

Sun POJO Service Engine User's Guide



Part No: 821-0873-10
February 2010

Copyright ©2010 Sun Microsystems, Inc. 4150 Network Circle, Santa Clara, CA 95054 U.S.A. All rights reserved.

Sun Microsystems, Inc. has intellectual property rights relating to technology embodied in the product that is described in this document. In particular, and without limitation, these intellectual property rights may include one or more U.S. patents or pending patent applications in the U.S. and in other countries.

U.S. Government Rights – Commercial software. Government users are subject to the Sun Microsystems, Inc. standard license agreement and applicable provisions of the FAR and its supplements.

This distribution may include materials developed by third parties.

Parts of the product may be derived from Berkeley BSD systems, licensed from the University of California. UNIX is a registered trademark in the U.S. and other countries, exclusively licensed through X/Open Company, Ltd.

Sun, Sun Microsystems, the Sun logo, the Solaris logo, the Java Coffee Cup logo, docs.sun.com, Java, and Solaris are trademarks or registered trademarks of Sun Microsystems, Inc. or its subsidiaries in the U.S. and other countries. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. in the U.S. and other countries. Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc.

The OPEN LOOK and Sun™ Graphical User Interface was developed by Sun Microsystems, Inc. for its users and licensees. Sun acknowledges the pioneering efforts of Xerox in researching and developing the concept of visual or graphical user interfaces for the computer industry. Sun holds a non-exclusive license from Xerox to the Xerox Graphical User Interface, which license also covers Sun's licensees who implement OPEN LOOK GUIs and otherwise comply with Sun's written license agreements.

Products covered by and information contained in this publication are controlled by U.S. Export Control laws and may be subject to the export or import laws in other countries. Nuclear, missile, chemical or biological weapons or nuclear maritime end uses or end users, whether direct or indirect, are strictly prohibited. Export or reexport to countries subject to U.S. embargo or to entities identified on U.S. export exclusion lists, including, but not limited to, the denied persons and specially designated nationals lists is strictly prohibited.

DOCUMENTATION IS PROVIDED "AS IS" AND ALL EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS AND WARRANTIES, INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT, ARE DISCLAIMED, EXCEPT TO THE EXTENT THAT SUCH DISCLAIMERS ARE HELD TO BE LEGALLY INVALID.

Contents

Using the Sun POJO Service Engine	5
POJO Service Engine Overview	6
Providing and Consuming JBI Services	6
Providing a JBI Service	7
Consuming a JBI Service	8
Consuming Services Asynchronously	10
Using Multiple Threads	11
Creating POJO Service Engine Projects	11
Creating a POJO Service Provider (Without Binding)	11
Creating a POJO Service Provider (With Binding)	15
Creating a POJO Service Provider in an Existing Java Application	21
Creating a POJO Service Consumer (Without Binding)	23
Creating a POJO Service Consumer (With Binding)	25
POJO Configuration Properties	27
Creating Composite Applications for POJO Service Engine Projects	30
Creating a Composite Application for a POJO Service Provider (Without Binding)	31
Creating a Composite Application for a POJO Service Provider (With Binding)	33
Creating a Composite Application for a POJO Service Consumer (Without Binding)	36
Creating a Composite Application for a POJO Service Consumer (With Binding)	39
Using POJO Services With BPEL	42
Invoking POJO Services from a Business Process	43
Calling a Business Process From a POJO Service	48
Configuring Runtime Properties for the POJO Service Engine	51
▼ To Configure POJO SE Runtime Properties	52
POJO Service Engine Runtime Property Descriptions	53
POJO Service Engine API Annotation and Classes	55
POJO Service Engine API Annotations	56
POJO Service Engine Non-Annotated Classes	57

JBI API Classes Relevant to the POJO Service Engine 57

Using the Sun POJO Service Engine

The POJO Service Engine provides simple and easy to use tools that allow you to incorporate Plain Old Java Objects (POJO) into the JBI applications you create.

What You Need to Know

The following topics provide information to help you understand the POJO Service Engine:

- [“POJO Service Engine Overview” on page 6](#)
- [“Providing and Consuming JBI Services” on page 6](#)
- [“POJO Service Engine API Annotation and Classes” on page 55](#)

What You Need to Do

The following topics provide instructions for creating service providers and consumers using the POJO Service Engine:

- [“Creating a POJO Service Provider \(Without Binding\)” on page 11](#)
- [“Creating a POJO Service Provider \(With Binding\)” on page 15](#)
- [“Creating a POJO Service Provider in an Existing Java Application” on page 21](#)
- [“Creating a POJO Service Consumer \(Without Binding\)” on page 23](#)
- [“Creating a POJO Service Consumer \(With Binding\)” on page 25](#)
- [“Creating a Composite Application for a POJO Service Provider \(Without Binding\)” on page 31](#)
- [“Creating a Composite Application for a POJO Service Provider \(With Binding\)” on page 33](#)
- [“Creating a Composite Application for a POJO Service Consumer \(Without Binding\)” on page 36](#)
- [“Creating a Composite Application for a POJO Service Consumer \(With Binding\)” on page 39](#)
- [“Invoking POJO Services from a Business Process” on page 43](#)
- [“Calling a Business Process From a POJO Service” on page 48](#)

- [“Configuring Runtime Properties for the POJO Service Engine” on page 51](#)

Reference Information

The following topics provide information about the POJO properties you need to configure:

- [“POJO Service Provider Properties” on page 27](#)
- [“POJO Service Properties for Binding” on page 28](#)
- [“POJO Service Consumer Properties” on page 29](#)
- [“POJO Service Engine Runtime Property Descriptions” on page 53](#)
- [POJO Service Engine Javadoc](#)

POJO Service Engine Overview

The POJO Service Engine allows you to build business integration applications based on JBI standards and using Plain Old Java Objects (POJO). The POJO Service Engine automates much of the annotation and generates the code framework in which you can define your applications. The service engine simplifies the process by defining very few annotation and API classes. It provides flexibility by use of method signatures and by handling synchronous and asynchronous messages in a message-oriented way.

The POJO Service Engine supports a message-oriented paradigm rather than service-oriented. WSDL documents can be used but are not required. The service engine allows you to define both service providers and consumers with or without bindings (WSDL documents). You can also call the POJO providers you create from a BPEL process, and you can call a BPEL process from a POJO service consumer.

Unlike the Java EE Service Engine, the POJO Service Engine does not require a web or EJB container. The message data structure does not need to be exposed in a service description language such as WSDL, although a WSDL document can be used if that is preferred. The POJO Service Engine can access JBI normalized message objects and message exchange objects directly, which supports RESTful services, provides options for streaming and handling non-XML data, and avoids unnecessary unmarshaling of the incoming messages to Java objects.

Providing and Consuming JBI Services

The following topics provide information to help you understand how services are provided and consumed using the POJO Service Engine:

- [“Providing a JBI Service” on page 7](#)
- [“Consuming a JBI Service” on page 8](#)
- [“Consuming Services Asynchronously” on page 10](#)
- [“Using Multiple Threads” on page 11](#)

Providing a JBI Service

With just two annotations, `@Provider` and `@Operation`, you can enable a Java class as JBI service. Below is an example of a simple POJO service provider.

```
package org.glassfish.openesb.pojo.sample;
import org.glassfish.openesb.pojose.api.annotation.Operation;
import org.glassfish.openesb.pojose.api.annotation.Provider;
@Provider
public class Echo {
    @Operation(outMessageType="EchoOperationResponse",
               outMessageTypeNS="http://sample.pojo.openesb.glassfish.org/Echo/")
    public String receive(String input) {
        return input;
    }
}
```

The properties for `@Operation` annotation are not mandatory, but may be needed if the JBI component interacting with the POJO SE expects the JBI message type QName on the JBI wrapper message element. Defaults are assumed for most of the information needed by JBI runtime. Some of the defaults are endpoint names and interface and service QNames. If not specified through `@POJO` annotation, the endpoint name defaults to the unqualified class name. The service and interface QName namespaces default to the HTTP URI made up of the package name in reverse with the endpoint name appended. For the above example, this would be `http://tst.pojo.glassfishesb.org/Echo/`. The local service name defaults to the endpoint name with "Service" appended. The local interface name defaults to the endpoint name with "Interface" appended. For the above example it will be "EchoService" and "EchoInterface" respectively.

Operation Parameter Types

The POJO SE supports the following input parameter types and return types for methods annotated with `@Operation`:

- `java.lang.String`
- `org.w3c.dom.Node`
- `javax.xml.transform.Source`
- `javax.jbi.messaging.NormalizedMessage`
- `Document`
- `javax.jbi.messaging.MessageExchange` (input parameter only)
- `void` (return type only)

When the return type is void, the JBI message exchange pattern is assumed to be InOnly. The POJO SE supports InOnly and InOut JBI message exchange patterns. For parameter and return types of String, Node, and Source, the POJO SE automates JBI WSDL 1.1 message unwrapping and wrapping.

Context

The POJO SE injects an instance of the class `org.glassfish.openesb.pojose.api.res.Context` when a field is annotated with `@Resource` (`org.glassfish.openesb.pojose.api.annotation.Resource`). The Context class includes the methods needed to retrieve an instance of the current `MessageExchange` object and a method to create a new `MessageExchange` objects for invoking JBI services.

Consuming a JBI Service

Using the Consumer instance inserted by the service engine, you can declare a field of the Consumer type (`org.glassfish.openesb.pojose.api.Consumer`) and annotate it with `ConsumerEndpoint` (`org.glassfish.openesb.pojose.api.annotation.ConsumerEndpoint`). The POJO SE uses the endpoint name and service QName specified on the `ConsumerEndpoint` annotation to find the `ServiceEndpoint` instance and insert it into the POJO instance before the operation method is called. You can optionally specify `inMessageTypeQN` and `operationQN` if required by the component being called.

EXAMPLE 1 POJO Service Consumer Example

First declare the Consumer field as shown below.

```
@ConsumerEndpoint(name="asiaBPELProcess",
    serviceQN="AsiaSvc",
    interfaceQN="{wwOrderProcessNS}wwOrderProcessPortType",
    operationQN="{wwOrderProcessNS}wwOrderProcessOperation",
    inMessageTypeQN="{wwOrderProcessNS}wwOrderProcessOperationRequest")
private Consumer asiaEp;
```

Use the Consumer instance inserted by the POJO SE, as shown below.

```
outputMsg = (Node) cons.sendSynchInOut(input, MessageObjectType.Node);
```

Below is a complete example.

```
package org.glassfish.openesb.pojo.cbr;
import org.glassfish.openesb.pojose.api.annotation.Provider;
import org.glassfish.openesb.pojose.api.annotation.Operation;
import org.glassfish.openesb.pojose.api.annotation.ConsumerEndpoint;
import org.glassfish.openesb.pojose.api.Consumer;
import org.glassfish.openesb.pojose.api.Consumer.MessageObjectType;

import org.w3c.dom.Node;
import java.util.logging.Level;
import java.util.logging.Logger;
```


EXAMPLE 1 POJO Service Consumer Example *(Continued)*

```

@Provider
public class WWOrderRouter {
    @ConsumerEndpoint(name="asiaBPELProcess",
        serviceQN="AsiaSvc",
        interfaceQN="{wwOrderProcessNS}wwOrderProcessPortType",
        operationQN="{wwOrderProcessNS}wwOrderProcessOperation",
        inMessageTypeQN="{wwOrderProcessNS}wwOrderProcessOperationRequest")
    private Consumer asiaEp;

    @ConsumerEndpoint(name="europeBPELProcess",
        serviceQN="EuropeSvc",
        interfaceQN="{wwOrderProcessNS}wwOrderProcessPortType",
        operationQN="{wwOrderProcessNS}wwOrderProcessOperation",
        inMessageTypeQN="{wwOrderProcessNS}wwOrderProcessOperationRequest")

@Resource
private Context ctx;

public WWOrderRouter() {
}

@Operation(outMessageTypeQN="{http://cbr.pojo.openesb.glassfish.org/WWOrderRouter/}
    WWOrderRouterOperationResponse")
public Node receive(Node input) {
    try {
        Node outputMsg = input;

        location = ...

        Consumer cons = null;
        if ("Asia".equals(location)) {
            svc2use = this.asiaSvcName;
            cons = asiaEp;
        } else {
            svc2use = this.europeSvcName;
            cons = europeEp;
        }
        outputMsg = (Node) cons.sendSynchInOut(input, MessageObjectType.Node);

        return outputMsg;
    } catch (Exception ex) {
        Logger.getLogger(WWOrderRouter.class.getName()).log(Level.SEVERE, null,
            ex);
    }
    return input;
}

```

EXAMPLE 1 POJO Service Consumer Example *(Continued)*

```
    }  
}
```

Getting the Consumer Instance Dynamically

Use a Context method to retrieve the instance of ServiceEndpoint, and use it again to retrieve the instance of Consumer from the Context instance.

```
QName svc2use = ....;  
String endpointName = ....;  
  
ServiceEndpoint se = this.ctx.getEndpoint(svc2use, endpointName);  
Consumer cons = this.ctx.getConsumer(se, this.consOpName, this.consInMsgType);  
outputMsg = (Node) cons.sendSynchInOut(input, MessageObjectType.Node);
```

Consuming Services Asynchronously

Consuming services in asynchronous mode can make using resources such as threads more efficient. Consuming services asynchronously in the POJO SE does not block the threads; instead, the control returns to the POJO code. This allows the POJO SE to execute more POJO services using fewer thread resources. Asynchronous service consumption is supported using annotated callback methods. Each of the callback methods are annotated in POJO using one of the following annotations: `@OnReply`, `@OnError`, `@OnFault`, or `@OnDone`.

The POJO SE calls the `OnReply` annotated method when the InOut message exchange pattern service is consumed asynchronously. This method takes two parameters. The first is of the type `ServiceEndpoint` and the second is one of the following types: `String`, `Source`, `Node`, `NormalizedMessage`, or `MessageExchange`.

The POJO SE calls the `OnFault` annotated method when the InOut message exchange pattern service is consumed asynchronously and the consumed service returns a fault message. This method also takes two parameters. The first is of the type `ServiceEndpoint` and second is of the type `MessageExchange`.

The POJO SE calls the `OnError` annotated method when the InOut message exchange pattern service is consumed asynchronously and the consumed service returns an error status. This method takes two parameters. The first one is of the type `ServiceEndpoint` and the second is of the type `MessageExchange`.

