



## Migration Guide





## Migration Guide

**Note**

Before using this information and the product it supports, read the information in Notices at the end of this book.

**Fourth Edition (December 2006)**

**© Copyright International Business Machines Corporation 2006. All rights reserved.**

US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

---

## Contents

### **Migrating applications using WebSphere Integration Developer . . . . . 1**

Migrating to WebSphere Integration Developer . . . . .	1
Migrating to WebSphere Process Server from WebSphere InterChange Server . . . . .	1
Migrating to WebSphere Integration Developer from WebSphere MQ Workflow . . . . .	35

Migrating source artifacts to WebSphere Integration Developer from WebSphere Studio Application Developer Integration Edition . . . . .	42
---	----

### **Notices . . . . . 105**



---

# Migrating applications using WebSphere Integration Developer

WebSphere® Integration Developer Version 6.0 provides the necessary tools to migrate your existing environment.

The following topics describe concept, reference, and step-by-step instructions for migrating to WebSphere Integration Developer:

---

## Migrating to WebSphere Integration Developer

WebSphere Integration Developer Version 6.0 provides the necessary tools to migrate your existing environment.

**Note:** WebSphere Integration Developer 6.0.2 projects *cannot* be used in WebSphere Integration Developer 6.0.1. Once you upgrade to WebSphere Integration Developer 6.0.2, you cannot take projects back to WebSphere Integration Developer 6.0.1.x. Support is also not available if you are a 6.0.2 user checking your code into a repository/exporting projects and then sharing them with a WebSphere Integration Developer 6.0.1 user.

The following topics describe concepts, reference information, and step-by-step instructions for migrating to WebSphere Integration Developer:

## Migrating to WebSphere Process Server from WebSphere InterChange Server

Migration from WebSphere InterChange Server to WebSphere Process Server is supported through the following functions in WebSphere Integration Developer.

**Note:** Refer to the release notes for information concerning limitations related to migration in this release of WebSphere Process Server.

- Automatic migration of source artifacts through migration tools that can be invoked from the following:
  - The **File** → **Import** menu of WebSphere Integration Developer
  - The Welcome Screen of WebSphere Integration Developer
  - The **reposMigrate** command Line utility
- Native support in the runtime of many WebSphere InterChange Server API's
- Support for the current WebSphere Business Integration Adapter technology so that existing adapters will be compatible with the WebSphere Process Server

Even though migration of source artifacts is supported, it is recommended that extensive analysis and testing be done to determine if the resulting applications will function as expected in WebSphere Process Server, or if they will need post-migration redesign. This recommendation is based on the following limitations in functional parity between WebSphere InterChange Server and this version of WebSphere Process Server. There is no support in this version of WebSphere Process Server that is equivalent to these WebSphere InterChange Server functions:

- Group Support
- Hot Deployment/Dynamic Update
- Scheduler - Pause Operation
- Security - Auditing

- Security - Fine Grain RBAC
- Security Descriptors are not migrated

### **Supported migration paths for WebSphere InterChange Server**

WebSphere Process Server migration tools support migration from WebSphere InterChange Server versions 4.2.2, 4.2.3 and 4.3.

Any WebSphere InterChange Server release prior to Version 4.2.2 will first need to migrate to version 4.2.2, 4.2.3 or 4.3 *before* migrating to WebSphere Process Server.

### **Preparing for migration from WebSphere InterChange Server**

Before migrating to WebSphere Process Server from WebSphere InterChange Server, you must first ensure that you have properly prepared your environment. WebSphere Process Server provides the necessary tools to migrate from WebSphere InterChange Server.

These migration tools can be invoked from:

- The **File** → **Import** menu of WebSphere Integration Developer
- The Welcome Screen of WebSphere Integration Developer

Input to the migration tools is a repository jar file exported from WebSphere InterChange Server. Therefore, before accessing the migration tools through any of these options, you must first:

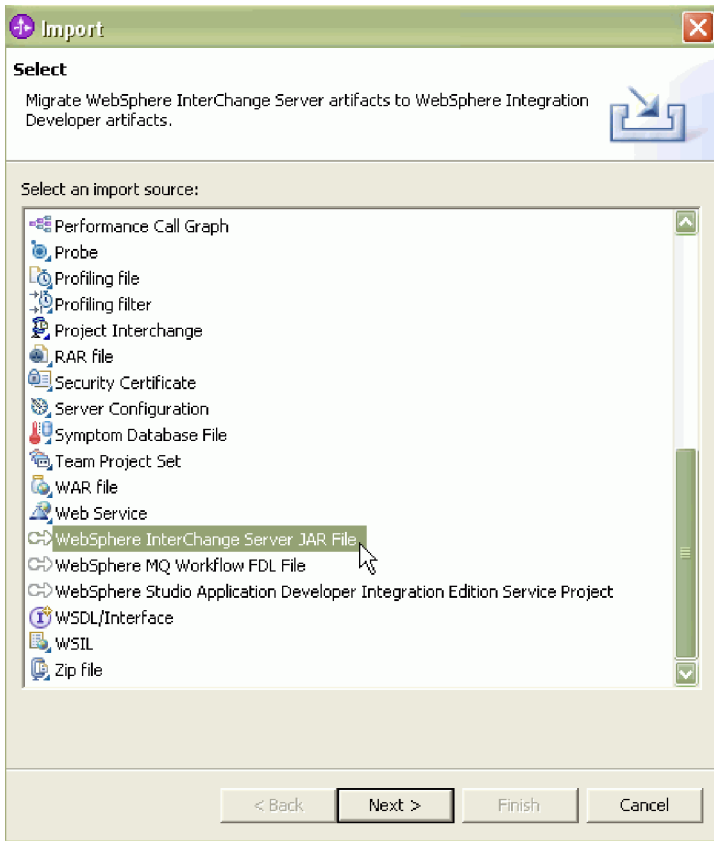
1. ensure that you are running a version WebSphere InterChange Server that can be migrated to WebSphere Process Server. See the topic "Supported migration paths for WebSphere InterChange Server".
2. export your source artifacts from WebSphere InterChange Server into a repository jar file using the WebSphere InterChange Server **repos\_copy** command as described in the documentation for WebSphere InterChange Server. This jar file will be input to the migration tools.

### **Migrating WebSphere InterChange Server using the Migration wizard**

You can use the WebSphere Integration Developer Migration wizard to migrate your existing WebSphere InterChange Server artifacts.

To use the Migration wizard to migrate your WebSphere InterChange Server artifacts, follow these steps:

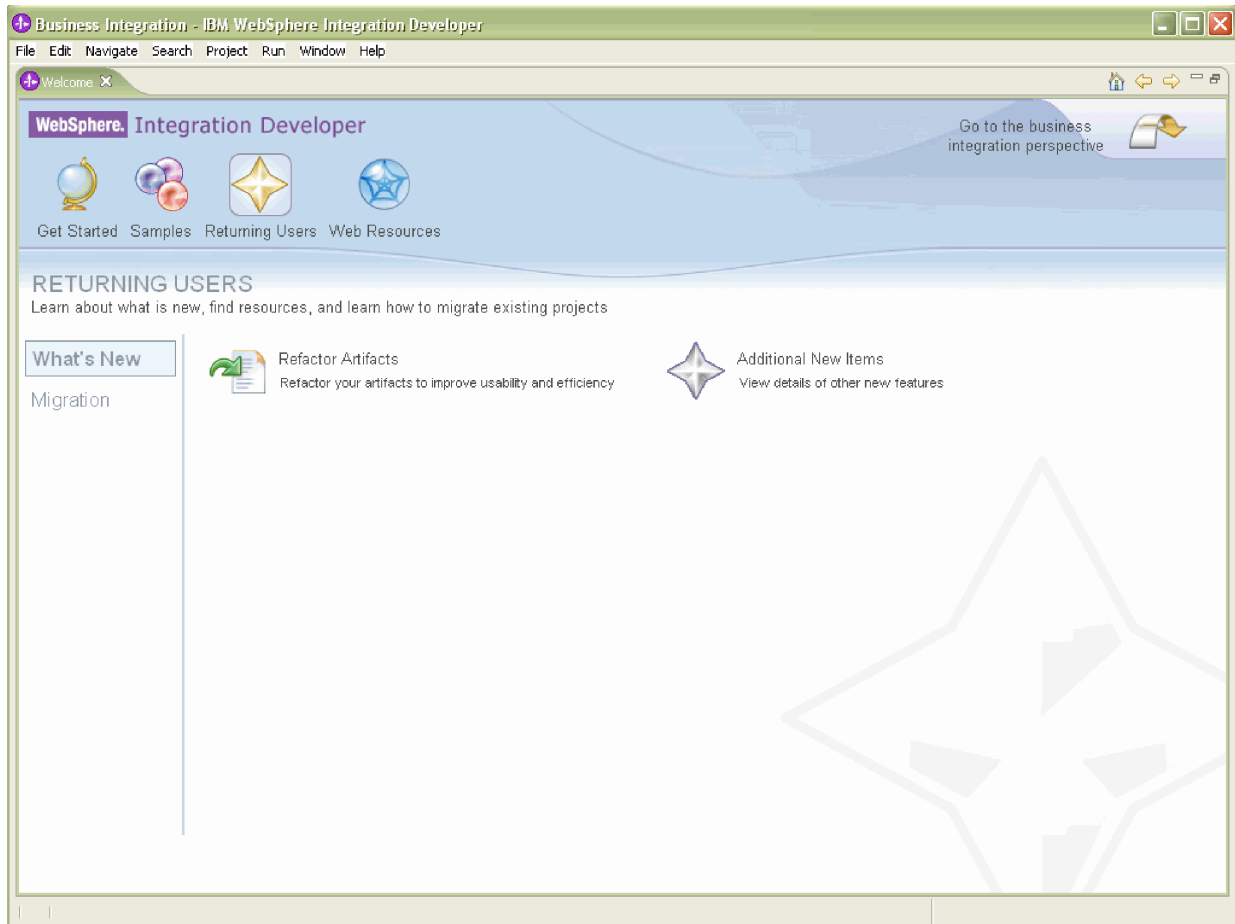
1. Invoke the wizard by selecting **File** → **Import** → **WebSphere InterChange Server JAR File**:



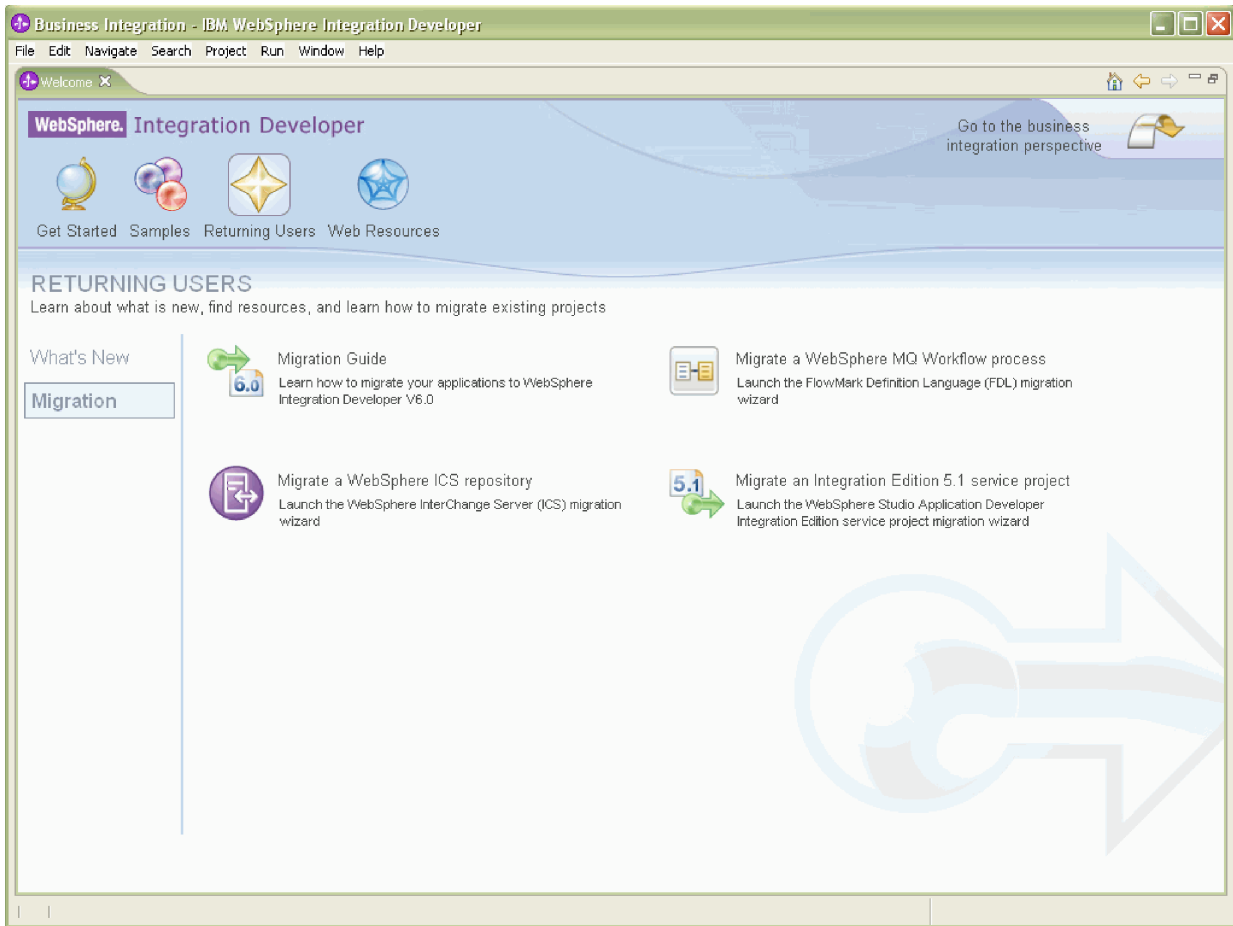
OR you can also open the Migration wizard from the Welcome page by clicking on the **Returning**



**Users** icon **Returning Users** to open the Returning Users page (Note that you can always return to the Welcome page by clicking on **Help** → **Welcome** ):



Click on **Migration** on the left side of the Returning Users page to open the Migration page:



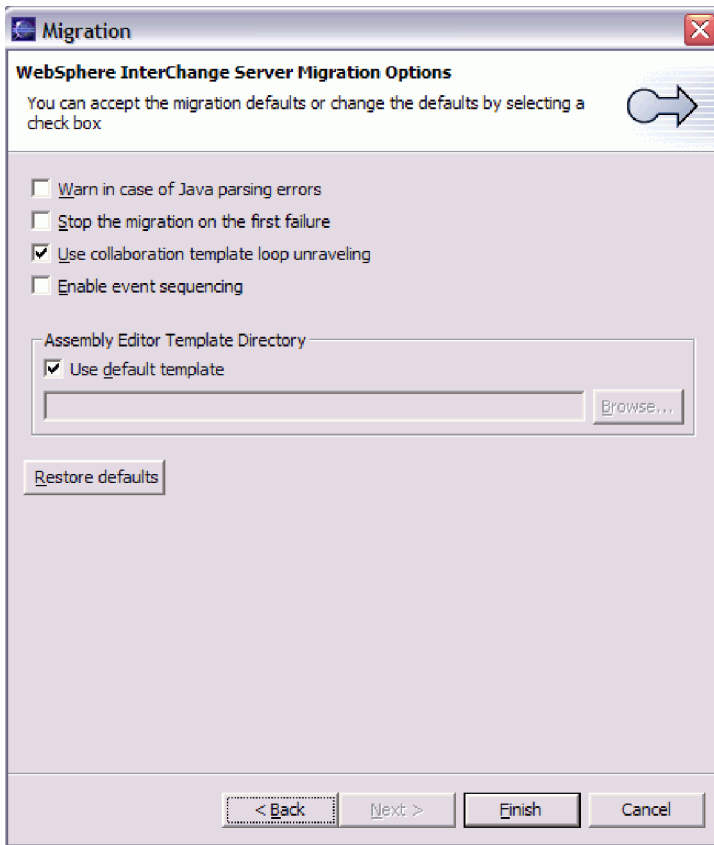
From the Migration page, select the **Migrate a WebSphere ICS repository** option



2. The Migration wizard opens. Enter the name of the source file in the **Source selection** field by clicking the **Browse** button and navigating to the file. Enter the library name in the relevant field. Click **Next**.



3. The Migration Options window opens. From here you can accept the migration defaults or select a check box to change the option.



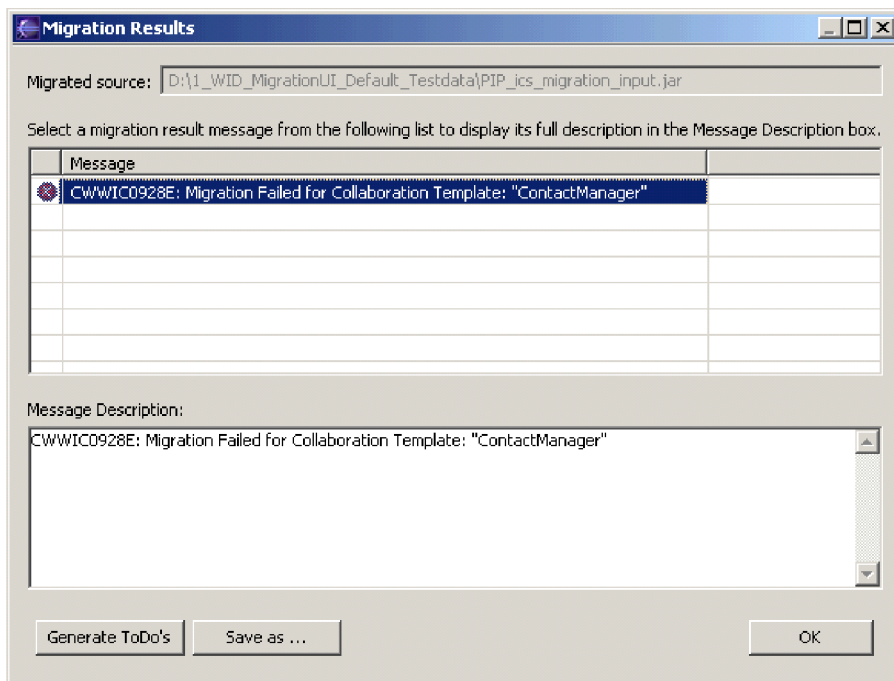
Click Finish.

### Verifying the WebSphere InterChange Server migration

If no errors are reported during the migration of the WebSphere InterChange Server jar file, then the migration of the artifacts was successful. If the migration has not completed successfully, a list of errors, warnings, and/or informational messages will be displayed. You can use these messages to verify the WebSphere InterChange Server migration.

**Note:** Due to the complexity of migrating from WebSphere InterChange Server to WebSphere Process Server, you are strongly urged to perform extensive testing of the resulting applications running in WebSphere Process Server to ensure they function as expected, before putting them into production.

The following page appears if migration messages were generated during the migration process:



In the Migration Results window, you can see the migration messages that were generated during the migration process. By selecting a message from the upper Message list, you can find more information regarding that message in the lower Message Description window. To keep all messages for future reference, click the **Generate ToDo's** button to create a list of "ToDo" tasks in the task view and/or click the **Save as...** button to save the messages in a text file in the filesystem.

## Working with migration failures from WebSphere InterChange Server

If your migration from WebSphere InterChange Server fails, there are a two ways in which to deal with the failures.

**Note:** You may prefer the first option as you will, initially, be more familiar with WebSphere InterChange Server. However, as you become more experienced with WebSphere Process Server and its new artifacts, you may choose to repair the migrated artifacts in WebSphere Integration Developer.

1. If the nature of the error permits, you can adjust the WebSphere InterChange Server artifacts using the WebSphere InterChange Server toolset, and export the jar file again and retry the migration.
2. You can fix any errors in the resulting WebSphere Process Server artifacts by editing the artifacts in WebSphere Integration Developer.

## WebSphere InterChange Server artifacts handled by the migration tools

The migration tools can automatically migrate some of the WebSphere InterChange Server artifacts.

The following artifacts can be migrated:

- **Business objects** become WebSphere Process Server business objects
- **Maps** become WebSphere Process Server maps
- **Relationships** become WebSphere Process Server relationships and roles
- **Collaboration templates** become WebSphere Process Server BPEL and WSDL
- **Collaboration objects** become WebSphere Process Server SCA artifacts
- **Connector definitions** become WebSphere Process Server SCA artifacts

The migration tools will create a Jython script that can be used with the **wsadmin** command line tool to configure resources in the WebSphere Process Server for the following WebSphere InterChange Server artifacts/resources:

- DBConnection Pools
- Relationship Databases
- Scheduler entries

The migration tools do *not* handle the following WebSphere InterChange Server artifacts:

- Benchmark artifacts

## Supported WebSphere InterChange Server APIs

In addition to the WebSphere InterChange Server source artifact migration tools provided in WebSphere Process Server and WebSphere Integration Developer, there is also support for many of the APIs that were provided in WebSphere InterChange Server. The migration tools work in conjunction with these WebSphere InterChange Server APIs by preserving your custom snippet code as much as possible when migrating.

**Note:** These APIs are provided only to support migrated WebSphere InterChange Server applications until they can be modified to use new Process Server APIs. These WebSphere InterChange Server APIs are all deprecated and are planned to be removed in a future release.

The supported WebSphere InterChange Server APIs in Process Server are listed below. These APIs provide functions in Process Server similar to the function that they provide in WebSphere InterChange Server. See the WebSphere InterChange Server documentation for a functional description of these APIs.

### CwBiDiEngine

#### AppSide\_Connector/

- BiDiBOTransformation(BusinessObject, String, String, boolean):BusinessObj
- BiDiBusObjTransformation(BusObj, String, String, boolean):BusObj
- BiDiStringTransformation(String, String, String):String

### JavaConnectorUtil

#### AppSide\_Connector/

- INFRASTRUCTURE\_MESSAGE\_FILE
- CONNECTOR\_MESSAGE\_FILE
- XRD\_WARNING
- XRD\_TRACE
- XRD\_INFO
- XRD\_ERROR
- XRD\_FATAL
- LEVEL1
- LEVEL2
- LEVEL3
- LEVEL4
- LEVEL5
- createBusinessObject(String):BusinesObjectInterface
- createBusinessObject(String, Locale):BusinesObjectInterface
- createBusinessObject(String, String):BusinesObjectInterface
- createContainer(String):CxObjectContainerInterface
- generateMsg(int, int, int, int, int, Vector):String
- generateMsg(int, int, int, int, Vector):String
- getBlankValue():String
- getEncoding():String

- getIgnoreValue():String
- getLocale():String
- isBlankValue(Object):boolean
- isIgnoreValue(Object):boolean
- isTraceEnabled(int):boolean
- logMsg(String)
- logMsg(String, int)
- traceWrite(int, String)

## **BusObj**

### **Collaboration/**

- BusObj(DataObject)
- BusObj(String)
- BusObj(String, Locale)
- copy(BusObj)
- duplicate():BusObj
- equalKeys(BusObj):boolean
- equals(Object):boolean
- equalsShallow(BusObj):boolean
- exists(String):boolean
- get(int):Object
- get(String):Object
- getBoolean(String):boolean
- getBusObj(String):BusObj
- getBusObjArray(String):BusObjArray
- getCount(String):int
- getDouble(String):double
- getFloat(String):float
- getInt(String):int
- getKeys():String
- getLocale():java.util.Locale
- getLong(String):long
- getLongText(String):String
- getString(String):String
- getType():String
- getValues():String
- getVerb():String
- isBlank(String):boolean
- isKey(String):boolean
- isNull(String):boolean
- isRequired(String):boolean
- keysToString():String
- set(BusObj)
- set(int, Object)
- set(String, boolean)

- set(String, double)
- set(String, float)
- set(String, int)
- set(String, long)
- set(String, Object)
- set(String, String)
- setContent(BusObj)
- setDefaultAttrValues()
- setKeys(BusObj)
- setLocale(java.util.Locale)
- setVerb(String)
- setVerbWithCreate(String, String)
- setWithCreate(String, boolean)
- setWithCreate(String, BusObj)
- setWithCreate(String, BusObjArray)
- setWithCreate(String, double)
- setWithCreate(String, float)
- setWithCreate(String, int)
- setWithCreate(String, long):
- setWithCreate(String, Object)
- setWithCreate(String, String)
- toString():String
- validData(String, boolean):boolean
- validData(String, BusObj):boolean
- validData(String, BusObjArray):boolean
- validData(String, double):boolean
- validData(String, float):boolean
- validData(String, int):boolean
- validData(String, long):boolean
- validData(String, Object):boolean
- validData(String, String):boolean

## **BusObjArray**

### **Collaboration/**

- addElement(BusObj)
- duplicate():BusObjArray
- elementAt(int):BusObj
- equals(BusObjArray):boolean
- getElements():BusObj[]
- getLastIndex():int
- max(String):String
- maxBusObjArray(String):BusObjArray
- maxBusObjs(String):BusObj[]
- min(String):String
- minBusObjArray(String):BusObjArray

- minBusObjs(String):BusObj[]
- removeAllElements()
- removeElement(BusObj)
- removeElementAt(int)
- setElementAt(int, BusObj)
- size():int
- sum(String):double
- swap(int, int)
- toString():String

## **BaseDLM**

### **DLM/**

- BaseDLM(BaseMap)
- getDBConnection(String):CwDBConnection
- getDBConnection(String, boolean):CwDBConnection
- getName():String
- getRelConnection(String):DtpConnection
- implicitDBTransactionBracketing():boolean
- isTraceEnabled(int):boolean
- logError(int)
- logError(int, Object[])
- logError(int, String)
- logError(int, String, String)
- logError(int, String, String, String)
- logError(int, String, String, String, String)
- logError(int, String, String, String, String, String)
- logError(String)
- logInfo(int)
- logInfo(int, Object[])
- logInfo(int, String)
- logInfo(int, String, String)
- logInfo(int, String, String, String)
- logInfo(int, String, String, String, String)
- logInfo(int, String, String, String, String, String)
- logInfo(String)
- logWarning(int)
- logWarning(int, Object[])
- logWarning(int, String)
- logWarning(int, String, String)
- logWarning(int, String, String, String)
- logWarning(int, String, String, String, String)
- logWarning(int, String, String, String, String, String)
- logWarning(String)
- raiseException(RuntimeEntityException)
- raiseException(String, int)

- raiseException(String, int, Object[])
- raiseException(String, int, String)
- raiseException(String, int, String, String)
- raiseException(String, int, String, String, String)
- raiseException(String, int, String, String, String, String)
- raiseException(String, int, String, String, String, String, String)
- raiseException(String, String)
- releaseRelConnection(boolean)
- trace(int, int)
- trace(int, int, Object[])
- trace(int, int, String)
- trace(int, int, String, String)
- trace(int, int, String, String, String)
- trace(int, int, String, String, String, String)
- trace(int, int, String, String, String, String, String)
- trace(int, String)
- trace(String)

#### **CwDBConnection**

##### **CwDBConnection/**

##### **CxCommon/**

- beginTransaction()
- commit()
- executePreparedSQL(String)
- executePreparedSQL(String, Vector)
- executeSQL(String)
- executeSQL(String, Vector)
- executeStoredProcedure(String, Vector)
- getUpdateCount():int
- hasMoreRows():boolean
- inTransaction():boolean
- isActive():boolean
- nextRow():Vector
- release()
- rollback()

#### **CwDBConstants**

##### **CwDBConnection/**

##### **CxCommon/**

- PARAM\_IN - 0
- PARAM\_INOUT - 1
- PARAM\_OUT - 2

#### **CwDBStoredProcedureParam**

##### **CwDBConnection/**

##### **CxCommon/**

- CwDBStoredProcedureParam(int, Array)

- CwDBStoredProcedureParam(int, BigDecimal)
- CwDBStoredProcedureParam(int, boolean)
- CwDBStoredProcedureParam(int, Boolean)
- CwDBStoredProcedureParam(int, byte[])
- CwDBStoredProcedureParam(int, double)
- CwDBStoredProcedureParam(int, Double)
- CwDBStoredProcedureParam(int, float)
- CwDBStoredProcedureParam(int, Float)
- CwDBStoredProcedureParam(int, int)
- CwDBStoredProcedureParam(int, Integer)
- CwDBStoredProcedureParam(int, java.sql.Blob)
- CwDBStoredProcedureParam(int, java.sql.Clob)
- CwDBStoredProcedureParam(int, java.sql.Date)
- CwDBStoredProcedureParam(int, java.sql.Struct)
- CwDBStoredProcedureParam(int, java.sql.Time)
- CwDBStoredProcedureParam(int, java.sql.Timestamp)
- CwDBStoredProcedureParam(int, Long)
- CwDBStoredProcedureParam(int, String)
- CwDBStoredProcedureParam(int, String, Object)
- getParamType():int getValue():Object

### **DataHandler (Abstract Class)**

**DataHandlers/  
crossworlds/  
com/**

- createHandler(String, String, String):DataHandler
- getBO(InputStream, Object):BusinessObjectInterface
- getBO(Object, BusinessObjectInterface, Object)
- getBO(Object, Object):BusinessObjectInterface
- getBO(Reader, BusinessObjectInterface, Object) (Abstract Method)
- getBO(Reader, Object):BusinessObjectInterface (Abstract Method)
- getBO(String, Object):BusinessObjectInterface
- getBOName(InputStream):String
- getBOName(Reader):String
- getBOName(String):String
- getBooleanOption(String):boolean
- getEncoding():String
- getLocale():Locale
- getOption(String):String
- getStreamFromBO(BusinessObjectInterface, Object):InputStream (Abstract Method)
- getStringFromBO(BusinessObjectInterface, Object):String (Abstract Method)
- setConfigMOName(String)
- setEncoding(String)
- setLocale(Locale)
- setOption(String, String)
- traceWrite(String, int)

## **NameHandler (Abstract Class)**

**DataHandlers/  
crossworlds/  
com/**

- `getBOName(Reader, String):String` (Abstract Method)

## **ConfigurationException** (extends `java.lang.Exception`)

**Exceptions/  
DataHandlers/  
crossworlds/  
com/**

## **MalformedDataException** (extends `java.lang.Exception`)

**Exceptions/  
DataHandlers/  
crossworlds/  
com/**

## **NotImplementedException** (extends `java.lang.Exception`)

**Exceptions/  
DataHandlers/  
crossworlds/  
com/**

## **BusinessObjectInterface**

**CxCommon/**

- `clone():Object`
- `dump():String`
- `getAppText():String`
- `getAttrCount():int`
- `getAttrDesc(int):CxObjectAttr`
- `getAttrDesc(String):CxObjectAttr`
- `getAttribute(String):Object`
- `getAttributeIndex(String):int`
- `getAttributeType(int):int`
- `getAttributeType(String):int`
- `getAttrName(int):String`
- `getAttrValue(int):Object`
- `getAttrValue(String):Object`
- `getBusinessObjectVersion():String`
- `getDefaultAttrValue(int):String`
- `getDefaultAttrValue(String):String`
- `getLocale():String`
- `getName():String`
- `getParentBusinessObject():BusinessObjectInterface`
- `getVerb():String`
- `getVerbAppText(String):String`
- `isBlank(int):boolean`
- `isBlank(String):boolean`
- `isIgnore(int):boolean`

- `isIgnore(String):boolean`
- `isVerbSupported(String):boolean`
- `makeNewAttrObject(int):Object`
- `makeNewAttrObject(String):Object`
- `setAttributeWithCreate(String, Object)`
- `setAttrValue(int, Object)`
- `setAttrValue(String, Object)`
- `setDefaultAttrValues()`
- `setLocale(Locale)`
- `setLocale(String)`
- `setVerb(String)`

