Sterling Integrator®

MESA Developer Studio

Version 5.0

Contents

Using MESA Developer Studio

About MESA Developer Studio

MESA™ Developer Studio is used to create and edit service definitions (using the SDK), maps, and business processes. You can also use MESA Developer Studio to remotely start and stop an application instance, install third party files, list directory contents and current processes, and display disk usage. For example, within the application you can only edit business processes using the Graphical Process Modeler (GPM) or a text editor. From MESA Developer Studio, you can also edit business processes using a code editor.

Note: Before you can being using MESA Developer Studio, you must first install it. See Installing MESA Developer Studio and the Installation Guide for your application.

Using MESA Developer Studio to Manage the Application

Note: MESA Developer Studio is designed to assist you with resource development. Changes made with the MESA Developer Studio plug-ins should be thoroughly tested in a development environment before moving them into production.

Use MESA Developer Studio to:

- Edit property files
- Work with business processes
- Work with maps
- Work with service definitions
- Start and stop application instances
- Install third-party files
- Manage application resources
- List directory contents
- List current processes
- Display disk usage

Available MESA Developer Studio Editors

The following editors are available in MESA Developer Studio to assist you in creating or editing properties, service definitions, and other code:

• Properties Editor
• JDBC Properties Editor
• Knowledgebase Properties Editor
• Service Definitions Editor
• BPML Editor

License Management Settings

Licenses provide you access to the different components offered by the Application that you have purchased. Without the proper licenses, Application does not operate. For example, after you purchase Application, you can purchase new components and open those components with a new license file. Occasionally, you may need to update your license file for either administrative purposes, or when your license file expires. You can view the components license status and update your license files through MESA Developer Studio.

To manage your license files:

1. From the MESA Developer Studio perspective, double-click an instance. The instance overview appears in the Control Editor.
2. Click the Settings tab at the bottom of the Control Editor pane.
3. Click Update Licenses.
4. Navigate to the appropriate license file. Click Open.

   The license file is automatically updated.

Create a MESA Developer Studio Project

Create a project in the Package Explorer to organize the files and resources you will use on your local system.

To create a project:

1. From the File menu, select **New** > **Project**.
2. Select **Java** > **Project**.
3. Click **Next.**
4. Type a name for the project.
5. Click **Finish**. A new project is added to the Package Explorer under the folder name Other Projects.

Note: If no direct connection is possible between the host where the application is installed and the Windows PC where Eclipse is installed, and you are using a proxy server, you must enable the HTTP proxy connection. From the Window menu, select **Preferences**. On the left, select Install/Update. In the Proxy settings section, add your proxy information and click **Apply.**

Managing Resources in MESA Developer Studio

Resources are files, templates, and documents that may be deployed in your application and that you can import and export from one system to another, such as when you are migrating from a test to production environment. You can check in, check out, lock, and unlock the following resources in MESA Developer Studio:

- Business Processes – Business process model definitions and their associated particulars

Note: In a cluster environment, business processes should be checked in from the application UI, not from MESA Developer Studio. Currently, MESA Developer Studio is not configured to set specific business processes.

- Maps - Translation maps used for converting file types including flat files, CSV, EBCDIC, EDI, SWIFT, ACH, XML and others to different formats
- Property Files - The property files for the application are available for editing in MESA Studio.

Caution: We strongly recommend that you do not edit these files directly . Instead, use MESA Studio to view the property files, then create or edit a new property file called "customer_overrides.properties" and add changes to property files there. This ensures that your customizations for property files are not lost or overwritten by a patch or upgrade. See the Property Files documentation for more specific information about creating and using the customer_overrides.properties file.

- XML schemas – Data that makes up XML schemas. In the application, XML schemas are used for several functions, including Reporting Services, Web Services, translation maps, and reporting.

Working with Business Processes

Business processes may be created and edited in MESA Developer Studio using either the BPML text editor or the GPM. Files with a .bp extension will by default open the GPM; however, they may also be edited by using a option to open the file with the BPML Editor.

Note: The default when you double-click a business process is for it to open in the GPM. If you want to edit a business process using the BPML Editor, right-click on the business process and select **Open With > BPML Editor**. MESA Developer Studio remembers this setting so that the next time you double-click on that business process, it will open in the BPML Editor.

