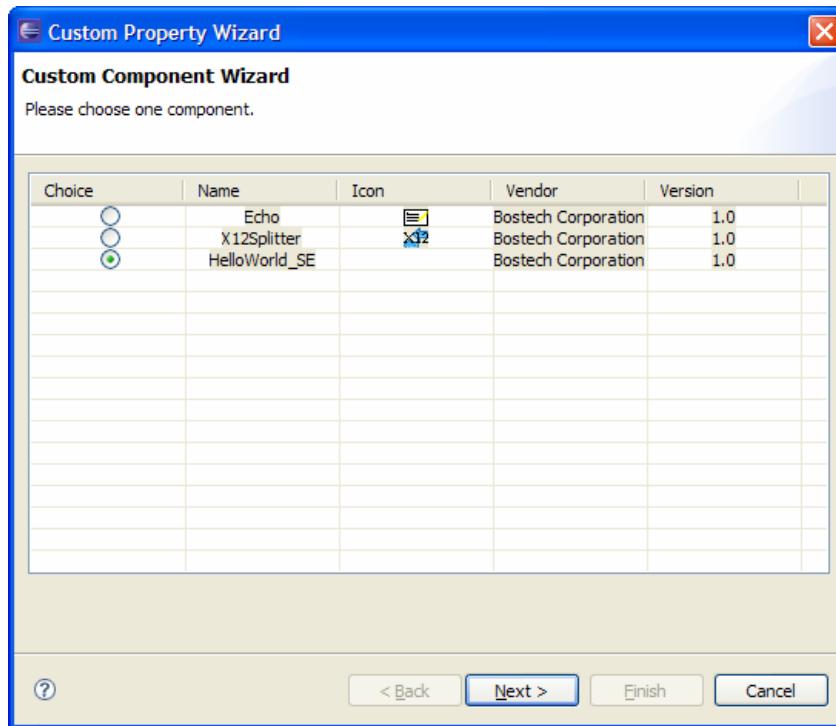


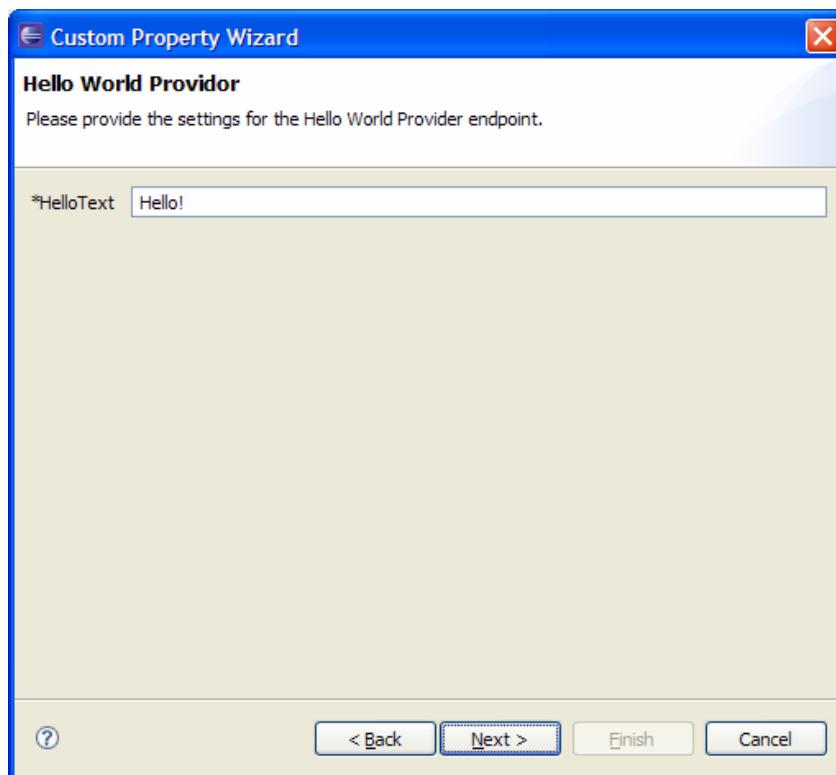
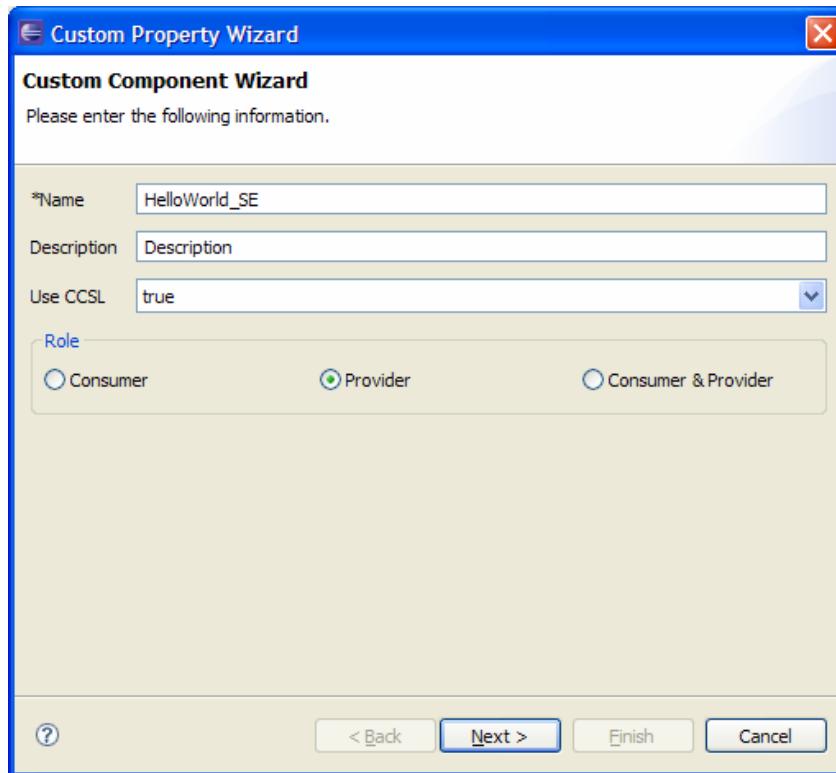