The POJO SE executes the `OnDone` annotated method when all the responses from asynchronously consumed services are received. Whenever POJO throws `FaultMessage`, `ErrorMessage`, or `Exception` back to the POJO SE, the POJO SE returns the fault message or error status back to the POJO service consumer. Further execution of callback and `OnDone`

methods are aborted and outstanding responses from asynchronously consumed services by this POJO instance are ignored. Where possible, the error status is returned.

Using Multiple Threads

The annotations described above for asynchronous consumption are also used in the multi-threaded execution model. In this model, the POJO SE executes the POJO instance methods annotated with `Operation`, `OnReply`, `OnFault`, and `OnError` concurrently when the response messages are received. Since many transaction managers only allow one thread to be associated with a transaction, by default the transaction is not resumed while executing any of the POJO instance methods such as those annotated with `Operation`, `OnReply`, `OnFault`, `OnError`, and `OnDone`. Transaction objects are also not propagated to asynchronously consumed services.

Creating POJO Service Engine Projects

The POJO Service Engine offers a variety of methods to create service providers and consumers using the NetBeans IDE. You can create providers and consumers with or without binding definitions (WSDL documents), and you can create providers from a wizard or from the Java Editor Palette. Consumers can also be created from the Palette.

Perform any of the following steps to create POJO service providers and consumers:

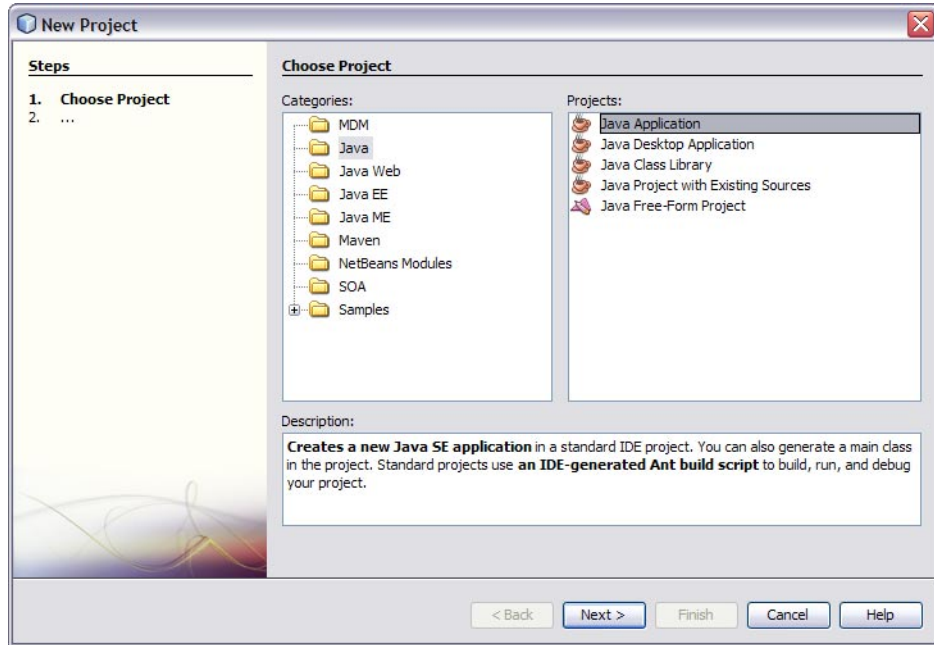
- [“Creating a POJO Service Provider \(Without Binding\)” on page 11](#)
- [“Creating a POJO Service Provider \(With Binding\)” on page 15](#)
- [“Creating a POJO Service Provider in an Existing Java Application” on page 21](#)
- [“Creating a POJO Service Consumer \(Without Binding\)” on page 23](#)
- [“Creating a POJO Service Consumer \(With Binding\)” on page 25](#)

Creating a POJO Service Provider (Without Binding)

This procedure creates a framework in a Java file for a POJO service provider using a wizard that guides you through the steps. The framework includes annotations and standard methods, and you can customize this framework to define the logic of the service. There is no WSDL document or binding involved in this procedure; a binding can be added when you create the composite application.

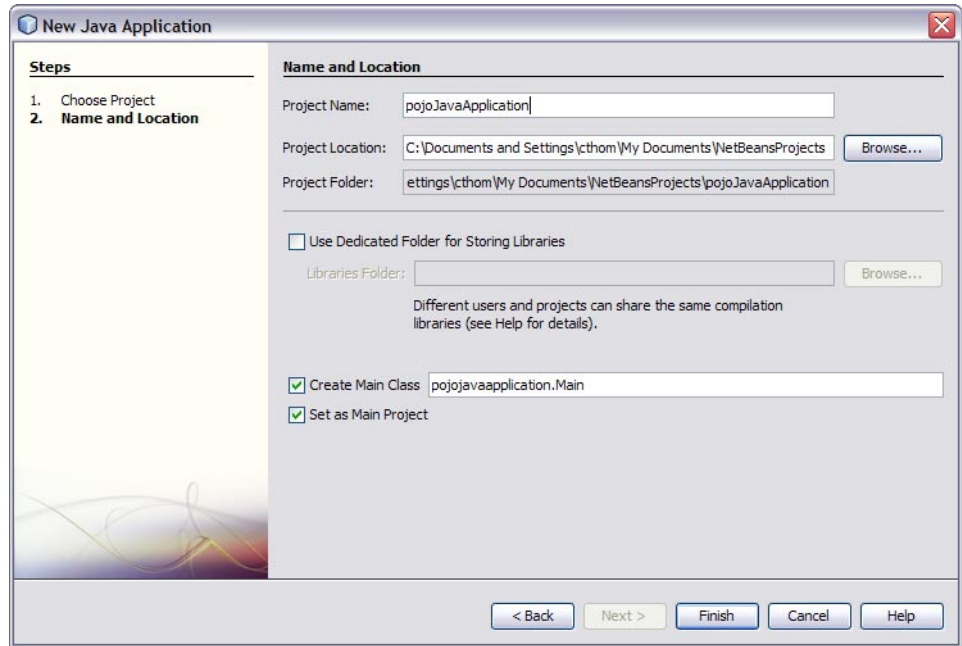
▼ To Create a POJO Service Provider (Without Binding)

- 1 Right-click in the NetBeans Projects window, and then select New Project.
The New Project Wizard appears.
- 2 Under Categories, select Java; under Projects, select Java Application.



- 3 Click Next.
The Name and Location window appears.

- 4 Enter a unique name for the project and a name for the main Java package and class.



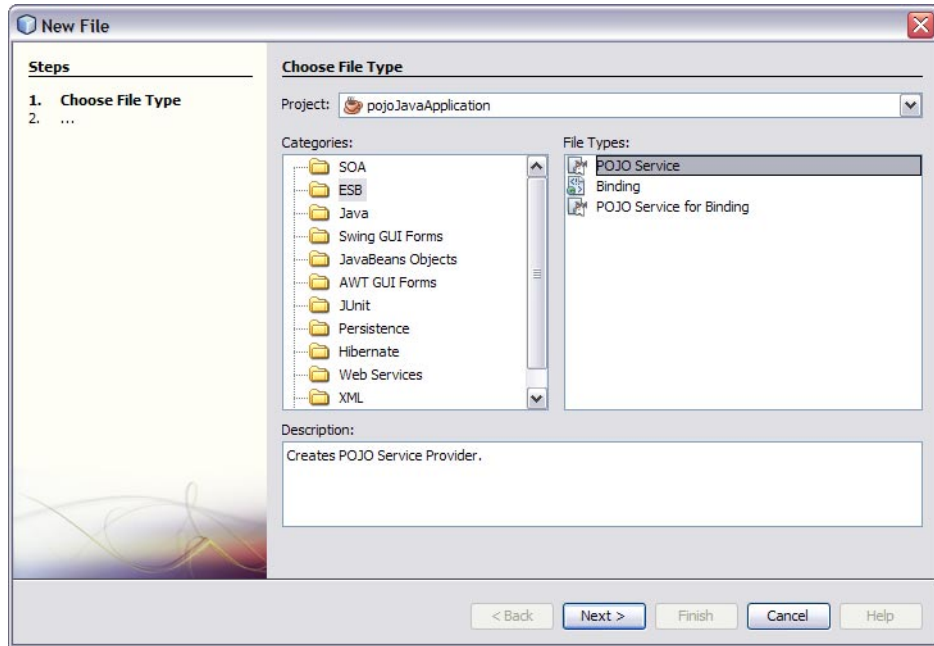
- 5 Click Finish.

The project structure is generated and appears in the Projects window.

- 6 Right-click the project you just created, point to New, and then select Other.

The New File Wizard appears.

- 7 Under Categories, select ESB; under Projects, select POJO Service.



- 8 Click Next.
The Name and Location window appears.
- 9 Fill in the Name and Location properties for the POJO service.
For more information, see [Table 1](#).
- 10 To modify advanced properties for the POJO service, click Advanced.
The POJO Provider – Advanced properties editor appears.
- 11 Modify any of the properties described in [Table 2](#), and then click OK.

12 On the New POJO Service Wizard, click Finish.

The new POJO class is generated in the project structure and any library JAR files needed to compile the project are added to the Libraries node of the project. The POJO file includes the `@Provider`, `@Operation`, and `@Resource` annotations.

```
@Provider
public class POJOProvider {

    /**
     * Constructor
     */
    public POJOProvider() {
    }

    /**
     * POJO Operation
     *
     * @param input input of type String input
     * @return String
     */
    @Operation (outMessageTypeQN="{http://pojo.openesb.glassfish.org/POJOProvider/}POJOProviderOperationResponse")
    public String receive(String input) {
        return input;
    }

    // Logger
    private static final Logger logger = Logger.getLogger(POJOProvider.class.getName());
    // POJO Context
    @Resource
    private Context jbiCtx;
}
```

Next Steps For instructions on creating a composite application for this project, see [“Creating a Composite Application for a POJO Service Provider \(Without Binding\)”](#) on page 31.

Creating a POJO Service Provider (With Binding)

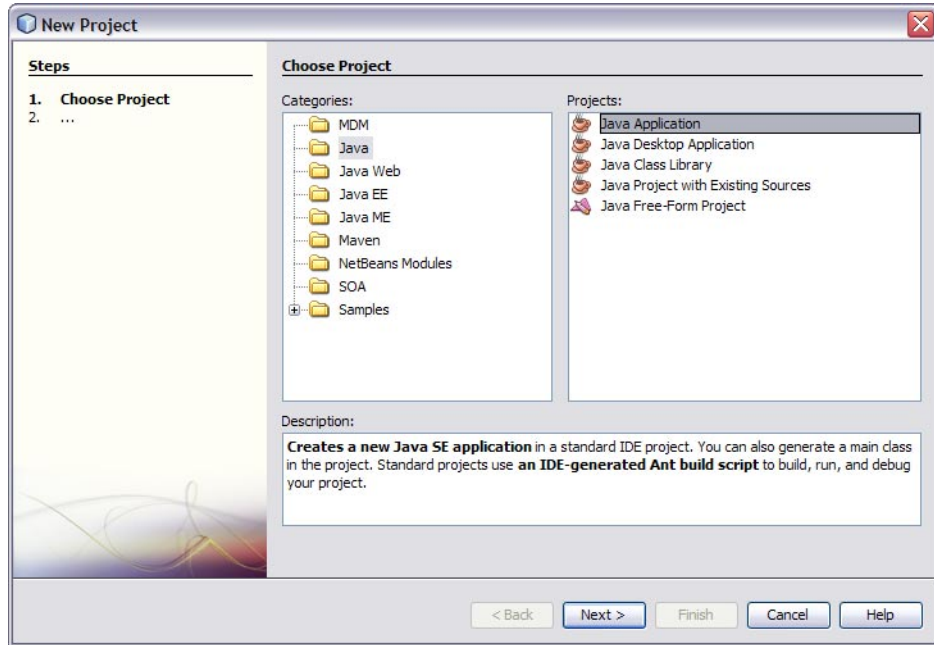
This procedure creates a POJO service provider using a wizard to guide you through the steps. This wizard includes the steps for configuring the binding component for the service provider and automatically generates the WSDL file.

▼ To Create a POJO Service Provider (With Binding)

- 1 Right-click in the NetBeans Projects window, and then select New Project.

The New Project Wizard appears.

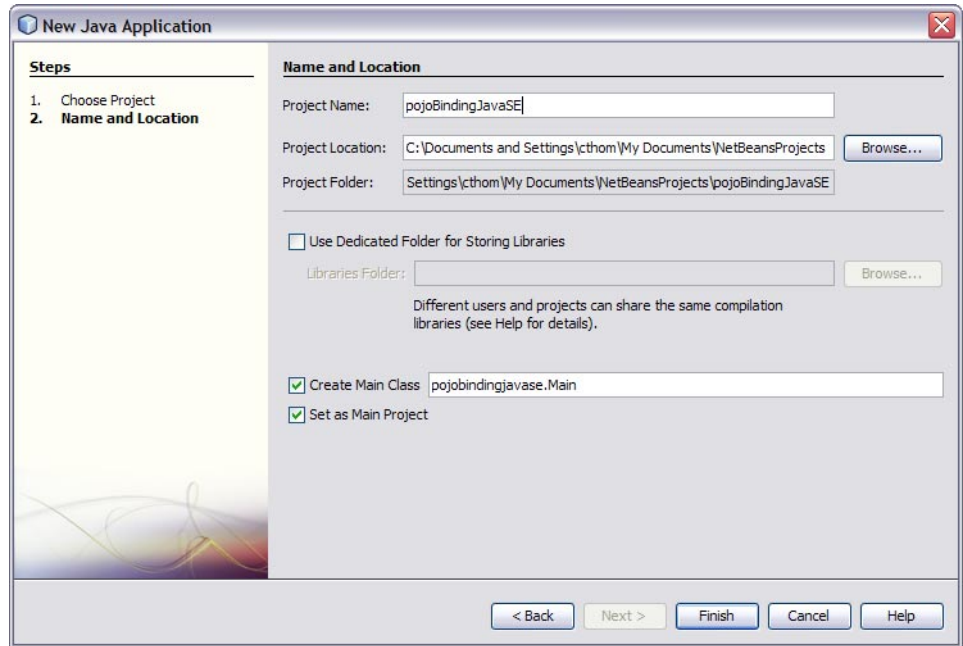
- 2 Under Categories, select Java; under Projects, select Java Application.



- 3 Click Next.