The BPML Editor allows XML-like text editing of business processes along with autofill of BPML activities. MESA Developer Studio allows you to start the GPM; however, it runs independently of Eclipse.

To use MESA Developer Studio to edit business processes, you must set up the GPM to be used in MESA Developer Studio. This enables you to launch the GPM from within Eclipse when you double-click on a business process name, either in your Package Explorer list of projects (local) or on your test server through a configured instance. (Only files that have been checked out from the server are displayed in that view.)

To set up the GPM:

1. In Eclipse, from the Window menu, select **Preferences.**
2. On the left, select the **Java > Installed JREs** category.
3. In the Installed JREs section, click **Add.**
4. Enter the following information:
   - JRE name - Type `GBM JRE`.
   - JRE home directory - Click **Browse** to select the directory. You must use a 1.5 JRE with the GPM. This is usually located in C:\Program Files\javasoft\jre\1.5.
5. Click **OK**.

   You are now ready to edit business process files from within MESA Developer Studio.

   Note: The first time you start the GPM from Eclipse, your application instance must be running.

For more information on managing business processes, see the business process documentation.

About Working with Schemas

An XML schema is an XML document that specifies the structure of a valid XML document. Comparable to a document template, an XML schema ensures that every item is in the correct form. An XML schema consists of the following components:

- XML declaration – defines the XML version that the schema uses
- Schema element – identifies the document as an XML schema
- Element declaration – defines the element
- Attribute declaration – defines the attribute

Schemas are used in your application to validate translation data in the Map Editor. When you submit an XML document using the application, the XML document is compared to the XML schema to ensure that the document is in the appropriate format and is valid. For more information on managing schemas, see the schema documentation.

Working with Properties Files

From Mesa Developer Studio, you can add, edit, or delete properties within existing files.

To open a property file:

1. From the Mesa Developer Studio perspective, expand the Properties Files folder.
2. Double-click on the property file name you want to work with to open it in the editor.
3. Do one of the following:
   - Click **Add** to add a property. Type a name and value for the property.
   - Select a property and click **Delete** to remove it.

Using the MESA Skin Editor

About the MESA Developer Studio Skin Editor

The MESA™ Developer Studio Skin Editor provides an interface where you can quickly change the appearance or branding of your application. This includes basic page colors, images, and fonts. The Skin Editor allows you to edit all five of the main application templates. You can change their properties so that they better reflect your company's brand image. The templates (pages) you can edit include:

- AFT/myAFT
- Dashboard
- Login
- MBI (mailbox)
- Admin
- Community Management

Basic properties include page colors, images, and fonts. Advanced properties include the use of Cascading Style Sheets (.css files). Changes are made through the Skin Editor Plug-in in the Eclipse interface. You download a copy of the current skin, make changes, then upload the revised skin to your application installation. You can revert to the default skin at any time. Multiple skin versions can be saved for future use.

Note: Your application must be running the first time MESA Developer Studio Skin Editor is used to verify the license and to download and deploy the skin.

Tips for Using the Skin Editor

When you open the Skin Editor, you must select a server from the Studio Tree View.

Task	Tip
Retrieve the current skin	Before editing a template, you must retrieve the current Application page skin from the instance. From the toolbar at the top of the page, select **Skin > Download skin**.

Task	Tip
	Note: If the current skin has not been downloaded, you are prompted to retrieve the skin from the instance.
Preview a skin	Selecting Preview to open the selected page in the preview pane and see changes you have made. **Note:** When a template is edited, all other pages sharing the same skin are updated.
Save skin changes	When the skin is ready to be deployed, select **Skin > Deploy changes**. The changes are saved to the remote instance of your application.
Undo or redo skin changes	Selecting **Skin > Undo skin edit** or redo the last change made by selecting **Skin > Redo skin edit** from the toolbar at the top of the page.
Restore the original (default) skin	Select**Skin > Apply default skin**. The original skin will be restored to the remote instance.

Using the Skin Editor to Edit a Application Template

Once the existing Application skin is downloaded, you can use the Skin Editor to make changes to an existing template. The top left view provides a list of editable Application pages. Click on a page to open the property editor in the bottom left view, and open a preview window on the right.