### **CXObjectAttr**

#### **CxCommon/**

- `BOOLEAN`
- `BOOLSTRING`
- `DATE`
- `DATESTRING`
- `DOUBLE`
- `DOUBSTRING`
- `FLOAT`
- `FLTSTRING`
- `INTEGER`
- `INTSTRING`
- `INVALID_TYPE_NUM`
- `INVALID_TYPE_STRING`
- `LONGTEXT`
- `LONGTEXTSTRING`
- `MULTIPLECARDSTRING`
- `OBJECT`
- `SINGLECARDSTRING`
- `STRING`
- `STRSTRING`
- `equals(Object):boolean`
- `getAppText():String`
- `getCardinality():String`
- `getDefault():String`
- `getMaxLength():int`
- `getName():String`
- `getRelationType():String`
- `getTypeName():String`
- `getTypeNum():String`
- `hasCardinality(String):boolean`
- `hasName(String):boolean`
- `hasType(String):boolean`

- isForeignKeyAttr():boolean
- isKeyAttr():boolean
- isMultipleCard():boolean
- isObjectType():boolean
- isRequiredAttr():boolean
- isType(Object):boolean

### **CxObjectContainerInterface**

#### **CxCommon/**

- getBusinessObject(int):BusinessObjectInterface
- getObjectCount():int
- insertBusinessObject(BusinessObjectInterface)
- removeAllObjects()
- removeBusinessObjectAt(int)
- setBusinessObject(int, BusinessObjectInterface)

### **DtpConnection**

#### **Dtp/**

#### **CxCommon/**

- beginTran()
- commit()
- executeSQL(String)
- executeSQL(String, Vector)
- executeStoredProcedure(String, Vector)
- getUpdateCount():int
- hasMoreRows():boolean
- inTransaction():boolean
- isActive():boolean
- nextRow():Vector
- rollback()

### **DtpDataConversion**

#### **Dtp/**

#### **CxCommon/**

- BOOL\_TYPE - 4
- CANNOTCONVERT - 2
- DATE\_TYPE - 5
- DOUBLE\_TYPE - 3
- FLOAT\_TYPE - 2
- INTEGER\_TYPE - 0
- LONGTEXT\_TYPE - 6
- OKTOCONVERT - 0
- POTENTIALDATALOSS - 1
- STRING\_TYPE - 1
- UNKNOWN\_TYPE - 999
- getType(double):int
- getType(float):int

- getType(int):int
- getType(Object):int
- isOKToConvert(int, int):int
- isOKToConvert(String, String):int
- toBoolean(boolean):Boolean
- toBoolean(Object):Boolean
- toDouble(double):Double
- toDouble(float):Double
- toDouble(int):Double
- toDouble(Object):Double
- toFloat(double):Float
- toFloat(float):Float
- toFloat(int):Float
- toFloat(Object):Float
- toInteger(double):Integer
- toInteger(float):Integer
- toInteger(int):Integer
- toInteger(Object):Integer
- toPrimitiveBoolean(Object):boolean
- toPrimitiveDouble(float):double
- toPrimitiveDouble(int):double
- toPrimitiveDouble(Object):double
- toPrimitiveFloat(double):float
- toPrimitiveFloat(int):float
- toPrimitiveFloat(Object):float
- toPrimitiveInt(double):int
- toPrimitiveInt(float):int
- toPrimitiveInt(Object):int
- toString(double):String
- toString(float):String
- toString(int):String
- toString(Object):String

## **DtpDate**

### **Dtp/**

#### **CxCommon/**

- DtpDate()
- DtpDate(long, boolean)
- DtpDate(String, String)
- DtpDate(String, String, String[], String[])
- addDays(int):DtpDate
- addMonths(int):DtpDate
- addWeekdays(int):DtpDate
- addYears(int):DtpDate
- after(DtpDate):boolean
- before(DtpDate):boolean

- calcDays(DtpDate):int
- calcWeekdays(DtpDate):int
- get12MonthNames():String[]
- get12ShortMonthNames():String[]
- get7DayNames():String[]
- getCWDate():String
- getDayOfMonth():String
- getDayOfWeek():String
- getHours():String
- getIntDay():int
- getIntDayOfWeek():int
- getIntHours():int
- getIntMilliseconds():int
- getIntMinutes():int
- getIntMonth():int
- getIntSeconds():int
- getIntYear():int
- getMaxDate(BusObjArray, String, String):DtpDate
- getMaxDateBO(BusObj[], String, String):BusObj[]
- getMaxDateBO(BusObjArray, String, String):BusObj[]
- getMinDate(BusObjArray, String, String):DtpDate
- getMinDateBO(BusObj[], String, String):BusObj[]
- getMinDateBO(BusObjArray, String, String):BusObj[]
- getMinutes():String
- getMonth():String
- getMSSince1970():long
- getNumericMonth():String
- getSeconds():String
- getShortMonth():String
- getYear():String
- set12MonthNames(String[], boolean)
- set12MonthNamesToDefault()
- set12ShortMonthNames(String[])
- set12ShortMonthNamesToDefault()
- set7DayNames(String[])
- set7DayNamesToDefault()
- toString():String
- toString(String):String
- toString(String, boolean):String

### **DtpMapService**

#### **Dtp/**

#### **CxCommon/**

- runMap(String, String, BusObj[], CxExecutionContext):BusObj[]

## **DtpSplitString**

**Dtp/**

**CxCommon/**

- DtpSplitString(String, String)
- elementAt(int):String
- firstElement():String
- getElementCount():int
- getEnumeration():Enumeration
- lastElement():String
- nextElement():String
- prevElement():String
- reset()

## **DtpUtils**

**Dtp/**

**CxCommon/**

- padLeft(String, char, int):String
- padRight(String, char, int):String
- stringReplace(String, String, String):String
- truncate(double):int
- truncate(double, int):double
- truncate(float):int
- truncate(float, int):double
- truncate(Object):int
- truncate(Object, int):double

## **BusObjInvalidVerbException** (extends InterchangeExceptions)

**Exceptions/**

**CxCommon/**

- getFormattedMessage()

## **IdentityRelationship**

**relationship/**

**utilities/**

**crossworlds/**

**com/**

- addMyChildren(String, String, BusObj, String, Object, CxExecutionContext)
- deleteMyChildren(String, String, BusObj, String, CxExecutionContext)
- deleteMyChildren(String, String, BusObj, String, Object, CxExecutionContext)
- foreignKeyLookup(String, String, BusObj, String, BusObj, String, CxExecutionContext)
- foreignKeyXref(String, String, String, BusObj, String, BusObj, String, CxExecutionContext)
- maintainChildVerb(String, String, String, BusObj, String, BusObj, String, CxExecutionContext, boolean, boolean)
- maintainCompositeRelationship(String, String, BusObj, Object, CxExecutionContext)
- maintainSimpleIdentityRelationship(String, String, BusObj, BusObj, CxExecutionContext)
- updateMyChildren(String, String, BusObj, String, String, String, String, CxExecutionContext)

## **MapExeContext**

### **Dtp/**

#### **CxCommon/**

- ACCESS\_REQUEST - "SUBSCRIPTION\_DELIVERY"
- ACCESS\_RESPONSE - "ACCESS\_RETURN\_REQUEST"
- EVENT\_DELIVERY - "SUBSCRIPTION\_DELIVERY"
- SERVICE\_CALL\_FAILURE - "CONSUME\_FAILED"
- SERVICE\_CALL\_REQUEST - "CONSUME"
- SERVICE\_CALL\_RESPONSE - "DELIVERBUSOBJ"
- getConnName():String
- getGenericBO():BusObj
- getInitiator():String
- getLocale():java.util.Locale
- getOriginalRequestBO():BusObj
- setConnName(String)
- setInitiator(String)
- setLocale(java.util.Locale)

## **Participant**

### **RelationshipServices/**

#### **Server/**

- Participant(String, String, int, BusObj)
- Participant(String, String, int, String)
- Participant(String, String, int, long)
- Participant(String, String, int, int)
- Participant(String, String, int, double)
- Participant(String, String, int, float)
- Participant(String, String, int, boolean)
- Participant(String, String, BusObj)
- Participant(String, String, String)
- Participant(String, String, long)
- Participant(String, String, int)
- Participant(String, String, double)
- Participant(String, String, float)
- Participant(String, String, boolean)
- getBoolean():boolean
- getBusObj():BusObj
- getDouble():double
- getFloat():float
- getInstanceId():int
- getInt():int
- getLong():long
- getParticipantDefinition():String
- getRelationshipDefinition():String
- getString():String INVALID\_INSTANCE\_ID
- set(boolean)

- set(BusObj)
- set(double)
- set(float)
- set(int)
- set(long)
- set(String)
- setInstanceId(int)
- setParticipantDefinition(String)
- setRelationshipDefinition(String)
- setParticipantDefinition(String)
- setRelationshipDefinition(String)

## Relationship

### RelationshipServices/

#### Server/

- addMyChildren(String, String, BusObj, String, Object, CxExecutionContext)
- addParticipant(Participant):int
- addParticipant(String, String, boolean):int
- addParticipant(String, String, BusObj):int
- addParticipant(String, String, double):int
- addParticipant(String, String, float):int
- addParticipant(String, String, int):int
- addParticipant(String, String, int, boolean):int
- addParticipant(String, String, int, BusObj):int
- addParticipant(String, String, int, double):int
- addParticipant(String, String, int, float):int
- addParticipant(String, String, int, int):int
- addParticipant(String, String, int, long):int
- addParticipant(String, String, int, String):int
- addParticipant(String, String, long):int
- addParticipant(String, String, String):int
- create(Participant):int
- create(String, String, boolean):int
- create(String, String, BusObj):int
- create(String, String, double):int
- create(String, String, float):int
- create(String, String, int):int
- create(String, String, long):int
- create(String, String, String):int
- deactivateParticipant(Participant)
- deactivateParticipant(String, String, boolean)
- deactivateParticipant(String, String, BusObj)
- deactivateParticipant(String, String, double)
- deactivateParticipant(String, String, float)
- deactivateParticipant(String, String, int)
- deactivateParticipant(String, String, long)

- deactivateParticipant(String, String, String)
- deactivateParticipantByInstance(String, String, int)
- deactivateParticipantByInstance(String, String, int, boolean)
- deactivateParticipantByInstance(String, String, int, BusObj)
- deactivateParticipantByInstance(String, String, int, double)
- deactivateParticipantByInstance(String, String, int, float)
- deactivateParticipantByInstance(String, String, int, int)
- deactivateParticipantByInstance(String, String, int, long)
- deactivateParticipantByInstance(String, String, int, String)
- deleteMyChildren(String, String, BusObj, String, CxExecutionContext)
- deleteMyChildren(String, String, BusObj, String, Object, CxExecutionContext)
- deleteParticipant(Participant)
- deleteParticipant(String, String, boolean)
- deleteParticipant(String, String, BusObj)
- deleteParticipant(String, String, double)
- deleteParticipant(String, String, float)
- deleteParticipant(String, String, int)
- deleteParticipant(String, String, long)
- deleteParticipant(String, String, String)
- deleteParticipantByInstance(String, String, int)
- deleteParticipantByInstance(String, String, int, boolean)
- deleteParticipantByInstance(String, String, int, BusObj)
- deleteParticipantByInstance(String, String, int, double)
- deleteParticipantByInstance(String, String, int, float)
- deleteParticipantByInstance(String, String, int, int)
- deleteParticipantByInstance(String, String, int, long)
- deleteParticipantByInstance(String, String, int, String)
- getNewID(String):int
- maintainCompositeRelationship(String, String, BusObj, Object, CxExecutionContext)
- maintainSimpleIdentityRelationship(String, String, BusObj, BusObj, CxExecutionContext)
- retrieveInstances(String, boolean):int[]
- retrieveInstances(String, BusObj):int[]
- retrieveInstances(String, double):int[]
- retrieveInstances(String, float):int[]
- retrieveInstances(String, int):int[]
- retrieveInstances(String, long):int[]
- retrieveInstances(String, String):int[]
- retrieveInstances(String, String, boolean):int[]
- retrieveInstances(String, String, BusObj):int[]
- retrieveInstances(String, String, double):int[]
- retrieveInstances(String, String, float):int[]
- retrieveInstances(String, String, int):int[]
- retrieveInstances(String, String, long):int[]
- retrieveInstances(String, String, String):int[]
- retrieveInstances(String, String[], boolean):int[]

- retrieveInstances(String, String[], BusObj):int[]
- retrieveInstances(String, String[], double):int[]
- retrieveInstances(String, String[], float):int[]
- retrieveInstances(String, String[], int):int[]
- retrieveInstances(String, String[], long):int[]
- retrieveInstances(String, String[], String):int[]
- retrieveParticipants(String, int):Participant[]
- retrieveParticipants(String, String, int):Participant[]
- retrieveParticipants(String, String[], int):Participant[]
- updateMyChildren(String, String, BusObj, String, String, String, String, CxExecutionContext)
- updateParticipant(String, String, BusObj)
- updateParticipantByInstance(Participant)
- updateParticipantByInstance(String, String, int)
- updateParticipantByInstance(String, String, int, BusObj)

### **UserStoredProcedureParam**

**Dtp/**

**CxCommon/**

- UserStoredProcedureParam(int, String, Object, String, String)
- getParamDataTypeJavaObj():String
- getParamDataTypeJDBC():int
- getParamIndex():int
- getParamIOType():String
- getParamName():String
- getParamValue():Object
- setParamDataTypeJavaObj(String)
- setParamDataTypeJDBC(int)
- setParamIndex(int)
- setParamIOType(String)
- setParamName(String)
- setParamValue(Object)
- PARAM\_TYPE\_IN - "IN"
- PARAM\_TYPE\_OUT - "OUT"
- PARAM\_TYPE\_INOUT - "INOUT"
- DATA\_TYPE\_STRING - "String"
- DATA\_TYPE\_INTEGER - "Integer"
- DATA\_TYPE\_DOUBLE - "Double"
- DATA\_TYPE\_FLOAT - "Float"
- DATA\_TYPE\_BOOLEAN - "Boolean"
- DATA\_TYPE\_TIME - "java.sql.Time"
- DATA\_TYPE\_DATE - "java.sql.Date"
- DATA\_TYPE\_TIMESTAMP - "java.sql.Timestamp"
- DATA\_TYPE\_BIG\_DECIMAL - "java.math.BigDecimal"
- DATA\_TYPE\_LONG\_INTEGER - "Long"
- DATA\_TYPE\_BINARY - "byte[]"
- DATA\_TYPE\_CLOB - "Clob"

- DATA\_TYPE\_BLOB - "Blob"
- DATA\_TYPE\_ARRAY - "Array"
- DATA\_TYPE\_STRUCT - "Struct"
- DATA\_TYPE\_REF - "Ref"

### **BaseCollaboration Collaboration/**

- BaseCollaboration(com.ibm.bpe.api.ProcessInstanceData)
- AnyException - "AnyException"
- AppBusObjDoesNotExist - "BusObjDoesNotExist"
- AppLogOnFailure - "AppLogOnFailure"
- AppMultipleHits - "AppMultipleHits"
- AppRequestNotYetSent - "AppRequestNotYetSent"
- AppRetrieveByContentFailed - "AppRetrieveByContent"
- AppTimeOut - "AppTimeOut"
- AppUnknown - "AppUnknown"
- AttributeException - "AttributeException"
- existsConfigProperty(String):boolean
- getConfigProperty(String):String
- getConfigPropertyArray(String):String[]
- getCurrentLoopIndex():int
- getDBConnection(String):CwDBConnection
- getDBConnection(String, boolean):CwDBConnection getLocale():java.util.Locale
- getMessage(int):String
- getMessage(int, Object[]):String
- getName():String
- implicitDBTransactionBracketing():boolean
- isCallerInRole(String):boolean
- isTraceEnabled(int):boolean
- JavaException - "JavaException"
- logError(int)
- logError(int, Object[])
- logError(int, String)
- logError(int, String, String)
- logError(int, String, String, String)
- logError(int, String, String, String, String)
- logError(int, String, String, String, String, String)
- logError(String)
- logInfo(int)
- logInfo(int, Object[])
- logInfo(int, String)
- logInfo(int, String, String)
- logInfo(int, String, String, String)
- logInfo(int, String, String, String, String)
- logInfo(int, String, String, String, String, String)

- logInfo(String)
- logWarning(int)
- logWarning(int, Object[])
- logWarning(int, String)
- logWarning(int, String, String)
- logWarning(int, String, String, String)
- logWarning(int, String, String, String, String)
- logWarning(int, String, String, String, String, String)
- logWarning(String)
- not(boolean):boolean ObjectException - "ObjectException"
- OperationException - "OperationException"
- raiseException(CollaborationException)
- raiseException(String, int)
- raiseException(String, int, Object[])
- raiseException(String, int, String)
- raiseException(String, int, String, String)
- raiseException(String, int, String, String, String)
- raiseException(String, int, String, String, String, String)
- raiseException(String, int, String, String, String, String, String)
- raiseException(String, String)
- ServiceCallException - "ConsumerException"
- ServiceCallTransportException - "ServiceCallTransportException"
- SystemException - "SystemException"
- trace(int, int)
- trace(int, int, Object[])
- trace(int, int, String)
- trace(int, int, String, String)
- trace(int, int, String, String, String)
- trace(int, int, String, String, String, String)
- trace(int, int, String, String, String, String, String)
- trace(int, String)
- trace(String)
- TransactionException - "TransactionException"

## **CxExecutionContext**

### **CxCommon/**

- CxExecutionContext()
- getContext(String):Object
- MAPCONTEXT - "MAPCONTEXT"
- setContext(String, Object)

## **CollaborationException**

### **Collaboration/**

- getMessage():String
- getMsgNumber():int
- getSubType():String

- getText():String
- getType():String
- toString():String

## **Filter**

### **crossworlds/**

#### **com/**

- Filter(BaseCollaboration)
- filterExcludes(String, String):boolean
- filterIncludes(String, String):boolean
- recurseFilter(BusObj, String, boolean, String, String):boolean
- recursePreReqs(String, Vector):int

## **Globals**

### **crossworlds/**

#### **com/**

- Globals(BaseCollaboration)
- callMap(String, BusObj):BusObj

## **SmartCollabService**

### **crossworlds/**

#### **com/**

- SmartCollabService()
- SmartCollabService(BaseCollaboration)
- doAgg(BusObj, String, String, String):BusObj
- doMergeHash(Vector, String, String):Vector
- doRecursiveAgg(BusObj, String, String, String):BusObj
- doRecursiveSplit(BusObj, String):Vector
- doRecursiveSplit(BusObj, String, boolean):Vector
- getKeyValues(BusObj, String):String
- merge(Vector, String):BusObj
- merge(Vector, String, BusObj):BusObj
- split(BusObj, String):Vector

## **StateManagement**

### **crossworlds/**

#### **com/**

- StateManagement()
- beginTransaction()
- commit()
- deleteBO(String, String, String)
- deleteState(String, String, String, int)
- persistBO(String, String, String, String, BusObj)
- recoverBO(String, String, String):BusObj
- releaseDBConnection()
- resetData()
- retrieveState(String, String, String, int):int
- saveState(String, String, String, String, int, int, double)

- setDBConnection(CwDBConnection)
- updateBO(String, String, String, String, BusObj)
- updateState(String, String, String, String, int, int)

#### **EventKeyAttrDef**

##### **EventManagement/**

##### **CxCommon/**

- EventKeyAttrDef()
- EventKeyAttrDef(String, String)
- public String keyName
- public String keyValue

#### **EventQueryDef**

##### **EventManagement/**

##### **CxCommon/**

- EventQueryDef()
- EventQueryDef(String, String, String, String, int)
- public String nameConnector
- public String nameCollaboration
- public String nameBusObj
- public String verb
- public int ownerType

#### **FailedEventInfo**

##### **EventManagement/**

##### **CxCommon/**

- FailedEventInfo()
- FailedEventInfo(String x6, int, EventKeyAttrDef[], int, int, String, String, int)
- public String nameOwner
- public String nameConnector
- public String nameBusObj
- public String nameVerb
- public String strTime
- public String strMessage
- public int wipIndex
- public EventKeyAttrDef[] strbusObjKeys
- public int nKeys
- public int eventStatus
- public String expirationTime
- public String scenarioName
- public int scenarioState

### **Mapping the WebSphere Process Sever DataObject from WebSphere InterChange Server XML**

If you use the Legacy Adapters to connect to WebSphere Process Server, the following algorithm will enable you to further understand how the WebSphere Process Sever DataObject was created from the WebSphere InterChange Server XML. This information shows where the data values have been placed, and also what data values have been chosen to replace the ones used in WebSphere InterChange Server.

## General

- For setting the verb in the ChangeSummary, all settings will be done with the **markCreate/Update/Delete** APIs.
- For setting the verb in the ChangeSummary/EventSummary, **Create**, **Update**, and **Delete** verbs will be set in the ChangeSummary, while all other verbs will be set in the EventSummary.
- For getting the verb from the ChangeSummary:
  - If isDelete is true, the verb will be **Delete**.

**Note:** The value of isCreate will be ignored if isDelete is true. This could happen if the creation was marked before the DataObject entered the hub at which point it was deleted, or if the creation and deletion both occurred in the hub.

- If isDelete is false and isCreate is true, the verb will be **Create**.
- If isDelete is false and isCreate is false, the verb will be **Update**.
- To avoid a DataObject being identified as **Create** instead of an intended **Update**, if logging is enabled, you must:
  - Suspend logging during the creation of the DataObject.
  - Resume logging for the update of the DataObject (or use the **markUpdated** API).

## Loading

Loading will load a WebSphere InterChange Server runtime XML into a WebSphere Business Integration BusinessGraph AfterImage instance.

- An instance of the appropriate BusinessGraph will be created.
- ChangeSummary Logging will be turned on, so that turning it on later will not clear the entries.
- ChangeSummary Logging will be paused to prevent unwanted information from entering the ChangeSummary.
- The attributes of the top level BusinessObject will be created in the DataObject (see the section "Attribute processing" below).
- If the top level BusinessObject has children BusinessObjects, these will be processed recursively.
- The attributes of these children BusinessObjects will be created in the DataObject (see the section "Attribute processing" below).
- The verb of the top level BusinessObject will be set to the top level verb of the BusinessGraph and set in the summaries.
- The verb of the children BusinessObjects will be set in the summaries.

## Saving

Saving will save a WebSphere Business Integration BusinessGraph AfterImage instance to a WebSphere InterChange Server runtime XML. An Exception will be thrown if the input BusinessGraph is not AfterImage.

- An instance of the WebSphere InterChange Server XML document will be created.
- All deleted DataObjects in the ChangeSummary will be treated as if they existed in their original positions.

**Note:** This undelete process will not preserve list order.

- The verb for the top level BusinessObject will be set to the top level verb of the BusinessGraph.
- The attributes of the top level BusinessObject will be set from the DataObject (see the section "Attribute processing" below).
- If the DataObject has children DataObjects, these will be processed recursively.
- The verb of the children DataObjects will be handled in the following manner:

- If there is no verb for the child DataObject in the EventSummary or ChangeSummary, the verb will be set to "" (empty string).
- If there is a verb present in the EventSummary, but not in the ChangeSummary, this verb will be used.
- If there is a verb present in the ChangeSummary, but not in the EventSummary, this verb will be used .
- If there is a verb present in both the ChangeSummary and EventSummary, it will be resolved in that the value in the ChangeSummary will be chosen over that in the EventSummary.
- The attributes of the children DataObjects will be set in the BusinessObject (see the section "Attribute processing" below).

### Attribute processing

- All values not covered below will be loaded/saved ASIS.
- ObjectEventId will be loaded into/saved from the EventSummary.
- For **CxBlank** and **CxIgnore**:
  - On the WebSphere Business Integration BusinessObject side of the conversion, **CxBlank** and **CxIgnore** will be set/identified as follows:
    - **CxIgnore** - unset or set with the Java™ value of null
    - **CxBlank** - type dependent value as shown in the table below
  - On the WebSphere InterChange Server XML side of the conversion, **CxBlank** and **CxIgnore** will be set/identified as follows:

Table 1. Setting CxBlank and CxIgnore

Type	CxIgnore	CxBlank
Int	Integer.MIN_VALUE	Integer.MAX_VALUE
Float	Float.MIN_VALUE	Float.MAX_VALUE
Double	Double.MIN_VALUE	Double.MAX_VALUE
String/date/longtext	"CxIgnore"	""
Children BusinessObjects	(empty element)	N/A

### Best practices for the WebSphere InterChange Server migration process

The following guidelines are intended to assist you in the development of integration artifacts for WebSphere InterChange Server. By adhering to these guidelines, you can ease the migration of WebSphere InterChange Server artifacts to WebSphere Process Server. These recommendations are meant to be used only as a guide for the development of new integration solutions. It is recognized that existing content may not adhere to these guidelines. It is also understood that there may be cases where it is necessary to deviate from these guidelines. In these cases care should be taken to limit the scope of the deviation to minimize the amount of rework required to migrate the artifacts. Note that the guidelines outlined here do not encompass best practices for the development of WebSphere InterChange Server artifacts in general. They are instead limited in scope to those considerations which may affect the ease in which artifacts can be migrated at a future time.

#### Best practice: General development:

There are several considerations which apply broadly to most of the integration artifacts. In general, the artifacts which leverage the facilities provided by the tooling and conform to the metadata models enforced by the tooling will migrate most smoothly. Also, artifacts with significant extensions and external dependencies are likely to require more manual intervention when migrating.

The following list summarizes the best practices for general development of WebSphere InterChange Server based solutions to help ease future migration:

- Use WebSphere InterChange Server for real-time, automated process integration solutions
- Document the system and component design
- Use the development tooling to edit integration artifacts
- Leverage best practices for defining rules with the tooling and Java snippets

Though it may seem obvious, it is important for integration solutions to adhere to the programming model and architecture provided by WebSphere InterChange Server. It is best suited to real-time, automated process integration solutions. Also, each of the integration components within WebSphere InterChange Server plays a well-defined role within the architecture. Significant deviations from this model will make it more challenging to migrate content to the appropriate artifacts on WebSphere Process Server.

Another general best practice which will greatly improve the success of future migration projects is to document the system design. Be sure to capture the integration architecture and design, including functional design and quality of service requirements, the interdependencies of artifacts shared across projects, and also the design decisions that were made during the deployment. This will assist in system analysis during migration, and will minimize any rework efforts.

For creating, configuring, and modifying artifact definitions, it is essential that only the development tooling provided is used. Avoid manual manipulation of artifact metadata (e.g. editing XML files directly), which may corrupt the artifact for migration.

When developing Java code within collaboration templates, maps, common code utilities, and other components, there are several considerations to take into account:

- Use only the published APIs
- Use Activity Editor
- Use adapters to access EISs
- Avoid external dependencies in Java snippet code
- Adhere to J2EE develop practices for portability
- Do not spawn threads or use thread synchronization primitives. If you must, these will need to be converted to use Asynchronous Beans when you migrate.
- Do not do any disk I/O using java.io.\* Use JDBC to store any data.
- Don't perform any functions that may be reserved for an EJB container such as socket I/O, classloading, loading native libraries, etc. If you must, these snippets would need manual conversion to use EJB container functions when you migrate.

Only use the APIs published in the product documentation for the artifacts. These are outlined in detail in the WebSphere InterChange Server development guides. While in many cases compatibility APIs will be provided in WebSphere Process Server, only the published APIs will be included. Although WebSphere InterChange Server has many internal interfaces which a developer might wish to use, there can be no assurance that these will be supported moving forward.

When designing business logic and transformation rules in maps and collaboration templates, be sure to use the Activity Editor tool to the greatest extent possible. This will ensure that the logic is described through metadata which can more readily be converted to the new artifacts. For operations that you wish to reuse in the tooling, use the "My Collections" feature of the Activity Editor wherever possible. Try to avoid field developed common code utility libraries, included as a Java archive (\*.jar) file in the classpath of WebSphere InterChange Server, as these will need to be migrated manually.

In any Java code snippets that may need to be developed, it is recommended that the code be as simple and atomic as possible. The level of sophistication in the Java code should be on the order of scripting,

involving basic evaluations, operations, and computations, data formatting, type conversions, etc. If more extensive or sophisticated application logic is required, consider using EJBs running in WebSphere Application Server to encapsulate the logic, and use web service calls to invoke it from WebSphere InterChange Server. Use standard JDK libraries rather than third party or external libraries which would need to be migrated separately. Also, collect all related logic within a single code snippet, and avoid using logic where connection and transaction contexts span multiple code snippets. With database operations, for example, code related to obtaining a connection, beginning and ending a transaction, and releasing the connection should be in one code snippet.

In general, ensure that code which is designed to interface with an Enterprise Information System (EIS) is placed within adapters, and not within maps or collaboration templates. This is generally a best practice for architecture design. Also, this will help avoid prerequisites for third party libraries and related considerations within the code, such as connection management and possible Java Native Interface (JNI) implementations.

Make the code as safe as possible by using appropriate exception handling. Also make the code compatible to run within a J2EE application server environment, even though it is currently running within a J2SE environment. Adhere to J2EE development practices, such as avoiding static variables, spawning threads, and disk I/O. These are excellent best practices to adhere to in general, but will be pertinent to portability.

**Best practice: Common code utilities:**

As noted earlier, avoiding the development of common code utility libraries for use across integration artifacts within the WebSphere InterChange Server environment is recommended. Where code reuse across integration artifacts is necessary, leveraging the “My Collections” feature of the Activity Editor tool is recommended. Also, consider using EJBs running in WebSphere Application Server to encapsulate the logic, and use web service calls to invoke them from WebSphere InterChange Server.

While it is possible that some common code utility libraries may execute appropriately on WebSphere Process Server, you will be responsible for the migration of the custom utilities.

**Best practice: Database connection pools:**

User-defined database connection pools are very useful within maps and collaboration templates for simple data lookups and for more sophisticated state management across process instances. A database connection pool in WebSphere InterChange Server will be rendered as a standard JDBC resource in WebSphere Process Server, and the basic function will be the same. The way connections and transactions are managed, however, may differ.

To maximize future portability, avoid keeping database transactions active across Java snippet nodes within a collaboration template or map. For example, code related to obtaining a connection, beginning and ending a transaction, and releasing the connection should be in one code snippet.

**Best practice: Business objects:**

The primary considerations for the development of business objects will be to use only the tooling provided to configure artifacts, to use explicit data types and lengths for data attributes, and to utilize only the documented APIs.

Business objects within WebSphere Process Server are based on Service Data Objects (SDOs), which utilize data attributes that are strongly typed. For business objects in WebSphere InterChange Server and adapters, data attributes are not strongly typed, and users sometimes specify string data types for non-string data attributes. To avoid issues in WebSphere Process Server, be explicit in the specification of data types.

Because business objects within WebSphere Process Server may be serialized at runtime as they are passed between components, it is important to be explicit with the required lengths for data attributes to minimize utilization of system resources. For this reason, do not use the maximum 255 character length for a string attribute, for example. Also, do not specify zero length attributes which currently default to 255 characters. Instead, specify the exact length required for attributes.

XSD NCName rules apply to business object attribute names in WebSphere Process Server, so, do not use any spaces or ":" in names for business object attributes. Business object attribute names with spaces or ":" are invalid in WebSphere Process Server. Rename business object attributes before migration.

If using an array in a business object, you can't rely on the order of the array when indexing into the array in Maps and/or Relationships. The construct that this migrates into in WebSphere Process Server, does not guarantee index order, particularly when entries are deleted.

Again, as stated earlier, it is important to use only the Business Object Designer tool to edit business object definitions, and to use only the published APIs for business objects within integration artifacts.

### **Best practice: Collaboration templates:**

Many of the guidelines we have described earlier apply to the development of collaboration templates.

To ensure processes are described appropriately with metadata, always use the Process Designer tool for the creation and modification of collaboration templates, and avoid editing the metadata files directly. Use the Activity Editor tool wherever possible to maximize the use of metadata to describe the required logic.

To minimize the amount of manual rework that may be required in migration, use only the documented APIs within collaboration templates. Avoid the use of static variables. Instead, use non-static variables and collaboration properties to address the requirements of the business logic. Avoid the use of Java qualifiers final, transient and native in Java snippets. These can not be enforced in the BPEL Java snippets that are the result of migrating the Collaboration Templates.