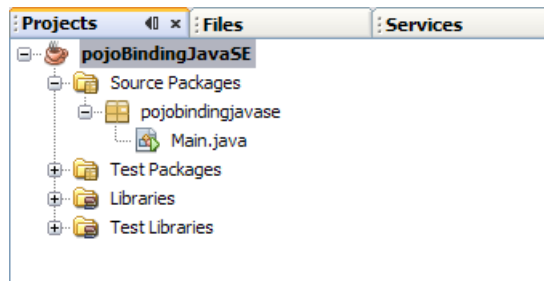
The Name and Location window appears.

- 4 Enter a unique name for the project and a name for the main Java package and class.



- 5 Click Finish.

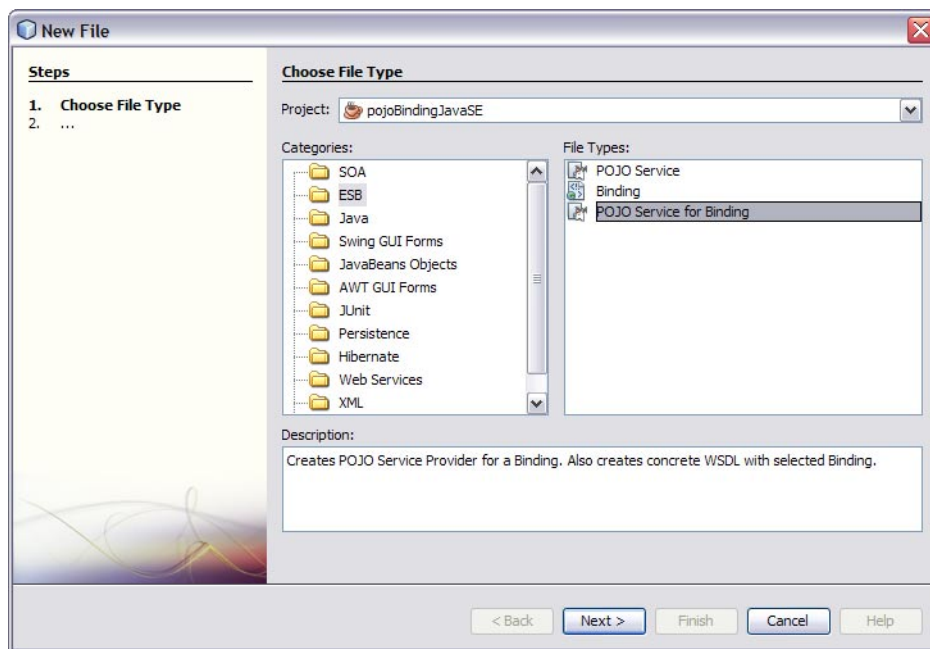
The project structure is generated and appears in the Projects window.



- 6 Right-click the project you just created, point to New, and then select Other.

The New File Wizard appears.

7 Under Categories, select ESB; under Projects, select POJO Service for Binding.

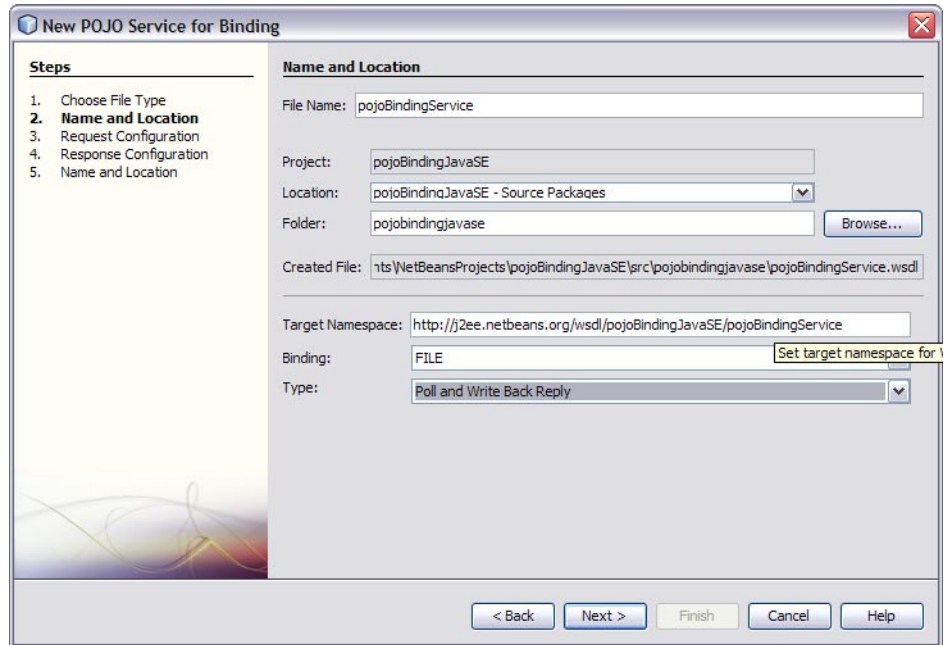


8 Click Next.

The Name and Location window for the WSDL file appears.

9 Fill in the Name and Location properties for the POJO service.

For more information, see [Table 3](#).



Note – The sequence of steps changes to reflect the binding component and type you choose. The subsequent steps will vary depending on those two properties.

10 Click Next.

The binding configuration window appears.

11 Enter information about the binding component and type, and then click Next.

Note – This page varies by binding component and type. For more information about a specific binding component, refer to the user's guide for that component or refer to the property descriptions at the bottom of the window. An example for the File Binding Component is shown below.

- 12 If a second page of binding configuration appears, enter the required information and then click

The screenshot shows a Windows-style dialog box titled "New POJO Service for Binding". On the left, a "Steps" pane lists five steps: 1. Choose File Type, 2. Name and Location, 3. Request Configuration, 4. **Response Configuration**, and 5. Name and Location. Step 4 is highlighted. The main area is titled "Request Configuration" and contains three sections: "File Polling", "Record Processing", and "Payload Processing".

File Polling

- File Name* (pattern): input.xml ☐ Is Regex
- Polling Directory*: C:\TEMP
- ☐ Poll Recursive
- Exclude Entries When Polling:
- Polling Directory Relative To: <Not S...
- Polling Interval (ms): 1000
- ☐ Enable Archive

Record Processing

- ☒ Multiple Record Delimited By: LINE FEED
- ☐ Maximum (Bytes) Per Record: 1

Payload Processing

- Message Type: text
- Character Encoding: <default> ☐ Add trailing EOL
- ☐ Forward as Attachment
- ☐ Remove trailing EOL

File Name

Defines the file name relative to the specified directory to read from or write to.

At the bottom are five buttons: "< Back", "Next >", "Finish", "Cancel", and "Help".

Next.

- 13 On the Name and Location window for the POJO service, fill in the fields described in [Table 1](#).

14 On the New POJO Service Wizard, click Finish.

The new POJO class and WSDL file are generated in the project structure and any library JAR files needed to compile the project are added to the Libraries node of the project. The POJO file includes the `@Provider`, `@Operation`, and `@Resource` annotations.

```
@Provider (name="pojoBindingService",
interfaceQN="{http://j2ee.netbeans.org/wsdl/pojoBinding/pojoBindingService}FileInboundPortType",
serviceQN="{http://j2ee.netbeans.org/wsdl/pojoBinding/pojoBindingService}FileInboundPortTypeService")
public class pojoBindingService {

    /**
     * Constructor
     */
    public pojoBindingService() {
    }

    /**
     * POJO Operation
     *
     * @param input input of type String input
     * @return String
     */
    @Operation (outMessageTypeQN="{http://j2ee.netbeans.org/wsdl/pojoBinding/pojoBindingService}PollOutputMessage";
    public String poll(String input) {
        return input;
    }

    // Logger
    private static final Logger logger = Logger.getLogger(pojoBindingService.class.getName());
    // POJO Context
    @Resource
    private Context jbiCtx;
}
```

Next Steps For instructions on creating a composite application for this project, see [“Creating a Composite Application for a POJO Service Provider \(With Binding\)”](#) on page 33.

Creating a POJO Service Provider in an Existing Java Application

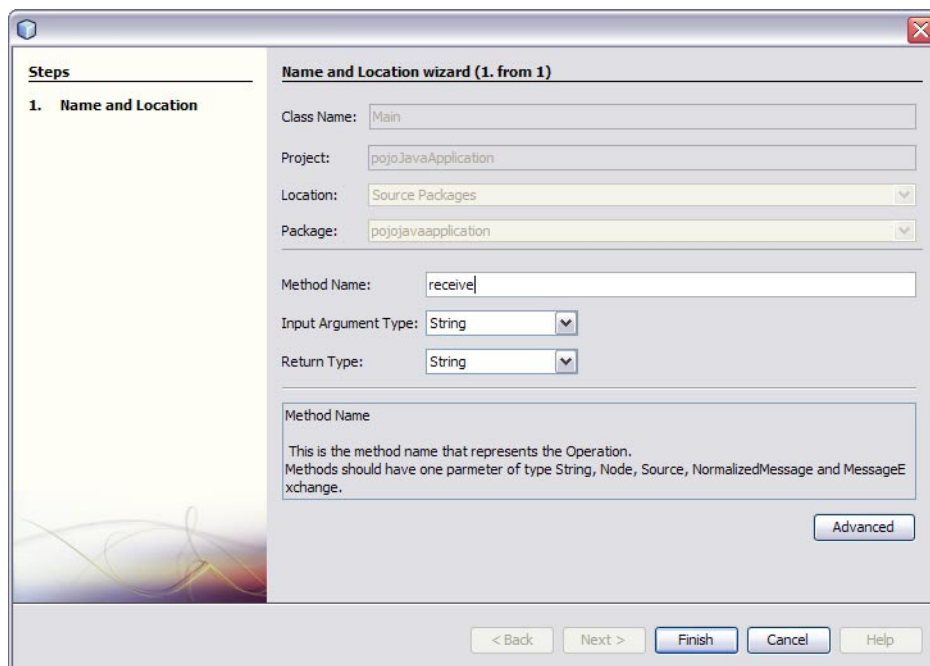
This procedure creates a POJO service provider in an existing Java file using the Palette on the NetBeans Java Editor. Dragging and dropping a provider into the Java code creates a framework for the service provider, which you can then customize with the needed processing logic. There is no WSDL document or binding involved in this procedure; a binding can be added when you create the composite application.

▼ To Create a POJO Service Provider in a Java Application

- 1 Open the Java file to which you want to add the POJO service provider.
- 2 If the Palette is not visible, click **Window** in the NetBeans toolbar and then click **Palette**.
The Palette appears to the right of the Java Editor.

- 3 **Drag and drop a POJO Provider from the Palette to the location in the Java file where you want to insert to POJO service provider.**

The Name and Location Wizard appears.



The image shows a 'Name and Location wizard (1. from 1)' dialog box. On the left, a 'Steps' pane shows '1. Name and Location'. The main area contains several input fields: 'Class Name' with 'Main', 'Project' with 'pojoJavaApplication', 'Location' with 'Source Packages', and 'Package' with 'pojojavaapplication'. Below these are 'Method Name' with 'receive', 'Input Argument Type' with 'String', and 'Return Type' with 'String'. A 'Method Name' section at the bottom explains that the method name represents the operation and should have one parameter of type String, Node, Source, NormalizedMessage, or MessageExchange. An 'Advanced' button is on the right. At the bottom are '< Back', 'Next >', 'Finish', 'Cancel', and 'Help' buttons.

- 4 **Enter the method name, input type, and return type.**

For more information about these fields, see [Table 1](#).

- 5 **To configure advanced properties, click Advanced.**

The Advanced properties editor appears.

- 6 **Modify any of the properties described in [Table 2](#).**

7 Click Finish.

Any library JAR files needed to compile the project are added to the Libraries node of the project. The `@Provider`, `@Operation`, and `@Resource` annotations are added at the insertion point.

```
@Provider
public class Main {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // TODO code application logic here
    }

    @Resource
    private Context jbiCtx;

    @Operation(outMessageTypeQN = "{http://pojojavaapplication/Main/}MainOperationResponse")
    public String receive(String var0) {
        return var0;
    }

}
```

Next Steps For instructions on creating a composite application for this project, see [“Creating a Composite Application for a POJO Service Provider \(Without Binding\)”](#) on page 31.

Creating a POJO Service Consumer (Without Binding)

This procedure creates a service consumer within an existing service provider definition. Dragging and dropping a consumer into the Java code creates a framework for the service consumer, which you can then customize with the needed processing logic. There is no WSDL document or binding involved in this procedure; a binding can be added when you create the composite application.

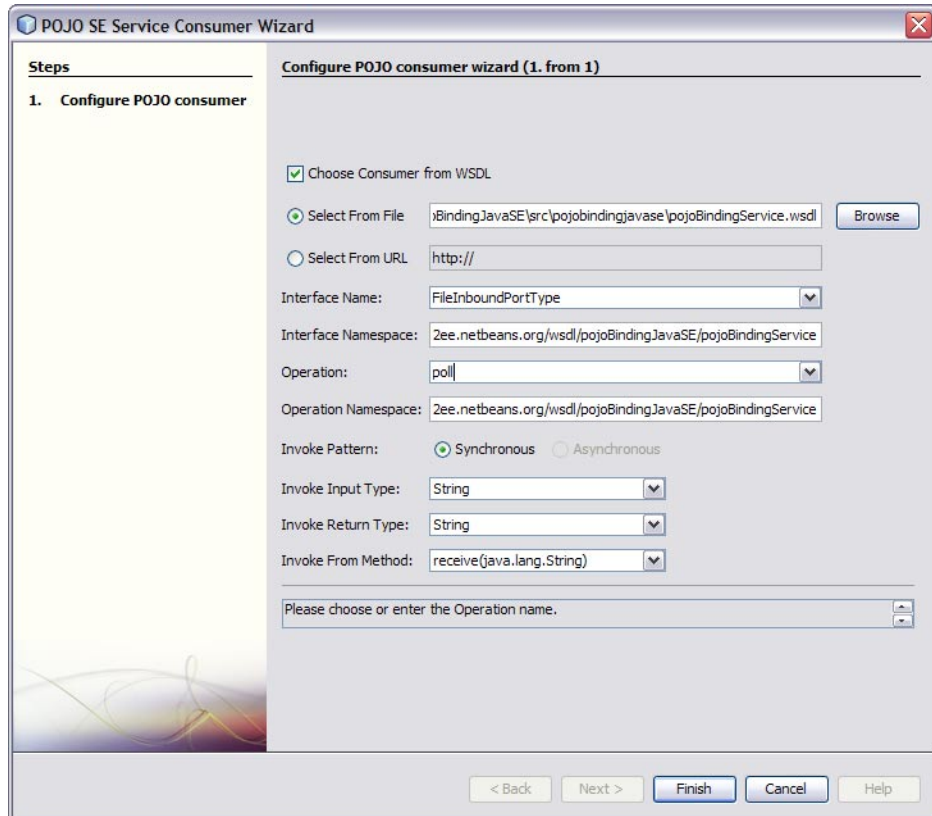
▼ To Create a POJO Service Consumer (Without Binding)

Before You Begin The POJO service consumer can only be added within an existing provider. Make sure you have a provider defined in the Java class before beginning this procedure.