To edit a template:

1. Open Eclipse and select the Skin Editor perspective.
2. Select the UI page type to be edited. The UI colors, image files, fonts, and properties appropriate for that page type are displayed.
3. Make desired changes and click **Save**. The changes are displayed in a preview.

   Note: You can refresh the view of the current skin at any time to see changes applied.

4. Specify the instance to apply the changes.
5. Test the connection to your instance.
6. Update the configured instance with the new look and feel information. The system confirms the update and displays a progress message.
7. You must restart/relaunch your application for the changes to take effect.

Use the advanced editing option in the Skin Editor to edit a Cascading Style Sheet (.css file) to your skin. If your company uses an existing style sheet for other applications, you can add it to the Skin Editor and use it to customize your application so that it will match your other applications.

You can also use advanced editing to view and make changes to either the default UI settings, or the UI settings for the specified page type.

Creating Services Using the MESA SDK

Creating Custom Services

The application can perform most common tasks needed by a user, but there are instances where functionality is needed that is not provided and an existing service is not available. This usually occurs in environments with legacy systems that do not use standards-based methods for communication. There are several ways to interact with these systems, including the Command Line Adapter 2 and the Script adapter, but the capabilities of these adapters are limited.

MESA Developer Studio Software Development Kit (SDK) provides the tightest integration with the application and enables you to create complex and complete services and adapters. These services use the same APIs as the services included with the application, so all of the benefits and infrastructure of the application are available. This flexibility and tight integration means that creating a service is more advanced and requires good knowledge of Java development, application APIs, and the APIs of any system that will be accessed by the service.

Note: Because all adapters are a type of service, this guide uses the term *service* for both services and adapters. This guide uses the term *adapter* when the information is unique to adapters.

What is a Service?

A service is a component that can be configured to carry out an activity in a business process.

What is a Parameter?

Parameters can be configured to define and control your service. Any parameters that you want to add to a service must be added to a parameter group.

What is a Parameter Group?

Parameter groups are logical groupings of similar parameters (for example, host name and port). It is acceptable to have a parameter group with only one parameter. There are three types of parameter groups, as described in the following table:

Parameter Group	Description	Can be edited in this location:
Global Definition	Widest scope. Applicable to all services of this type. They have a constant value for all instances of a service.	Application interface: Deployment > Services > Installation/Setup.
Instance Definition	Specific to a single copy of a service. Can have different values for each instance of a business process that calls the service instance.	Page in a service configuration wizard accessed through Application interface, **Deployment > Services > Configuration**.
Workflow Definition	Specific to a single invocation of a service. Can have different values every time the service is called.	Graphical Process Modeler.

What is an Adapter?

An adapter is a type of service that communicates with external systems to move data in and out of the application.

What is a Method?

A method is the Java equivalent of functions, subroutines, or procedures in other programming languages.

Knowledge Prerequisites for Creating Custom Services

Creating custom services and adapters for use with the application requires specialized programming knowledge and skills, as well as a solid understanding of the application.

The following list includes the types of knowledge and experience necessary for successfully creating custom services and adapters:

- Java (J2SE) programming knowledge
- General operational and architectural knowledge of the application
- Eclipse programming experience

The following knowledge and experience are helpful, but not required:

- Multi-threaded programming experience in Java
- Ability to write custom APIs and user exits

Classes Available in MESA Developer Studio SDK

In MESA Developer Studio SDK, you have access to certain java classes for use in creating custom services and adapters. You can find detailed information about each class in the MESA Studio javadocs, which are located in the *install_dir*/install/studiodocs folder of your application installation. The classes are grouped by functional area:

- ASI (Application Server Independence)
- IFC (Integration Framework Collection)

• AFC (Application Framework Collection)

The following tables describe the classes available for use with the MESA Studio SDK.

ASI Classes

“Application Server Independence” refers to the architecture used for the B2B applications: no third party application server is needed. Javadocs for these classes are located at *install_dir*/install/studiodocs/asi_javadocs.