To maximize future portability, avoid using explicit connection release calls and explicit transaction bracketing (i.e. explicit commits & explicit rollbacks) for User Defined Database Connection Pools. Instead, make use of the container-managed implicit connection clean-up and implicit transaction bracketing. Also, avoid keeping system connections and transactions active across Java snippet nodes within a collaboration template. This applies to any connection to an external system, as well as user-defined database connection pools. As we described earlier, operations with an external EIS should be managed within an adapter, and code related to database operation should be contained within one code snippet. This may be necessary within a collaboration which, when rendered as a BPEL business process component may be selectively deployed as an interruptible flow. In this case, the process may be comprised of several separate transactions, with only state and global variable information passed between the activities. The context for any system connection or related transaction which spanned these process transactions would be lost.

Do not use special characters in Collaboration Template property names. These special characters are invalid in BPEL property names that they will be migrated into. Rename properties to remove these special characters before migrating to avoid syntactical errors in the BPEL generated by migration.

Do not reference variables using "this." For example, Instead of "this.inputBusObj" use just "inputBusObj"

Use class-level scoping on variables instead of scenario-scoped variables. Scenario-scoping is not carried forward during migration.

Initialize all variables declared in Java snippets with a default value. For example, "Object myObject = null;" Be sure all variables are initialized during declaration before migrating.

Ensure that there are no Java import statements in the user modifiable sections of your Collaboration Templates. In the definition of the Collaboration Template, use the import fields to specify Java packages to import.

### **Best practice: Maps:**

Many of the guidelines we have described for collaboration templates also apply to maps.

To ensure maps are described appropriately with metadata, always use the Map Designer tool for the creation and modification of maps, and avoid editing the metadata files directly. Use the Activity Editor tool wherever possible to maximize the use of metadata to describe the required logic.

When referencing child business object in a map, use a submap for the child business objects.

Avoid using Java code as the "value" in a SET since that is not valid in WebSphere Process Server. Use constants instead. For example, if the set value is "xml version=" + "1.0" + " encoding=" + "UTF-8" this will not validate in WebSphere Process Server. Instead, change it to "xml version=1.0 encoding=UTF-8" before you migrate.

To minimize the amount of manual rework that may be required in migration, use only the documented APIs within collaboration templates. Avoid the use of static variables. Instead, use non-static variables and collaboration properties to address the requirements of the business logic. Avoid the use of Java qualifiers final, transient and native in Java snippets.

If using an array in a business object, you can't rely on the order of the array when indexing into the array in Maps. The construct that this migrates into in WebSphere Process Server, does not guarantee index order, particularly when entries are deleted.

To maximize future portability, avoid using explicit connection release calls and explicit transaction bracketing (i.e. explicit commits & explicit rollbacks) for User Defined Database Connection Pools. Instead, make use of the container-managed implicit connection clean-up and implicit transaction bracketing. Also, avoid keeping system connections and transactions active in Java snippet code across transformation node boundaries. This applies to any connection to an external system, as well as user-defined database connection pools. As we described earlier, operations with an external EIS should be managed within an adapter, and code related to database operation should be contained within one code snippet.

### **Best practice: Relationships:**

For relationships, remember that while relationship definitions will be able to be migrated for use in WebSphere Process Server, the relationship table schema and instance data may be reused by WebSphere Process Server, and also shared concurrently between WebSphere InterChange Server and WebSphere Process Server.

The key considerations for relationships are to use only the tooling provided to configure the related components, and to use only the published APIs for relationships within integration artifacts.

It is important to use only the Relationship Designer tool to edit relationship definitions. Additionally, allow only WebSphere InterChange Server to configure the relationship schema, which is generated automatically upon deployment of relationship definitions. Do not alter the relationship table schema directly with database tools or SQL scripts.

Also, if you must manually modify relationship instance data within the relationship table schema, be sure to use the facilities provided by the Relationship Designer.

As we stated earlier, it is important to use only the published APIs for relationships within integration artifacts.

**Best practice: Access framework clients:**

Do not develop any new clients adopting the CORBA IDL interface APIs. This will not be supported in WebSphere Process Server.

## Migrating to WebSphere Integration Developer from WebSphere MQ Workflow

WebSphere Integration Developer provides the necessary tools to migrate from WebSphere MQ Workflow.

The Migration wizard enables you to convert FDL definitions of business processes that you exported from the build time component of WebSphere MQ Workflow into corresponding artifacts in WebSphere Integration Developer. The generated artifacts comprise XMLSchema definitions for business objects, WSDL definitions, BPEL, import and component definitions, and TEL definitions.

The conversion tool requires a semantically complete FDL definition of a process model that you export from WebSphere MQ Workflow build time with the option **export deep**. This option ensures that all necessary data, program, and subprocess specifications are included. Also, ensure that any user defined process execution server definitions (UPES) referenced in your WebSphere MQ Workflow process model is also selected when you export FDL from the WebSphere MQ Workflow build time.

**Note:** The Migration wizard does *not* cover the migration of:

- WebSphere MQ Workflow runtime instances
- Program applications that are invoked by a WebSphere MQ Workflow program execution agent (PEA) or WebSphere MQ Workflow process execution server (PES for z/OS<sup>®</sup>)

For more information on migrating using the **FDL2BPEL** conversion tool, see WebSphere MQ Workflow Support Site.

### Preparing for migration from WebSphere MQ Workflow

Before migrating to WebSphere Integration Developer from WebSphere MQ Workflow, you must first ensure that you have properly prepared your environment.

The scope and completeness of mapping depends on how far you adhere to the following guidelines for migration:

- Ensure that FDL program activities are associated to a user defined process execution server (UPES) if they are not pure **staff** activities.
- Ensure that staff assignments for WebSphere MQ Workflow program activities are compliant to TEL default **staff verbs**.
- Use short and simple names to improve the readability of migrated process models. Note that FDL names may be illegal BPEL names. The Migration wizard automatically converts FDL names to valid BPEL names.

The Migration wizard will produce syntactically correct business process editor constructs even for WebSphere MQ Workflow constructs that cannot be migrated (PEA or PES program activities, some dynamic staff assignments, and so on), which need to be manually adapted to executable business process editor artifacts.

The following table outlines the applied mapping rules:

Table 2. Mapping rules

WebSphere MQ Workflow	WebSphere Integration Developer
Process	<i>Process with execution mode: longRunning; Partner links for inbound and outbound interfaces of process</i>
Source and Sink	<i>Variables for process input and process output; Receive activity and reply activity</i>
Program activity	<i>Invoke activity</i>
Process activity	<i>Invoke activity</i>
Empty activity	<i>FMCINTERNALNOOP activity</i>
Block	<i>Scope with embedded BPEL activities</i>
Exit condition of activity	<i>While activity (enclosing the actual activity)</i>
Start condition of activity	<i>Join condition of activity</i>
Staff assignment of activity	<i>Human task activity</i>
Input container and output container of activity	<i>Variables used to specify the input/output of invoke activity</i>
Control connector; Transition condition	<i>Link; Transition condition</i>
Data connector	<i>Assign activity</i>
Global data container	<i>Variable</i>

**Note:** You should initially try the migration process with small projects, if possible. The Migration wizard will simplify the conversion of your WebSphere MQ Workflow process models into business process editor process models, but you should be aware that the processes cannot be mapped one-to-one as you are creating a new programming model. The semantic scopes of the underlying process specification languages (FDL and BPEL) share an area of intersection, but they do not overlap in total. Otherwise, you could not expect any new benefits from business process editor. Web services represent a promising new technology that claim replacing deprecated solutions by new ones.

In general, you should always review and possibly modify the generated artifacts. Additional effort may be necessary to either make a successful migration possible or to complete the migration task.

### Migrating WebSphere MQ Workflow using the Migration wizard

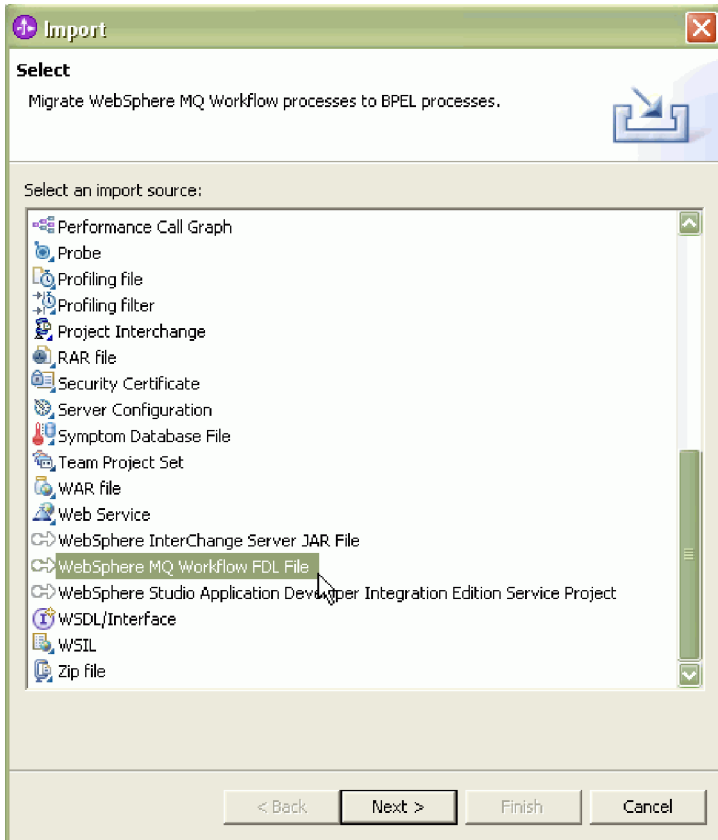
The Migration wizard enables you to convert FDL definitions of business processes that you exported from the build time component of WebSphere MQ Workflow into corresponding artifacts in WebSphere Integration Developer. The generated artifacts comprise XMLSchema definitions for business objects, WSDL definitions, BPEL, import and component definitions, and TEL definitions.

**Note:** The Migration wizard does *not* cover the migration of:

- WebSphere MQ Workflow runtime instances
- Program applications that are invoked by a WebSphere MQ Workflow program execution agent (PEA) or WebSphere MQ Workflow process execution server (PES for z/OS)

To use the Migration wizard to migrate your WebSphere MQ Workflow artifacts, follow these steps:

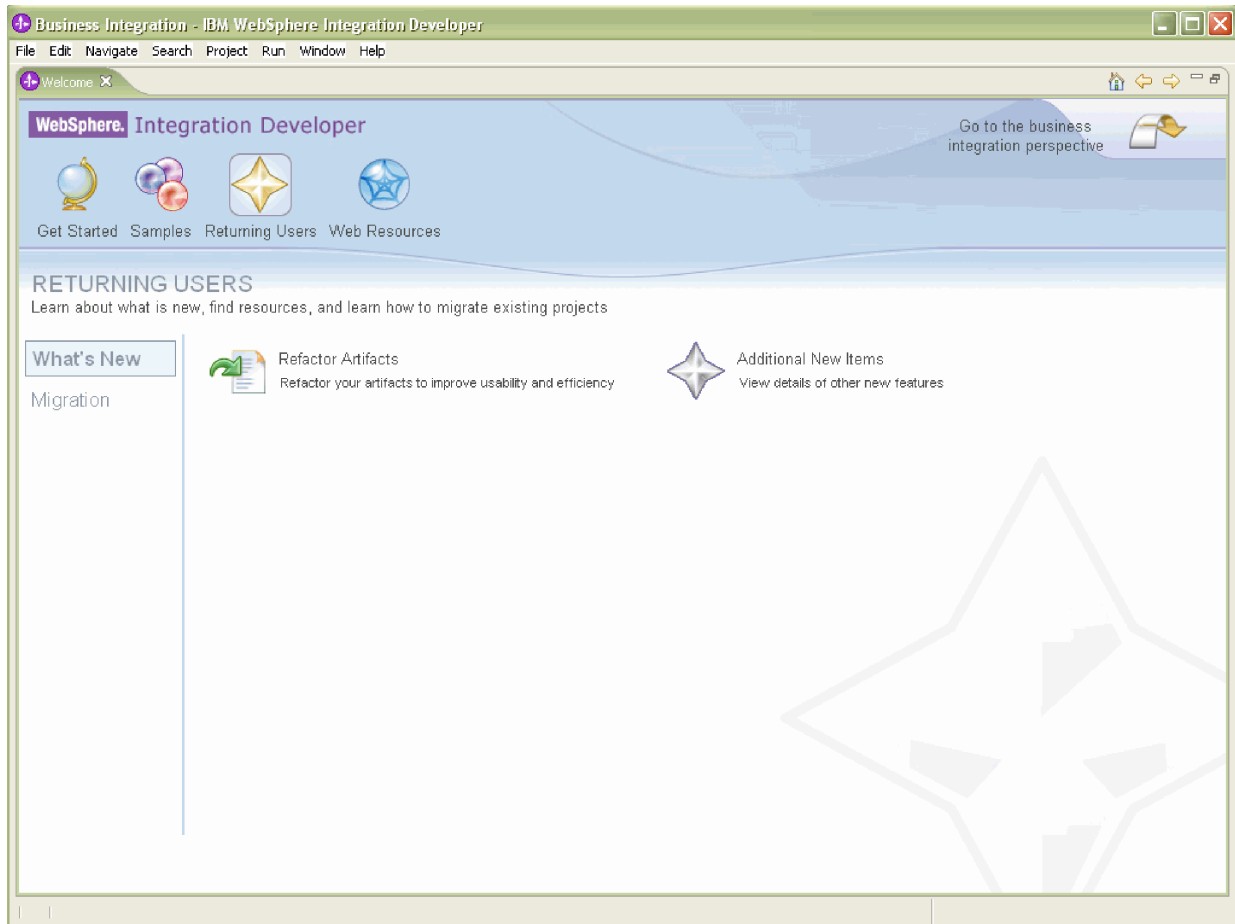
1. Invoke the wizard by selecting **File** → **Import** → **WebSphere MQ Workflow FDL File**:



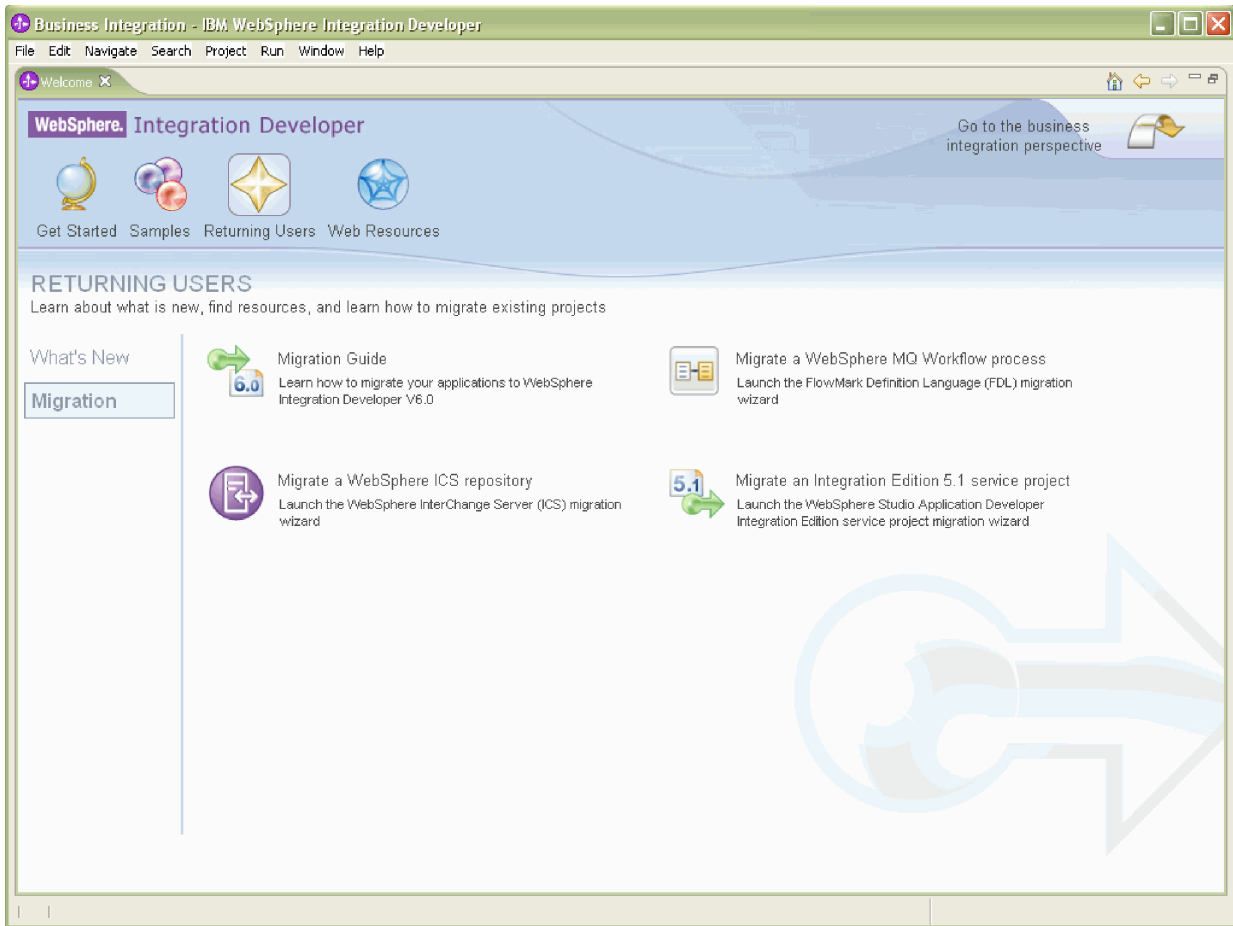
OR you can also open the Migration wizard from the Welcome page by clicking on the **Returning**



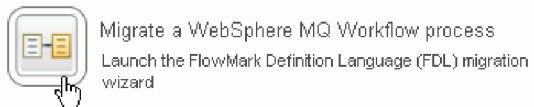
**Users** icon [Returning Users](#) to open the Returning Users page (Note that you can always return to the Welcome page by clicking on **Help** → **Welcome** ):



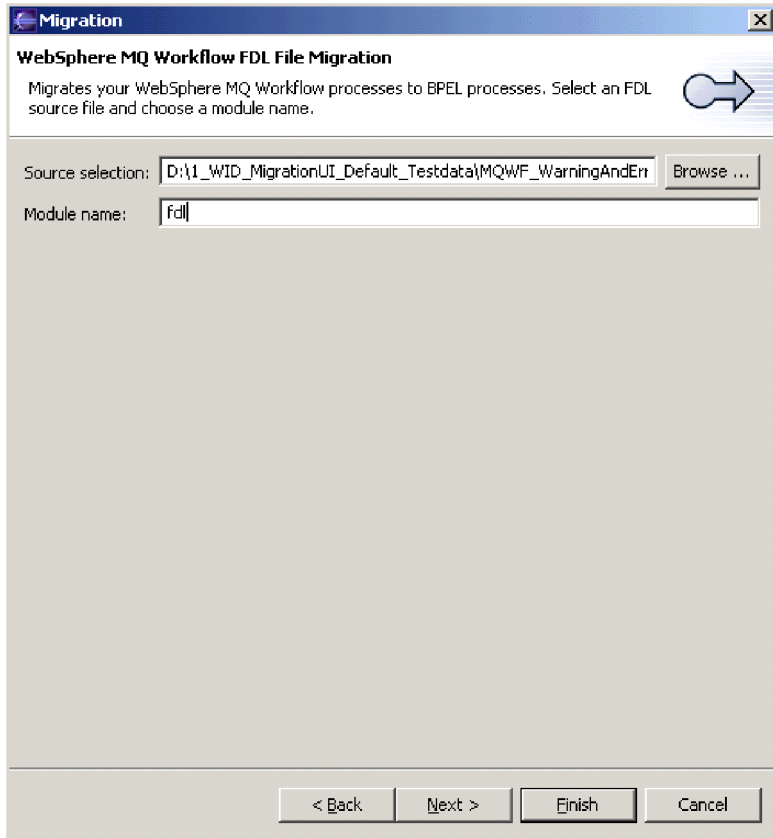
Click on **Migration** on the left side of the Returning Users page to open the Migration page:



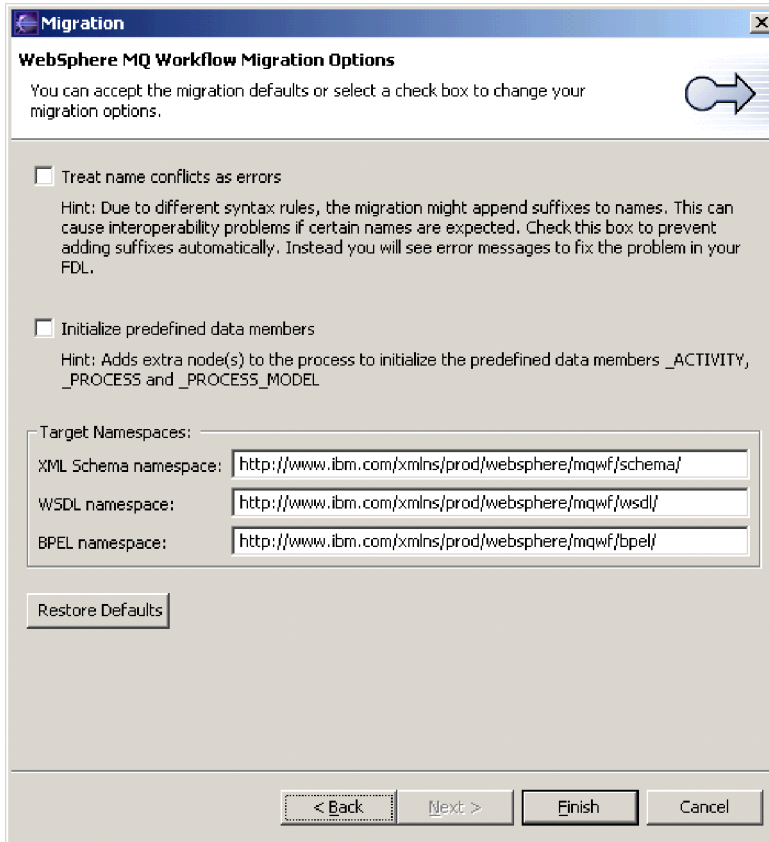
From the Migration page, select the **Migrate a WebSphere MQ Workflow process** option



2. The Migration wizard opens. Enter the absolute path and name of the FDL file into the Source selection field or select one from the file system by clicking the **Browse** button and navigating to the file. Enter the module name in the relevant field (you must enter a module name in the Module name field before you are able to proceed). Click **Next**:



3. The Migration Options page opens. From here you can accept the migration defaults or select a check box to change the option. By selecting the **Treat name conflicts as errors** check box, you can prevent the automatic addition of suffixes which could result in interoperability errors. The **Initialize predefined data members** check box adds extra nodes to the process to initialize the predefined data members:

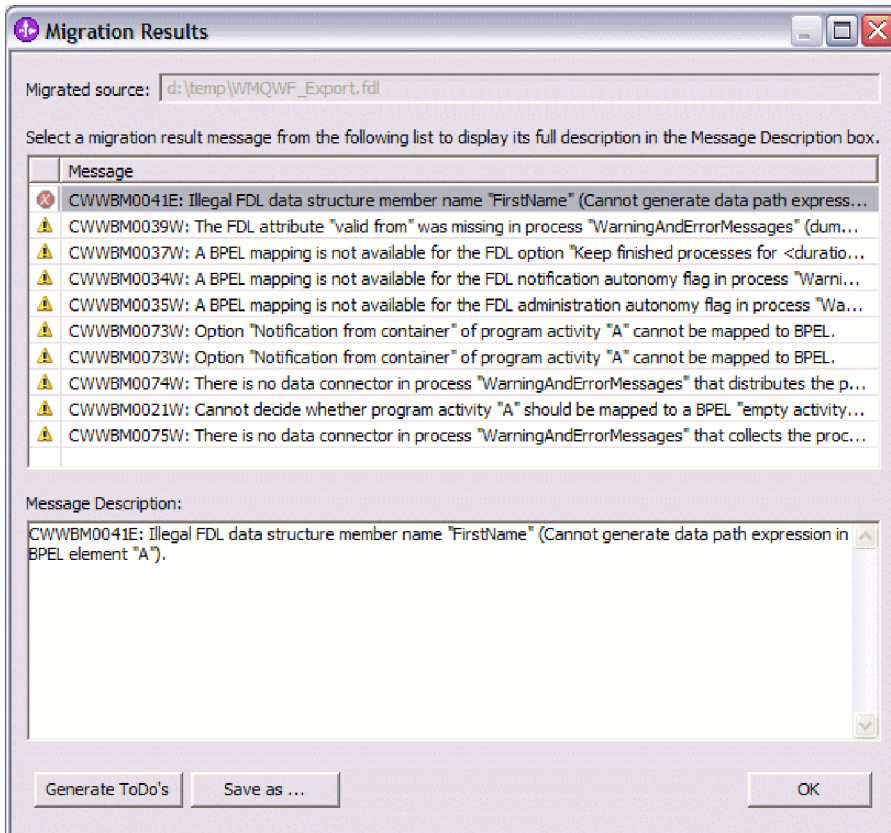


Click **Finish**.

## Verifying the WebSphere MQ Workflow migration

If the migration completes with a list of errors, warnings, and/or informational messages, they will be displayed in the Migration Results window. Otherwise, the wizard window will close.

The following page appears if migration messages were generated during the migration process:



In the Migration Results window, you can see the migration messages that were generated during the migration process. By selecting a message from the upper Message list, you can find more information regarding that message in the lower Message Description window. To keep all messages for future reference, click the **Generate ToDo's** button to create a list of "ToDo" tasks in the task view and/or click the **Save as...** button to save the messages in a text file in the filesystem.

### Limitations of the migration process (from WebSphere MQ Workflow)

There are certain limitations involved with the WebSphere MQ Workflow migration process.

- The migration of FDL will generate invoke activities for UPES activities and the corresponding WSDLs. However, the runtime environment significantly differs between IBM® WebSphere MQ Workflow and IBM WebSphere Process Server in terms of the techniques that are used to correlate invocation messages and their responses.
- The runtime engines of IBM WebSphere MQ Workflow and IBM WebSphere Process Server handle uninitialized data differently. While in IBM WebSphere MQ Workflow this did not cause errors, IBM WebSphere Process Server is handling this situation with an exception and stops executing the process. To run migrated applications correctly in IBM WebSphere Process Server, ensure that all variables and sub-structures are initialized before they are used with Assign, Invoke, Staff, and Reply activities.

### Migrating source artifacts to WebSphere Integration Developer from WebSphere Studio Application Developer Integration Edition

Source artifacts can be migrated from WebSphere Studio Application Developer Integration Edition to WebSphere Integration Developer. Migrating the source artifacts in an application involves migrating them to the new WebSphere Integration Developer programming model so that new functionality and features can be used. The application can then be redeployed and installed to the WebSphere Process Server.

Many features available in WebSphere Business Integration Server Foundation 5.1 have moved down into the base WebSphere Application Server 6.0. See this site for tips on migrating those features: [Tips for migrating programming model extensions](#).

To fully migrate a WebSphere Studio Application Developer Integration Edition service project, there are three fundamental tasks to complete:

1. Preparing source artifacts for migration. These actions may need to be performed in WebSphere Studio Application Developer Integration Edition.
2. Use the Migration wizard to automatically migrate the artifacts to the Business Integration Module project.
3. Use WebSphere Integration Developer to manually complete the migration. This involves fixing any Java code that could not be automatically migrated and verifying the wiring of the migrated artifacts.

**Note:** The runtime migration (upgrade path) will not be provided in WebSphere Process Server 6.0.x, therefore, this source artifact migration path will be the only option for migrating WebSphere Studio Integration Edition service projects in 6.0.x.

## Supported migration paths for migrating source artifacts

Before beginning to migrate source artifacts from WebSphere Studio Application Developer Integration Edition, you should review the supported migration paths that are supported by WebSphere Integration Developer.

The Migration wizard offers the ability to migrate one WebSphere Studio Application Developer Integration Edition Version 5.1 (or above) service project at a time. It will *not* migrate an entire workspace.

The Migration wizard does not migrate application binaries – it will only migrate source artifacts found in a WebSphere Studio Application Developer Integration Edition service project.

## Preparing source artifacts for migration

Before migrating source artifacts to WebSphere Integration Developer from WebSphere Studio Application Developer Integration Edition, you must first ensure that you have properly prepared your environment.

The following steps describe how to prepare your environment before migrating source artifacts to WebSphere Integration Developer from WebSphere Studio Application Developer Integration Edition:

1. Ensure that you have a backup copy of the entire 5.1 workspace before attempting to migrate.
2. Review the migration section of the Rational® Application Developer Information Center to determine the best way to migrate the non-WBI-specific projects in your workspace: [Migrating from WebSphere Studio V5.1, 5.1.1, or 5.1.2](#)
3. Review the Web service section of the Rational Application Developer Information Center for background information on the Web service functionality provided by Rational Application Developer: [Developing Web services](#)
4. Ensure that you have all of the appropriate WebSphere Integration Developer features enabled. If you don't have these features enabled, you may not see the menu options that will be discussed below. To enable the important features:
  - In WebSphere Integration Developer, go to the **Window** menu item and select **Preferences**.
  - Go to the **Workbench** then select the **Capabilities** category.
  - Select all features under the following categories:
    - Advanced J2EE
    - Enterprise Java
    - Integration Developer
    - Java Developer

- Web Developer (typical)
  - Web Service Developer
  - XML Developer
  - Click **OK**.
5. Use a new workspace directory for WebSphere Integration Developer. It is not recommended that you open WebSphere Integration Developer in an old WebSphere Studio Application Developer Integration Edition workspace that contains service projects, as those projects must first be migrated to a format that can be read by WebSphere Integration Developer. The recommended steps to do this are:
    - a. Copy all non-service projects from the old workspace to the new workspace. Do *not* copy the 5.1 EJB, Web, and EAR projects created when deploy code was generated for a 5.1 service project. The new 6.0 deploy code will be regenerated automatically when the BI module is built.
    - b. Open WebSphere Integration Developer in the blank workspace and import all non-service projects by clicking on **File** → **Import** → **Existing Project into Workspace** and select the projects that you have copied over to the new workspace.
      - If the project is a J2EE project, then you should migrate it to the 1.4 level by using the Rational Application Developer Migration wizard:
        - 1) Right-click the project and select **Migration** → **J2EE Migration Wizard...**
        - 2) Review the warning statements on the first page and, unless otherwise indicated, click **Next**.
        - 3) Ensure that the **J2EE project** is checked in the Projects list. Leave **Migrate project structure** and **Migrate J2EE specification level** checked. Choose J2EE version **1.4** and **Target Server WebSphere Process Server v6.0**.
        - 4) Select any other options that might be appropriate for your J2EE project and click **Finish**. If this step completes successfully, you will see a message **Migration finished successfully**.
        - 5) If there are errors in the J2EE project after migration, you should remove all classpath entries that refer to v5 .jar files or libraries and add the **JRE System Library** and **WPS Server Target** libraries to the classpath instead (discussed below). This should resolve most, if not all, of the errors.
      - For WebSphere Business Integration EJB projects with Extended Messaging (CMM) or Container Managed Persistence over Anything (CMP/A), the IBM EJB Jar Extension descriptor files must be migrated after the 5.1 project has been imported into the 6.0.x workspace. See "Migrating WebSphere Business Integration EJB Projects" for more information.
      - Fix the classpath for each non-service project imported in to the workspace. To add the JRE and WebSphere Process Server libraries to the classpath, right-click on the imported project and select **Properties**. Go to the **Java Build Path** entry and select the **Libraries** tab. Then do the following:
        - 1) Select **Add library** → **JRE System Library** → **Alternate JRE - WPS Server v6.0 JRE** → **Finish**.
        - 2) Then select **Add library** → **WPS Server Target** → **Configure wps server classpath** → **Finish**.
  6. By default, WebSphere Integration Developer generates the deploy code during build time.
  7. In order to fully migrate the .bpel files within a service project, you must ensure that all .wsdl and .xsd files referenced by the .bpel files can be resolved in a business integration project in the new workspace:
    - If the .wsdl and/or .xsd files are in the same service project as the .bpel file then no further action is required.
    - If the .wsdl and/or .xsd files are in a different service project than the one you are migrating, the 5.1 artifacts must be reorganized using WebSphere Studio Application Developer Integration Edition prior to migration. The reason for this is that Business Integration Module projects may not share artifacts. Here are the two options for reorganizing the 5.1 artifacts:
      - In WebSphere Studio Application Developer Integration Edition, create a new Java project that will hold all the common artifacts. Place all .wsdl and .xsd files that are shared by more than one service project into this new Java project. Add a dependency on this new Java project to all service projects that use these common artifacts. In WebSphere Integration Developer, create a

new Business Integration Library project with the same name as the 5.1 shared Java project before migrating any of the service projects. Manually copy the old .wsdl and .xsd files from the 5.1 shared Java project to this new BI Library project folder. This must be done before migrating the BPEL service projects.