- 1 **Open the Java file to which you want to add the POJO service consumer.**
- 2 **If the Palette is not visible, click Window in the NetBeans toolbar and then click Palette.**
The Palette appears to the right of the Java Editor.

- 3 Drag and drop a POJO Consumer from the Palette to the location in the service provider in the Java file where you want to insert to POJO service consumer.

The POJO SE Service Consumer Wizard appears.



- 4 Fill in the consumer properties, as described in [“POJO Service Consumer Properties” on page 29](#).
- 5 Click Finish.

The `@ConsumerEndpoint` annotation and related methods are added at the insertion point.

Next Steps For instructions on creating a composite application for this project, see [“Creating a Composite Application for a POJO Service Consumer \(Without Binding\)” on page 36](#).

Creating a POJO Service Consumer (With Binding)

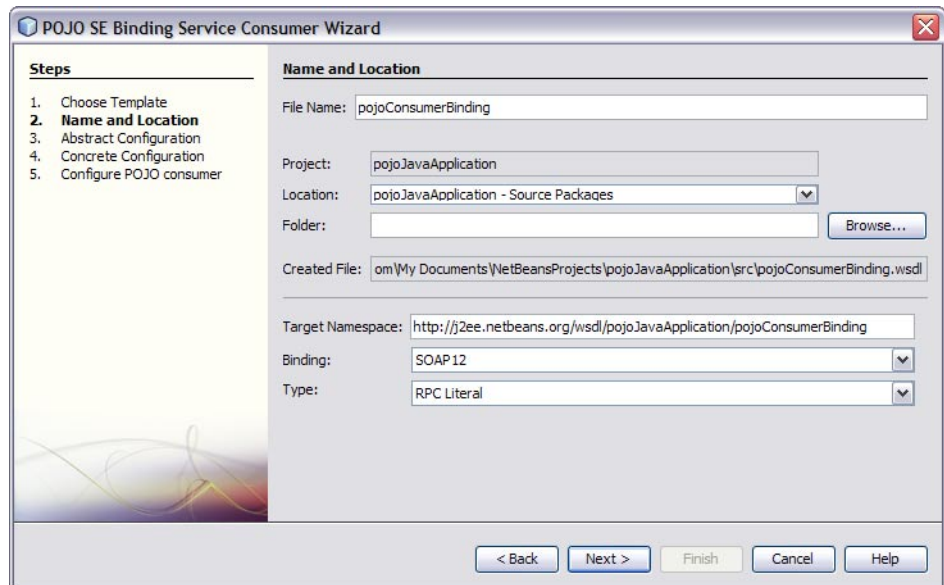
This procedure creates a POJO service consumer in an existing Java file using the Palette on the NetBeans Java Editor. Dragging and dropping a consumer into the Java code creates a framework for the service consumer, which you can then customize with the needed processing logic. When you add the consumer, you also configure the binding component for the service consumer and the WSDL file is automatically generated.

▼ To Create a POJO Service Consumer (With Binding)

Before You Begin The POJO service consumer can only be added within an existing provider. Make sure you have a provider defined in the Java class before beginning this procedure.

- 1 **Open the Java file to which you want to add the POJO service consumer.**
- 2 **If the Palette is not visible, click Window in the NetBeans toolbar and then click Palette.**
The Palette appears to the right of the Java Editor.
- 3 **Drag and drop a POJO Binding Consumer from the Palette to the location in the service provider in the Java file where you want to insert to POJO service consumer.**

The POJO SE Binding Service Consumer Wizard appears.



- 4 **Fill in the name and location properties, as described in “POJO Service Properties for Binding” on page 28.**

Note – The sequence of steps changes to reflect the binding component and type you choose. The subsequent steps will vary depending on those two properties.

5 Click Next.

The abstract binding configuration window appears.

6 Enter information about the binding component and type, and then click Next.

Note – This page varies by binding component and type. For more information about a specific binding component, refer to the user's guide for that component or refer to the property descriptions at the bottom of the window. An example for the HTTP Binding Component is shown below.

7 If a second page of binding configuration appears, enter the required information and then click

The screenshot shows the 'POJO SE Binding Service Consumer Wizard' window, specifically the 'Abstract Configuration' tab. On the left, a 'Steps' pane lists five steps: 1. Choose Template, 2. Name and Location, 3. Abstract Configuration (highlighted), 4. Concrete Configuration, and 5. Configure POJO consumer. The main area contains configuration fields for 'Port Type Name' (set to 'pojoConsumerBindingPortType'), 'Operation Name' (set to 'pojoConsumerBindingOperation'), and 'Operation Type' (set to 'Request-Response Operation'). Below these are sections for 'Input', 'Output', and 'Fault', each with a table for 'Message Part Name' and 'Element Or Type'. The 'Input' and 'Output' sections each have one row with 'part1' and 'xsd:string'. Each section has 'Add' and 'Remove' buttons. At the bottom, there is a checkbox for 'Generate partnerlinktype automatically.' and a set of navigation buttons: '< Back', 'Next >', 'Finish', 'Cancel', and 'Help'.

Next.

8 On the Name and Location window for the POJO service, the interface and operation fields are automatically populated. Fill in the fields described in Table 4 beginning with Invoke Pattern.

9 Click Finish.

The `@ConsumerEndpoint` annotation and related methods are added at the insertion point.

Next Steps For instructions on creating a composite application for this project, see [“Creating a Composite Application for a POJO Service Consumer \(With Binding\)”](#) on page 39.

POJO Configuration Properties

When you create a POJO service consumer or provider, with or without binding, you need to configure certain properties so the POJO framework and optionally the WSDL documents can be generated correctly. The following topics describe the properties you need to configure. Note that these do not include specific binding component properties, which you need to configure if you are using bindings. For more information about these properties, see the user's guide for the specific binding component you are using.

- [“POJO Service Provider Properties”](#) on page 27
- [“POJO Service Properties for Binding”](#) on page 28
- [“POJO Service Consumer Properties”](#) on page 29

POJO Service Provider Properties

The following tables list and describe the properties you can configure when creating a new POJO service. These properties appear on the Name and Location page of the New POJO Service Wizard and on the Advanced Properties Editor, which is accessed from the Name and Location page.

TABLE 1 POJO Service Provider Name and Location Properties

Property	Description
Class Name	A unique name for the Java class to create for the POJO service.
Location	The location in the project structure to create the Java class and package.
Package	A unique Java package name for the class.
Method Name	The name of the method that represents the operation. This will be automatically generated in the Java class.
Input Argument Type	<p>The input type for the operation. This represents the parameter type for the operation method. Select one of the following values:</p> <ul style="list-style-type: none"> ■ String (<code>java.lang.string</code>) ■ Source (<code>javax.xml.transform.Source</code>) ■ Normalized Message (<code>javax.jbi.messaging.NormalizedMessage</code>) ■ Node (<code>org.w3c.dom.Node</code>) ■ Document ■ Message Exchange (<code>javax.jbi.messaging.MessageExchange</code>)

TABLE 1 POJO Service Provider Name and Location Properties *(Continued)*

Property	Description
Return Type	<p>The output type for the operation. This represents the return type for the operation method. Select one of the following values:</p> <ul style="list-style-type: none"> ■ String (<code>java.lang.string</code>) ■ Source (<code>javax.xml.transform.Source</code>) ■ Normalized Message (<code>javax.jbi.messaging.NormalizedMessage</code>) ■ Node (<code>org.w3c.dom.Node</code>) ■ Document ■ Void

TABLE 2 POJO Service Provider Advanced Properties

Property	Description
Endpoint Name	The name of the POJO provider's endpoint.
Interface Name	The local interface name for the POJO provider. If this property is left blank, it defaults to the class name with "Interface" appended. For example, <code>MyPojoInterface</code> .
Interface Namespace	Additional interface namespaces for the POJO provider.
Service Name	The local service name for the POJO provider. If this property is left blank, it defaults to the class name with "Service" appended. For example, <code>MyPojoService</code> .
Service Namespace	The namespace of the JBI service. If this property is left blank, the default namespace will be an HTTP URL with the package name of class reversed and the endpoint name as the path. For example, if the package is <code>org.glassfish.openesb.pojo</code> and the endpoint name is <code>MyPojoProvider</code> , the URL will be <code>http://pojo.openesb.glassfish.org/MyPojoProvider/</code> .
Output Message Type Name	The WSDL message type for output messages for a return type of String, Node, or Source if the consuming JBI component expects the WSDL 1.1 wrapper message to contain these attributes.
Output Message Type Namespace	The namespace of the output message type specified above.

POJO Service Properties for Binding

The following table lists and describes the properties you can configure when creating a new POJO service provider or consumer with a binding component. These properties appear on the Name and Location page of the New POJO Service for Binding Wizard and the POJO SE Binding Service Consumer Wizard.

TABLE 3 Name and Location Properties for Binding

Property	Description
File Name	A unique name for the binding WSDL document for the POJO service.
Location	The name of the node in the project structure in which to create the WSDL document.
Folder	The name of the folder under the above node in which to create the WSDL document.
Target Namespace	The target namespace for the binding WSDL document.
Binding	The type of binding component to use for the POJO service. This determines the template for the WSDL document.
Type	The type of binding. Note that for some binding components, only a subset of their types is supported.

POJO Service Consumer Properties

The following table lists and describes the properties you can configure when creating a new POJO service consumer. These properties appear on the POJO SE Service Consumer Wizard.

TABLE 4 POJO Consumer Wizard Properties

Property	Description
Choose Consumer From WSDL	An indicator of whether to create the consumer endpoint from an existing WSDL document. You can specify a local path and filename or the URL for the WSDL document.
Select From File	The path and file name of the local WSDL document to use. Pressing the Tab key after entering this value populates the interface and operation fields.
Select From URL	The URL to the remote WSDL document to use. Pressing the Tab key after entering a URL populates the interface and operation fields.
Interface Name	The name of the consumer interface.
Interface Namespace	The namespace for the consumer interface.
Operation	The type of binding operation for the consumer.
Operation Namespace	The namespace for the binding operation.
Invoke Pattern	An indicator of whether to invoke the consumer synchronously or asynchronously. Select the appropriate checkbox. Both options may not be available for some operations.

TABLE 4 POJO Consumer Wizard Properties *(Continued)*

Property	Description
Invoke Input Type	<p>The input type for the operation. This represents the parameter type for the operation method. Select one of the following values:</p> <ul style="list-style-type: none"> ■ <code>String (java.lang.string)</code> ■ <code>Source (javax.xml.transform.Source)</code> ■ <code>Normalized Message (javax.jbi.messaging.NormalizedMessage)</code> ■ <code>Node (org.w3c.dom.Node)</code> ■ <code>Document</code> ■ <code>Message Exchange (javax.jbi.messaging.MessageExchange)</code>
Return Type	<p>The output type for the operation. This represents the return type for the operation method. Select one of the following values:</p> <ul style="list-style-type: none"> ■ <code>String (java.lang.string)</code> ■ <code>Source (javax.xml.transform.Source)</code> ■ <code>Normalized Message (javax.jbi.messaging.NormalizedMessage)</code> ■ <code>Node (org.w3c.dom.Node)</code> ■ <code>Document</code> ■ <code>Void</code>
Invoke From Method	<p>The method from which the consumer is invoked. When you drag and drop the POJO Consumer from the Palette into a Java method, that method name is automatically populated here.</p>

Creating Composite Applications for POJO Service Engine Projects

The way you create a Composite Application varies depending on whether you created a POJO service provider or consumer, and whether you created it with or without binding. The following topics describe the different methods.

- [“Creating a Composite Application for a POJO Service Provider \(Without Binding\)” on page 31](#)
- [“Creating a Composite Application for a POJO Service Provider \(With Binding\)” on page 33](#)
- [“Creating a Composite Application for a POJO Service Consumer \(Without Binding\)” on page 36](#)
- [“Creating a Composite Application for a POJO Service Consumer \(With Binding\)” on page 39](#)

Creating a Composite Application for a POJO Service Provider (Without Binding)

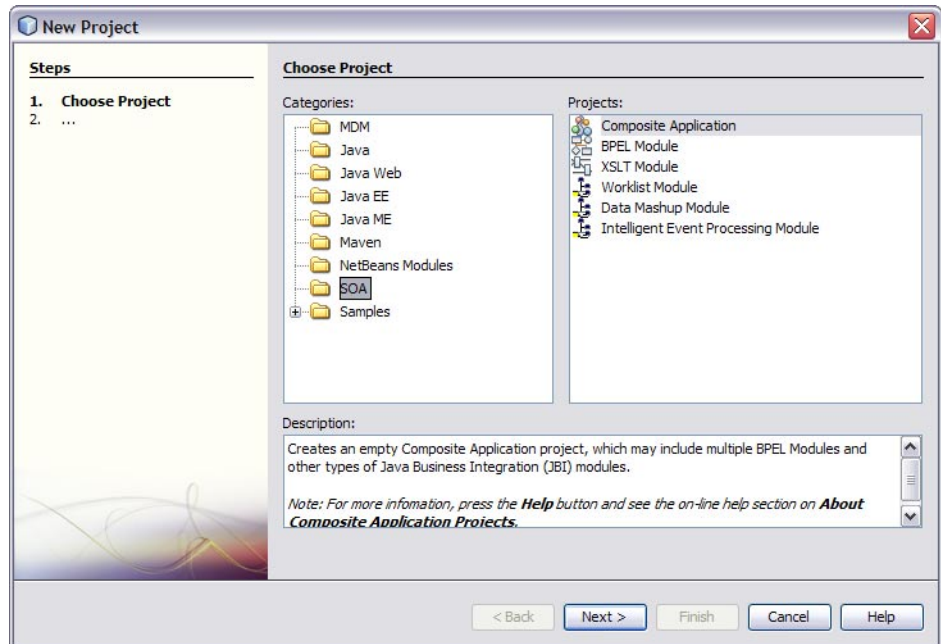
When you create a POJO service provider without any binding, you can apply the binding to the service assembly in the Composite Application.