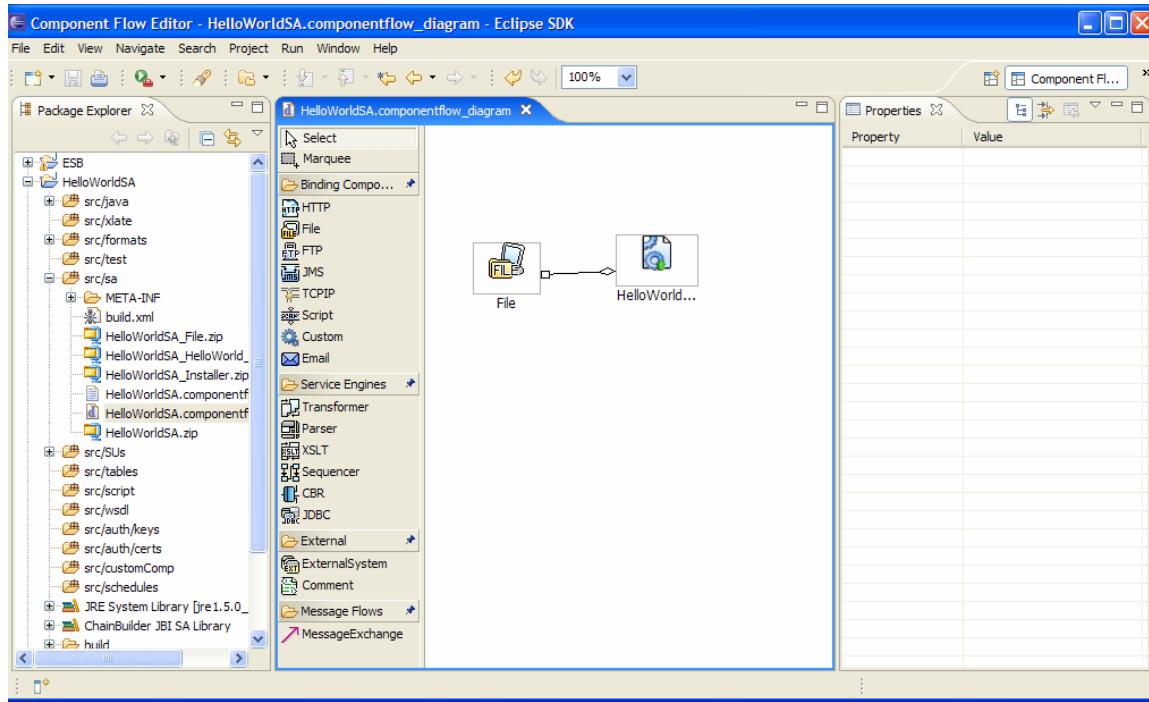


Then, add the HelloWorld\_SE component to the flow, by selecting the Custom component from the palette and selecting the HelloWorld\_SE component.





Then draw a connection from the File Component to the HelloWorld component.



Save the file and do a Deploy. The project can then be executed with the cbesb\_run <project name> command.

The following shows the results that are written to the log when a file is placed in the "inbox" folder.

```
INFO - HelloWorld_SEProviderProcessor - Received In-Out Message Exchange.  
INFO - HelloWorld_SEProviderProcessor - Hello!
```

A file is also created in the "outbox" folder containing the message "Hello!" as well.

## 4. ChainBuilder ESB Community

ChainForge.net is the internet's premier destination to share ChainBuilder and JBI knowledge with your peers.

Join the ChainBuilder ESB Community:

<http://www.chainforge.net/community>

As a member you can view content and contribute to a Forum:

<http://www.chainforge.net/community/forums.html>

Read ChainBuilder ESB related Blogs:

<http://www.chainforge.net/blogs>