- Another option is to keep a local copy of these shared .wsdl and .xsd artifacts in each service project such that there are no dependencies between service projects.
- If the .wsdl and/or .xsd files are in any other type of project (usually other Java Projects), you should create a Business Integration Library project with the same name as the 5.1 project. You should also set up the new library project's classpath, adding the entries from the 5.1 Java project if any. This type of project is useful for storing shared artifacts.

You are now ready to begin the migration process.

## Migrating service projects using the WebSphere Integration Developer Migration wizard

The WebSphere Integration Developer Migration wizard enables the migration of service projects.

### Note:

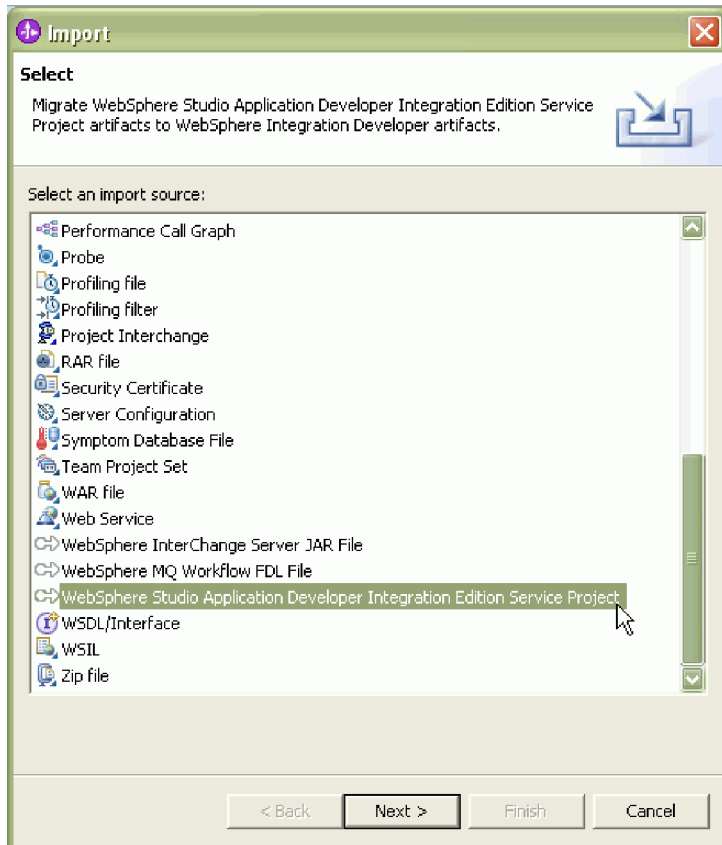
- You must migrate the service projects in their dependency order. For example, if a BPEL in service project A makes a process-to-process call to a BPEL in service project B, then service project B *must* be migrated before service project A. Otherwise, the process-to-process call cannot be configured correctly.
- You are not required to disable autobuild in WebSphere Integration Developer Migration 6.0.2 and beyond as autobuild is automatically turned off during the migration process.

The Migration wizard does the following:

1. Creates a new business integration module (the module name is defined by you)
2. Migrates the service project's classpath entries to the new module
3. Copies all WebSphere Business Integration Server Foundation source artifacts from the selected source project to this module
4. Migrates the BPEL extensions in WSDL files
5. Migrates the business processes (.bpel files) from BPEL4WS version 1.1 to the new level supported by WebSphere Process Server, which is built on BPEL4WS version 1.1 with major capabilities of the upcoming WS-BPEL version 2.0 specification
6. Creates an SCA component for each .bpel process
7. Generates a monitoring .mon file for each BPEL process to preserve the default monitoring behavior from WebSphere Studio Application Developer Integration Edition (if necessary)
8. Creates imports and exports depending on the deploy options chosen in WebSphere Studio Application Developer Integration Edition
9. Wires the BPEL component to its partnerlinks (imports, exports, and Java components)

To migrate service projects using the WebSphere Integration Developer Migration wizard, follow these steps:

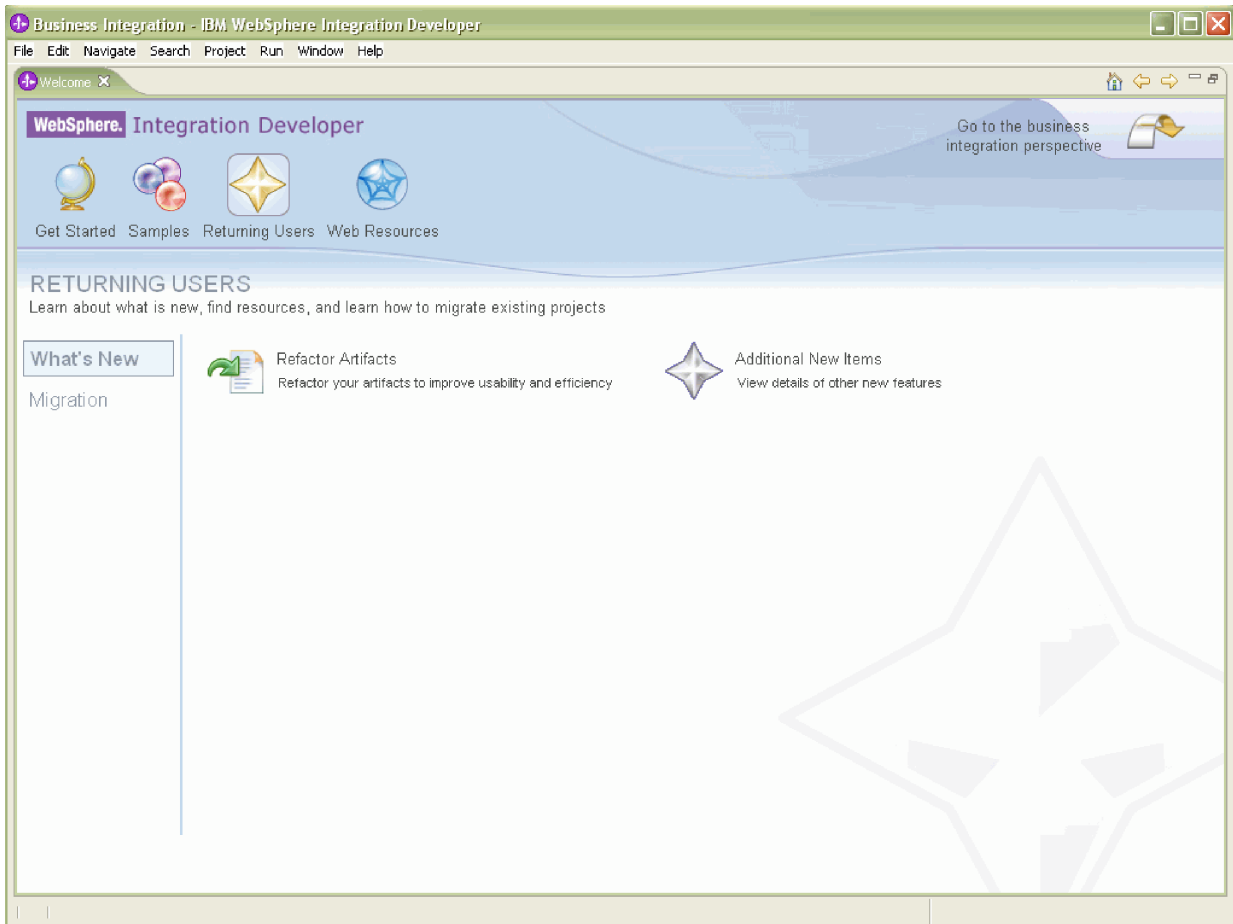
1. Invoke the wizard by selecting **File** → **Import** → **WebSphere Studio Application Developer Integration Edition Service Project**.



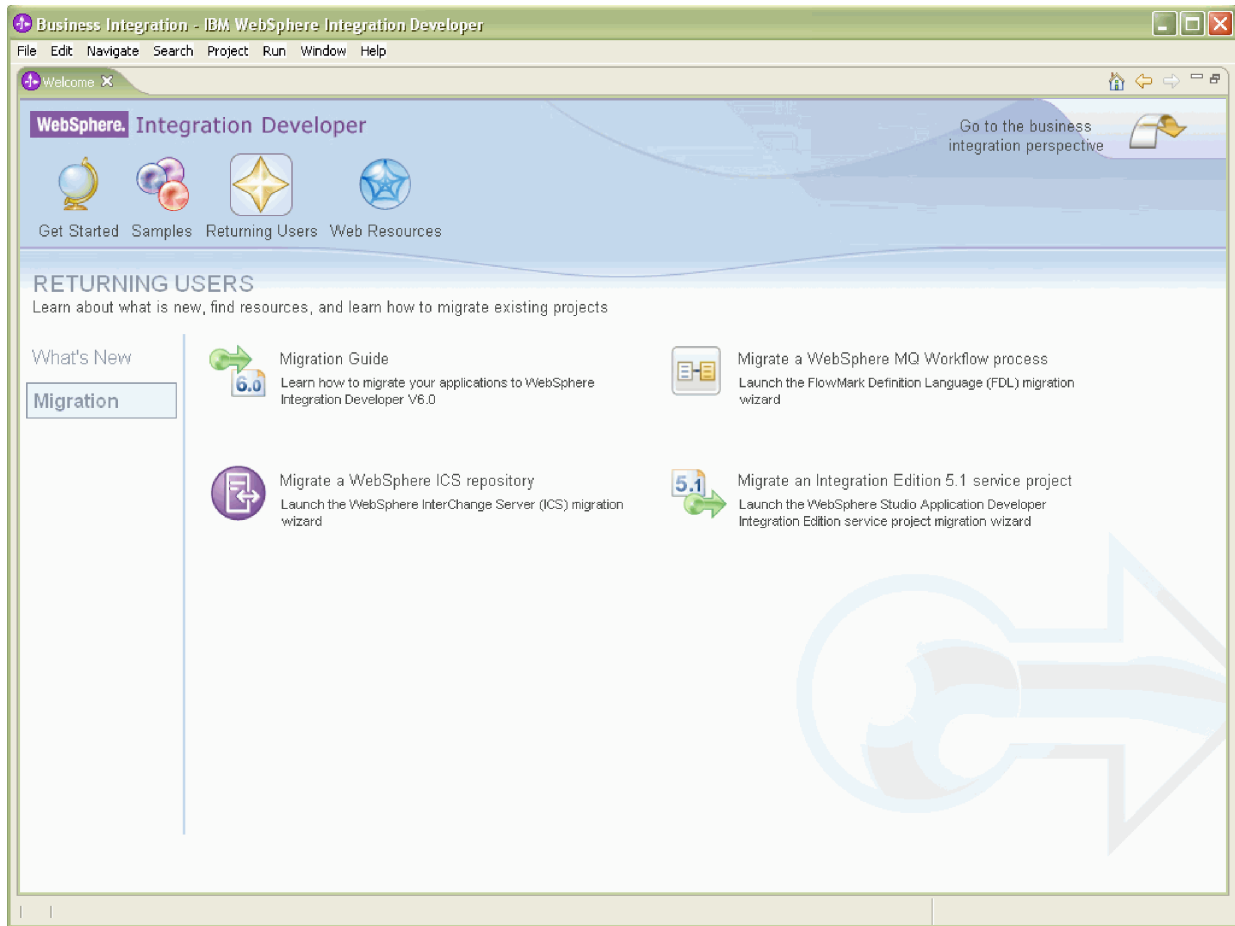
OR you can also open the Migration wizard from the Welcome page by clicking on the **Returning**



**Users** icon [Returning Users](#) to open the Returning Users page (Note that you can always return to the Welcome page by clicking on **Help** → **Welcome**):



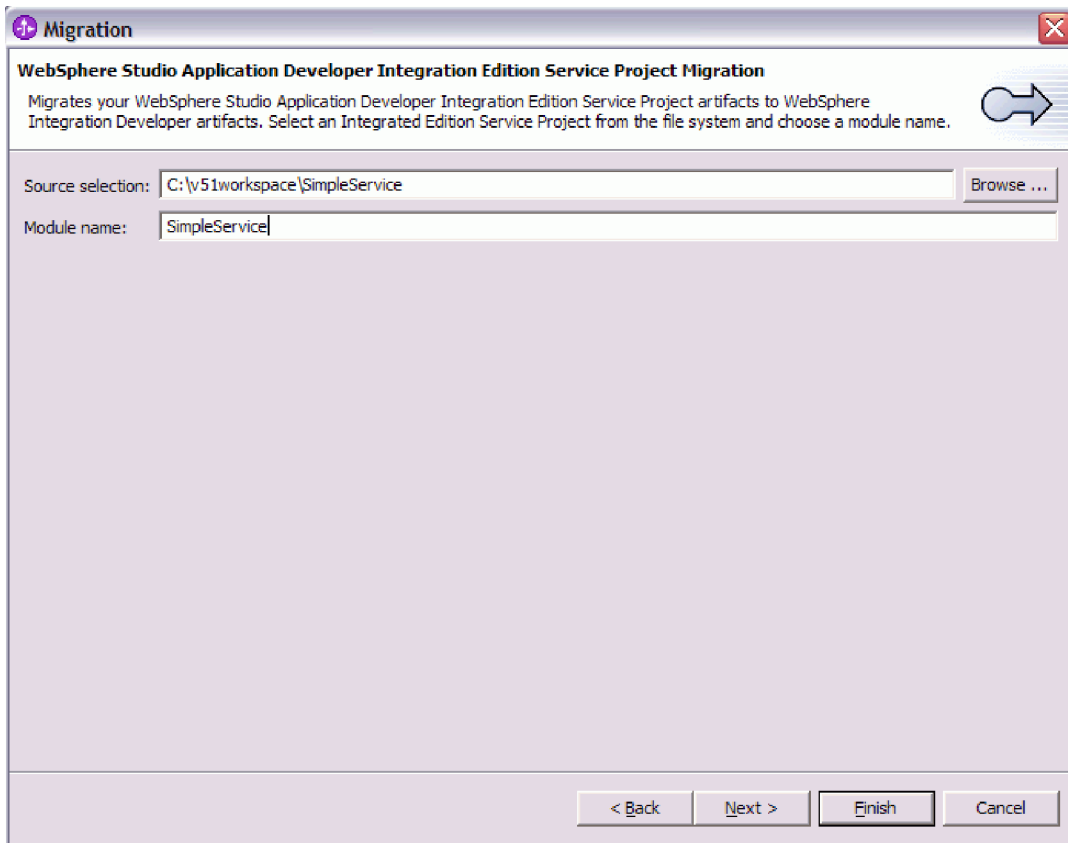
Click on **Migration** on the left side of the Returning Users page to open the Migration page:



From the Migration page, select the **Migrate an Integration Edition 5.1 service project** option

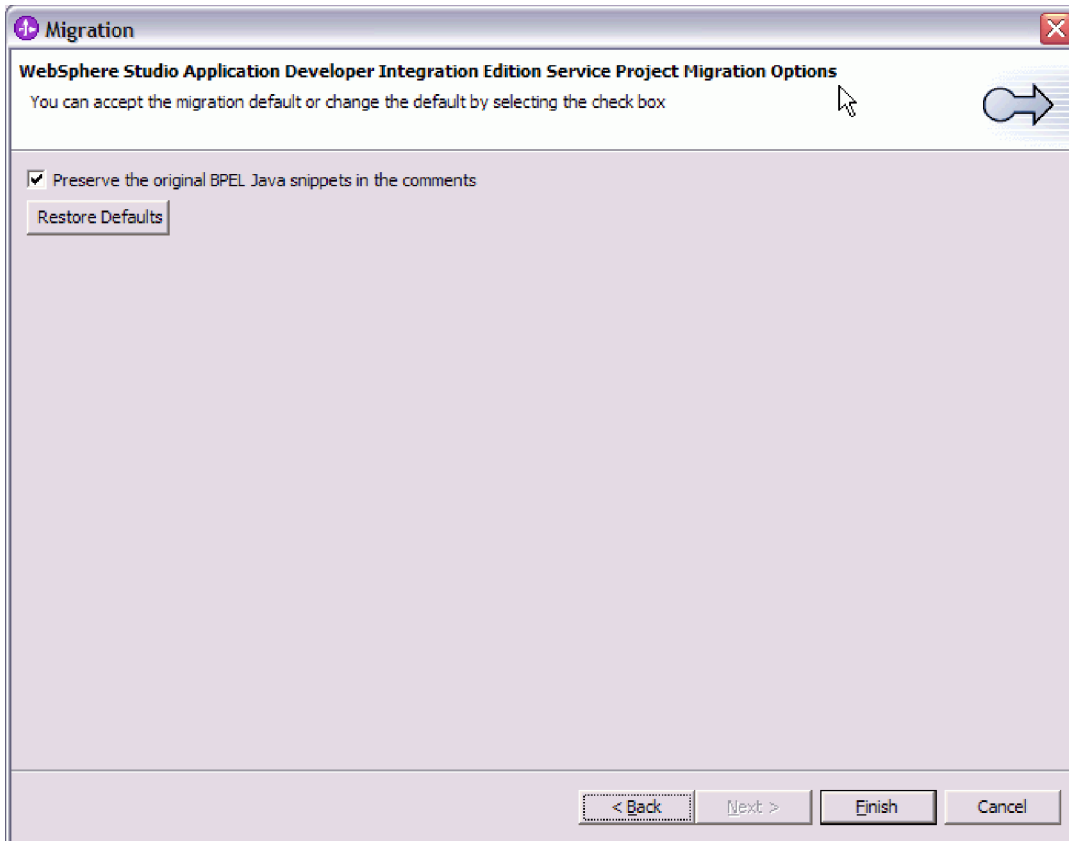


2. The Migration wizard opens. Enter the path for the Source Selection or click the **Browse** button to find it. Also enter the Module name of the location of the WebSphere Studio Application Developer Integration Edition Service Project to migrate:



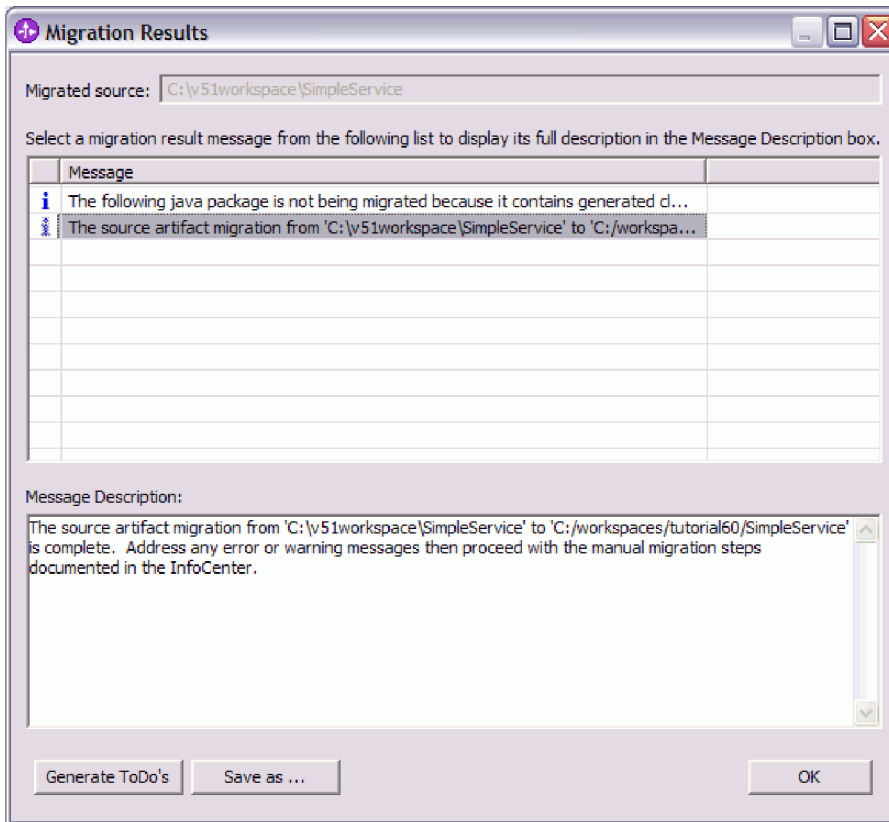
**Note:** It is recommended that you choose the name of the Service Project as the module name because if there are other projects in the WebSphere Studio Application Developer Integration Edition workspace that are dependent on this project, you won't have to update the dependent projects' classpaths after importing them in to WebSphere Integration Developer.

3. From the Migration options, select the Preserve original BPEL Java snippets in the comments check box:



Click **Finish**.

4. After the migration process has completed, the Migration Results window opens:



A log file containing these migration messages will automatically get generated to the 6.0 workspace's .metadata folder. The log file will be named ".log".

After the Migration wizard has completed, build the Business Integration module that was created and try to resolve any build errors. Inspect all migrated .bpel files: ensure that they are fully migrated and can be opened in the WebSphere Integration Developer BPEL Editor. There are some BPEL Java snippets that can not be automatically migrated. If you see any errors in the BPEL Java snippets, see "Migrating to the SCA Programming Model" for steps needed to fix the errors. Also, if you used the Migration wizard to migrate a service project to a BI Module, open the module dependency editor to ensure that the dependencies are set correctly. To do this, switch to the Business Integration perspective and double click on the BI module project. From there you can add dependencies on business integration library projects, Java projects, and J2EE projects.

## Manually migrating the application

After the Migration wizard has successfully migrated the artifacts to the new Business Integration module, the artifacts must be wired together to create an application that adheres to the SCA model. Note that even though the Migration wizard attempts to successfully migrate the artifacts, manual verification should also be done. The information in this section can be used to help ensure that the migration was correct.

1. Open WebSphere Integration Developer and switch to the Business Integration perspective. You should see the module(s) that were created by the Migration wizard (one module for each service project that was migrated). The first artifact listed under the module project is the module's assembly file (it has the same name as the module).
2. Double-click the assembly file to open it in the Assembly Editor where SCA components can be created and wired together to obtain similar functionality to the Version 5.1 application. If there were any BPEL processes in the WebSphere Studio Application Developer Integration Edition service project, the migration wizard should have created default SCA components for each of those processes and they will be in the Assembly Editor.

3. Select a component and go to the Properties view where the Description, Details, and Implementation properties will be displayed and can be edited.

The following information further describes how to manually rewire the application using the tools available in WebSphere Integration Developer:

### **Creating SCA Components and SCA Imports for the services in the application for rewiring:**

All projects will require some rewiring after migration in order to reconnect the services the way they were in 5.1. For example, all migrated business processes must be rewired to their business partners. An SCA Component or Import must be created for all other service types. For WebSphere Studio Application Developer Integration Edition service projects that interact with systems or entities external to the project, an SCA Import can be created in order for the migrated project to access those entities as services according to the SCA model. **Note:** The migration utility attempts to do this automatically, however, you can refer to the following information to help verify what the tool did.

For WebSphere Studio Application Developer Integration Edition service projects that interact with entities within the project (for example, a business process, transformer service or Java class), an SCA Import can be created in order for the migrated project to access those entities as services according to the SCA model.

The following sections provide details on the SCA Import or SCA Components to create based on the type of service that must be migrated:

#### *Migrating a Java service:*

You can migrate a Java service to an SCA Java Component.

If the WebSphere Studio Application Developer Integration Edition Service Project was dependent on other Java projects, copy the existing projects into the new workspace directory and import them into WebSphere Integration Developer using the **File** → **Import** → **Existing Project into Workspace** wizard.

In WebSphere Studio Application Developer Integration Edition when generating a new Java service from an existing Java class, the following options were given:

- Create XSD schemas for complex data types:
  - Within the interface WSDL file
  - As a new file for each data type
- Support error handling capability:
  - Generate fault
  - Do not generate fault
- Other details about the service to generate such as binding and service names

There are many new components in 6.0 that offer new functionality such as data mapping, interface mediation, business state machines, selectors, business rules, and more. First you should determine whether one of these new component types can replace the custom Java component. If that is not possible, follow the migration path described below.

Import the service project using the Migration wizard. This will result in the creation of a business integration module with the WSDL Messages, PortTypes, Bindings, and Services generated in WebSphere Studio Application Developer Integration Edition.

In the Business Integration perspective, expand the module to see its contents. Open the Assembly Editor by double-clicking the first item under the module project (it will have the same name as the project).

You have the following options:

*Pros and cons for each of the Java service rewiring options:*

There are pros and cons for each of the Java service rewiring options.

The following list describes both options and the pros and cons of each:

- The first option is likely to give better performance at runtime because invoking a Web service is slower than invoking a Java component.
- The first option can propagate context whereas a Web service invocation does not propagate context in the same way.
- The second option does not involve creating any custom code.
- The second option may not be possible for some Java interface definitions, as generating a Java service has limitations. See the Rational Application Developer documentation here: [Limitations of Web services](#)
- The second option may result in an interface change and hence a change to the SCA consumer.
- The second option requires that a WebSphere Process Server 6.0 server is installed and has been configured to work with WebSphere Integration Developer. To see the installed runtimes that are configured to work with WebSphere Integration Developer, go to **Window** → **Preferences** → **Server** → **Installed Runtimes** and select the **WebSphere Process Server v6.0** entry if it exists and ensure that it points to the location where the product is installed. Ensure that this entry is checked if the server does exist and unchecked if this server is not actually installed. You can also click the **Add...** button if you want to add another server.
- If the Java component was built in WebSphere Studio Application Developer Integration Edition using the top-down approach where the Java skeleton was generated from a WSDL, then the parameters in and out of this Java class will probably subclass WSIFFormatPartImpl. If this is the case then you choose option 1 to generate a new SCA style Java skeleton from the original WSDL/XSDs or option 2 to generate a new generic Java skeleton (not dependent on the WSIF or DataObject APIs) from the original WSDL interface.

*Creating the custom Java component: option 1:*

The recommended migration technique is to use the WebSphere Integration Developer Java Component type that allows you to represent the Java service as an SCA component. During migration, custom Java code must be written to convert between the SCA Java interface style and the existing Java component's interface style.

To create the custom Java component, follow these steps:

1. Under the module project, expand **Interfaces** and select the WSDL interface that was generated for this Java class in WebSphere Studio Application Developer Integration.
2. Drag and drop this interface onto the Assembly Editor. A dialog will pop up asking you to select the type of component to create. Select **Component with No Implementation Type** and click **OK**.
3. A generic component will appear on the Assembly diagram. Select it and go to the **Properties** view.
4. On the **Description** tab, you can change the name and display name of the component to something more descriptive.
5. On the **Details** tab you will see that this component has one interface - the one that you dragged and dropped onto the Assembly Editor.
6. Ensure that the Java class that you are trying to access is on the classpath of the service project if it is not contained within the service project itself.
7. Right-click on the module project and select **Open Dependency Editor...** Under the **Java**, section ensure that the project containing the old Java class is listed. If it is not, add it by clicking the **Add...** button.

8. Back in the Assembly Editor, right-click the component that you just created and select **Generate Implementation...** → **Java** Then select the package where the Java implementation will be generated. This creates a skeleton Java service that adheres to the WSDL interface according to the SCA programming model, where complex types are represented by an object that is a `commonj.sdo.DataObject` and simple types are represented by their Java Object equivalents.

The following code examples show:

1. Relevant definitions from the 5.1 WSDL interface
2. The WebSphere Studio Application Developer Integration Edition 5.1 Java methods that correspond to the WSDL
3. The WebSphere Integration Developer 6.0 Java methods for the same WSDL

The following code shows the relevant definitions from the 5.1 WSDL interface:

```
<types>
  <schema xmlns="http://www.w3.org/2001/XMLSchema"
    attributeFormDefault="qualified"
    elementFormDefault="unqualified"
    targetNamespace="http://migr.practice.ibm.com/"
    xmlns:xsd1="http://migr.practice.ibm.com/">

    <complexType name="StockInfo">
      <all>
        <element name="index" type="int"/>
        <element name="price" type="double"/>
        <element name="symbol" nillable="true"
          type="string"/>
      </all>
    </complexType>
  </schema>
</types>

<message name="getStockInfoRequest">
  <part name="symbol" type="xsd:string"/>
</message>
<message name="getStockInfoResponse">
  <part name="result" type="xsd1:StockInfo"/>
</message>

<operation name="getStockInfo" parameterOrder="symbol">
  <input message="tns:getStockInfoRequest"
    name="getStockInfoRequest"/>
  <output message="tns:getStockInfoResponse"
    name="getStockInfoResponse"/>
</operation>
```

The following code shows the WebSphere Studio Application Developer Integration Edition 5.1 Java methods that correspond to the WSDL:

```
public StockInfo getStockInfo(String symbol)
{
  return new StockInfo();
}

public void setStockPrice(String symbol, float newPrice)
{
  // set some things
}
```

The following code shows the WebSphere Integration Developer 6.0 Java methods for the same WSDL:

```
public DataObject getStockInfo(String aString) {
  //TODO Needs to be implemented.
  return null;
}
```

```

}

public void setStockPrice(String symbol, Float newPrice) {
    //TODO Needs to be implemented.
}

```

Now you will need to fill in code where you see the “//TODO” tags in the generated Java implementation class. There are two options:

1. Move the logic from the original Java class to this class, adapting it to use DataObjects
  - This is the recommended option if you had chosen the top-down approach in WebSphere Studio Application Developer Integration Edition and want your Java component to deal with DataObject parameters. This rework is necessary because the Java classes generated from WSDL definitions in WebSphere Studio Application Developer Integration Edition have WSIF dependencies that should be eliminated.
2. Create a private instance of the old Java class inside this generated Java class and write code to:
  - a. Convert all parameters of the generated Java implementation class into parameters that the old Java class expects
  - b. Invoke the private instance of the old Java class with the converted parameters
  - c. Convert the return value of the old Java class into the return value type declared by the generated Java implementation method
  - d. This option is recommended for consumption scenarios where the WSIF service proxies must be consumed by new 6.0 style Java components.

Once you have completed one of the above options, you must rewire the Java service. There should not be any references, therefore you just need to rewire the Java component’s interface:

- If this service is invoked by a business process in the same module, then create a wire from the appropriate business process reference to this Java component’s interface.
- If this service is invoked by a business process in another module, create an **Export with SCA Binding** and from the other module, drag and drop this export onto that module’s Assembly Editor to create the corresponding **Import with SCA Binding**. Wire the appropriate business process reference to that Import.
- If this service was published in WebSphere Studio Application Developer Integration Edition to expose it externally, then see the section “Creating SCA Exports to access the migrated service” for instructions on how to republish the service.

*Creating a Java Web service: option 2:*

An alternative option to consider is the Rational Application Developer Web services tooling that allows you to create a Web service around a Java class.

**Note:** See the information at the following site before attempting to migrate using this method: [Creating a Web service from a Java bean](#)

**Note:** This option requires that a Web service runtime be configured through WebSphere Integration Developer before invoking the Web service wizard.

If you had taken a bottom-up approach in WebSphere Studio Application Developer Integration Edition to generate WSDL around the Java class, then follow these steps:

1. Create a new Web project and copy the Java class that you would like to build a service around to this Web project’s Java source folder.
2. Right-click on the enterprise application project that is the container for the Java class you are creating a service around.