▼ To Create a Composite Application for a POJO Service Provider (Without Binding)

- 1 Right-click in the NetBeans Projects window, and then select New Project.

The New Project Wizard appears.

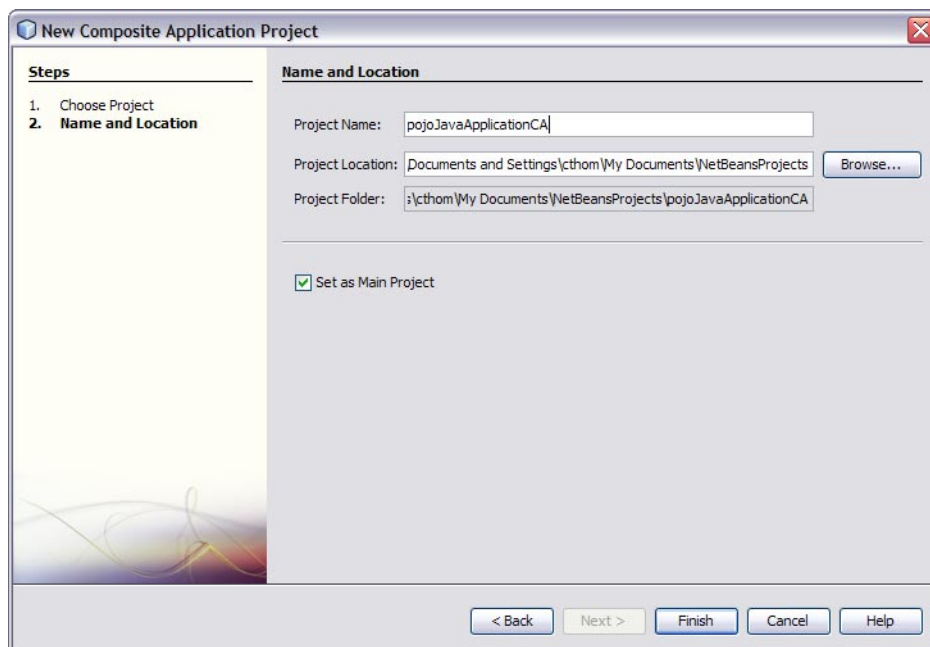
- 2 Under Categories, select SOA; under Projects, select Composite Application.



- 3 Click Next.

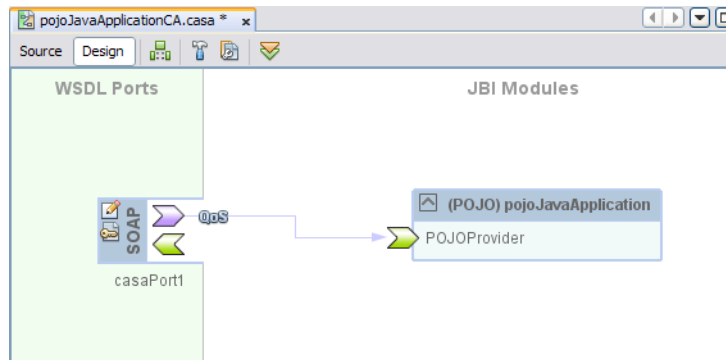
The Name and Location window appears.

- 4 Enter a unique name for the project. You can also modify the location of the project files.



- 5 Click Finish.
The new project appears in the projects list and the Composite Application appears in the CASA Editor.
- 6 Drag the POJO Service Engine project to the JBI Module section of the CASA Editor.
- 7 Click Build.
- 8 From the CASA Editor Palette, drag the appropriate WSDL Binding to the WSDL Ports section of the CASA Editor.

- 9 Drag a connection from the WSDL port endpoint to the POJO service endpoint.



- 10 Save the changes to the Composite Application.
- 11 To deploy the application, do the following:
- Make sure the GlassFish server is running.
 - In the Projects window, right-click the Composite Application project and then select Deploy.

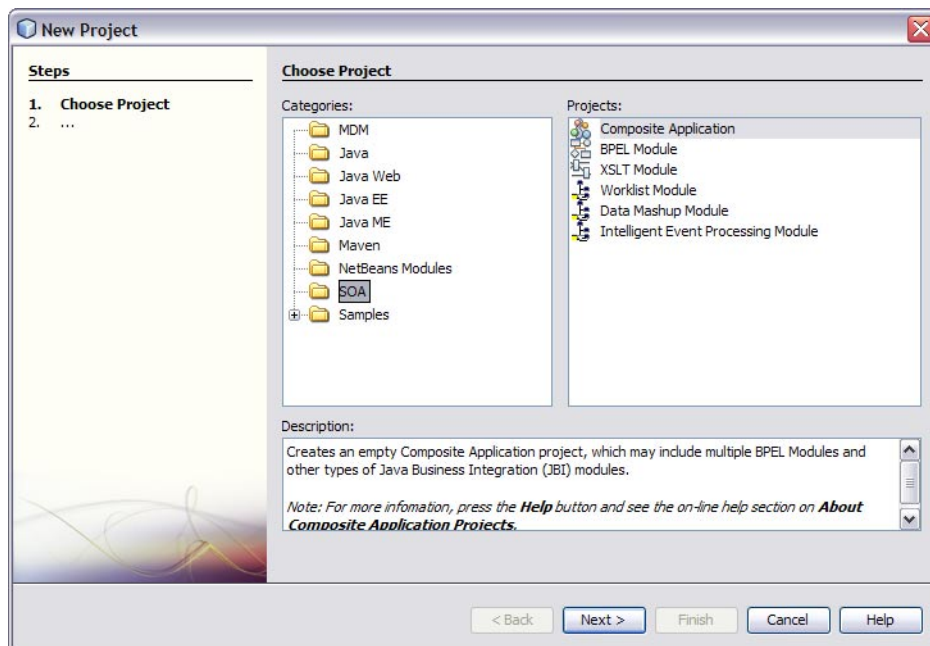
Note – If the POJO Service Engine is not started, deploying the Composite Application will automatically start it for you.

Creating a Composite Application for a POJO Service Provider (With Binding)

When you create a Composite Application for a POJO service provider with binding, the CASA Editor automates most of the work for you.

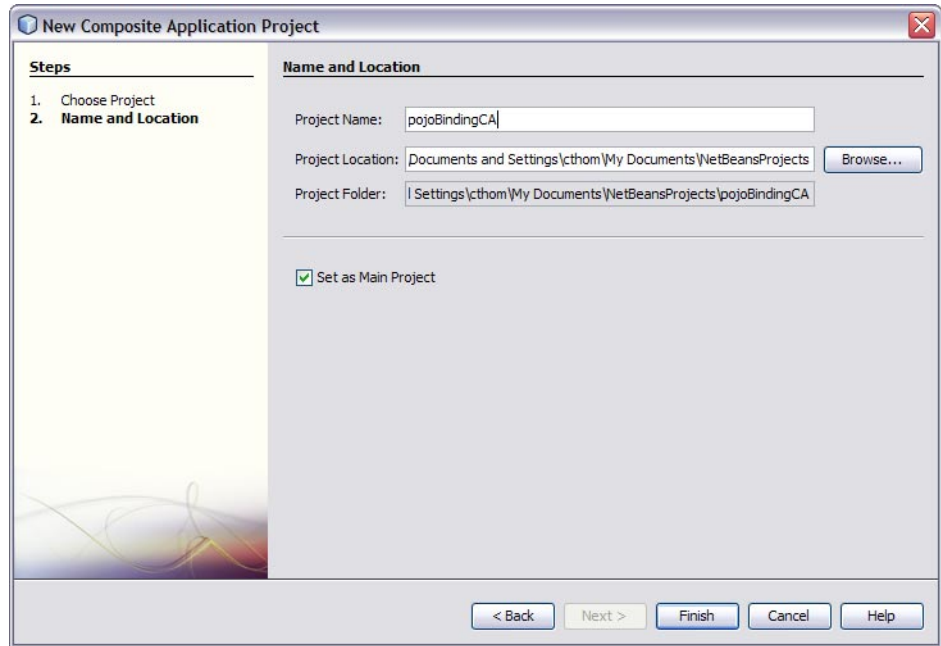
▼ To Create a Composite Application for a POJO Service Provider (With Binding)

- 1 Right-click in the NetBeans Projects window, and then select New Project.
The New Project Wizard appears.
- 2 Under Categories, select SOA; under Projects, select Composite Application.



- 3 Click Next.
The Name and Location window appears.

- 4 Enter a unique name for the project. You can also modify the location of the project files.



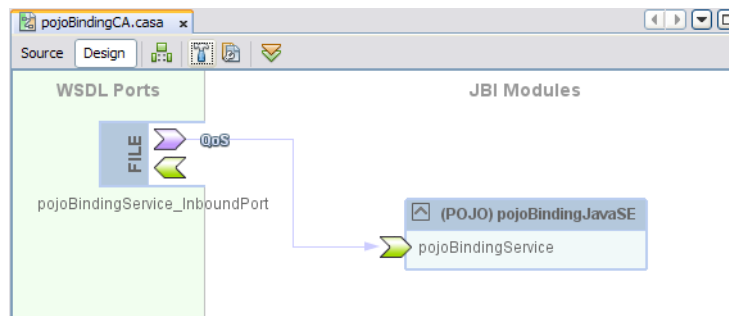
- 5 Click Finish.

The new project appears in the projects list and the Composite Application appears in the CASA Editor.

- 6 Drag the POJO Service Engine project to the JBI Module section of the CASA Editor.

- 7 Click Build.

The WSDL port and JBI Module both appear in the CASA Editor with the endpoints connected.



- 8 Save the changes to the Composite Application.

- 9 To deploy the application, do the following:
 - a. Make sure the GlassFish server is running.
 - b. In the Projects window, right-click the Composite Application project and then select Deploy.

Note – If the POJO Service Engine is not started, deploying the Composite Application will automatically start it for you.

Creating a Composite Application for a POJO Service Consumer (Without Binding)

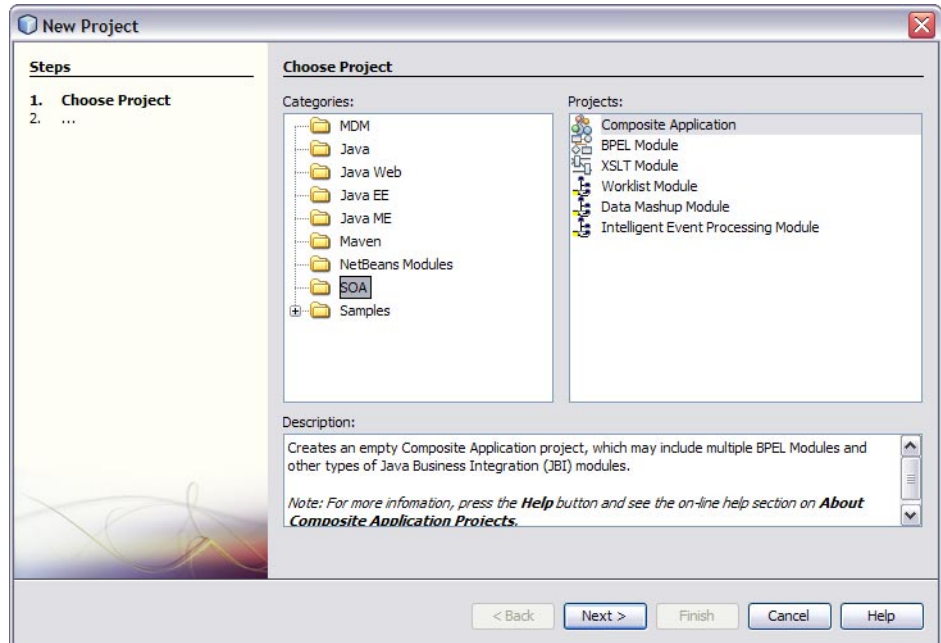
When you create a POJO service consumer without any binding, you can apply the binding to the service assembly in the Composite Application.

▼ To Create a Composite Application for a POJO Service Consumer (Without Binding)

- 1 Right-click in the NetBeans Projects window, and then select New Project.

The New Project Wizard appears.

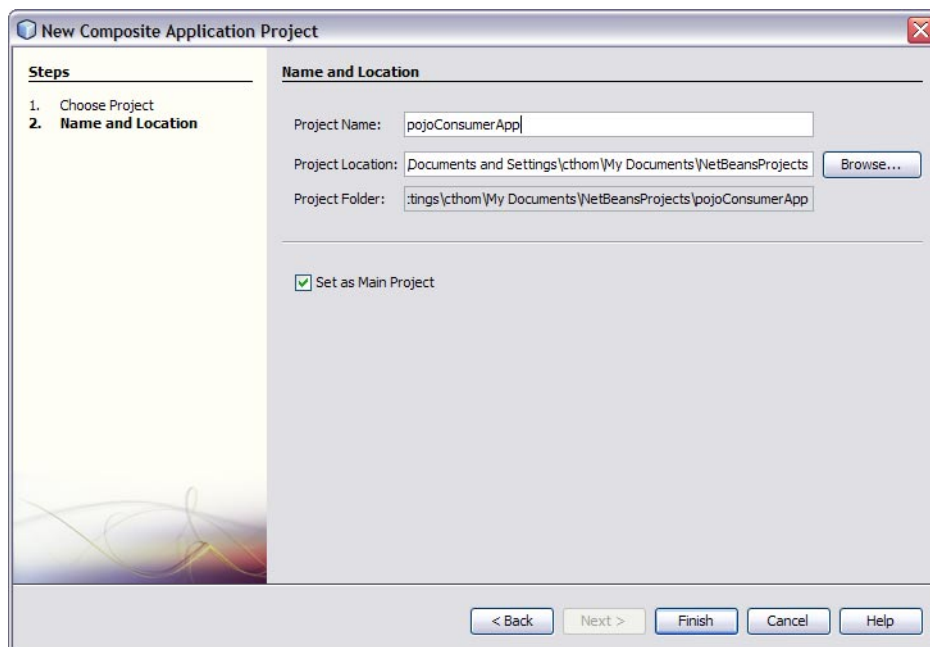
- 2 Under Categories, select SOA; under Projects, select Composite Application.



- 3 Click Next.

The Name and Location window appears.

- 4 Enter a unique name for the project. You can also modify the location of the project files.



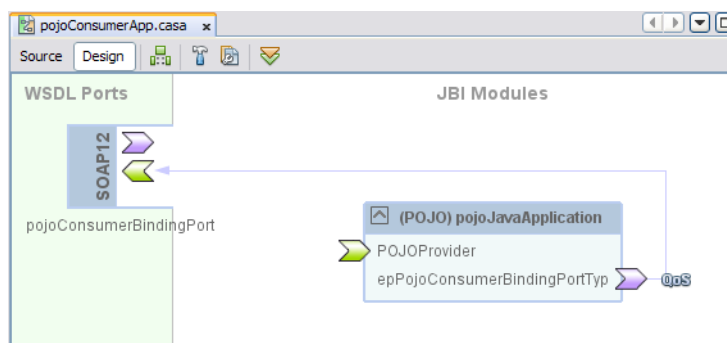
- 5 Click Finish.

The new project appears in the projects list and the Composite Application appears in the CASA Editor.

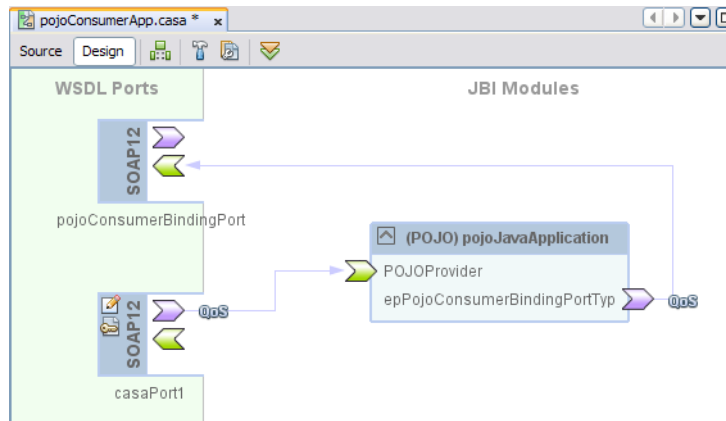
- 6 Drag the POJO Service Engine project to the JBI Module section of the CASA Editor.

- 7 Click Build.

If you configured the POJO consumer from an existing WSDL document, the WSDL port for the consumer appears and is connected to the POJO application.



If you added the consumer to a provider that was created with binding, the WSDL port for the provider also appears and is connected to the POJO application.



- 8 If you did not configure the consumer from an existing WSDL document, drag the appropriate WSDL binding from the Palette to the WSDL Ports section of the CASA Editor and connect it to the consumer endpoint as shown above.
- 9 If the provider was not created with binding, drag the appropriate WSDL binding from the Palette to the WSDL Ports section of the CASA Editor and connect it to the provider endpoint as shown above.
- 10 Save the changes to the Composite Application.
- 11 To deploy the application, do the following:
 - a. Make sure the GlassFish server is running.
 - b. In the Projects window, right-click the Composite Application project and then select Deploy.

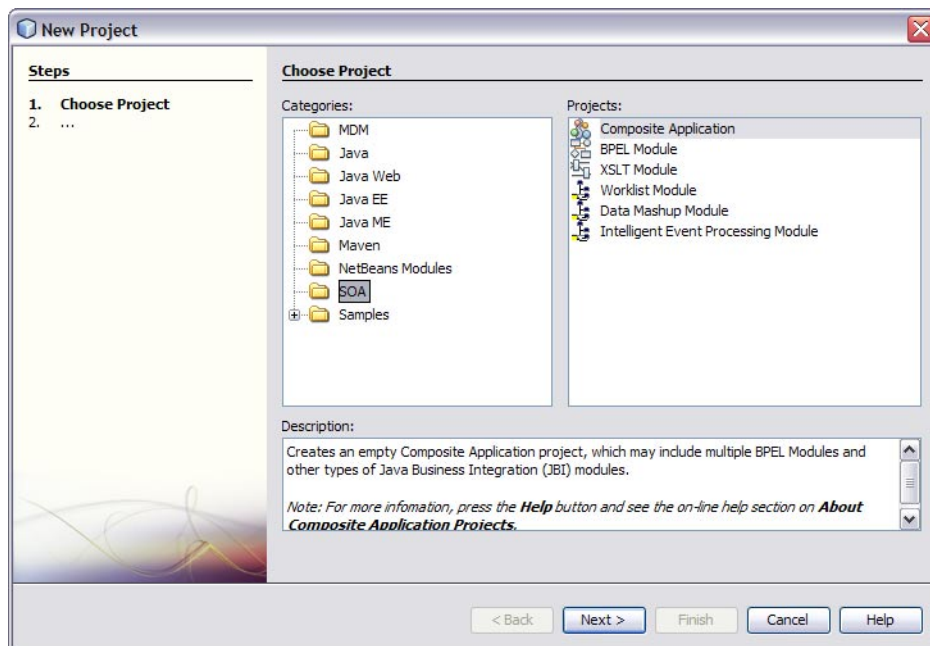
Note – If the POJO Service Engine is not started, deploying the Composite Application will automatically start it for you.

Creating a Composite Application for a POJO Service Consumer (With Binding)

When you create a Composite Application for a POJO service consumer with binding, building the Composite Application automates most of the work for you.

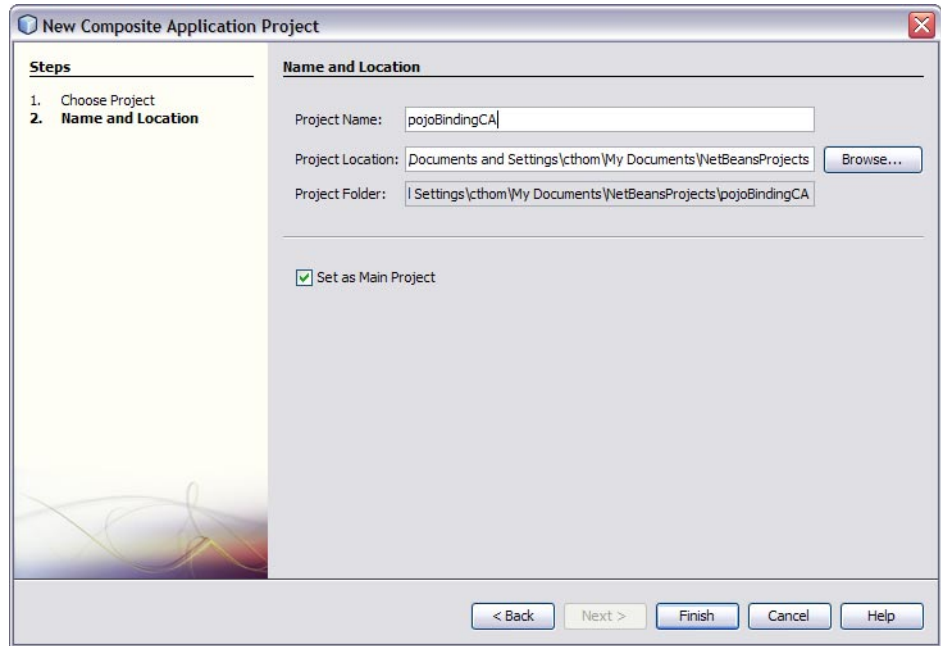
▼ To Create a Composite Application for a POJO Service Consumer (With Binding)

- 1 Right-click in the NetBeans Projects window, and then select New Project.
The New Project Wizard appears.
- 2 Under Categories, select SOA; under Projects, select Composite Application.



- 3 Click Next.
The Name and Location window appears.

- 4 Enter a unique name for the project. You can also modify the location of the project files.



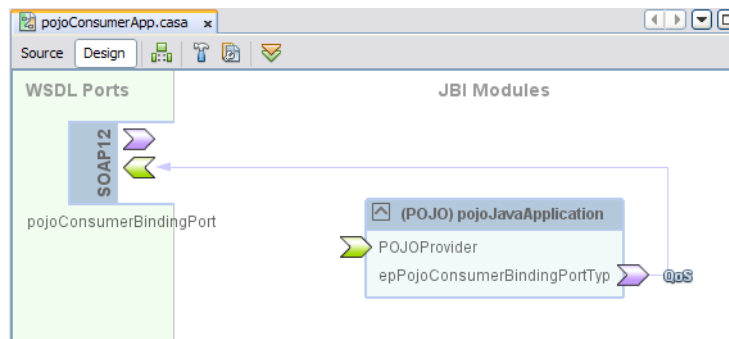
- 5 Click Finish.

The new project appears in the projects list and the Composite Application appears in the CASA Editor.

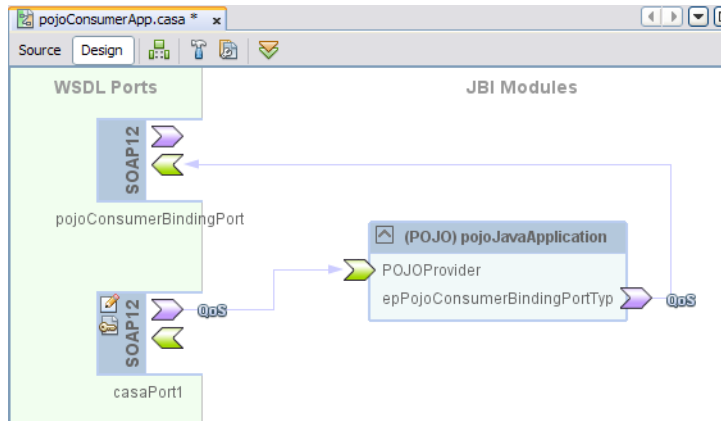
- 6 Drag the POJO Service Engine project to the JBI Module section of the CASA Editor.

- 7 Click Build.

The WSDL port for the consumer appears and is connected to the POJO application.



If you added the consumer to a provider that was created with binding, the WSDL port for the provider also appears and is connected to the POJO application.



- 8 If the provider was not created with binding, drag the appropriate WSDL binding from the Palette to the WSDL Ports section of the CASA Editor and connect it to the provider endpoint as shown above.
- 9 Save the changes to the Composite Application.
- 10 To deploy the application, do the following:
 - a. Make sure the GlassFish server is running.
 - b. In the Projects window, right-click the Composite Application project and then select Deploy.

Note – If the POJO Service Engine is not started, deploying the Composite Application will automatically start it for you.

Using POJO Services With BPEL

You can call a POJO service from a BPEL business process and you can call a business process from the POJO service. The following procedures explain how to create the BPEL Modules and POJO projects, and then how to create the Composite Applications.

- “Invoking POJO Services from a Business Process” on page 43
- “Calling a Business Process From a POJO Service” on page 48

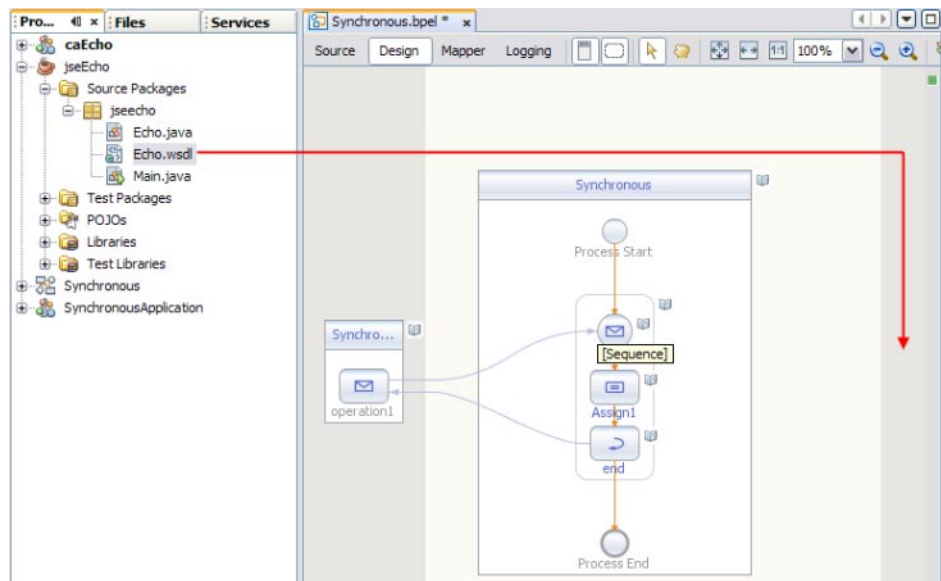
Invoking POJO Services from a Business Process

This procedure provides general instructions for calling a POJO service from a business process. You can view a tutorial with more detailed information for completing this process at <http://wiki.open-esb.java.net/Wiki.jsp?page=POJOTutorialEchoServiceInvokedByBPEL>.

▼ To Invoke a POJO Service from a Business Process

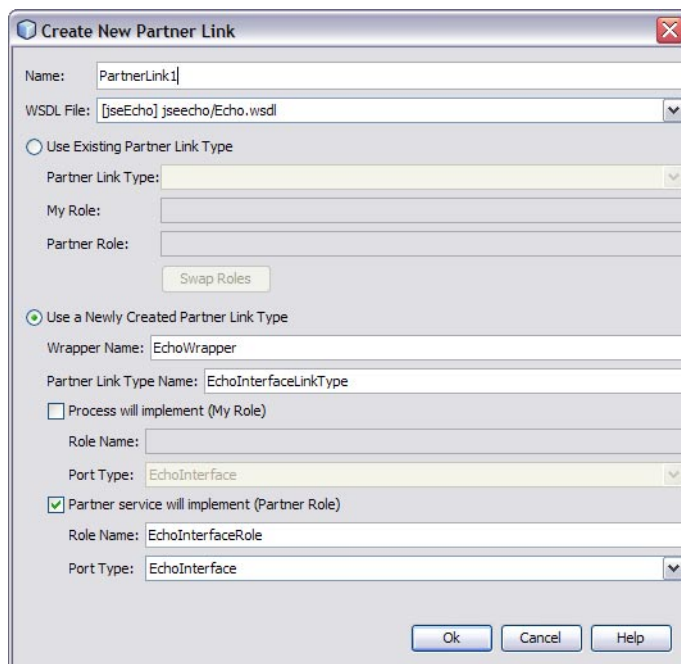
Before You Begin This task requires that the POJO service being called is already created. For information on creating a POJO service, see “Creating POJO Service Engine Projects” on page 11.