3. Select **Properties**, go to the **Server** properties and ensure that the **Target runtime** is set to **WebSphere Process Server v6.0** and **Default server** is set to the installed **WebSphere Process Server v6.0**.
4. Start the test server and deploy this application to the server and ensure that it starts successfully.
5. Next, right-click on the Java class that you would like to create a service around and select **Web Services** → **Create Web service**.
6. For **Web Service Type** select **Java bean Web Service** and uncheck the **Start Web service in Web project** option unless you want to deploy the web service right away. You can optionally select to generate a client proxy as well. Click **Next**.
7. The Java class that you right-clicked will be shown, click **Next**.
8. You must now configure your service deployment options. Click the **Edit...** button. For the server type choose **WPS Server v6.0** and for the Web service runtime choose **IBM WebSphere** and J2EE version **1.4**. If you are not able to select a valid combination by doing this, then see the section "Preparing for Migration" for information on migrating J2EE projects to the v1.4 level. Click **OK**.
9. For the Service project, enter the name of the Web project. Also select the appropriate EAR project. Click **Next**. Note that you may have to wait for several minutes.
10. On the Web Service Java Bean Identity panel, select the WSDL file that will contain the WSDL definitions. Choose the methods that you would like to expose on the Web service and choose the appropriate style/encoding (Document/Literal, RPC/Literal, or RPC/Encoded). Select the **Define custom mapping for package to namespace** option and select a namespace that is unique to the Java class being migrated for all Java packages used by this Java class's interface (the default namespace will be unique to the package name which may cause conflicts if you create another Web Service that uses the same Java classes). Complete the other parameters if appropriate.
11. Click **Next** and on the **Web Service package to namespace mapping panel**, click the **Add** button and in the row that is created, enter the name of the package of the Java bean, then add the custom namespace that uniquely identifies this Java class. Continue to add mappings for all Java packages used by the Java bean interface.
12. Click **Next**. Note that you may have to wait for several minutes.
13. Click **Finish**. After completing the wizard, you should copy the generated WSDL file that describes the Java service to the business integration module project if the service project was a consumer of the Java service. It can be found in the generated router Web project under the folder `WebContent/WEB-INF/wsdl`. Refresh/rebuild the business integration module project.
14. Switch to the Business Integration perspective and expand the module and then the **Web Service Ports** logical category.
15. Select the port that was created in the previous steps and drag and drop it onto the Assembly Editor and select to create an **Import with Web Service Binding**. Select the Java class's WSDL interface if prompted. Now the SCA component that consumed the Java component in 5.1 can be wired to this Import to complete the manual rewiring migration steps.

Note that the interface may be slightly different than the 5.1 interface, and you may need to insert an Interface Mediation component in between the 5.1 consumer and the new Import. To do this, click on the **wire** tool in the Assembly Editor and wire the SCA source component to this new **Import with Web Service Binding**. As the interfaces are different, you will be prompted: **Source and target nodes do not have matching interfaces**. Choose to **create an interface mapping between the source and target node**. Double-click on the mapping component that was created in the Assembly Editor. This will open the mapping editor. See the Information Center for instructions on creating an interface mapping.

If you had taken a top-down approach in WebSphere Studio Application Developer Integration Edition, generating Java classes from a WSDL definition, then follow these steps:

1. Create a new Web project and copy the WSDL file that you would like to Java skeleton to this Web project's source folder.

2. Right-click on the WSDL file containing the PortType that you want to generate a Java skeleton from and select **Web Services** → **Generate Java bean skeleton**.
3. Choose the Web service type **Skeleton Java bean Web Service** and complete the wizard.

After completing the wizard, you should have Java classes that implement the service interface and are not dependent on WSIF APIs.

*Migrating an EJB service:*

You can migrate an EJB service to an SCA Import with stateless session bean binding.

If the WebSphere Studio Application Developer Integration Edition Service Project was dependent on another EJB, EJB client, or Java project, import those existing projects using the **File** → **Import** → **Existing Project into Workspace** wizard. This was usually the case when an EJB was referenced from a service project. If any WSDL or XSD files that are referenced from the service project exist in another type of project, create a new Business Integration Library with the same name as the old non-service project, and copy all of those artifacts to the library.

Import the service project using the Migration wizard. This will result in the creation of a business integration module with the WSDL Messages, PortTypes, Bindings, and Services generated in WebSphere Studio Application Developer Integration Edition.

In the Business Integration perspective, expand the module to see its contents. Open the Assembly Editor by double-clicking the first item under the module project (it will have the same name as the project).

You have the following options:

*Pros and cons for each of the EJB service rewiring options:*

There are pros and cons for each of the EJB service rewiring options.

The following list describes both options and the pros and cons of each:

- The first option is likely to give better performance at runtime because invoking a Web service is slower than invoking an EJB.
- The first option can propagate context whereas a Web service invocation does not propagate context in the same way.
- The second option does not involve creating any custom code.
- The second option may not be possible for some EJB interface definitions, as generating an EJB service has limitations. See the Rational Application Developer documentation here: [Limitations of Web services](#)
- The second option may result in an interface change and hence a change to the SCA consumer.
- The second option requires that a WebSphere Process Server 6.0 server is installed and has been configured to work with WebSphere Integration Developer. To see the installed runtimes that are configured to work with WebSphere Integration Developer, go to **Window** → **Preferences** → **Server** → **Installed Runtimes** and select the **WebSphere Process Server v6.0** entry if it exists and ensure that it points to the location where the product is installed. Ensure that this entry is checked if the server does exist and unchecked if this server is not actually installed. You can also click the **Add...** button if you want to add another server.
- If the Java component was built in WebSphere Studio Application Developer Integration Edition using the top-down approach where the EJB skeleton was generated from a WSDL, then the parameters in and out of this Java class will probably subclass WSIFFormatPartImpl. If this is the case then you choose option 2 to generate a new generic EJB skeleton (not dependent on the WSIF or DataObject APIs) from the original WSDL interface.

*Creating the custom EJB component: option 1:*

The recommended migration technique is to use the WebSphere Integration Developer Import with Stateless Session Binding type that allows you to invoke a stateless session EJB as an SCA component. During migration, custom Java code must be written to convert between the SCA Java interface style and the existing EJB interface style.

**Note:** Even though the migration tool automatically handles this, any changes made after migration to the interfaces and data types (business objects) involved in the EJB interface will require manual updates to the conversion code mentioned here. Errors may be displayed in WebSphere Integration Developer depending on the type of change made.

To create the custom EJB component, follow these steps:

1. Under the module project, expand **Interfaces** and select the WSDL interface that was generated for this EJB in WebSphere Studio Application Developer Integration.
2. Drag and drop this interface onto the Assembly Editor. A dialog will pop up asking you to select the type of component to create. Select **Component with No Implementation Type** and click **OK**.
3. A generic component will appear on the Assembly diagram. Select it and go to the **Properties** view.
4. On the **Description** tab, you can change the name and display name of the component to something more descriptive. Choose a name like your EJB's name, but append a postfix such as "JavaMed" as this is going to be a Java component that mediates between the WSDL interface generated for the EJB in WebSphere Studio Application Developer Integration and the Java interface of the EJB.
5. On the **Details** tab you will see that this component has one interface - the one that you dragged and dropped onto the Assembly Editor.
6. Back in the Assembly Editor, right-click the component that you just created and select **Generate Implementation... → Java** Then select the package where the Java implementation will be generated. This creates a skeleton Java service that adheres to the WSDL interface according to the SCA programming model, where complex types are represented by an object that is a `com.ibm.sdo.DataObject` and simple types are represented by their Java Object equivalents.

The following code examples show:

1. Relevant definitions from the 5.1 WSDL interface
2. The WebSphere Studio Application Developer Integration Edition 5.1 Java methods that correspond to the WSDL
3. The WebSphere Integration Developer 6.0 Java methods for the same WSDL

The following code shows the relevant definitions from the 5.1 WSDL interface:

```
<types>
  <schema xmlns="http://www.w3.org/2001/XMLSchema"
    attributeFormDefault="qualified"
    elementFormDefault="unqualified"
    targetNamespace="http://migr.practice.ibm.com/"
    xmlns:xsd1="http://migr.practice.ibm.com/">

    <complexType name="StockInfo">
      <all>
        <element name="index" type="int"/>
        <element name="price" type="double"/>
        <element name="symbol" nillable="true"
          type="string"/>
      </all>
    </complexType>
  </schema>
</types>

<message name="getStockInfoRequest">
  <part name="symbol" type="xsd:string"/>
</message>
```

```

<message name="getStockInfoResponse">
  <part name="result" type="xsd:StockInfo"/>
</message>

<operation name="getStockInfo" parameterOrder="symbol">
  <input message="tns:getStockInfoRequest"
    name="getStockInfoRequest"/>
  <output message="tns:getStockInfoResponse"
    name="getStockInfoResponse"/>
</operation>

```

The following code shows the WebSphere Studio Application Developer Integration Edition 5.1 Java methods that correspond to the WSDL:

```

public StockInfo getStockInfo(String symbol)
{
  return new StockInfo();
}

public void setStockPrice(String symbol, float newPrice)
{
  // set some things
}

```

The following code shows the WebSphere Integration Developer 6.0 Java methods for the same WSDL:

```

public DataObject getStockInfo(String aString) {
  //TODO Needs to be implemented.
  return null;
}

public void setStockPrice(String symbol, Float newPrice) {
  //TODO Needs to be implemented.
}

```

Eventually you need to fill in real code where you see the “//TODO” tags in the generated Java implementation class. First you need to create a reference from this Java component to the actual EJB so that it can access the EJB according to the SCA programming model:

1. Keep the Assembly Editor open and switch to the J2EE perspective. Locate the EJB project containing the EJB that you are creating a service for.
2. Expand its **Deployment Descriptor: <project-name>** item and locate the EJB. Drag and drop it onto the Assembly Editor. If warned about project dependencies needing to be updated, select the **Open the module dependency editor...** check box and click **OK**.
3. Under the J2EE section ensure that the EJB project is listed and if it is not, add it by clicking the **Add...** button.
4. Save the module dependencies and close that editor. You will see that a new Import was created in the Assembly Editor. You can select it and go to the Properties view on the Description tab to change the import’s name and display name to something more meaningful. On the Binding tab you will see that the import type is automatically set to **Stateless Session Bean Binding** and the JNDI name of the EJB is already set appropriately.
5. Select the Wire tool from the palette in the Assembly Editor.
6. Click on the Java component and release the mouse.
7. Next click on the EJB Import and release the mouse.
8. When asked **A matching reference will be created on the source node. Do you want to continue?**, click **OK**. This creates a wire between the two components.
9. Select the Java component in the Assembly Editor and in the Properties view under the Details tab, expand References and select the reference to the EJB that was just created. You can update the reference’s name if the generated name is not very descriptive or appropriate. Remember the name of this reference for future use.

## 10. Save the Assembly diagram.

You must use the SCA programming model to invoke the EJB from the generated Java class. Open the generated Java class and follow these steps to write the code that will invoke the EJB service. For the generated Java implementation class:

1. Create a private variable (whose type is that of your remote EJB interface):

```
private YourEJBInterface ejbService = null;
```

2. If there are complex types in your EJB interface, then also create a private variable for the BOFactory:

```
private BOFactory boFactory = (BOFactory)
    ServiceManager.INSTANCE.locateService("com/ibm/websphere/bo
    /BOFactory");
```

3. In the constructor of the Java implementation class, use the SCA APIs to resolve the EJB reference (remember to fill in the name of the EJB reference that you wrote down a few steps back) and set the private variable equal to this reference:

```
// Locate the EJB service
this.ejbService = (YourEJBInterface)
    ServiceManager.INSTANCE.locateService("name-of-your-ejb-reference");
```

For each “//TODO” in the generated Java implementation class:

1. Convert all parameters into the parameter types that the EJB expects.
2. Invoke the appropriate method on the EJB reference using the SCA programming model, sending the converted parameters.
3. Convert the return value of the EJB into the return value type declared by the generated Java implementation method

```
/**
 * Method generated to support the implementing WSDL port type named
 * "interface.MyBean".
 */
public DataObject getStockInfo(String aString) {
    DataObject boImpl = null;

    try {

        // invoke the EJB method
        StockInfo stockInfo = this.ejbService.getStockInfo(aString);

        // formulate the SCA data object to return.
        boImpl = (DataObject)
            this.boFactory.createClass(StockInfo.class);

        // manually convert all data from the EJB return type into the
        // SCA data object to return
        boImpl.setInt("index", stockInfo.getIndex());
        boImpl.setString("symbol", stockInfo.getSymbol());
        boImpl.setDouble("price", stockInfo.getPrice());
    } catch (RemoteException e) {
        e.printStackTrace();
    }
    return boImpl;
}

/**
 * Method generated to support the implementing WSDL port type named
 * "interface.MyBean".
 */
public void setStockPrice(String symbol, Float newPrice) {
    try {
        this.ejbService.setStockPrice(symbol, newPrice.floatValue());
    } catch (RemoteException e) {
```

```

    e.printStackTrace();
}
}

```

*Creating an EJB Web service: option 2:*

An alternative option to consider is the Rational Application Developer Web services tooling that allows you to create a Web service around an EJB.

**Note:** See the information at the following site before attempting to migrate using this method: [Creating a Web service from an enterprise bean \(EJB\) using the WebSphere run-time environment](#)

**Note:** This option requires that a Web service runtime be configured through WebSphere Integration Developer before invoking the Web service wizard.

To create a Web service around an EJB, follow these steps:

1. Right-click on the enterprise application project that is the container for the EJB that you are creating a service around.
2. Select **Properties**, go to the **Server** properties and ensure that the **Target runtime** is set to **WebSphere Process Server v6.0** and **Default server** is set to the installed **WebSphere Process Server v6.0**.
3. Start the test server and deploy this application to the server and ensure that it starts successfully.
4. In the J2EE perspective, expand the **EJB project** in the Project Explorer view. Expand the **Deployment Descriptor** then the **Session Beans** category. Select the bean that you want to generate the Web service around.
5. Right-click and select **Web Services** → **Create Web service**.
6. For **Web Service Type** select **EJB Web Service** and uncheck the **Start Web service in Web project** option unless you want to deploy the Web service right away. Click **Next**.
7. Ensure that the EJB that you right-clicked is selected here and click **Next**.
8. You must now configure your service deployment options. Click the **Edit...** button. For the server type choose **WPS Server v6.0** and for the Web service runtime choose **IBM WebSphere** and J2EE version **1.4**. If you are not able to select a valid combination by doing this, then see the section "Preparing for Migration" for information on migrating J2EE projects to the v1.4 level. Click **OK**.
9. For the Service project, enter the name of the EJB project containing the EJB. Also select the appropriate EAR project. Click **Next**. Note that you may have to wait for several minutes.
10. On the Web Service EJB Configuration panel, select the appropriate router project to use (choose the name of the router Web project you would like to be created and this project will be added to the same enterprise application as the original EJB. Select the desired transport (**SOAP over HTTP** or **SOAP over JMS**). Click **Next**.
11. Select the WSDL file that will contain the WSDL definitions. Choose the methods that you would like to expose on the Web service and choose the appropriate style/encoding (Document/Literal, RPC/Literal, or RPC/Encoded). Select the **Define custom mapping for package to namespace** option and select a namespace that is unique to the EJB being migrated for all Java packages used by the EJB (the default namespace will be unique to the package name which may cause conflicts if you create another Web Service that uses the same Java classes). Complete the other parameters if appropriate. There are limitations around each style/encoding combinations. See the limitations for more information: [Limitations of Web services](#)
12. Click **Next** and on the **Web Service package to namespace mapping** panel, click the **Add** button and in the row that is created, enter the name of the package of the your EJB, then the custom namespace that uniquely identifies this EJB. Continue to add mappings for all Java packages used by the EJB interface.
13. Click **Next**. Note that you may have to wait for several minutes.

14. Click **Finish**. After completing the wizard, you should copy the generated WSDL file that describes the EJB service to the business integration module project if the service project was a consumer of the EJB service. It can be found in the generated router Web project under the folder WebContent/WEB-INF/wsdl. Refresh/rebuild the business integration module project.
15. Switch to the Business Integration perspective and expand the migrated module and then the **Web Service Ports** logical category.
16. Select the port that was generated in the previous steps and drag and drop it onto the Assembly Editor and select to create an **Import with Web Service Binding**. Select the EJB's WSDL interface if prompted. Now the SCA component that consumed the EJB in 5.1 can be wired to this Import to complete the manual rewiring migration steps.

If you had taken a top-down approach in WebSphere Studio Application Developer Integration Edition, generating an EJB skeleton from a WSDL definition, then follow these steps:

1. Create a new Web project and copy the WSDL file that you would like to generate the EJB skeleton from to this Web project's source folder.
2. Right-click on the WSDL file containing the PortType that you want to generate an EJB skeleton from and select **Web Services** → **Generate Java bean skeleton**.
3. Choose the Web service type **Skeleton EJB Web Service** and complete the wizard.

After completing the wizard, you should have an EJB that implements the service interface and is not dependent on WSIF APIs.

Note that the interface may be slightly different than the 5.1 interface, and you may need to insert an Interface Mediation component in between the 5.1 consumer and the new Import. To do this, click on the **wire** tool in the Assembly Editor and wire the SCA source component to this new **Import with Web Service Binding**. As the interfaces are different, you will be prompted: **Source and target nodes do not have matching interfaces**. Choose to **create an interface mapping between the source and target node**. Double-click on the mapping component that was created in the Assembly Editor. This will open the mapping editor. See the Information Center for instructions on creating an interface mapping.

Once you have completed this, you must rewire the EJB service. There should not be any references, therefore you just need to rewire the Java component's interface:

- If this service is invoked by a business process in the same module, then create a wire from the appropriate business process reference to this EJB component.
- If this service is invoked by a business process in another module, create an **Export with SCA Binding** and from the other module, drag and drop this export onto that module's Assembly Editor to create the corresponding **Import with SCA Binding**. Wire the appropriate business process reference to that Import.
- If this service was published in WebSphere Studio Application Developer Integration Edition to expose it externally, then see the section "Creating SCA Exports to access the migrated service" for instructions on how to republish the service.

#### *Migrating a Business Process to Business Process Service Invocation:*

This scenario applies to a business process that invokes another business process, where the second business process is invoked using a WSIF Process Binding. This section shows how to migrate a BPEL to BPEL service invocation using a wire or an Import/Export with SCA Binding.

To migrate a process (BPEL) binding service project for an outbound service, follow these steps:

1. In the Business Integration perspective, expand the module to see its contents. Open the Assembly Editor by double-clicking the first item under the module project (it will have the same name as the project).
2. There are several scenarios where a BPEL process can invoke another BPEL process. Find the scenario below that applies to your application:

- If the BPEL being invoked is in the same module, create a wire from the appropriate reference on the first BPEL component to the appropriate interface on the target BPEL component.
- If the BPEL being invoked is in another module (where the other module is a migrated service project):
  - a. Create an **Export with SCA Binding** for the second business process in its module assembly diagram.
  - b. Expand the second module's assembly icon in the navigator in the Business Integration view. You should see the export that you just created.
  - c. Drag and drop the export from the Business Integration view under the second module onto the open assembly editor of the first module. This will create an Import with SCA Binding in the first module. If this service was published in WebSphere Studio Application Developer Integration Edition to expose it externally, then see the section, "Creating SCA Exports to access the migrated service".
  - d. Wire the appropriate reference on the first business process to the import that you just created in that module.
  - e. Save the Assembly diagram.
- To achieve late binding when invoking the second business process:
  - a. Leave the first business process component's reference unwired. Open the first process in the BPEL editor and under the **Reference Partners** section, select the partner that corresponds to the second BPEL process to invoke using late binding.
  - b. In the Properties view on the **Description** tab, enter the name of the second business process in the **Process Template** field.
  - c. Save the business process. You have now finished setting up the late bound invocation.

#### *Migrating a Web Service (SOAP/JMS):*

You can migrate a Web Service (SOAP/JMS) to an SCA Import with Web Service binding.

To migrate a SOAP/JMS service project for an outbound service migration, follow these steps:

1. First, you will need to import the service project using the Migration wizard. This will result in the creation of a Business Integration module with the WSDL Messages, PortTypes, Bindings, and Services generated in WebSphere Studio Application Developer Integration Edition. Note that if the IBM Web Service (SOAP/JMS) that this application will invoke is also a WebSphere Studio Application Developer Integration Edition Web service that will be migrated, there may have been updates to that Web service during migration. If this is the case, you should use that Web service's migrated WSDL files here.
2. In the Business Integration perspective, expand the module so that you can see its contents. Open the Assembly Editor by double-clicking the first item under the module project (it will have the same name as the project).
3. Next, add an Import that will allow the application to interact with the IBM Web Service (via SOAP/JMS) according to the SCA programming model. Ensure that the WSDL interface, binding, and service definitions are present in the migrated module or in a library that the migrated module is dependent on.
4. In the Business Integration perspective, expand the migrated module and open its Assembly Diagram in the Assembly Editor.
5. Expand the Web Service Ports logical category and drag and drop the port that corresponds to the service you want to invoke onto the Assembly Editor.
6. Choose to create an **Import with Web Service Binding**.
7. After creating the import, select it in the Assembly Editor and go to the Properties view. Under the Binding tab you will see the port and service that the import is bound to.
8. Save the assembly diagram.

Once you have completed this, you must rewire the service:

- If this service is invoked by a business process in the same module, then create a wire from the appropriate business process reference to this Import.
- If this service is invoked by a business process in another module, create an **Export with SCA Binding** and from the other module, drag and drop this export onto that module's Assembly Editor to create the corresponding **Import with SCA Binding**. Wire the appropriate business process reference to that Import.
- Save the assembly diagram.

*Migrating a Web Service (SOAP/HTTP):*

You can migrate a Web Service (SOAP/HTTP) to an SCA Import with Web Service binding.

To migrate a SOAP/HTTP service project for an outbound service migration, follow these steps:

1. First, you will need to import the service project using the Migration wizard. This will result in the creation of a Business Integration module with the WSDL Messages, PortTypes, Bindings, and Services generated in WebSphere Studio Application Developer Integration Edition. Note that if the IBM Web Service (SOAP/HTTP) that this application will invoke is also a WebSphere Studio Application Developer Integration Edition Web service that will be migrated, there may have been updates to that Web service during migration. If this is the case, you should use that Web service's migrated WSDL files here.
2. In the Business Integration perspective, expand the module so that you can see its contents. Open the Assembly Editor by double-clicking the first item under the module project (it will have the same name as the project).
3. Next, add an Import that will allow the application to interact with the IBM Web Service (via SOAP/HTTP) according to the SCA programming model. Ensure that the WSDL interface, binding, and service definitions are present in the migrated module or in a library that the migrated module is dependent on.
4. In the Business Integration perspective, expand the migrated module and open its Assembly Diagram in the Assembly Editor.
5. Expand the Web Service Ports logical category and drag and drop the port that corresponds to the service you want to invoke onto the Assembly Editor.
6. Choose to create an **Import with Web Service Binding**.
7. After creating the import, select it in the Assembly Editor and go to the Properties view. Under the Binding tab you will see the port and service that the import is bound to.
8. Save the assembly diagram.

Once you have completed this, you must rewire the service:

- If this service is invoked by a business process in the same module, then create a wire from the appropriate business process reference to this Import.
- If this service is invoked by a business process in another module, create an **Export with SCA Binding** and from the other module, drag and drop this export onto that module's Assembly Editor to create the corresponding **Import with SCA Binding**. Wire the appropriate business process reference to that Import.
- Save the assembly diagram.

*Migrating a JMS service:*

You can migrate a JMS service to an SCA Import with JMS binding.

**Note:** If the JMS message is being sent to a WebSphere Business Integration Adapter, then see the section "Migrating Interactions with WebSphere Business Integration Adapter" in the link below.

To migrate a JMS service project for an outbound service migration, follow these steps:

1. First, you will need to import the service project using the Migration wizard. This will result in the creation of a Business Integration module with the WSDL Messages, PortTypes, Bindings, and Services generated in WebSphere Studio Application Developer Integration Edition.
2. In the Business Integration perspective, expand the module so that you can see its contents. Open the Assembly Editor by double-clicking the first item under the module project (it will have the same name as the project).
3. Next, add an Import that will allow the application to interact with a JMS queue according to the SCA programming model.
4. In the Assembly Editor expand the migrated module project and expand the **Interfaces** category and find the WSDL PortType that describes the Web service that the application will invoke. Drag and drop it onto the Assembly Editor.
5. A **Component Creation** dialog will allow you to select the type of component to create. Choose **Import with No Binding**.
6. You will see that a new Import was created in the Assembly Editor and if you select it and go to the Properties view, on the Description tab you can change the import's name and display name to something more meaningful.
7. You can refer to the 5.1 WSDL binding and service files to find details about the JMS service that you are migrating and use them to fill in the details of the 6.0 "Import with JMS Binding". Locate the 5.1 JMS binding and service WSDL files within the 5.1 service project (they are usually named \*JMSBinding.wsdl and \*JMSService.wsdl). Inspect the binding and service information captured there. From the binding, you can determine whether text or object messages were used and whether any custom data format bindings were used. If there were any, you should consider writing a custom data binding for your 6.0 "Import with JMS Binding" as well. From the service, you can find the initial context factory, JNDI connection factory name, JNDI destination name, and destination style (queue).
8. Right-click the import and select **Generate Binding** then **JMS Binding**. You will be prompted to enter the following parameters:

**Select JMS messaging domain:**

- Point-to-Point
- Publish-Subscribe
- Domain-Independent

**Select how data is serialized between Business Object and JMS Message:**

- Text
- Object
- User-supplied

**If User-supplied is selected then:**

Specify fully qualified name of com.ibm.websphere.sca.jms.data.JMSDataBinding implementation class. You should specify a user-defined data binding if your application needs to set any JMS header properties that are not normally available in the JMS Import Binding. In this case, you can create a custom data binding class that extends the standard JMS data binding "com.ibm.websphere.sca.jms.data.JMSDataBinding" and add custom code to access the JMSMessage directly. See the JMS examples in "Creating and modifying bindings for import and export components" from the link below.

**Inbound connectivity is using default JMS function selector class:**

<selected> or <deselected>

9. Select the import that you just created. In the Properties view, go to the Binding tab. You can manually fill in all the binding information listed there to the same values that you specified before in WebSphere Studio Application Developer Integration Edition. The binding information that you may specify is:
  - JMS Import Binding (this is the most important)

- Connection
- Resource Adapter
- JMS Destinations
- Method Bindings

Once you have completed this, you must rewire the service:

- If this service is invoked by a business process in the same module, then create a wire from the appropriate business process reference to this Import.
- If this service is invoked by a business process in another module, create an **Export with SCA Binding** and from the other module, drag and drop this export onto that module's Assembly Editor to create the corresponding **Import with SCA Binding**. Wire the appropriate business process reference to that Import.
- Save the assembly diagram.

*Migrating a J2C-IMS service:*

You can migrate a J2C-IMS service to an SCA Import with EIS Binding or SCA Import with Web Service Binding.

Do not use any of the WebSphere Studio Application Developer Integration Edition artifacts that were generated for this IMS™ service. You will need to recreate the service using the wizards available in WebSphere Integration Developer and manually rewire the application.

**Note:** Turn Auto-Build on or build the module manually.

You have the following options:

**Note:** For both options, note that if a BPEL service invokes this IMS service, the BPEL will need to change slightly, as the interface exposed by the EIS service will be slightly different than the old 5.1 interface. To do this, open the BPEL editor and adjust the partner link that corresponds to the EIS service and use the new interface (WSDL file) generated when performing the steps above. Make any necessary changes to the BPEL activities for the new WSDL interface of the EIS service.

*Pros and cons for each of the J2C-IMS service rewiring options:*

There are pros and cons for each of the J2C-IMS service rewiring options.

The following list describes both options and the pros and cons of each:

- The first option uses the standard SCA componentry to invoke the IMS service.
- The first option has some limitations:
  - The SDO version 1 specification API does not provide access to the COBOL or C byte array – this will impact customers working with IMS multi-segments.
  - The SDO version 1 specification for serialization does not support COBOL redefines or C unions.
- The second option uses the standard JSR 109 approach to connect to the IMS service. This functionality is available as part of Rational Application Developer.

*Create an SCA Import to invoke the IMS service: option 1:*

You can create an SCA Import with EIS Binding that will use DataObjects to store the message/data to communicate with the IMS system.

To create an SCA Import to invoke the IMS service, follow these steps:

1. Create a new business integration module project to house this new IMS service.

2. To recreate the EIS service, go to **File** → **New** → **Other** → **Business Integration** → **Enterprise Service Discovery**.
3. This wizard allows you to import a service from an EIS system. It is very similar to the WebSphere Studio Application Developer Integration Edition wizard that created the WSIF-based EIS service in 5.1. You can import the new J2C IMS resource adapter in this wizard. You should browse to the directory where WebSphere Integration Developer is installed and drill down to **Resource Adapters** → **ims15** → **imsico9102.rar**.

**Note:** See the Information Center for more information on completing the saving properties and operations panels. During the Enterprise Service Discovery wizard, when you add an operation you will also be able to create business objects for the input or output data type of the operation. This requires that you have the C or COBOL source file that you used in the WebSphere Studio Application Developer Integration Edition wizard. These files should have been copied to the old service project so that you can point to the source files there. You can also import the business objects using the separate wizard **File** → **New** → **Other** → **Business Integration** → **Enterprise Data Discovery**.

4. Once you have completed the wizard, open the Business Integration perspective and expand the module so that you can see its contents. You should see new business objects listed under the module's Data Types and new interfaces listed under Interfaces.
5. Open the Assembly Editor by double-clicking the first item under the module project (it will have the same name as the project). You should see that an Import exists on the canvas, this Import has an EIS Binding and it represents the service that you just created.

Now see the section entitled "Creating SCA Exports to access the migrated service" for instructions on how to expose this service to consumers.

*Create a Web service around the J2C service: option 2:*

You can create a J2C Web service and if the consumer of the service is an SCA component, consume the service as an IBM Web Service (SOAP/HTTP or SOAP/JMS).

To create a Web service around the J2C service, follow these steps:

1. Create the J2C Java Bean by clicking **File** → **New** → **J2C** → **J2C Java Bean**
2. Choose the 1.5 version of the **IMS Connector for Java** and click **Next**.
3. Check **Managed Connection** and enter the JNDI lookup name. Click **Next**.
4. Specify the project, package, and name for the new Java bean. The bean consists of an interface and an implementation class. Click **Next**.
5. Add a Java method for each function or service you want to access from the EIS. Additional methods can be added later in the Java source editor through the Snippets View. When you click the **Add...** button, choose the name for the method and click **Next**.
6. Now you can choose **Browse...** to reuse existing types or **New...** to launch the CICS/IMS Java Data Binding Wizard (where you can refer to a COBOL or C source file) for the input and output data types.
7. Once you are finished creating Java methods, Click **Next**.
8. Complete the remaining steps in this wizard to create your J2C Java Bean.
9. Create the Web Service by clicking **File** → **New** → **J2C** → **Web Page, Web Service, or EJB from J2C Java Bean** to create the Web service around your J2C Java Bean.
10. Complete the wizard.

The consumers of this service can now use the WSDL service that is generated by this wizard to invoke the IMS service.

*Migrating a J2C-CICS ECI service:*

You can migrate a J2C-CICS ECI service to an SCA Import with EIS Binding or SCA Import with Web Service Binding.

Follow the instructions in the topic "Migrating a J2C-IMS service", but ensure to import the following RAR file *instead* of the IMS RAR file:

- Browse to the directory where WebSphere Integration Developer is installed and drill down to **Resource Adapters** → **cics15** → **cicseci.rar** .

If you follow the second option to create a J2C Web service, then choose the v1.5 **ECIResourceAdapter** on the second panel of the J2C Java Bean creation wizard.

Also, see the topic "Migrating a J2C-IMS service".