- 1 **Create the business process with any necessary activities and WSDL documents.**
For more information about creating business processes, see the *BPEL Designer and Service Engine User's Guide*.
- 2 **Add an Invoke activity to the point in the business process where you want to invoke the POJO service.**
- 3 **Drag the WSDL document from the POJO project in the Projects window to the right partner link panel of the BPEL Editor.**



The Create New Partner Link dialog box appears.

4 Modify the partner link information, or accept the default values.



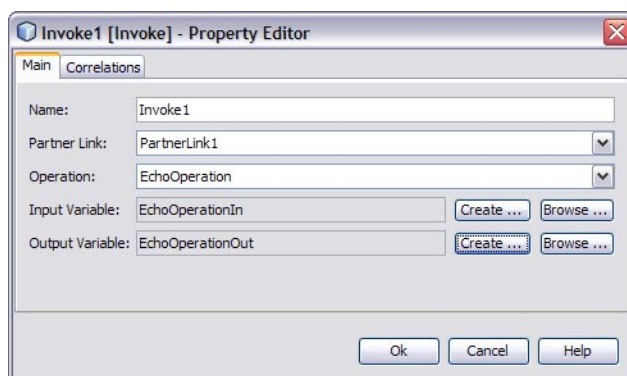
The "Create New Partner Link" dialog box is shown. It has a title bar with a close button. The "Name" field contains "PartnerLink1". The "WSDL File" dropdown shows "[jseEcho] jseecho/Echo.wsdl". There are two radio buttons: "Use Existing Partner Link Type" (unselected) and "Use a Newly Created Partner Link Type" (selected). Under "Use Existing Partner Link Type", there are fields for "Partner Link Type", "My Role", and "Partner Role", along with a "Swap Roles" button. Under "Use a Newly Created Partner Link Type", there are fields for "Wrapper Name" (EchoWrapper), "Partner Link Type Name" (EchoInterfaceLinkType), a checkbox for "Process will implement (My Role)" (unchecked), "Role Name", "Port Type" (EchoInterface), a checkbox for "Partner service will implement (Partner Role)" (checked), "Role Name" (EchoInterfaceRole), and "Port Type" (EchoInterface). At the bottom are "Ok", "Cancel", and "Help" buttons.

5 Click OK.

6 In the business process, double-click the Invoke activity.

The activity Properties Editor appears.

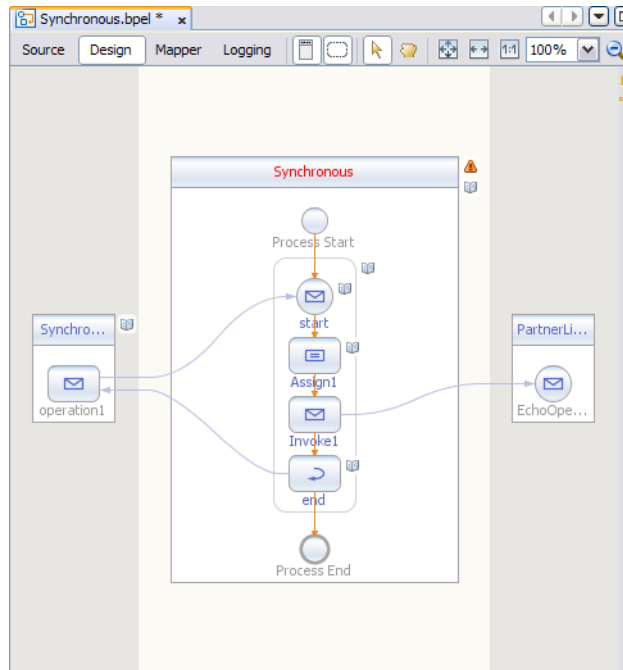
7 Define the information for the partner link, as shown in the example below.



The "Invoke1 [Invoke] - Property Editor" dialog box is shown. It has a title bar with a close button. There are two tabs: "Main" (selected) and "Correlations". In the "Main" tab, there are fields for "Name" (Invoke1), "Partner Link" (PartnerLink1), "Operation" (EchoOperation), "Input Variable" (EchoOperationIn), and "Output Variable" (EchoOperationOut). Next to each variable field are "Create ..." and "Browse ..." buttons. At the bottom are "Ok", "Cancel", and "Help" buttons.

8 Click OK.

A connection appears between the Invoke activity and the partner link from the POJO project.

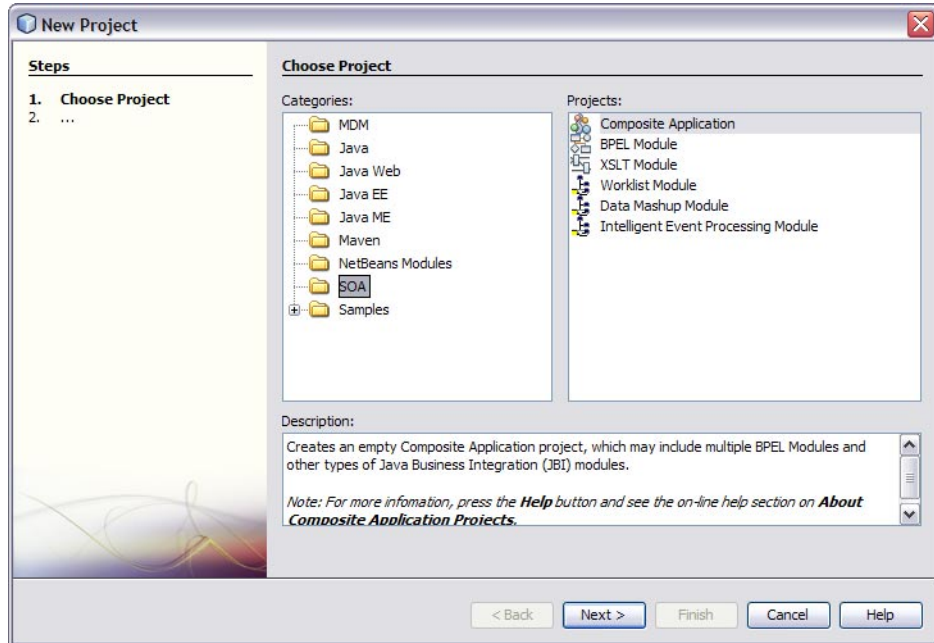


▼ To Create the Composite Application

- 1 Right-click in the Projects window, and then select New Project.

The New Project Wizard appears.

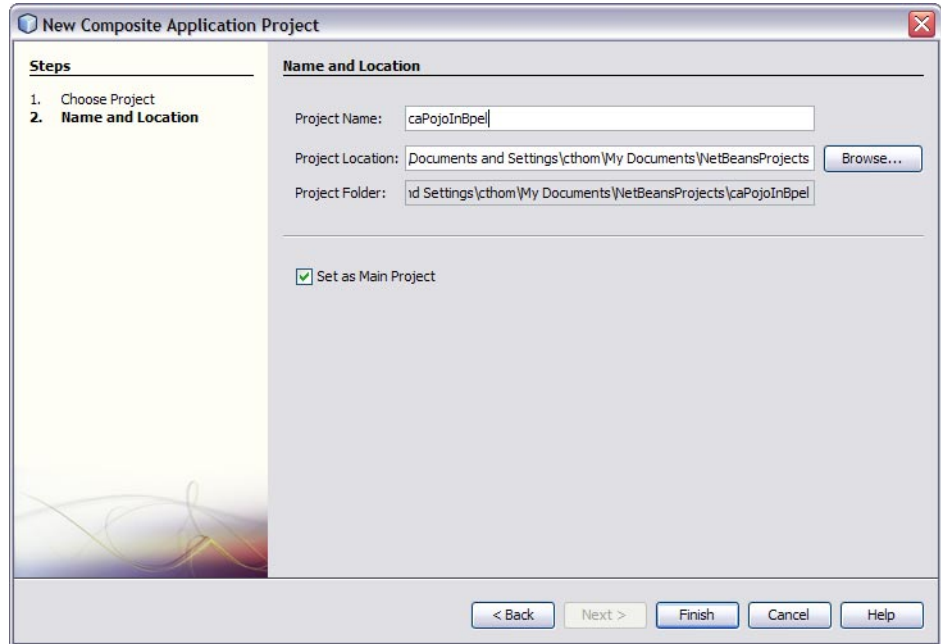
- 2 Under Categories, select SOA; under Projects, select Composite Application.



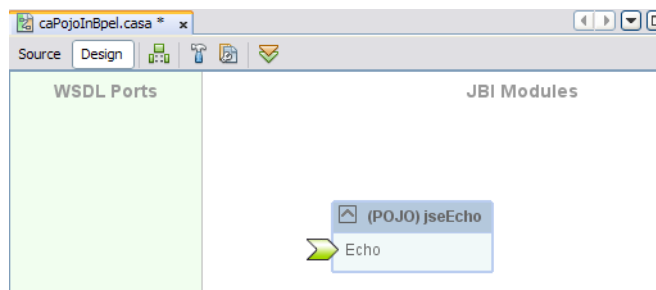
- 3 Click Next.

The Name and Location window appears.

- 4 Enter a name for the Composite Application and modify the location of the project files if necessary.



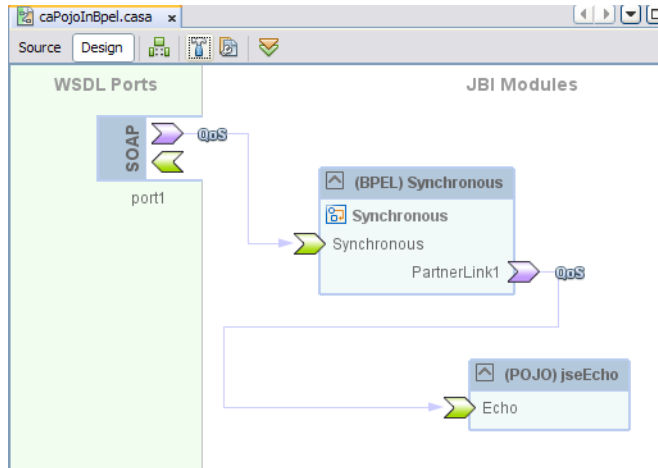
- 5 Click Finish.
The new Composite Application appears in the CASA Editor.
- 6 Drag the POJO Service Engine project from the Projects window to the JBI Modules section of the CASA Editor.
- 7 In the CASA Editor toolbar, click Build Project.
The service, along with the endpoint, appears in the CASA Editor.



- 8 Drag the BPEL project from the Projects window to the JBI Modules section of the CASA Editor.

9 In the CASA Editor toolbar, click Build Project.

The business process, endpoints, ports, and connections appear in the CASA Editor.



10 To deploy the application, do the following:

- a. Make sure the GlassFish server is running.
- b. In the Projects window, right-click the Composite Application project and then select Deploy.

Note – If the POJO Service Engine is not started, deploying the Composite Application will automatically start it for you.

Calling a Business Process From a POJO Service

This procedure provides general instructions for calling a business process from a POJO service. You can view a tutorial with more detailed information for completing this process at <http://wiki.open-esb.java.net/Wiki.jsp?page=POJOJBISamplePOJO2BPEL>.

▼ To Call a Business Process From a POJO Service

Before You Begin This task requires that the business process being called is already created. For information on creating a business process, see *BPEL Designer and Service Engine User's Guide*.

1 Create the business process with any necessary activities and WSDL documents.

For more information about creating business processes, see the *BPEL Designer and Service Engine User's Guide*.

- 2 **Create a new Java Application project for a POJO consumer, as described under “Creating a POJO Service Consumer (Without Binding)” on page 23 or “Creating a POJO Service Consumer (With Binding)” on page 25.**

Standard code is automatically generated for the consumer.

- 3 **Modify the code between the Consumer Invoke comments to handle the input from the business process.**

Below is a simple example of adding a response string to the message received from the business process.

```
@Operation (outMessageTypeQN="{http://jseecho/Echo/}EchoOperationResponse")
public String receive(String input) {
    /* Consumer Invoke - Begin */
    {
        String inputMessage = null;
        try {
            String outputMsg = (String) sepEchoInterfaceEchoOperation.
                sendSynchInOut(inputMessage, org.glassfish.opensb.pojo.se.
                    api.Consumer.MessageObjectType.String);
            return "Hello from POJO: " + outputMsg;
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }
    /* Consumer Invoke - End */
    return "Hello from POJO: " + input;
}

// Logger
private static final Logger logger = Logger.getLogger(Echo.class.getName());
// POJO Context
@Resource
private Context jbiCtx;
@ConsumerEndpoint(serviceQN =
    "{http://jseecho/Echo/}epEchoInterfaceEchoOperatioService",
    interfaceQN = "{http://jseecho/Echo/}EchoInterface",
    name = "epEchoInterfaceEchoOperatio",
    operationQN = "{http://jseecho/Echo/}EchoOperation",
    inMessageTypeQN = "{http://jseecho/Echo/}EchoOperationRequest")
private Consumer sepEchoInterfaceEchoOperation;
```

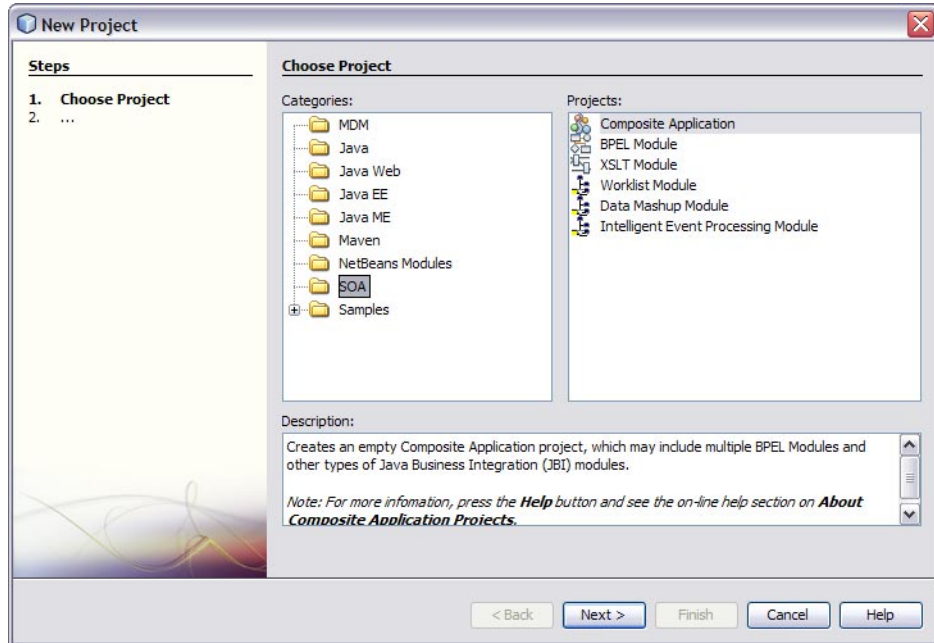
- 4 **When you are done modifying the code, click Save on the NetBeans toolbar.**

▼ To Create the Composite Application

- 1 Right-click in the Projects window, and then select New Project.

The New Project Wizard appears.

- 2 Under Categories, select SOA; under Projects, select Composite Application.



- 3 Click Next.

The Name and Location window appears.

- 4 Enter a name for the Composite Application and modify the location of the project files if necessary.

- 5 Click Finish.

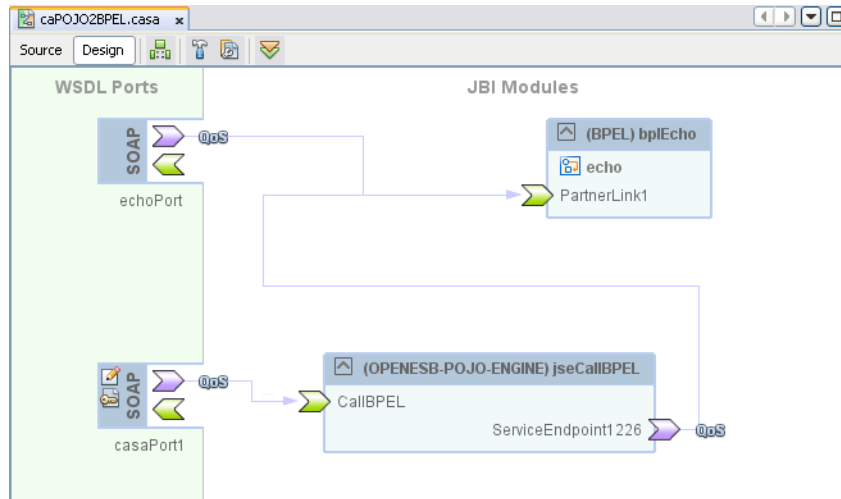
The new Composite Application appears in the CASA Editor.

- 6 Drag the POJO Service Engine project from the Projects window to the JBI Modules section of the CASA Editor.

- 7 Drag the BPEL project from the Projects window to the JBI Modules section of the CASA Editor.

8 In the CASA Editor toolbar, click Build Project.

The business process, POJO service, endpoints, ports, and connections appear in the CASA Editor.



9 To deploy the application, do the following:

- a. Make sure the GlassFish server is running.
- b. In the Projects window, right-click the Composite Application project and then select Deploy.

Note – If the POJO Service Engine is not started, deploying the Composite Application will automatically start it for you.

Configuring Runtime Properties for the POJO Service Engine

The POJO SE Properties Editor allows you to view information about the service engine, configure threading properties, view statistics on the runtime components, and set log levels for various POJO SE components.

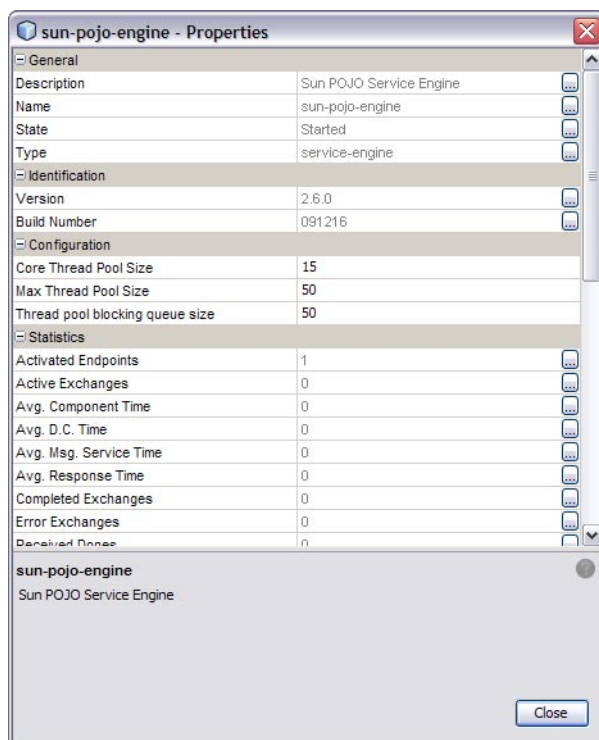
The following topics provide instructions for configuring the runtime properties and a reference of the available properties:

- [“To Configure POJO SE Runtime Properties” on page 52](#)
- [“POJO Service Engine Runtime Property Descriptions” on page 53](#)

▼ To Configure POJO SE Runtime Properties

- 1 From the Services window of the NetBeans IDE, expand the Servers node.
- 2 If the application server is not already started, right-click the server and then select Start.
- 3 Under the application server, expand JBI and expand Service Engines.
- 4 If the POJO SE is not started, right-click sun-pojo-engine and then select Start.
- 5 Right-click sun-pojo-engine and then select Properties.

The Properties Editor appears.



- 6 Modify any of the properties listed in [“POJO Service Engine Runtime Property Descriptions” on page 53](#).

Note – Statistic properties are automatically updated by the POJO SE. You do not need to modify these properties.

7 To apply the changes, stop and restart the POJO SE.

POJO Service Engine Runtime Property Descriptions

The following table lists and describes each POJO Service Engine runtime property.

TABLE 5 POJO SE General Runtime Properties

Property	Description
Description	A general description of the JBI component.
Name	A unique name for the POJO SE in the JBI environment. If you install more than one POJO Service Engine in a JBI environment, make sure that each has a unique name. When the service unit deploys the component, it is matched with the target component name defined in its descriptor file, <code>jbi.xml</code> , which can be modified as needed.
State	The current state of the JBI component. This value can be either Started, Stopped, or Shutdown.
Type	The type of JBI component (service-engine or binding-component).

TABLE 6 POJO SE Identification Runtime Properties

Property	Description
Version	The version number of the installed service engine.
Build Number	The build number of the installed service engine.

TABLE 7 POJO SE Configuration Runtime Properties

Property	Description	Default Value
Core Thread Pool Size	The number of core threads in the thread pool executor for processing inbound messages.	15
Max Thread Pool Size	The maximum number of threads in the thread pool executor for processing inbound messages.	50
Thread Pool Blocking Queue Size	The number of thread for the blocking queue.	50

TABLE 8 POJO SE Runtime Statistics

Property	Description
Activated Endpoints	The number of activated endpoints.

TABLE 8 POJO SE Runtime Statistics *(Continued)*

Property	Description
Active Exchanges	The number of active exchanges.
Avg. Component Time	The average message exchange component time in milliseconds.
Avg. D.C. Time	The average message exchange delivery channel time in milliseconds.
Avg. Msg. Service Time	The average message exchange message service time in milliseconds.
Avg. Response Time	The average message exchange response time in milliseconds.
Completed Exchanges	The total number of completed exchanges.
Error Exchanges	The total number of error exchanges.
Received Dones	The total number of received dones.
Received Errors	The total number of received errors.
Received Faults	The total number of received faults.
Received Replies	The total number of received replies.
Received Requests	The total number of received requests.
Sent Dones	The total number of sent dones.
Sent Errors	The total number of sent errors.
Sent Faults	The total number of sent faults.
Sent Replies	The total number of sent replies.
Sent Requests	The total number of sent requests.
Up Time	The up time of this component in milliseconds.

The Loggers properties specify the level of logging for each event. You can set the logging level for each logger to any of the following levels:

- **FINEST**: provides highly detailed tracing
- **FINER**: provides more detailed tracing
- **FINE**: provides basic tracing
- **CONFIG**: provides static configuration messages
- **INFO**: provides informative messages
- **WARNING**: messages indicate a warning
- **SEVERE**: messages indicate a severe failure
- **OFF**: no logging messages

By default, these are all set to the INFO level.

TABLE 9 POJO SE Logger Runtime Properties

Property	POJO Component
sun-pojo-engine	org.glassfish.openesb.pojose
DeploymentLookup	org.glassfish.openesb.pojose.com.sun.jbi.common.qos.descriptor. DeploymentLookup
MessagingChannel	org.glassfish.openesb.pojose.com.sun.jbi.common.qos.messaging. MessagingChannel
PojoSE Annotation Processor	org.glassfish.openesb.pojose.core.anno.processor. POJOAnnotationProcessor
PojoSE Util	org.glassfish.openesb.pojose.core.util.Util
PojoSE Bootstrap	org.glassfish.openesb.pojose.jbi.PojoSEBootstrap
PojoSE Component Manager	org.glassfish.openesb.pojose.jbi.PojoSEComponentManager
PojoSEConfiguration	org.glassfish.openesb.pojose.jbi.PojoSEConfiguration
PojoSE Life Cycle	org.glassfish.openesb.pojose.jbi.PojoSELifeCycle
PojoSE Service Unit Manager	org.glassfish.openesb.pojose.jbi.PojoSEServiceUnitManager
PojoSE Executor	org.glassfish.openesb.pojose.jbi.nmr.BasePojoExecutor
PojoSE Service Unit	org.glassfish.openesb.pojose.jbi.su.PojoSEServiceUnit
PojoSE Inbound Processor	org.glassfish.openesb.pojose.jbi.thread.InboundProcessor

POJO Service Engine API Annotation and Classes

The POJO Service Engine provides a very simple API for defining consumers and providers using POJO. Some of the objects are annotated for ease of use. The complete POJO Service Engine API Javadoc is at <https://open-esb.dev.java.net/nonav/pojose/javadoc/>.

The following sections list and describe the POJO annotations and classes, along with the relevant JBI classes:

- “POJO Service Engine API Annotations” on page 56
- “POJO Service Engine Non-Annotated Classes” on page 57
- “JBI API Classes Relevant to the POJO Service Engine” on page 57

POJO Service Engine API Annotations

The annotations listed and described below are provided in the POJO Service Engine API to simplify the necessary coding.

- **Provider** - `org.glassfish.openesb.pojose.api.annotation.Provider`
A class-level annotation that designates a Java class as a POJO service.
- **Operation** - `org.glassfish.openesb.pojose.api.annotation.Operation`
A method-level annotation that designates an operation as a POJO service method.
- **ConsumerEndpoint** -
`org.glassfish.openesb.pojose.api.annotation.ConsumerEndpoint`
A field-level annotation that designates fields of the type
`org.glassfish.openesb.pojose.api.Consumer`. The POJO SE injects this instance before calling the POJO operation method.
- **Resource** - `org.glassfish.openesb.pojose.api.annotation.Resource`
A field-level annotation that designates field of the type
`org.glassfish.openesb.pojose.api.res.Context`. The POJO SE injects the instance before calling the POJO operation method.
- **OnReply** - `org.glassfish.openesb.pojose.api.annotation.OnReply`
A method-level annotation that designates a method to be invoked when the reply message from the asynchronously called JBI service is received.

Note – Fault and error messages are only handled by methods annotated with `OnFault` and `OnError` and will not get routed to methods annotated with `OnReply`. If you do not have `OnFault` or `OnError` annotated methods, faults or error messages are ignored after they are logged at the Severe level.

- **OnDone** - `org.glassfish.openesb.pojose.api.annotation.OnDone`
A method-level annotation that designates a method to be invoked when all the outstanding reply messages from asynchronously called JBI services are received.
- **OnError** - `org.glassfish.openesb.pojose.api.annotation.OnError`
A method-level annotation that designates a method to be invoked when an asynchronous JBI services call results in a JBI error status. See the note above for information about using `OnReply`, `OnError`, and `OnFault`.
- **OnFault** - `org.glassfish.openesb.pojose.api.annotation.OnFault`
A method-level annotation that designates a method to be invoked when an asynchronous JBI services call results in a JBI fault status.

POJO Service Engine Non-Annotated Classes

The classes listed and described below define the non-annotated objects of the POJO Service engine API.

- **Context** - `org.glassfish.openesb.pojose.api.res.Context`

When a field is annotated with `@Resource` (`org.glassfish.openesb.pojose.api.annotation.Resource`), the POJO SE inserts an instance of `Context` before invoking the POJO instance method. A `Context` instance can be used to look up JNDI resources, look up service endpoints, retrieve an instance of `Consumer`, access a `MessageExchange` object, and create a new `MessageExchange` object.

- **Consumer** - `org.glassfish.openesb.pojose.api.Consumer`

When a field is annotated with `@ConsumerEndpoint` (`org.glassfish.openesb.pojose.api.annotation.ConsumerEndpoint`), the POJO SE inserts an instance of `Consumer` before invoking the POJO instance method. A `Consumer` instance can be used to consume JBI services both synchronously and asynchronously. `Consumer.sendSynchInOnly()` and `sendSynchInOut` methods are used to send messages in synchronous mode, and `sendInOnly()` and `sendInOut` are used to send and receive messages asynchronously. A `Consumer` instance can also be used to create a `MessageExchanges` object specific to the `ServiceEndpoint` that annotates the field.

- **ErrorMessage** - `org.glassfish.openesb.pojose.api.ErrorMessage`

A POJO class can throw an `ErrorMessage` object from its `@Operation` method to send the JBI error status to the caller of the service.

Note – You need to specifically throw a `FaultMessage` exception to return a fault message; otherwise, throw an `ErrorMessage` exception to return the `MessageExchange` with the error status and exception. If there is a runtime exception, the `MessageExchange` is returned with a status of "error" and an exception.

- **FaultMessage** - `org.glassfish.openesb.pojose.api.FaultMessage`

A POJO class can throw a `FaultMessage` object from its `@Operation` method to send the JBI fault status to the caller of the service. See the above note for information about when to use `ErrorMessage` and when to use `FaultMessage`.

JBI API Classes Relevant to the POJO Service Engine

The following classes from the JBI API are used with the POJO Service Engine:

- `NormalizedMessage` (`javax.jbi.messaging.NormalizedMessage`)
- `MessageExchange` (`javax.jbi.messaging.MessageExchange`)
- `ServiceEndpoint` (`javax.jbi.servicedesc.ServiceEndpoint`)