*Migrating a J2C-CICS EPI service:*

There is no direct support for the J2C-CICS EPI service in WebSphere Integration Developer. In order to access this service from an SCA module, you will need to migrate using the *consumption scenario*.

See the topic "The consumption scenario for service migration" for instructions on migrating this service type to WebSphere Integration Developer.

*Migrating a J2C-HOD service:*

There is no direct support for the J2C-HOD service in WebSphere Integration Developer. In order to access this service from an SCA module, you will need to migrate using the *consumption scenario*.

See the topic "The consumption scenario for service migration" for instructions on migrating this service type to WebSphere Integration Developer.

*Migrating a transformer service:*

You can migrate a transformer service to an SCA Data Map and Interface Map where possible. You can also use the *consumption scenario* to access this service from an SCA module.

The data map and interface map components are new in version 6.0. They offer similar function to the transformer service from 5.1 but they do not have the full XSL transform capability. If you are not able to replace your transformer service with one of these components, then you must migrate using the consumption scenario as there is no direct support for the transformer service in WebSphere Integration Developer. Follow the steps documented in the "The consumption scenario for service migration" section to access this service from an SCA module.

*The consumption scenario for service migration:*

In the cases where there is no direct counterpart for a WebSphere Studio Application Developer Integration Edition service type, a consumption scenario is needed to consume the old WebSphere Studio Application Developer Integration Edition service as-is when redesigning the application in WebSphere Integration Developer.

Here are the steps to perform in WebSphere Studio Application Developer Integration Edition *before* invoking the Migration wizard:

1. Create a new Java project to hold this client proxy code. Do not put this client proxy code in the service project because the 5.1-style generated messages and Java bean classes will be skipped by the automatic Migration wizard that migrates service projects.

2. Open WebSphere Studio Application Developer Integration Edition and right-click the WSDL file containing the transformer binding and service and select **Enterprise Services** → **Generate Service Proxy**. You will be asked what type of proxy to create, but only **Web Services Invocation Framework (WSIF)** will be available. Click **Next**.
3. You can now specify the package and name of the service proxy Java class to create (you will create the proxy in the current service project). Click **Next**.
4. You can now specify the proxy style, choose **Client Stub**, select the desired operations to include in the proxy, and click **Finish**. This creates a Java class that exposes the same methods as the WebSphere Studio Application Developer Integration Edition service, where the arguments to the Java methods are the parts of the source WSDL message.

You can now migrate to WebSphere Integration Developer:

1. Copy the client proxy Java project to the new workspace and import it by going to the **File** → **Import** → **Existing Project into Workspace**.
2. Import the service project using the Migration wizard. This will result in the creation of a Business Integration module with the WSDL Messages, PortTypes, Bindings, and Services generated in WebSphere Studio Application Developer Integration Edition.
3. In the Business Integration perspective, expand the module so that you can see its contents. Open the Assembly Editor by double-clicking the first item under the module project (it will have the same name as the project).
4. To create the custom Java component, under the module project, expand **Interfaces** and select the WSDL interface that was generated for this transformer service in WebSphere Studio Application Developer Integration Edition.
5. Drag and drop this interface onto the Assembly Editor. A dialog will pop up asking you to select the type of component to create. Select **Component with No Implementation Type** and click **OK**.
6. A generic component will appear on the Assembly diagram. Select it and go to the **Properties** view.
7. On the **Description** tab, you can change the name and display name of the component to something more descriptive (in this case name it something like your EJB's name but append a postfix such as "JavaMed" as this is going to be a Java component that mediates between the WSDL interface generated for the transformer service in WebSphere Studio Application Developer Integration Edition and the Java interface of the transformer client proxy).
8. On the **Details** tab you will see that this component has one interface - the one that you dragged and dropped onto the Assembly Editor.
9. Back in the Assembly Editor, right-click the component that you just created and select **Generate Implementation...** → **Java** Then select the package where the Java implementation will be generated. This creates a skeleton Java service that adheres to the WSDL interface according to the SCA programming model, where complex types are represented by an object that is a `commonj.sdo.DataObject` and simple types are represented by their Java Object equivalents.

Now you will need to fill in code where you see the `//TODO` tags in the generated Java implementation class. There are two options:

1. Move the logic from the original Java class to this class, adapting it to the new data structure.
2. Create a private instance of the old Java class inside this generated Java class and write code to:
  - a. Convert all parameters of the generated Java implementation class into parameters that the old Java class expects
  - b. Invoke the private instance of the old Java class with the converted parameters
  - c. Convert the return value of the old Java class into the return value type declared by the generated Java implementation method

Once you have completed the above options, you must rewire the client proxy. There should not be any "references", therefore you just need to rewire the Java component's interface:

- If this service is invoked by a business process in the same module, then create a wire from the appropriate business process reference to this Java component's interface.
- If this service is invoked by a business process in another module, create an **Export with SCA Binding** and from the other module, drag and drop this export onto that module's Assembly Editor to create the corresponding **Import with SCA Binding**. Wire the appropriate business process reference to that Import.
- If this service was published in WebSphere Studio Application Developer Integration Edition to expose it externally, then see the section "Creating SCA Exports to access the migrated service" for instructions on how to republish the service.

### Creating SCA Exports to access the migrated service:

An SCA Export must be created to make the migrated service available to external consumers according to the SCA model for all services that deployment code was generated for in the WebSphere Studio Application Developer Integration Edition service project. This includes all services invoked by clients external to the application. **Note:** The migration utility attempts to do this automatically, however, you can refer to the following information to help verify what the tool did.

If from WebSphere Studio Application Developer Integration Edition, you right-clicked the BPEL process or other Service WSDL and selected **Enterprise Services** → **Generate Deploy Code**, you must perform the manual migration steps below. Note that WebSphere Integration Developer is different from WebSphere Studio Application Developer Integration Edition in that it stores all of the deployment options. When the project is built, the deployment code is automatically updated in the generated EJB and Web projects so there is no option to manually **Generate Deploy Code** anymore.

Five binding options were given under the Interfaces for Partners section of the Generate BPEL Deploy Code wizard. The following inbound BPEL service migration information provides more details on the Export type and properties to create based on the deployment binding type(s) that were selected in WebSphere Studio Application Developer Integration Edition:

- EJB
- IBM Web Service (SOAP/JMS)
- IBM Web Service (SOAP/HTTP)
- Apache Web Service (SOAP/HTTP)
- JMS

*Migrating the EJB and the EJB process bindings:*

The EJB and EJB process bindings can be migrated to the recommended SCA construct.

In WebSphere Studio Application Developer Integration Edition this binding type enabled clients to communicate with a BPEL process or other service type by invoking an EJB. Note that this binding type was not optional for microprocesses – it was always selected as the generated EJB was used internally by the other binding types.

The JNDI name of the generated EJB was automatically generated as a combination of the BPEL's name, target namespace, and valid-from timestamp. For example, these attributes can be found by examining the BPEL process's properties in the BPEL editor on the Description and Server content tabs:

*Table 3. Generated namespace*

Process name	MyService
Target namespace	http://www.example.com/process87787141/
Valid From	Jan 01 2003 02:03:04

The generated namespace for this example is then com/example/www/process87787141/MyService20030101T020304.

In WebSphere Studio Application Developer Integration Edition when the EJB binding was selected as the deployment type, there were no options given.

There are four options for migrating the WebSphere Studio Application Developer Integration Edition process binding. The type of client(s) that access the service will determine which migration option(s) below to perform:

**Note:** After the manual migration steps have been completed, the client must be migrated to the new programming model as well. See the appropriate topic for the following client types:

*Table 4. Further information for migrating clients*

Client type	For further information see
EJB client that invokes the generated session bean. Such a client would invoke an EJB method corresponding to the BPEL operation to invoke	"Migrating the EJB client"
WSIF client that uses the EJB process binding	"Migrating the EJB process binding client"
Generic business process choreographer EJB API	"Migrating the business process choreographer generic EJB API client"
Generic business process choreographer Messaging API	"Migrating the business process choreographer generic Messaging API client"
Another BPEL process in the same module	N/A: Wire BPEL components together using Assembly Editor
Another BPEL process in a different module	N/A: Create an <b>Import with SCA Binding</b> in the referencing module, and configure its binding to point to the <b>Export with SCA Binding</b> that you create below in Option 1

It is important to note that if the business process passes a reference to itself outside of its module (via a service reference), you must always follow option 1 below (you can always perform more than one of these options) to create an Export with SCA Binding for the business process. Only one business process per module may pass its service reference outside of the module because its export must be marked as the module default export. This is done by specifying "true" for the attribute called "default" of an export, as in:

Default endpoint reference

You must manually mark this business process's export as the default by right-clicking the Export in the Business Integration view and selecting **Open With** and then selecting **Text Editor**.

*Migration option 1 for the EJB and EJB process binding:*

The first migration option for the WebSphere Studio Application Developer Integration Edition EJB process binding is to make business processes accessible to another component in the same module.

In the Assembly Editor, to wire this other component to the BPEL component, follow these steps:

1. Select the **Wire** item from the toolbar.
2. Click on the other component to select it as the source of the wire.
3. Click the **BPEL SCA** component to select it as the target of the wire.
4. Save the assembly diagram.

*Migration option 2 for the EJB and EJB process binding:*

The second migration option for the WebSphere Studio Application Developer Integration Edition EJB process binding is to make business processes accessible to other SCA modules and clients.

**Note:** These steps are mandatory if the generic business process choreographer APIs will be used to invoke the business process.

The Export with SCA Binding makes an SCA component accessible by other SCA modules. To create an Export with an SCA binding, follow these steps:

1. Open the Assembly Editor for the module created by the migration wizard.
2. Create an Export with SCA Binding for each BPEL process interface that had an EJB binding generated for it in WebSphere Studio Application Developer Integration Edition:
  - a. Right-click the BPEL component in the Assembly Editor.
  - b. Select **Export...**
  - c. Select **SCA Binding**.
  - d. If there are multiple interfaces for the process, select the interface(s) to export with this binding type.
  - e. Once the SCA Export is created, select the export in the Assembly Editor and in the Properties view, select the **Description** content pane. The Export's name and description are listed and may be modified as necessary.
  - f. Save the assembly diagram.

*Migration option 3 for the EJB and EJB process binding:*

The third migration option for the WebSphere Studio Application Developer Integration Edition EJB process binding is to make modules accessible by a non-SCA entity (for example, a JSP or a Java client).

The Standalone Reference makes an SCA component accessible by any external client. To create a Standalone Reference, follow these steps:

1. Open the Assembly Editor for the module created by the Migration wizard.
2. Create a Standalone Reference for each BPEL process interface that had an EJB binding generated for it in WebSphere Studio Application Developer Integration Edition:
  - a. Select the **Standalone References** item from the toolbar.
  - b. Click the canvas of the Assembly Editor to create a Standalone References SCA entity.
  - c. Select the **Wire** item from the toolbar.
  - d. Click the **Standalone References** entity to select it as the source of the wire.
  - e. Click the **BPEL SCA** component to select it as the target of the wire.
  - f. You will see an alert **Matching reference will be created on the source node. Would you like to continue?**, click **OK**.
  - g. Select the **Standalone References** entity that was just created and in the Properties view select the **Description** content pane.
  - h. Expand the **References** link and select the reference that was just created. The reference's name and description are listed and may be modified as necessary.
  - i. If there are multiple interfaces for the process, select the interface(s) to export with this binding type.
  - j. Save the assembly diagram.

*Migration option 4 for the EJB and EJB process binding:*

The fourth migration option for the WebSphere Studio Application Developer Integration Edition EJB process binding is to make business processes accessible by a Web Services client.

The Export with Web service binding makes an SCA component accessible by an external web services client. To create an Export with Web Service binding, follow these steps:

1. Open the Assembly Editor for the module created by the migration wizard.
2. Create an Export with SCA Binding for each BPEL process interface that had an EJB binding generated for it in WebSphere Studio Application Developer Integration Edition:
  - a. Right-click the BPEL component in the Assembly Editor.
  - b. Select **Export...**
  - c. Select **Web Service Binding**
  - d. If there are multiple interfaces for the process, select the interface(s) to export with this binding type.
  - e. Select the transport: **soap/http** or **soap/jms**.
  - f. Once the Web services Export has been created, select the export in the Assembly Editor and in the Properties view, select the **Description** content pane. The Export's name and description are listed and may be modified as necessary.
  - g. Save the assembly diagram.

*Migrating the JMS and the JMS process bindings:*

The JMS and JMS process bindings can be migrated to the recommended SCA construct.

In WebSphere Studio Application Developer Integration Edition, this binding type gave clients the ability to communicate with a BPEL process or other service type by sending a message to an MDB. Note that this binding type was not optional for long-running processes and it was always selected. In fact, this binding type was the **only** binding type allowed for request-response interfaces of long-running processes. For the other service types, an MDB would be generated and it would invoke the appropriate service.

The JNDI name used by the JMS binding was a combination of the BPEL's name, target namespace, and valid-from timestamp.

In WebSphere Studio Application Developer Integration Edition when the JMS binding was selected as the deployment type for a BPEL process, the following options were given:

- **JNDI Connection Factory** - the default is **jms/BPECF** (this is the JNDI name of the target business process container's queue connection factory)
- **JNDI Destination Queue** - the default is **jms/BPEIntQueue** (this is the JNDI name of the target business process container's internal queue)
- **JNDI Provider URL: Server supplied or Custom** - you must enter an address. The default is **iiop://localhost:2809**

There are five options for migrating the WebSphere Studio Application Developer Integration Edition JMS process binding. The type of client(s) that access the service will determine which migration option(s) below to perform:

**Note:** After the manual migration steps have been completed, the client must be migrated to the new programming model as well. See the appropriate topic for the following client types:

*Table 5. Further information for migrating clients*

Client type	For further information see
WSIF Client that uses the JMS process binding	"Migrating the business process choreographer generic Messaging API client and the JMS process binding client"
Generic business process choreographer EJB API	"Migrating the business process choreographer generic EJB API client"

Table 5. Further information for migrating clients (continued)

Client type	For further information see
Generic business process choreographer Messaging API Migrating the business	"Migrating the business process choreographer generic Messaging API client"
Another BPEL process in the same module	N/A: Wire BPEL components together using Assembly Editor
Another BPEL process in a different module	N/A: Create an <b>Import with SCA Binding</b> in the referencing module, and configure its binding to point to the <b>Export with SCA Binding</b> that you create below in Option 1.

It is important to note that if the business process passes a reference to itself outside of its module (via a service reference), you must always follow option 1 below (you can always perform more than one of these options) to create an Export with SCA Binding for the business process. Only one business process per module may pass its service reference outside of the module because its export must be marked as the module default export. This is done by specifying "true" for the attribute called "default" of an export, as in:

Default endpoint reference

You must manually mark this business process's export as the default by right-clicking the Export in the Business Integration view and selecting **Open With** and then select **Text Editor**.

*Migration option 1 for the JMS and JMS process binding:*

The first migration option for the WebSphere Studio Application Developer Integration Edition JMS process binding is to make business processes accessible to another component in the same module.

In the Assembly Editor, to wire this other component to the BPEL component, follow these steps:

1. Select the **Wire** item from the toolbar.
2. Click on the other component to select it as the source of the wire.
3. Click the **BPEL SCA** component to select it as the target of the wire.
4. Save the assembly diagram.

*Migration option 2 for the JMS and JMS process binding:*

The second migration option for the WebSphere Studio Application Developer Integration Edition JMS process binding is to make business processes accessible to other SCA modules and clients.

The Export with SCA Binding makes an SCA component accessible by other SCA modules. To create an Export with an SCA binding, follow these steps:

1. Open the Assembly Editor for the module created by the migration wizard.
2. Create an Export with SCA Binding for each BPEL process interface that had a JMS binding generated for it in WebSphere Studio Application Developer Integration Edition:
  - a. Right-click the BPEL component in the Assembly Editor.
  - b. Select **Export...**
  - c. Select **SCA Binding**.
  - d. If there are multiple interfaces for the process, select the interface(s) to export with this binding type.
  - e. Once the SCA Export is created, select the export in the Assembly Editor and in the Properties view, select the **Description** content pane. The Export's name and description are listed and may be modified as necessary.
  - f. Save the assembly diagram.

*Migration option 3 for the JMS and JMS process binding:*

The third migration option for the WebSphere Studio Application Developer Integration Edition JMS process binding is to make business processes accessible by a non-SCA entity (for example, a JSP or a Java client).

The Standalone Reference makes an SCA component accessible by any external client. To create a Standalone Reference, follow these steps:

1. Open the Assembly Editor for the module created by the migration wizard.
2. Create a Standalone Reference for each BPEL process interface that had a JMS binding generated for it in WebSphere Studio Application Developer Integration Edition:
  - a. Select the **Standalone References** item from the toolbar.
  - b. Click the canvas of the Assembly Editor to create a Standalone References SCA entity.
  - c. Select the **Wire** item from the toolbar.
  - d. Click the **Standalone References** entity to select it as the source of the wire.
  - e. Click the **BPEL SCA** component to select it as the target of the wire.
  - f. You will see an alert **Matching reference will be created on the source node. Would you like to continue?**, click **OK**.
  - g. Select the **Standalone References** entity that was just created and in the Properties view select the **Description** content pane.
  - h. Expand the **References** link and select the reference that was just created. The reference's name and description are listed and may be modified as necessary.
  - i. If there are multiple interfaces for the process, select the interface(s) to export with this binding type.
  - j. Save the assembly diagram.

*Migration option 4 for the JMS and JMS process binding:*

The fourth migration option for the WebSphere Studio Application Developer Integration Edition JMS process binding is to make business processes accessible by a Web services client.

The Export with Web service binding makes an SCA component accessible by an external web services client. To create an Export with Web service binding, follow these steps:

1. Open the Assembly Editor for the module created by the migration wizard.
2. Create an Export with SCA Binding for each BPEL process interface that had a JMS binding generated for it in WebSphere Studio Application Developer Integration Edition:
  - a. Right-click the BPEL component in the Assembly Editor.
  - b. Select **Export...**
  - c. Select **Web Service Binding**
  - d. If there are multiple interfaces for the process, select the interface(s) to export with this binding type.
  - e. Select the transport: **soap/http** or **soap/jms**.
  - f. Once the Web services Export has been created, select the export in the Assembly Editor and in the Properties view, select the **Description** content pane. The Export's name and description are listed and may be modified as necessary.
  - g. Save the assembly diagram.

*Migration option 5 for the JMS and JMS process binding:*

The fifth migration option for the WebSphere Studio Application Developer Integration Edition JMS process binding is to make business processes accessible by a JMS client.

The Export with JMS binding makes an SCA component accessible by an external JMS client. To create an Export with JMS binding, follow these steps:

1. For BPEL services, you will need to create and reference new queue resources, as the 5.1 JMS process binding was quite different from the standard 5.1 JMS binding. For non-BPEL services, you can find the values you selected for the JMS deployment code in WebSphere Studio Application Developer Integration Edition 5.1 by finding the WSDL file named **JMSBinding.wsdl** and **JMSService.wsdl** in the appropriate package underneath the generated EJB project's **ejbModule/META-INF** folder and inspecting the binding and service information captured there. From the binding, you can determine whether text or object messages were used and whether any custom data format bindings were used. If there were any, you should consider writing a custom data binding for your 6.0 **Export with JMS Binding** as well. From the service, you can find the initial context factory, JNDI connection factory name, JNDI destination name, and destination style (queue).
2. Open the Assembly Editor for the module created by the migration wizard.
3. Create an Export with JMS Binding for each BPEL process interface that had a JMS binding generated for it in WebSphere Studio Application Developer Integration Edition by right-clicking the BPEL component in the Assembly Editor.
4. Select **Export...**
5. Select **JMS Binding**
6. If there are multiple interfaces for the process, select the interface(s) to export with this binding type.
7. On the next panel (JMS Export Binding attributes), select **JMS messaging domain**. Define this attribute as **Point-to-Point**.
8. Select **how data is serialized between Business Object and JMS Message** and enter the following values (it is recommended that you select **Text** instead of **Object** because text, which is usually XML, is independent of the runtime and enables service integration between disparate systems):
  - a. For **Text**, select to use the **Default JMS function selector** or enter the fully qualified name of the FunctionSelector implementation class.
  - b. For **Object**, select to use the **Default JMS function selector** or enter the fully qualified name of the FunctionSelector implementation class.
  - c. For **User Supplied**, enter the fully-qualified name of the JMSDataBinding implementation class. You will need to select **User Supplied** if your application needs access to any JMS header properties that are not readily available in the JMS Import Binding. In this case, then you must create a custom data binding class that extends the standard JMS data binding **com.ibm.websphere.sca.jms.data.JMSDataBinding** and add custom code to access the JMSMessage directly. Then you will provide the name of your custom class for this field. See the JMS examples in "Creating and modifying bindings for import and export components" from the link below.
  - d. For **User Supplied**, select to use the **Default JMS function selector** or enter the fully qualified name of the FunctionSelector implementation class.
9. Once the Export with JMS Binding has been created, select the export in the Assembly Editor and in the Properties view, select the **Description** content pane. The Export's name and description are listed and may be modified as necessary.
10. Select the Binding content pane to see many more options.
11. Save the assembly diagram.

#### *Migrating the IBM Web Service binding (SOAP/JMS):*

The IBM Web Service binding (SOAP/JMS) for a BPEL process or other service type can be migrated to the recommended SCA construct.

In WebSphere Studio Application Developer Integration Edition, this binding type gave clients the ability to communicate with a BPEL process or other service type by invoking an IBM Web Service, where the communication protocol was JMS and the message adhered to the SOAP encoding rules.

The following is an example of the conventions used when generating an IBM Web Service (SOAP/JMS) for a 5.1 BPEL service. The JNDI name of the generated IBM Web Service was a combination of the BPEL's name, target namespace, and valid-from timestamp, as well as the name of the interface (WSDL port type that the deployment code was generated for). For example, these attributes can be found by examining the BPEL process' properties in the BPEL editor on the Description and Server content tabs:

Table 6. Generated namespace

Process name	MyService
Target namespace	http://www.example.com/process87787141/
Valid From	Jan 01 2003 02:03:04
Interface	ProcessPortType

The generated namespace for this example is then com/example/www/process87787141/MyService20030101T020304\_ProcessPortTypePT.

In WebSphere Studio Application Developer Integration Edition when the IBM Web Service binding (SOAP/JMS) was selected as the deployment type for a BPEL process or other service type, the following options were given:

- For Document Style, the default was **DOCUMENT** / other option: **RPC**
- For Document Use, the default was **LITERAL** / other option: **ENCODED**
- For JNDI Provider URL, it was either **Server supplied** or **Custom** (an address must be entered, the default is iiop://localhost:2809)
- For Destination Style, the default was **queue** / other option was **topic**
- For JNDI Connection Factory, the default was **jms/qcf** (this is the JNDI name of the queue connection factory for the generated MDB queue)
- For JNDI Destination Queue, the default was **jms/queue** (this is the JNDI name of the generated MDB queue)
- For MDB Listener Port, the default was <Service Project Name>**MdbListenerPort**

A WSDL file specifying the IBM Web Service SOAP/JMS binding and service is created in the generated EJB project but not in the service project itself. This means that you must manually locate that file and copy it to your business integration module project if it is important that the IBM Web Service client code must not change. By default, this WSDL file was created in the EJB project at ejbModule/META-INF/wsdl/<business process name>\_<business process interface port type name>\_JMS.wsdl

The WSDL PortType and Messages of the business process interface are actually copied to this WSDL file as well rather than referencing the existing WSDL PortType and Messages defined in the service project.

If it is important that the IBM Web Service client code remain unchanged after migration, then the information in this file will be needed for the manual migration steps below.

There are two options for migrating the WebSphere Studio Application Developer Integration Edition SOAP/JMS process binding. The choice will have to be made whether to migrate the client to the SCA programming model or to leave it as a web services client:

**Note:** After the manual migration steps have been completed, the client must be migrated to the new programming model as well. See the appropriate topic for the following client types:

Table 7. Further information for migrating clients

Client type	For further information see
IBM Web service client	"Migrating the IBM Web service (SOAP/JMS) client"

Migration option 1 for the IBM Web Service binding (SOAP/JMS):

The first migration option for the WebSphere Studio Application Developer Integration Edition SOAP/JMS binding is to make the service accessible to a Web services client.

The Export with Web Service Binding makes an SCA component accessible by an external Web services client. To create an Export with Web Service Binding, follow these steps:

1. Open the Assembly Editor for the module created by the migration wizard.
2. Create an Export with SCA Binding for each service interface that had an IBM Web Service (SOAP/JMS) binding generated for it in WebSphere Studio Application Developer Integration Edition:
  - a. Right-click the SCA component in the Assembly Editor.
  - b. Select **Export...**
  - c. Select **Web Service Binding**.
  - d. If there are multiple interfaces for the component, select the interface(s) to export with this binding type.
  - e. Select the transport **soap/jms**.
3. Once the Web Services Export is created, select the export in the Assembly Editor and in the Properties view, select the **Description** content pane. The Export's name and description are listed and may be modified as necessary.
4. Save the assembly diagram.
5. Select the Binding content pane and you will see that an IBM Web Service WSDL Binding and Service has been generated directly in the module's project folder. It will be named *component-that-was-exported* Export *WSDL PortType name* Jms\_Service.wsdl. If you inspect that file, you will find that the Document/Literal wrapped binding is used by default, as it is the preferred style in 6.0. This is the WSDL that IBM Web Service clients will use to invoke the service.
6. Follow these steps to generate a new web service binding and service if preserving client code is desired:
  - a. Copy the 5.1 WSDL file from the 5.1 generated EJB project at `ejbModule/META-INF/wsdl/business process name/business process interface port type name`JMS.wsdl to the business integration module project.
  - b. After copying over the file and rebuilding the module, you may see error messages because the XML schema types, WSDL messages, and WSDL port types used by the Web service are duplicated in the IBM Web Service WSDL file in 5.1. To fix this, delete those duplicate definitions from the IBM Web Service binding/service WSDL and in their place add a WSDL import for the real interface WSDL. **Note:** It is important to note that when WebSphere Studio Application Developer Integration Edition generated the IBM Web Service deployment code, it did modify the schema definitions in some cases. This could cause inconsistencies for existing clients that use the IBM Web Service WSDL. For example, the "elementFormDefault" schema attribute was set to "qualified" in the inline schema generated in the IBM Web Service WSDL even if the original schema definition was not qualified. This would cause the following error to be generated during runtime: WSWS3047E: Error: Cannot deserialize element.
  - c. Right-click on this WSDL file you just copied to the business integration module and select **Open With** then **WSDL Editor**.
  - d. Go to the Source tab. Delete all WSDL PortTypes and Messages defined in this file.
  - e. Now you will see the error: The '<portType>' port type specified for the '<binding>' binding is undefined. To fix this, in the WSDL editor in the Graph tab, right-click in the Imports section and select **Add Import**.
  - f. In the Properties view on the General tab, click the ... button to the right of the Location field. Browse to the interface WSDL where the WSDL message and port type definitions are located and click **OK** to import the interface WSDL into the service/binding WSDL.
  - g. Save the WSDL file.
  - h. Refresh/rebuild the project. Switch to the Business Integration perspective. Open the module's Assembly Diagram in the Assembly Editor.

- i. In the project explorer view, expand the module that you are migrating and expand the **Web Service Ports** logical category. You should see the port that exists in the binding/service WSDL listed. Drag and drop it on to the Assembly Editor.
- j. Choose to create an **Export with Web Service Binding** and select the appropriate port name. This will create the Export that uses the old binding/service such that existing Web service clients do not have to change. If you select the export you just created in the Assembly Editor and go to the Properties view, on the Binding tab you should see that the 5.1 port and service names have been filled in for you.
- k. Save all changes.
- l. Just before deploying the application, you can change the generated Web project's configuration to match the 5.1 service address (you have to make these changes every time you make any changes to the SCA module that cause this file to be regenerated). If you look at the IBM Web Service WSDL *Service* definition that you are reusing from 5.1 you will see the service address that the 5.1 client is coded to: `<wsdl:soap:address location="http://localhost:9080/MyServiceWeb/services/MyServicePort"/>`
- m. In order to make the 6.0 generated Web project artifacts match this old service address, you should modify the generated Web project's deployment descriptor. Open the deployment descriptor in WebSphere Integration Developer and on the Servlets tab, add an additional URL Mapping that is very similar to the existing URL mapping for that export, with the same servlet name but a different URL pattern.
- n. Also, if you need to modify the context root of this web project such that it matches the context root in the original service address (in this example the context root is "MyServiceWeb"), then you can open the deployment descriptor for the J2EE Enterprise Application that this web project is in and change the context root of that web module to match the old service address's context root. You may see the following error which you can ignore: `CHKJ3017E: Web Project: <WEB PROJ NAME> is mapped to an invalid Context root: <NEW CONTEXT ROOT> in EAR Project: <APP NAME>`.

*Migration option 2 for the IBM Web Service binding (SOAP/JMS):*

The second migration option for the WebSphere Studio Application Developer Integration Edition SOAP/JMS process binding is to make business processes accessible to a non-SCA entity (for example, JSP or a Java client).

The Standalone Reference makes an SCA component accessible by any external client. To create a Standalone Reference, follow these steps:

1. Open the Assembly Editor for the module created by the migration wizard.
2. Create a Standalone Reference for each BPEL process interface that had an IBM Web Service (SOAP/JMS) binding generated for it in WebSphere Studio Application Developer Integration Edition:
  - a. Select the **Standalone References** item from the toolbar.
  - b. Click the canvas of the Assembly Editor to create a Standalone References SCA entity.
  - c. Select the **Wire** item from the toolbar.
  - d. Click the **Standalone References** entity to select it as the source of the wire.
  - e. Click the **BPEL SCA** component to select it as the target of the wire.
  - f. You will see an alert **Matching reference will be created on the source node. Would you like to continue?**, click **OK**.
  - g. Select the **Standalone References** entity that was just created and in the Properties view select the **Description** content pane.
  - h. Expand the **References** link and select the reference that was just created. The reference's name and description are listed and may be modified as necessary.
  - i. If there are multiple interfaces for the process, select the interface(s) to export with this binding type.
  - j. Save the assembly diagram.

### *Migrating the IBM Web Service binding (SOAP/HTTP):*

The IBM Web Service binding (SOAP/HTTP) for a BPEL process or other service type can be migrated to the recommended SCA construct.

In WebSphere Studio Application Developer Integration Edition, this binding type gave clients the ability to communicate with a BPEL process or other service type by invoking an IBM Web Service, where the communication protocol was HTTP and the message adhered to the SOAP encoding rules.

The following is an example of the conventions used when generating an IBM Web Service (SOAP/HTTP) for a 5.1 BPEL service. The JNDI name of the generated IBM Web Service was a combination of the BPEL's name, target namespace, and valid-from timestamp, as well as the name of the interface (WSDL port type that the deployment code was generated for). For example, these attributes can be found by examining the BPEL process' properties in the BPEL editor on the Description and Server content tabs:

*Table 8. Generated namespace*

Process name	MyService
Target namespace	http://www.example.com/process87787141/
Valid From	Jan 01 2003 02:03:04
Interface	ProcessPortType

The generated namespace for this example is then `com/example/www/process87787141/MyService20030101T020304_ProcessPortTypePT`.

In WebSphere Studio Application Developer Integration Edition when the IBM Web Service binding (SOAP/HTTP) was selected as the deployment type for a BPEL process or other service type, the following options were given:

- For Document Style, the default was **RPC / other option: DOCUMENT**
- For Document Use, the default was **ENCODED / other option: LITERAL**
- For Router Address, the default was **http://localhost:9080**

A WSDL file specifying the IBM Web Service SOAP/HTTP binding and service is created in the generated Web and EJB projects but not in the service project itself. This means that you must manually locate that file and copy it to your business integration module project if it is important that the IBM Web Service client code must not change. By default, this WSDL file was created in the Web project at `WebContent/WEB-INF/wsdl/<business process name>_<business process interface port type name>_HTTP.wsdl`

The WSDL PortType and Messages of the business process interface are actually copied to this WSDL file as well rather than referencing the existing WSDL PortType and Messages defined in the service project.

If it is important that the IBM Web Service client code remain unchanged after migration, then the information in this file will be needed for the manual migration steps below.

There are two options for migrating the WebSphere Studio Application Developer Integration Edition SOAP/HTTP process binding. The choice will have to be made whether to migrate the client to the SCA programming model or to leave it as a Web services client:

**Note:** After the manual migration steps have been completed, the client must be migrated to the new programming model as well. See the appropriate topic for the following client types:

Table 9. Further information for migrating clients

Client type	For further information see
IBM Web service client	"Migrating the IBM Web service (SOAP/HTTP) client"

Migration option 1 for the IBM Web Service (SOAP/HTTP) binding:

The first migration option for the WebSphere Studio Application Developer Integration Edition SOAP/HTTP process binding is to make business processes accessible to a Web services client.

The Export with Web Service Binding makes an SCA component accessible by an external Web services client. To create an Export with Web Service Binding, follow these steps:

1. Open the Assembly Editor for the module created by the Migration wizard.
2. Create an Export with SCA Binding for each BPEL process interface that had an IBM Web Service (SOAP/HTTP) binding generated for it in WebSphere Studio Application Developer Integration Edition by right-clicking the BPEL component in the Assembly Editor.
3. Select **Export...**
4. Select **Web Service Binding**.
5. If there are multiple interfaces for the component, select the interface(s) to export with this binding type.
6. Select the transport **soap/http**.
7. Once the Web Services Export is created, select the export in the Assembly Editor and in the Properties view, select the **Description** content pane. The Export's name and description are listed and may be modified as necessary.
8. Save the assembly diagram.
9. Follow these steps to generate a new web service binding and service if preserving client code is desired:
  - a. Copy the 5.1 WSDL file from the 5.1 generated EJB project at `ejbModule/META-INF/wsdl/business process name/business process interface port type name_HTTP.wsd` to the business integration module project.
  - b. After copying over the file and rebuilding the module, you may see error messages as the XML schema types, WSDL messages, and WSDL port types used by the Web service are duplicated in the IBM Web Service WSDL file in 5.1. To fix this, delete those duplicate definitions from the IBM Web Service binding/service WSDL and in their place add a WSDL import for the real interface WSDL. **Note:** It is important to note that when WebSphere Studio Application Developer Integration Edition generated the IBM Web Service deployment code, it did modify the schema definitions in some cases. This could cause inconsistencies for existing clients that use the IBM Web Service WSDL. For example, the "elementFormDefault" schema attribute was set to "qualified" in the inline schema generated in the IBM Web Service WSDL even if the original schema definition was not qualified. This would cause the following error to be generated during runtime: `WSWS3047E: Error: Cannot deserialize element`.
  - c. Right-click on this WSDL file you just copied to the business integration module and select **Open With** then **WSDL Editor**.
  - d. Go to the Source tab. Delete all WSDL PortTypes and Messages defined in this file.
  - e. Now you will see the error: The '<portType>' port type specified for the '<binding>' binding is undefined. To fix this, in the WSDL editor in the Graph tab, right-click in the Imports section and select **Add Import**.
  - f. In the Properties view on the General tab, click the ... button to the right of the Location field. Browse to the interface WSDL where the WSDL message and port type definitions are located and click **OK** to import the interface WSDL into the service/binding WSDL.
  - g. Save the WSDL file.

- h. Refresh/rebuild the project. Switch to the Business Integration perspective. Open the module's Assembly Diagram in the Assembly Editor.
- i. In the project explorer view, expand the module that you are migrating and expand the **Web Service Ports** logical category. You should see the port that exists in the binding/service WSDL listed. Drag and drop it on to the Assembly Editor.
- j. Choose to create an **Export with Web Service Binding** and select the appropriate port name. This will create the Export that uses the old binding/service such that existing Web service clients do not have to change. If you select the export you just created in the Assembly Editor and go to the Properties view, on the Binding tab you should see that the 5.1 port and service names have been filled in for you.
- k. Save all changes.
- l. Just before deploying the application, you can change the generated Web project's configuration to match the 5.1 service address (you have to make these changes every time you make any changes to the SCA module that cause this file to be regenerated). If you look at the IBM Web Service WSDL service definition that you are reusing from 5.1, you will see the service address that the 5.1 client is coded to: `<wsdl:soap:address location="http://localhost:9080/MyServiceWeb/services/MyServicePort"/>`
- m. In order to make the 6.0 generated Web project artifacts match this old service address, you should modify the generated Web project's deployment descriptor. Open the deployment descriptor in WebSphere Integration Developer and on the Servlets tab, add an additional URL Mapping that is very similar to the existing URL mapping for that export, with the same servlet name but a different URL pattern.
- n. Also, if you need to modify the context root of this web project such that it matches the context root in the original service address (in this example the context root is "MyServiceWeb"), then you can open the deployment descriptor for the J2EE Enterprise Application that this web project is in and change the context root of that web module to match the old service address's context root. You may see the following error which you can ignore: `CHKJ3017E: Web Project: <WEB PROJ NAME> is mapped to an invalid Context root: <NEW CONTEXT ROOT> in EAR Project: <APP NAME>.`

*Migration option 2 for the IBM Web Service (SOAP/HTTP) binding:*

The second migration option for the WebSphere Studio Application Developer Integration Edition SOAP/HTTP process binding is to make business processes accessible to a non-SCA entity (for example, JSP or a Java client).

The Standalone Reference makes an SCA component accessible by any external client. To create a Standalone Reference, follow these steps:

1. Open the Assembly Editor for the module created by the migration wizard.
2. Create a Standalone Reference for each interface that had an IBM Web Service (SOAP/HTTP) binding generated for it in WebSphere Studio Application Developer Integration Edition:
  - a. Select the **Standalone References** item from the toolbar.
  - b. Click the canvas of the Assembly Editor to create a Standalone References SCA entity.
  - c. Select the **Wire** item from the toolbar.
  - d. Click the **Standalone References** entity to select it as the source of the wire.
  - e. Click the **SCA** component to select it as the target of the wire.
  - f. You will see an alert **Matching reference will be created on the source node. Would you like to continue?**, click **OK**.
  - g. Select the **Standalone References** entity that was just created and in the Properties view select the **Description** content pane.
  - h. Expand the **References** link and select the reference that was just created. The reference's name and description are listed and may be modified as necessary.

- i. If there are multiple interfaces for the process, select the interface(s) to export with this binding type.
- j. Save the assembly diagram.

#### *Migrating the Apache Web Service binding (SOAP/HTTP):*

The Apache Web Service binding (SOAP/HTTP) for a BPEL process or other service type can be migrated to the recommended SCA construct.

In WebSphere Studio Application Developer Integration Edition, this binding type gave clients the ability to communicate with a BPEL process or other service type by invoking an Apache Web Service.

In WebSphere Studio Application Developer Integration Edition when the Apache Web Service binding was selected as the deployment type for a BPEL process or other service type, the following options were given:

- For Document Style, it was **RPC** (no other option available)
- For SOAP action , it was **URN:WSDL PortType name**
- For Address, it was `http://localhost:9080/Service Project NameWeb/servlet/rpcrouter`
- For Use Encoding, the default was **yes** (If **yes**, then Encoding Style was set to: `http://schemas.xmlsoap.org/soap/encoding/`)

A WSDL file specifying the Apache SOAP binding and service is created in the service project. By default it is created in the same directory as the service it is wrapping with the name `<business process name>_<business process interface port type name>_SOAP.wsdl`. The WSDL PortType and Messages of the business process interface are used by this binding and service directly. After migration, you should *not* use this WSDL for anything aside from perhaps using the same namespace, port, and service names in the new WSDL that will be generated for you in Version 6.

There are two options for migrating the WebSphere Studio Application Developer Integration Edition Web Service process binding. The choice will have to be made whether to migrate the client to the SCA programming model or to leave it as an IBM Web Services programming model. There is no binding that is equivalent to the Apache Web Service (SOAP/HTTP) binding type anymore in the 6 SCA programming model.

You should migrate this Apache Web service to use the IBM Web Service engine. See the topic "Migrating the IBM Web Service (SOAP/HTTP) binding" for instructions on how to perform this migration and create an IBM Web Service (SOAP/HTTP).

#### **Migrating to the SCA programming model:**

For any free-form Java code that interacts with a WebSphere Studio Application Developer Integration Edition service, this section will show how to migrate from the WSIF programming model to the new SCA programming model where the data flowing through the application is stored in Eclipse Service Data Objects (SDOs). This section will also show you how to manually migrate the most common client types to the new programming model.

For any BPEL processes that contain Java snippets, this section explains how to migrate from the old Java snippet API to the new Java snippet API where the data flowing through the application is stored in Eclipse Service Data Objects (SDOs). Whenever possible, the snippets are migrated automatically by the migration wizard but there are snippets that the migration wizard can not fully migrate, meaning manual steps are required to complete the migration.

Here is a summary of the programming model changes:

#### **V5.1 Programming Model**

1. WSIF and WSDL based
2. Generated proxies for services
3. Beans and format handlers for types

#### V6.0 Programming Model (more Java-centric)

1. SCA services based on SDOs with doclet tags
2. Interface bindings for services
3. SDOs and Databindings for types

*Migrating WSIFMessage API calls to SDO APIs:*

The following section details how to migrate from the old WebSphere Business Integration Server Foundation Version 5.1 programming model where the data flowing through the application is represented as WSIFMessage objects with a generated interface that was strongly-typed to the new WebSphere Process Server Version 6.0 programming model where the data is represented as Service Data Objects (SDOs) and no strongly-typed interface is generated.

*Table 10. Changes and Solutions for migrating WSIFMessage API calls to SDO APIs*

Change	Solution
WSIFMessage-based wrapper classes are no longer generated for WSDL message types, nor are the Java bean helper classes generated for complex schema types.	When writing code that interacts with SCA services, the generic SDO APIs must be used to manipulate the commonj.sdo.DataObject messages that hold the data that flows through the application.  WSDL message definitions that have a single simple-typed part will now be represented by a simple Java type that directly represents the part instead of having a wrapper around the actual data. If the single message part is a complex type, then the data is represented as a DataObject that adheres to the complex type definition.  WSDL message definitions that have multiple parts now correspond to a DataObject that has properties for all of the message parts, where complexTypes are represented as "reference-type" properties of the parent DataObject, accessible via the getDataObject and setDataObject methods.
Strongly-typed getter methods for WSIFMessage parts and generated Java beans should not be used.	Weakly-typed SDO API should be used to get the DataObject properties.
Strongly-typed setter methods for BPEL variables' message parts are no longer available.	Weakly-typed SDO API must be used to set the DataObject properties.
Weakly-typed getter methods for WSIFMessage properties should no longer be used.	Weakly-typed SDO API must be used to set the DataObject properties.
Weakly-typed setter methods for WSIFMessage properties should no longer be used.	Weakly-typed SDO API must be used to set the DataObject properties.
All WSIFMessage API calls should be migrated to the SDO API where possible.	Migrate the call to an equivalent SDO API call where possible. Redesign logic if not possible.

*Migrating WebSphere Business Integration Server Foundation client code:*

This section shows how to migrate the various client types that were possible for the WebSphere Business Integration Server Foundation 5.1 service types.

*Migrating the EJB client:*

This topic shows how to migrate clients that use an EJB interface to invoke a service.

1. Drag and drop the Export with SCA Binding from the migrated module onto this new module's Assembly Editor. This will create an Import with SCA Binding. In order for a client to obtain a reference to this import, a Standalone Reference must be created.
2. On the palette, select the Standalone References item. Click the Assembly Editor canvas once to create a new standalone reference for this new module.
3. Select the wire tool and click the service reference and then click **Import**.
4. Click **OK** when alerted that a matching reference will be created on the source node.
5. You will be asked: **It is easier for a Java client to use a Java interface with this reference – would you like to convert the WSDL reference to a compatible Java reference?:**
  - a. Answer **Yes** if you would like the client to look up this service and cast it as a Java class to invoke it using a Java interface. This new Java interface takes the name of the WSDL PortType, where the package of the interface is derived from the namespace of the WSDL PortType. There is a method defined for each operation defined on the WSDL PortType, and each WSDL message part is represented as an argument to the interface methods.
  - b. Answer **No** if you would like the client to look up this service and use the generic `com.ibm.websphere.sca.Service` interface to invoke it using the `invoke` operation as a generic SCA service.
6. Rename the standalone reference to a more meaningful name if appropriate by selecting the Standalone References component in the Assembly Editor. Go to the Properties view, to the Details tab, drilling down to and selecting the reference that was just created, and modifying the name. Remember the name you chose for this reference because the client will need to use this name when invoking the `locateService` method of the `com.ibm.websphere.sca.ServiceManager` instance.
7. Click **Save** to save the Assembly diagram.

The client must have this new module on its local classpath in order to access the migrated EJB module running on the server.

The following shows what the client code looks like for a service of type "CustomerInfo":

```
// Create a new ServiceManager
ServiceManager serviceManager = ServiceManager.INSTANCE;

// Locate the CustomerInfo service
CustomerInfo customerInfoService = (CustomerInfo) serviceManager.locateService
("<name-of-standalone-reference-from-previous-step");

// Invoke the CustomerInfo service
System.out.println(" [getMyValue] getting customer info...");
DataObject customer = customerInfoService.getCustomerInfo(customerID);
```

The client must change how the message is constructed. In the messages were based on the `WSIFMessage` class but now they should be based on the `commonj.sdo.DataObject` class.

*Migrating the EJB process binding client:*

This topic shows how to migrate clients that use the WSIF EJB process binding to access a BPEL service.

Clients that used the EJB Process Binding to invoke a business process should now use either the SCA API to invoke the service (the migrated business process must have an Export with SCA Binding) or the IBM Web Service Client API to invoke the service (the migrated business process must have an Export with Web Service Binding).

See the topics "Migrating the EJB Client", "Migrating the IBM Web Service (SOAP/JMS) client", or "Migrating the IBM Web Service (SOAP/HTTP) client" for more information on generating such clients.

*Migrating the IBM Web Service (SOAP/JMS) client:*

This topic shows how to migrate clients that use Web Service APIs (SOAP/JMS) to invoke a service.

No migration is needed for existing clients during migration. Note that you must manually modify the generated Web project (create a new servlet mapping) and sometimes have to modify the Web project's context root in the enterprise application deployment descriptor to publish the service to the exact same address that it was published to on WebSphere Business Integration Server Foundation. See the topic "Migrating the IBM Web Service binding (SOAP/JMS)".

It is important to note that unlike 5.1 where a WSIF or RPC client proxy could be generated, in 6.0 the tools only support RPC client generation because RPC is the 6.0 preferred API over the WSIF API.

**Note:** To generate new client proxy from WebSphere Integration Developer, you must have a WebSphere Process Server or WebSphere Application Server installed.

1. Ensure that you have a WebSphere Process Server or WebSphere Application Server installed.
2. In the Resources or Java perspective, find the WSDL file corresponding to the **Export with Web Service Binding** then right-click and select **Web Services** → **Generate Client** (Note that this wizard is very similar to the 5.1 wizard).
3. For Client Proxy Type choose **Java proxy** and click **Next**.
4. The location of the WSDL should be filled in. Click **Next**.
5. Next you must select the appropriate options to specify your client environment configuration including the Web service runtime and server, J2EE version, client type (Java, EJB, Web, Application Client). Click **Next**.
6. Finish the remaining steps to generate the client proxy.

*Migrating the IBM Web Service (SOAP/HTTP) client:*

This topic shows how to migrate clients that use Web Service APIs (SOAP/HTTP) to invoke a service.

No migration is needed for existing clients during migration. Note that you must manually modify the generated Web project (create a new servlet mapping) and sometimes have to modify the Web project's context root in the enterprise application deployment descriptor to publish the service to the exact same address that it was published to on WebSphere Business Integration Server Foundation. See the topic "Migrating the IBM Web Service binding (SOAP/HTTP)".

If design changes have occurred and you would like to generate a new client proxy, the following steps will show you how to do that. It is important to note that unlike 5.1 where a WSIF or RPC client proxy could be generated, in 6.0 the tools only support RPC client generation because RPC is the 6.0 preferred API over the WSIF API.

**Note:** To generate new client proxy from WebSphere Integration Developer, you must have a WebSphere Process Server or WebSphere Application Server installed.

1. Ensure that you have a WebSphere Process Server or WebSphere Application Server installed.
2. Select the WSDL file corresponding to the **Export with Web Service Binding** then right-click and select **Web Services** → **Generate Client** (Note that this wizard is very similar to the 5.1 wizard).
3. For Client Proxy Type choose **Java proxy** and click **Next**.
4. The location of the WSDL should be filled in. Click **Next**.
5. Next you must select the appropriate options to specify your client environment configuration including the Web service runtime and server, J2EE version, client type (Java, EJB, Web, Application Client). Click **Next**.
6. Finish the remaining steps to generate the client proxy.

*Migrating the Apache Web Service (SOAP/HTTP) client:*

The Apache Web Service client APIs are not appropriate for invoking a WebSphere Integration Developer service. Client code must be migrated to use the IBM Web Service (SOAP/HTTP) client APIs.

See the topic, "Migrating the IBM Web Service (SOAP/HTTP) client" for more information.

In 5.1 if a client proxy was automatically generated, that proxy used WSIF APIs to interact with the service. In 6.0 the tools only support RPC client generation because RPC is the 6.0 preferred API over the WSIF API.

**Note:** To generate new client proxy from WebSphere Integration Developer, you must have a WebSphere Process Server or WebSphere Application Server installed.

1. Ensure that you have a WebSphere Process Server or WebSphere Application Server installed.
2. Select the WSDL file corresponding to the **Export with Web Service Binding** then right-click and select **Web Services** → **Generate Client** (Note that this wizard is very similar to the 5.1 wizard).
3. For Client Proxy Type choose **Java proxy** and click **Next**.
4. The location of the WSDL should be filled in. Click **Next**.
5. Next you must select the appropriate options to specify your client environment configuration including the Web service runtime and server, J2EE version, client type (Java, EJB, Web, Application Client). Click **Next**.
6. Finish the remaining steps to generate the client proxy.

*Migrating the JMS client:*

Clients that communicated with a 5.1 service via the JMS API (sending a JMS message to a queue) may require manual migration. This topic shows how to migrate clients that use JMS APIs (sending a JMS message to a queue) to invoke a service.

You must ensure that the **Export with JMS Binding** that you created in a previous step will be able to accept this text or object message with no changes. You may need to write a custom data binding to accomplish this. See the section "Migrating the JMS and the JMS process bindings" for more information.

The client must change how the message is constructed. The messages were previously based on the WSIFMessage class but now they should be based on the commonj.sdo.DataObject class. See the section "Migrating WSIFMessage API calls to SDO APIs" for more details on how to do this manual migration.

*Migrating the business process choreographer generic EJB API client:*

This topic shows how to migrate clients that use the 5.1 process choreographer generic EJB API to invoke a BPEL service.

There is a new version of the Generic EJB API that uses DataObjects as its message format. The client must change how the message is constructed. The messages were previously based on the WSIFMessage class but now they should be based on the commonj.sdo.DataObject class. Note that the Generic EJB API has not changed significantly, as the ClientObjectWrapper still provides a message wrapper around the particular message format.

```
Ex: DataObject dobj = myClientObjectWrapper.getObject();  
String result = dobj.getInt("resultInt");
```

The JNDI name of the old Generic EJB that takes WSIFMessage objects is:

```
GenericProcessChoreographerEJB  
JNDI Name: com/ibm/bpe/api/BusinessProcessHome  
Interface: com.ibm.bpe.api.BusinessProcess
```

There are two generic EJBs in 6.0 as the human task operations are now available as a separate EJB. The 6.0 JNDI names of these Generic EJBs are:

GenericBusinessFlowManagerEJB  
JNDI Name: com/ibm/bpe/api/BusinessFlowManagerHome  
Interface: com.ibm.bpe.api.BusinessFlowManager

HumanTaskManagerEJB  
JNDI Name: com/ibm/task/api/TaskManagerHome  
Interface: com.ibm.task.api.TaskManager

*Migrating the business process choreographer generic Messaging API client and the JMS process binding client:*

There is no generic messaging API in WebSphere Process Server 6.0. See the section "Migrating the JMS and JMS process bindings" to choose a different way to expose the business process to consumers and rewrite the client according to the chosen binding.

*Migrating the business process choreographer Web client:*

This topic shows how to migrate the 5.1 process choreographer Web client settings and custom JSPs.

The Migration wizard preserves the 5.1 Web client settings and in the Human Task Editor, you may not edit these settings. You should create new Web client settings and JSPs using the WebSphere Integration Developer 6.0.

### **Migrating Web Client modifications**

In 5.1 you could modify the look and feel of the Struts-based Web client by modifying its JSP **Header.jsp** and style sheet **dwc.css**.

Since the 6.0 Web client (renamed to the **Business Process Choreographer Explorer**) is based on Java Server Faces (JSF) instead of Struts, automatic migration of Web client modifications is not possible. Therefore, it is recommended that you see the "Business Process Choreographer Explorer" documentation for details on customizing the 6.0 version of this application.

User-defined JSPs could be defined for business processes and for staff activities. The Web client uses these JSPs to display input and output messages for the process and activities.

These JSPs are particularly useful when:

1. Messages have non-primitive parts to enhance the usability of the message's data structure.
2. You want to extend the Web client's capabilities.

There are more and different options available when specifying Web client settings for a 6.0 process, so you will have to use WebSphere Integration Developer to redesign the Web client settings for migrated processes and activities:

1. Select the process canvas or an activity in the process.
2. In the Properties view, select the **Client** tab to redesign the Web client settings.
3. Manually migrate any user-defined JSP:
  - a. See the "Migrating to the SCA programming model" section for programming model changes.
  - b. The Web client uses the Generic APIs to interact with business processes. See the sections that show how to migrate calls to these generic APIs..
4. Specify the name of the new JSP in the 6.0 Web client settings for the process

**Note:** Mapping JSPs are not needed with the 6.0 Business Process Choreographer Explorer because DataObjects do not need any custom mapping.

*Migrating WebSphere Business Integration Server Foundation BPEL Java snippets:*

For any BPEL processes that contain Java snippets, this section details how to migrate from the old Java snippet API to the new Java snippet API where the data flowing through the application is stored as Eclipse Service Data Objects (SDOs).

See the section "Migrating from the WSIFMessage API calls to SDO APIs" for migration steps to perform specific to the WSIFMessage to SDO transition.

Whenever possible, the snippets are automatically migrated by the migration wizard but there are snippets that the migration wizard can not fully migrate. This requires extra manual steps to complete the migration. See the Limitations topic for details on the types of Java snippets that must be migrated manually. Whenever one of these snippets is encountered, the Migration wizard will explain why it can not be automatically migrated and issue a warning or error message.

The following table detail the changes in the BPEL Java snippet programming model and API from Process Choreographer version 5.1 to 6.0:

*Table 11. Changes and solutions for migrating WebSphere Business Integration Server Foundation BPEL Java snippets*

Change	Solution
WSIFMessage-based wrapper classes are no longer generated for WSDL message types, nor are the Java bean helper classes generated for complex schema types.	<p>BPEL variables can be directly accessed by name. Note that for BPEL variables whose WSDL message definition has a single-part, these variables will now directly represent the part instead of having a wrapper around the actual data. Variables whose message type has multiple parts will have a DataObject wrapper around the parts (where the wrapper in WebSphere Application Developer Integration Edition was a WSIFMessage).</p> <p>Because BPEL variables can be used directly in 6.0 snippets, there is less need for local variables than there was in 5.1.</p> <p>The strongly-typed getters for the BPEL variables implicitly initialized the WSIFMessage wrapper object around the message parts. There is no 'wrapper' object for BPEL variables whose WSDL message definition has only a single part: in this case the BPEL variables directly represent the part (In the case where the single part is an XSD simple type, the BPEL variable will be represented as the Java object wrapper type such as java.lang.String, java.lang.Integer, etc). BPEL variables with multi-part WSDL message definitions are handled differently: there is still a wrapper around the parts and this DataObject wrapper must be explicitly initialized in the 6.0 Java snippet code if it has not already been set by a previous operation.</p> <p>If any local variables from the 5.1 snippets had the same name as the BPEL variable there may be conflicts so try to remedy this situation if possible.</p>
WSIFMessage objects are no longer used to represent BPEL variables.	If any custom Java classes invoked from the Java snippets have a WSIFMessage parameter it will need to be migrated such that it accepts/returns a DataObject.
Strongly-typed getter methods for BPEL variables are no longer available.	The variables can be directly accessed by name. Note that for BPEL variables whose WSDL message definition has a single-part will now directly represent the part instead of having a wrapper around the actual data. Variables whose message type has multiple parts will have a DataObject wrapper around the parts (where the wrapper in WebSphere Application Developer Integration Edition was a WSIFMessage).
Strongly-typed setter methods for BPEL variables are no longer available.	The variables can be directly accessed by name. Note that for BPEL variables whose WSDL message definition has a single-part, these variables will now directly represent the part instead of having a wrapper around the actual data. Variables whose message type has multiple parts will have a DataObject wrapper around the parts (where the wrapper in WebSphere Application Developer Integration Edition was a WSIFMessage).

Table 11. Changes and solutions for migrating WebSphere Business Integration Server Foundation BPEL Java snippets (continued)

Change	Solution
<p>Weakly-typed getter methods for BPEL variables that return a WSIFMessage are no longer available.</p>	<p>The variables can be directly accessed by name. Note that for BPEL variables whose WSDL message definition has a single-part, these variables will now directly represent the part instead of having a wrapper around the actual data. Variables whose message type has multiple parts will have a DataObject wrapper around the parts (where the wrapper in WebSphere Application Developer Integration Edition was a WSIFMessage).</p> <p>Note that there were two variations of the <code>getVariableAsWSIFMessage</code> method:</p> <pre>getVariableAsWSIFMessage(String variableName) getVariableAsWSIFMessage(String variableName, boolean forUpdate)</pre> <p>For a Java snippet activity, the default access is read-write. You can change this to read-only by specifying <code>@bpe.readOnlyVariables</code> with the list of names of the variables in a comment in the snippet. For example, you could set variable B and variable D to read only as follows:</p> <pre>variableB.setString("/x/y/z", variableA.getString("/a/b/c")); // @bpe.readOnlyVariables names="variableA" variableD.setInt("/x/y/z", variableC.getInt("/a/b/c")); // @bpe.readOnlyVariables names="variableC"</pre> <p>Additionally, if you have a Java snippet in a condition, the variables are read-only by default, but you can make them read-write by specifying <code>@bpe.readWriteVariables...</code></p>
<p>Weakly-typed setter methods for BPEL variables are no longer available.</p>	<p>The variables can be directly accessed by name. Note that for BPEL variables whose WSDL message definition has a single-part, these variables will now directly represent the part instead of having a wrapper around the actual data. Variables whose message type has multiple parts will have a DataObject wrapper around the parts (where the wrapper in WebSphere Application Developer Integration Edition was a WSIFMessage).</p>
<p>Weakly-typed getter methods for BPEL variables message parts are not appropriate for single-part messages and have changed for multi-part messages.</p>	<p>Migrate to the weakly-typed getter method for BPEL variables (DataObject's) properties.</p> <p>Note that for BPEL variables whose WSDL message definition has a single-part, the BPEL variable directly represents the part and the variable should be accessed directly without using a getter method.</p> <p>There were two variations of the <code>getVariablePartAsObject</code> method:</p> <pre>getVariablePartAsObject(String variableName, String partName) getVariablePartAsObject(String variableName, String partName, boolean forUpdate)</pre> <p>For multi-part messages, equivalent functionality is provided by this method in 6.0:</p> <pre>getVariableProperty(String variableName, QName propertyName);</pre> <p>In 6.0 there is no notion of using a variable for read-only access (which was the case in 5.1 for the first method above as well as the second method with <code>forUpdate='false'</code>). The variable is directly used in the 6.0 snippet and it is always able to be updated.</p>

Table 11. Changes and solutions for migrating WebSphere Business Integration Server Foundation BPEL Java snippets (continued)

Change	Solution
Weakly-typed setter methods for BPEL variables' message parts are not appropriate for single-part messages and have changed for multi-part messages.	<p>Migrate to the weakly-typed setter method for BPEL variables' (DataObject's) properties.</p> <p>Note that for BPEL variables whose WSDL message definition has a single-part, the BPEL variable directly represents the part and the variable should be accessed directly without using a setter method.</p> <p>Calls to the following method must be migrated:  <code>setVariableObjectPart(String variableName, String partName, Object data)</code></p> <p>For multi-part messages, equivalent functionality is provided by this method in 6.0:  <code>setVariableProperty(String variableName, QName propertyName, Serializable value);</code></p>
Strongly-typed getter methods for BPEL partner links are no longer available.	Migrate to the weakly-typed getter methods for BPEL partner links.
Strongly-typed setter methods for BPEL partner links are no longer available.	Migrate to the weakly-typed setter methods for BPEL partner links.
Strongly-typed getter methods for BPEL correlation sets are no longer available.	<p><b>V5.1 Snippet:</b>  <code>String corrSetPropStr =  getCorrelationSetCorrSetAPropertyCustomerName();  int corrSetPropInt =  getCorrelationSetCorrSetBPropertyCustomerId();</code></p> <p><b>V6.0 Snippet:</b>  <code>String corrSetPropStr = (String) getCorrelationSetProperty  ("CorrSetA", new QName("CustomerName"));  int corrSetPropInt = ((Integer) getCorrelationSetProperty  ("CorrSetB", new QName("CustomerId"))).intValue();</code></p>
Additional parameter needed for the weakly-typed getter methods for BPEL activity custom properties.	<p><b>V5.1 Snippet:</b>  <code>String val = getActivityCustomProperty("propName");</code></p> <p><b>V6.0 Snippet:</b>  <code>String val = getActivityCustomProperty  ("name-of-current-activity", "propName");</code></p>
Additional parameter needed for the weakly-typed setter methods for BPEL activity custom properties.	<p><b>V5.1 Snippet:</b>  <code>String newVal = "new value";  setActivityCustomProperty("propName", newVal);</code></p> <p><b>V6.0 Snippet:</b>  <code>String newVal = "new value";  setActivityCustomProperty("name-of-current-activity",  "propName", newVal);</code></p>

Table 11. Changes and solutions for migrating WebSphere Business Integration Server Foundation BPEL Java snippets (continued)

Change	Solution
The raiseFault(QName faultQName, Serializable message) method no longer exists.	Migrate to the raiseFault(QName faultQName, String variableName) where possible; otherwise migrate to the raiseFault(QName faultQName) method or create a new BPEL variable for the Serializable object.

*Migrating interactions with WebSphere Business Integration Adapters:*

If the JMS Client is a WebSphere Business Integration Adapter, you may need to use the Enterprise Service Discovery tooling to create the Import with JMS Binding. This Import uses a special data binding in order to serialize the SDO to the exact format that the WebSphere Business Integration Adapter expects.

To access the Enterprise Service Discovery tooling, follow these steps:

1. Go to **File** → **New** → **Other** → **Business Integration** and select **Enterprise Service Discovery**. Click **Next**.
2. Choose **WebSphere Business Integration Adapter Artifact Importer**. Click **Next**.
3. Enter the path of the WebSphere Business Integration Adapter's configuration (.cfg) file and the directory that contains the XML schema of the business objects that the adapter uses. Click **Next**.
4. Examine the query that is generated for you, and if it is correct click **Run Query**. In the **Objects discovered by query** list, select the objects that you want to add (one by one) and click the >> **Add** button.
5. Accept the configuration parameters for the business object and click **OK**.
6. Repeat for each business object.
7. Click **Next**.
8. For the **Runtime business object format** select **SDO**. For the **Target project** select the module you just migrated. Leave the **Folder** field blank.
9. Click **Finish**.

This tool will migrate the old XSDs to the format expected by the special data binding so remove the old WebSphere Business Integration Adapter's XSDs from the module and use the new XSDs. If the module will not receive messages from the adapter, delete the Exports generated by this tool. If the module will not send any messages to the adapter, delete the Import. See the information center for more information on this feature.

*Migrating WSDL interfaces that have SOAP-encoded array types:*

This section shows how to migrate or handle XML schemas that have SOAP-encoded array types.

Soap-encoded array types that have the RPC style will be treated as unbounded sequences of a concrete type in 6.0. It is not recommended that you create any XSD types that reference the soapend:Array types in any way, as the programming model is moving towards the Document/Literal wrapped style instead of the RPC style (although this could change).

There will be cases when an SCA application must invoke an external service that does use the soapend:Array type. There is no way to avoid this in some cases, so the following shows how to handle this situation:

Sample WSDL code:

```
<xsd:complexType name="Vendor">
  <xsd:all>
    <xsd:element name="name" type="xsd:string" />
  </xsd:all>
</xsd:complexType>
```

```

    <xsd:element name="phoneNumber" type="xsd:string" />
  </xsd:all>
</xsd:complexType>
</xsd:schema>

<xsd:complexType name="Vendors">
  <xsd:complexContent mixed="false">
    <xsd:restriction base="soapenc:Array">
      <xsd:attribute wsdl:arrayType="tns:Vendor[]" ref="soapenc:arrayType"
        xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/" />
    </xsd:restriction>
  </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="VendorsForProduct">
  <xsd:all>
    <xsd:element name="productId" type="xsd:string" />
    <xsd:element name="vendorList" type="tns:Vendors" />
  </xsd:all>
</xsd:complexType>

<xsd:complexType name="Product">
  <xsd:all>
    <xsd:element name="productId" type="xsd:string" />
    <xsd:element name="productName" type="xsd:string" />
  </xsd:all>
</xsd:complexType>

<message name="doFindVendorResponse">
  <part name="returnVal" type="tns:VendorsForProduct" />
</message>

<operation name="doFindVendor">
  <input message="tns:doFindVendor" />
  <output message="tns:doFindVendorResponse" />
</operation>

```

Sample code for a client of this Web Service:

```

// Locate the vendor service and find the doFindVendor operation
Service findVendor=(Service)ServiceManager.INSTANCE.locateService("vendorSearch");
OperationType doFindVendorOperationType=findVendor.getReference().getOperationType("doGoogleSearch");

// Create the input DataObject
DataObject doFindVendor=DataFactory.INSTANCE.create(doFindVendorOperationType.getInputType());
doFindVendor.setString("productId", "12345");
doFindVendor.setString("productName", "Refrigerator");

// Invoke the FindVendor service
DataObject findVendorResult = (DataObject)findVendor.invoke(doFindVendorOperationType, doFindVendor);

// Display the results
int resultProductId=findVendorResult.getString("productId");

DataObject resultElements=findVendorResult.getDataObject("vendorList");
Sequence results=resultElements.getSequence(0);
for (int i=0, n=results.size(); i
for (int i=0, n=results.size(); i

```

Here is another example where the data object's root type is a soapenc:Array. Note how the sampleElements DataObject is created using the second schema listed above. The DataObject's type is first obtained, and then the property for sampleStructElement is obtained. This is really a placeholder property and used only to get a valid property to use when adding the DataObjects to the sequence. A pattern like this can be used in your scenario:

Sample WSDL code:

```

<s:schema elementFormDefault="qualified" targetNamespace="http://soapinterop.org/xsd">
  <s:import namespace="http://schemas.xmlsoap.org/soap/encoding/" />
  <s:import namespace="http://schemas.xmlsoap.org/wsdl/" />
  <s:complexType name="SOAPStruct">
    <s:sequence>
      <s:element minOccurs="1" maxOccurs="1" form="unqualified" name="varInt" type="s:int" />
      <s:element minOccurs="1" maxOccurs="1" form="unqualified" name="varString" type="s:string" />
      <s:element minOccurs="1" maxOccurs="1" form="unqualified" name="varFloat" type="s:float" />
    </s:sequence>
  </s:complexType>

  <s:complexType name="ArrayOfSOAPStruct">
    <s:complexContent mixed="false">
      <s:restriction base="soapenc:Array">
        <s:attribute wsdl:arrayType="s0:SOAPStruct[]" ref="soapenc:arrayType" />
      </s:restriction>
    </s:complexContent>
  </s:complexType>
</s:schema>

<wsdl:message name="echoStructArraySoapIn">
  <wsdl:part name="inputStructArray" type="s0:ArrayOfSOAPStruct" />
</wsdl:message>
<wsdl:message name="echoStructArraySoapOut">
  <wsdl:part name="return" type="s0:ArrayOfSOAPStruct" />
</wsdl:message>

<wsdl:operation name="echoStructArray">
  <wsdl:input message="tns:echoStructArraySoapIn" />
  <wsdl:output message="tns:echoStructArraySoapOut" />
</wsdl:operation>

<schema targetNamespace="http://sample/elements"
  xmlns="http://www.w3.org/2001/XMLSchema"
  xmlns:tns="http://sample/elements">

  <element name="sampleStringElement" type="string"/>

  <element name="sampleStructElement" type="any"/>

</schema>

```

Sample code for a client of this Web Service:

```

// Create the input DataObject and get the SDO sequence for the any
// element
DataFactory dataFactory=DataFactory.INSTANCE;
DataObject arrayOfStruct = dataFactory.create("http://soapinterop.org/xsd", "ArrayOfSOAPStruct");
Sequence sequence=arrayOfStruct.getSequence("any");

// Get the SDO property for the sample element that we want to use
// here to populate the sequence
// We have defined this element in an XSD file, see SampleElements.xsd
DataObject sampleElements=dataFactory.create("http://sample/elements",
"DocumentRoot");
Property property = sampleElements.getType().getProperty("sampleStructElement");

// Add the elements to the sequence
DataObject item=dataFactory.create("http://soapinterop.org/xsd", "SOAPStruct");
item.setInt("varInt", 1);
item.setString("varString", "Hello");
item.setFloat("varFloat", 1.0f);
sequence.add(property, item);
item=dataFactory.create("http://soapinterop.org/xsd", "SOAPStruct");
item.setInt("varInt", 2);
item.setString("varString", "World");
item.setFloat("varFloat", 2.0f);

```

```

sequence.add(property, item);

// Invoke the echoStructArray operation
System.out.println("[client] invoking echoStructArray operation");
DataObject echoArrayOfStruct = (DataObject)interopTest.invoke("echoStructArray", arrayOfStruct);

// Display the results
if (echoArrayOfStruct!=null) {
    sequence=echoArrayOfStruct.getSequence("any");
    for (int i=0, n=sequence.size(); i<n; i++) {
        item=(DataObject)sequence.getValue(i);
        System.out.println("[client] item varInt = "+
            item.getInt("varInt")+
            varString="+item.getString("varString")+
            varFloat="+item.getFloat("varFloat"));
    }
}

```

## Migrating WebSphere Business Integration EJB projects:

In WebSphere Studio Application Developer Integration Edition, EJB projects could have special WebSphere Business Integration features such as Extended Messaging (CMM) and CMP/A (Component-managed persistence anywhere). The deployment descriptors for such projects must be migrated and this section shows how to perform that migration.

To perform this migration, follow these steps:

1. Copy the WebSphere Business Integration EJB project to the new 6.0 workspace and import it from WebSphere Integration Developer using the **File** → **Import** → **Existing Project into Workspace** wizard. Optionally, you can also run the J2EE Migration wizard.
2. Close all instances of WebSphere Integration Developer running in the 6.0 workspace.
3. Run the following script which will migrate the WebSphere Business Integration deployment descriptors in the EJB project:

### On Windows:

```
%WID_HOME%\wstools\eclipse\plugins\com.ibm.wbit.migration.wsadie_6.0.0\WSADIEEJBProjectMigration.bat
```

### On Linux:

```
$WID_HOME/wstools/eclipse/plugins/com.ibm.wbit.migration.wsadie_6.0.0/WSADIEEJBProjectMigration.sh
```

The following parameters are supported, where the workspace and project name are mandatory:

```
Usage:      WSADIEEJBProjectMigration.bat
           [-e eclipse-folder] -d workspace -p project
```

eclipse-folder: The location of your eclipse folder -- usually it's the 'eclipse' found under your product installation folder.

workspace: The workspace containing the WSADIE EJB project to be migrated.

project: The name of the project to migrate.

For example,

```
WSADIEEJBProjectMigration.bat -e "C:\IBM\WID6\eclipse" -d "d:\my60workspace" -p "MyWBIEJBProject"
```

4. When you open WebSphere Integration Developer you will need to refresh the EJB project to get the updated files.
5. Search for the file **ibm-web-ext.xmi** in the EJB project. If one is found, ensure that the following line is present in the file under the element:

```
<webappext:WebAppExtension> element:
<webApp href="WEB-INF/web.xml#WebApp"/>
```
6. Remove the old deployment code that was generated in 5.1. Regenerate the deployment code by following the WebSphere Application Server guidelines for doing so.

## Performing a manual fix for namespace collisions:

In WebSphere Studio Application Developer Integration Edition 5.1, defining two different XSD or WSDL types with the same name and target namespace was supported. In WebSphere Integration Developer 6.0 it is not supported. Manual migration is required if you encounter duplicate definition errors after building your migrated projects.

To remedy this problem, follow these steps:

1. If the definitions are the same, delete one of them and then clean and rebuild your project. Correct any errors that might result by pointing existing WSDL/XSD files to the file containing the definition that you did *not* delete.
2. If the definitions are *not* the same and you must use both of the definitions in your migrated service, rename the definition name or the target namespace. If there are only a few definition in the entire file that are duplicates, then changing their names is recommended. If all of the definitions in the file are duplicates, then changing the target namespace of all of the definitions is recommended. Clean and rebuild the project, ensuring that the artifacts that you want to use in the definition(s) that you modified reference the new definition name or namespace.
3. In the case where you have two import statements for the same namespace in a WSDL file, you can remedy this problem by chaining the imports such that one of these WSDLs imports the other, which imports the next one, etc., so that there is only one import for this namespace per WSDL file. Then clean and rebuild the project.

## Manually deleting 5.1 Web Services Invocation Framework (WSIF) definitions:

After you have completed migrating your source artifacts, you should delete all 5.1 WSIF Binding and Service WSDL definitions from your 6.0 projects that are no longer being used. The consumption scenario for service migration is the only case where a WSIF Binding or Service would still be in use.

The following WSDL namespaces indicate that a binding or service definition is a 5.1 WSIF Service and can be discarded if no longer used:

### EJB WSIF Namespace:

<http://schemas.xmlsoap.org/wsdl/ejb/>

### Java WSIF Namespace:

<http://schemas.xmlsoap.org/wsdl/java/>

### JMS WSIF Namespace:

<http://schemas.xmlsoap.org/soap/jms/>

### Business Process WSIF Namespace:

<http://schemas.xmlsoap.org/wsdl/process/>

### Transformer WSIF Namespace:

<http://schemas.xmlsoap.org/wsdl/transformer/>

### IMS WSIF Namespace:

<http://schemas.xmlsoap.org/wsdl/ims/>

### CICS-ECI WSIF Namespace:

<http://schemas.xmlsoap.org/wsdl/cicseci/>

### CICS-EPI WSIF Namespace:

<http://schemas.xmlsoap.org/wsdl/cicsepi/>

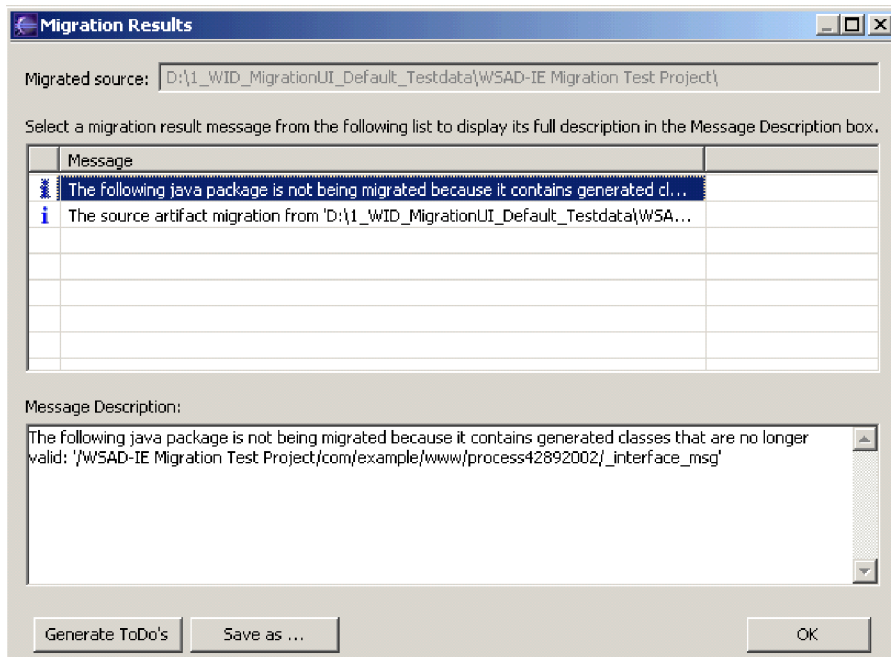
### HOD WSIF Namespace:

<http://schemas.xmlsoap.org/wsdl/hod3270/>

## Verifying the source artifact migration

If the migration completes with a list of errors, warnings, and/or informational messages, they will be displayed in the Migration Results window. Otherwise, the wizard window will close.

The following page appears if migration messages were generated during the migration process:



In the Migration Results window, you can see the migration messages that were generated during the migration process. By selecting a message from the upper Message list, you can find more information regarding that message in the lower Message Description window. To keep all messages for future reference, click the **Generate ToDo's** button to create a list of "ToDo" tasks in the task view and/or click the **Save as...** button to save the messages in a text file in the filesystem. Examine each message to see if any action needs to be taken to immediately fix an artifact that couldn't be fully migrated.

To verify that a portion of the migration is complete, switch to the Business Integration perspective and ensure that all processes and WSDL interfaces from the old service project appear in the new module. Build the project and fix any errors that prevent the project from building.

After performing the manual migration steps required to complete the migration of the business integration application, export the application as an EAR file and install it to a WebSphere Process Server, configuring the appropriate resources.

Perform the manual migration steps required to migrate any client code or generate new client code using WebSphere Integration Developer. Ensure that the client can access the application and that the application exhibits the same behavior it did on the previous runtime environment.

## Working with source artifact migration failures

If your source artifact migration from WebSphere Studio Application Developer Integration Edition fails, there are a number of ways in which to deal with the failures.

The following are some of the possible source artifact migration failures:

- If you receive the following message:  
"Migration Error Message"  
Reason: Fatal Migration Failure  
Message: Contact your IBM Representative

you should check the WebSphere Integration Developer log file in the .metadata folder of the new workspace to see the details of the error. If possible, resolve the cause of the error, delete the module that was created in the new workspace, and try the migration again.

If the Migration wizard completes without this message, a list of info, warning, and error messages will be displayed. These signify that some portion of the service project could not be automatically migrated and that manual changes must be performed to complete the migration.

### Best practice: Source artifact migration process

There are a number of best practices for the WebSphere Studio Application Developer Integration Edition source artifact migration process.

The following practices show how to design WebSphere Studio Application Developer Integration Edition services to ensure that they will migrate successfully to the new programming model:

- Try to use the **Assign** activity wherever possible (as opposed to the transformer service which is only needed when an advanced transformation is needed). You should use this practice because intermediate componentry must be constructed in order for the an SCA module to invoke a transformer service. Additionally, there is no special tooling support in WebSphere Integration Developer for the transformer services created in 5.1 (you must use the WSDL or XML editor to modify the XSLT embedded in the WSDL file if you need to change the behavior of the transformer service).
- Specify one part per WSDL message as per the Web Services Interoperability (WS-I) spec and the 6.0 preferred style.
- Use the WSDL doc-literal style as this is the preferred style in 6.0.
- Ensure that all complex types are given a name and that each complex type can be uniquely identified by its target namespace and name. The following shows the recommended way to define complex types and elements of that type (complex type definition followed by an element definition that uses it):

```
<schema attributeFormDefault="qualified"
  elementFormDefault="unqualified"
  targetNamespace="http://util.claimshandling.bpe.samples.websphere.ibm.com"
  xmlns="http://www.w3.org/2001/XMLSchema"
  xmlns:tns="http://util.claimshandling.bpe.samples.websphere.ibm.com">

<complexType name="Duration">
  <all>
    <element name="hours" type="int"/>
    <element name="minutes" type="int"/>
    <element name="days" type="int"/>
  </all>
</complexType>
<element name="DurationElement" type="tns:Duration"/>
</schema>
```

The following example is an anonymous complex type that should be avoided as it can cause problems when an SDO is serialized to XML (element containing an anonymous complex type definition):

```
<schema attributeFormDefault="qualified"
  elementFormDefault="unqualified"
  targetNamespace="http://util.claimshandling.bpe.samples.websphere.ibm.com"
  xmlns="http://www.w3.org/2001/XMLSchema"
  xmlns:tns="http://util.claimshandling.bpe.samples.websphere.ibm.com">
<element name="DurationElement">
  <complexType>
    <all>
      <element name="hours" type="int"/>
      <element name="minutes" type="int"/>
      <element name="days" type="int"/>
    </all>
  </complexType>
</element>
```

```

    </all>
  </complexType>
</element>
</schema>

```

- If publishing a service for external consumers, generate service deploy code using IBM Web Services (as opposed to Apache SOAP/HTTP) because IBM Web Services are directly supported in 6.0 and Apache Web services are not.
- There are two ways to organize WSDL and XSD files in 5.1 to minimize the amount of reorganizing you must do during migration. In 6.0, shared artifacts such as WSDL and XSD files must be located in BI projects (Business Integration *modules* and *libraries*) in order to be referenced by a BI service:
  - Keep all WSDL files shared by more than one service project in a Java project that the service projects can reference. During migration to 6.0 you will create a new Business Integration Library with the same name as the 5.1 shared Java project. Copy all of the artifacts from the 5.1 shared Java project to the library so that the migration wizard can resolve the artifacts when it migrates the services projects that use those artifacts
  - Keep a local copy of all WSDL/XSD files that a service project references in the service project itself. WebSphere Studio Application Developer Integration Edition service projects will be migrated to a Business Integration Module in WebSphere Integration Developer and a module can not have dependencies on other modules (a service project with dependencies on another service project for the sake of sharing WSDL or XSD files will not migrate cleanly).
- Avoid using the Business Process Choreographer Generic Messaging API (Generic MDBs) as it will not be provided in 6.0. An MDB interface offering late binding will *not* be available in 6.0.
- Use the Business Process Choreographer Generic EJB API as opposed to invoking the generated session beans that are specific to a particular version of a process. These session beans will not be generated in V6.0.x.
- If you have a business process with multiple replies for the same operation, ensure that if any of them has client settings that all replies for that operation have the same client settings as in 6.0 only one set of client settings is supported per operation reply.
- Design BPEL Java snippets according to the following guidelines:
  - Avoid sending WSIFMessage parameters to any custom Java classes - try not to depend on the WSIFMessage data format where possible.
  - Avoid using the WSIF metadata APIs if possible.
- Avoid creating top-down EJB or Java services where possible because the Java/EJB skeleton that gets generated from the WSDL PortTypes/Messages will be dependent on WSIF classes (for example, WSIFFormatPartImpl). Instead, create the Java/EJB interfaces first and generate a service around the Java class/EJB (bottom-up approach).
- Avoid creating or using WSDL interfaces that reference the soapenc:Array type because this type of interface is not natively supported in the SCA programming model
- Avoid creating message types whose high-level element is an array type (maxOccurs attribute is greater than one) because this type of interface is not natively supported in the SCA programming model.
- Define your WSDL interfaces precisely - avoid XSD complexTypes that have references to the xsd:anyType type where possible.
- For any WSDL and XSDs that you generate from an EJB or Java bean, ensure that the target namespace is unique (the Java class name and package name are represented by the target namespace) in order to avoid collisions when migrating to WebSphere Process Server V6. In WebSphere Process Server V6, two different WSDL/XSD definitions that have the same name and target namespace are not allowed. This situation often occurs when the Web Service wizard or Java2WSDL command is used without specifying the target namespace explicitly (the target namespace will be unique for the package name of the EJB or Java bean, but not for the class itself so problems will occur when a web service is generated for two or more EJB or Java beans in the same package). The solution is to specify a custom package to namespace mapping in the Web Service wizard or to use the **-namespace** Java2WSDL command line option to ensure that the namespace of the generated files is unique for the given class.

- Use unique namespaces for every WSDL file where possible. There are limitations around importing two different WSDL files with the same namespace according the WSDL 1.1 specification, and in WebSphere Integration Developer 6.0 these limitations are strictly enforced.

## Limitations of the migration process (for source artifact migration)

There are certain limitations involved with the WebSphere Studio Application Developer Integration Edition source artifact migration process.

The following lists detail some of the limitations of the migration process for source artifact migration:

### General limitations

- WebSphere Studio Application Developer Integration Edition did not strictly enforce consistency between the WSDLs and other artifacts in projects. WebSphere Integration Developer is much stricter and will report inconsistencies that WebSphere Studio Application Developer Integration Edition did not (and also which ran on the WebSphere Business Integration Server Foundation runtime without any issue).
- Although WebSphere Studio Application Developer Integration Edition allowed multiple identical Web Service Binding and Service definitions (name and namespace), WebSphere Integration Developer does not. You must resolve these duplicates manually before migration (in WebSphere Studio Application Developer Integration Edition) or after migration (in WebSphere Integration Developer). An example is that in WebSphere Studio Application Developer Integration Edition, all of the generated service definitions in the WSDL files with different names (ending in `_EJB`, `_JMS`, etc.) looked like:

```
<service name="OrderProcessIntfcService">
```

To fix the duplicate, simply append the binding type to the name attribute. For the `*_EJB.wsdl` file, it would be changed to

```
<service name="OrderProcessIntfcServiceEJB">
```

For the `*_JMS.wsdl` file, it would be changed to

```
<service name="OrderProcessIntfcServiceJMS">
```

However, once this name is changed, the Export generated in WebSphere Integration Developer to use this service will also need to be changed to use the right name.

- This Migration wizard cannot handle entire WebSphere Studio Application Developer Integration Edition workspaces. It is meant to migrate one WebSphere Studio Application Developer Integration Edition service project at a time.
- The Migration wizard does not migrate application binaries, it only migrates source artifacts found in a WebSphere Studio Application Developer Integration Edition service project.
- Business Rule Beans are deprecated in WebSphere Process Server 6.0 but there is an option during WebSphere Process Server install to install support for the deprecated Business Rule Beans such that they will run “as-is” on a WebSphere Process Server 6.0 server. There is no tooling support for the old Business Rule Beans however, and if you want the old Business Rule Bean artifacts to compile in the tools, you must follow the WebSphere Integration Developer documentation to install those deprecated features over top of the embedded WebSphere Process Server 6.0 test server and then manually add the deprecated jar files to the project classpath as external jars. You should use the new Business Rule tooling available in WebSphere Integration Developer to recreate their business rules according to the 6.0 specification.
- Extended Messaging support is deprecated in WebSphere Process Server 6.0 but there is an option during WebSphere Process Server install to install support for the deprecated Extended Messaging features such that existing applications can run “as-is” on a WebSphere Process Server 6.0 server. There is no tooling support for the old Extended Messaging features however, and if you want the old Extended Messaging artifacts to compile in the tools, you must follow the WebSphere Integration

Developer documentation to install those deprecated features over top of the embedded WebSphere Process Server 6.0 test server and then manually add the deprecated jar files to the project classpath as "external jars".

- The standard provided JMS data binding does not provide access to custom JMS header properties. A custom data binding must be written for the SCA services to get access to any custom JMS header properties.

### SCA Programming Model limitations

- The SDO version 1 specification does not provide access to the COBOL or C byte array – this will impact those working with IMS multi-segments.
- The SDO version 1 specification for serialization does not support COBOL redefines or C unions.
- When redesigning your source artifacts according to the SCA programming model, please note that the document/literal wrapped WSDL style (which is the default style for new artifacts created using the WebSphere Integration Developer tools) does not support method overloading. The other WSDL styles are still supported so it is recommended that another WSDL style/encoding other than document/literal wrapped be used for these cases.
- Native support for arrays is limited. In order to invoke an external service that exposes a WSDL interface with soapenc:Array types, you will need to create a WSDL interface that defines an element whose "maxOccurs" attribute is greater than one (this is the recommended approach for designing an array type).

### BPEL migration process technical limitations

- **Multiple replies per BPEL operation** - In WebSphere Business Integration Server Foundation a business process could have one receive activity and multiple reply activities for the same operation. If you have a business process with multiple replies for the same operation, ensure that if any of them has client settings that all replies for that operation have the same client settings as in 6.0 only one set of client settings is supported per operation reply.
- **Limitations of BPEL Java snippet migration** - The programming model has changed significantly from WebSphere Studio Application Developer Integration Edition to WebSphere Integration Developer and not all supported WebSphere Studio Application Developer Integration Edition APIs can be directly migrated to corresponding WebSphere Integration Developer APIs. Any Java logic can be found in the BPEL Java snippets so that the automatic migration tool may not be able to convert every Java snippet to the new programming model. Most of the standard snippet API calls will be automatically migrated from the 5.1 Java snippet programming model to the 6.0 Java snippet programming model. WSIF API calls are migrated to DataObject API calls where possible. Any custom Java classes that accept WSIFMessage objects will need manual migration such that they accept and return commonj.sdo.DataObject objects instead:
  - **WSIFMessage metadata APIs** - Manual migration may be needed for some WSIFMessage metadata and other WSIF APIs.
  - **EndpointReference/EndpointReferenceType APIs** - These classes are not automatically migrated. Manual migration is needed as the partner link getter/setter methods deal with commonj.sdo.DataObject objects instead of the com.ibm.websphere.srm.bpel.wsaddressing.EndpointReferenceType objects from 5.1.
  - **Complex types with duplicate names** - If an application declares complex types (in WSDLs or XSDs) with identical namespaces and local names, or different namespaces but identical local names, Java snippets that use these types may not be migrated correctly. Verify the snippets for correctness after the migration wizard has completed.
  - **Complex types with local names identical to Java classes in the java.lang package** - If an application declares complex types (in WSDLs or XSDs) with local names that are identical to classes in the java.lang package of J2SE 1.4.2, Java snippets that use the corresponding java.lang class may not be migrated correctly. Verify the snippets for correctness after the migration wizard has completed.

- **Read-only and read-write BPEL variables** - In any 5.1 Java snippet code, it was possible to set a BPEL variable to "read-only", meaning that any changes made to this object will not affect the BPEL variable's value at all. It was also possible to set a BPEL variable to "read-write", meaning that any changes made to the object would be reflected for the BPEL variable itself. The following shows four ways that a Java snippet could be accessed as "read-only" in any 5.1 BPEL Java snippet:

```
getMyInputVariable()
getMyInputVariable(false)
getVariableAsWSIFMessage("MyInputVariable")
getVariableAsWSIFMessage("MyInputVariable", false)
```

Here are the two ways that a BPEL variable could be accessed as "read-write" in any 5.1 BPEL Java snippet:

```
getMyInputVariable(true)
getVariableAsWSIFMessage("MyInputVariable", true)
```

In 6.0, read-only and read-write access to BPEL variables is handled on a "per-snippet basis," meaning you can add a special comment to the BPEL Java snippet to specify whether updates to the BPEL variable should be discarded or maintained after the snippet has finished executing. Here are the default access settings for the 6.0 BPEL Java snippet types:

```
BPEL Java Snippet Activity
Default Access: read-write
Override Default Access with comment containing:
    @bpe.readOnlyVariables names="variableA,variableB"
```

```
BPEL Java Snippet Expression (Used in a Timeout, Condition, etc)
Default Access: read-only
Override Default Access with comment containing:
    @bpe.readWriteVariables names="variableA,variableB"
```

During migration, these comments will automatically be created when a variable was accessed in a way that is not the default in 6.0. In the case that there is a conflict (meaning that the BPEL variable was accessed as "read-only" and as "read-write" in the same snippet), a warning is issued and the access is set to "read-write". If you receive any such warnings, ensure that setting the BPEL variable access to "read-write" is correct for your situation. If this is not correct, you should correct it manually using the WebSphere Integration Developer BPEL editor.

- **Many-valued primitive properties in complex types** - In 5.1, multi-valued properties are represented by arrays of the property type. As such, calls to get and set the property use arrays. In 6.0, java.util.List is used for this representation. Automatic migration will handle all cases where the many-valued property is some type of Java object, but in the case that the property type is a Java primitive (int, long, short, byte, char, float, double, and boolean), calls to get and set the entire array are not converted. Manual migration in such a case might require adding a loop to wrap/unwrap the primitives in/from their corresponding Java wrapper class (Integer, Long, Short, Byte, Character, Float, Double, and Boolean) for use in the rest of the snippet.
- **Instantiation of generated classes representing complex types** - In 5.1, generated classes of complex types defined in an application could be easily instantiated in a Java snippet using the default no-argument constructor. An example of this is:

```
MyProperty myProp = new MyProperty();
InputMessageMessage myMsg = new InputMessageMessage();
myMsg.setMyProperty(myProp);
```

In 6.0, a special factory class must be used to instantiate these types, or, an instance of the containing type may be used to create the sub-type. If a BPEL process variable InputVariable was defined as having type InputMessage, then the 6.0 version of the preceding snippet would be:

```
com.ibm.websphere.bo.BOFactory boFactory=
    (com.ibm.websphere.bo.BOFactory)
    com.ibm.websphere.sca.ServiceManager.INSTANCE.locateService(
        "com/ibm/websphere/bo/BOFactory");
```

```
commonj.sdo.DataObject myMsg =  
    boFactory.createByType(getVariableType("InputVariable"));  
commonj.sdo.DataObject myProp =  
    myMsg.createDataObject("MyProperty");
```

The snippet converter attempts to make this change, but if the order in which the original instantiations occur does not follow the parent-then-child pattern, manual migration will be needed (i.e. the converter does not attempt to intelligently reorder the instantiation statements in the snippet).

- In WebSphere Business Integration Server Foundation 5.1, dynamic references were represented as WSDL message parts of type `EndpointReferenceType` or element `EndpointReference` from the namespace:

<http://wsaddressing.bpel.srm.websphere.ibm.com>

Such references will be migrated to the standard service-ref element type from the standard business process namespace:

<http://schemas.xmlsoap.org/ws/2004/03/business-process/>  
<http://schemas.xmlsoap.org/ws/2004/08/addressing>

See the BPEL Editor documentation for instructions on manually importing these schema definitions into your project so that all references resolve properly.

- **BPEL variable message type** - A WSDL message type must be specified for all BPEL variables used in Java snippets. Java snippets that access BPEL variables without the "messageType" attribute specified can not be migrated.



---

## Notices

U.S. Government Users Restricted Rights - Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this documentation in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this documentation. The furnishing of this documentation does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing  
IBM Corporation  
North Castle Drive  
Armonk, NY 10504-1785  
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation  
Licensing  
2-31 Roppongi 3-chome, Minato-ku  
Tokyo 106, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OR CONDITIONS OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

*Intellectual Property Dept. for WebSphere Integration Developer  
IBM Canada Ltd.  
8200 Warden Avenue  
Markham, Ontario L6G 1C7  
Canada*

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this documentation and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples may include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

#### COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

© (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. 2000, 2006. All rights reserved.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

## **Programming interface information**

Programming interface information is intended to help you create application software using this program.

General-use programming interfaces allow you to write application software that obtain the services of this program's tools.

However, this information may also contain diagnosis, modification, and tuning information. Diagnosis, modification and tuning information is provided to help you debug your application software.

**Warning:** Do not use this diagnosis, modification, and tuning information as a programming interface because it is subject to change.

## **Trademarks and service marks**

See <http://www.ibm.com/legal/copytrade.shtml>.







Printed in USA

SC10-4211-03