ASI Class	Description
ActivityData	Used to store adapter-specific information about an active adapter session. This information is formatted and displayed as status information.
Document	Represents incoming data (e.g., HTML or XML document) to the Workflow system. Conceptually, it is the handle of the persisted document in permanent storage (i.e., the database), and provides the primary access to the document's data.
IAdapterRMI	Public interface for all adapters to implement method names.
InitialWorkFlowContext	Facilitates initiating and continuing workflows with or without input data. It contains methods for supplying input data to the workflow in the form of key, value pairs and as a Document with supporting data that describes the Document such as a Document Name, Document Subject, Document Content Type, and so on. In addition there are methods for specifying the workflow and version of the workflow to be initiated or continued.
IService	Service interface that will be implemented by the flat file services.
LockManager	The LockManager utility allows users to write classes that use a common resource that should not be updated concurrently by multiple classes. Classes can safely use such a resource by first "locking" the resource using this utility and then performing the desired operations on that resource then "unlock" the resource for use by another class. Classes that try to use a locked resource will receive an exception that the resource is currently in use.
ServicesControllerImpl	Implementation class of the ServicesController interface. Used to manage and report on the status of different adapters.
WFCBase	Extends BaseWFCConstants. Implements java.io.Serializable.
WorkFlowContext	All-encompassing structure that contains all the information regarding a workflow. All data members, get/set methods should be in WFCBase.
WorkFlowException	Thrown in response to events such as abnormal conditions defined within a workflow, or run-time server exceptions.

IFC Classes

IFC classes are used to support the B2B application integration with other applications. Javadocs for these classes are located at *install_dir*/install/studiodocs/ifc_javadocs.

IFC Class	Description
Event	Represents an event that occurs in the system.

IFC Class	Description
EventProcessor	All generated events are sent here. Hands the event to all the appropriate listeners for actual processing. This singleton class provides a single, global point of entry for all events generated in the system. However, to prevent it from becoming a bottleneck, rather than have one processor synchronized against all the threads in the system, it is actually a threadlocal singleton. This means that a new instance is created for every thread that asks for one. This reduces the need for synchronization because there will be an event processor per thread.
EventProcessorFactory	Instantiates event processors.
IEventProcessor	Event processor interface.
SemaphoreManager	Uses an API generated by the Entity framework, and allows users to share one single resource within the limit of count. The resource can be a connection to extenal system or simply a database connection.

AFC Classes

AFC classes include frameworks and classes that supply functions such as alert management, monitoring, event management, UI frameworks, and reporting. Javadocs for these classes are located at *install_dir*/install/studiodocs/afc_javadocs.

AFC Class	Description
BaseDocument	Extends java.lang.Object. Implements java.io.Serializable.
DocumentInputStream	AppendCipherInputStream.java extends DocumentInputStream and re-implements some of it methods, which means that any public method added to this class should be also added to AppendCipherInputStream.java.
DocumentOutputStream	DocumentCipherOutputStream.java extends DocumentOutputStream and reimplements some of it methods, which means that any public method added to this class should be also added to DocumentCipherOutputStream.java.
JDBCInputStreamWrapper	Wrapper to an input stream for reading a blob from the database.
JDBCOutputStreamWrapper	Wrapper for writing the blob data to the database.
JDBCService	A singleton class that provides access to one or many connection pools defined in a property file.
Manager	A management interface to the util.frame. Holds configuration values and allows components to access vendor specific configurations.

About MESA Developer Studio SDK

The MESA Developer Studio SDK helps you create and edit custom services and adapters using the Eclipse development environment. The MESA Developer Studio SDK is designed as an Eclipse plug-in and is installed locally on your computer. Use the SDK to create a service, build and export a service package within the Eclipse development environment, then install and test it with the application.

Like the MESA Studio plug-in, the SDK plug-in runs independently from the application in the Eclipse IDE.

Upgrading from Previous Versions

MESA Developer Studio SDK is available for use with the application. You cannot open projects created with the deprecated Service SDK in MESA Developer Studio SDK; however, you can import your existing java files from an old project into a new SDK project. MESA Developer Studio SDK includes all of the previously available Service SDK features, and includes new and enhanced features such as code editors, validation, consistency check, and wizards that guide you through specific tasks.

Use the MESA Developer Studio SDK Cheat Sheet

MESA Developer Studio SDK provides a cheat sheet to guide you through the service development process. The SDK Cheat Sheet provides you with information and step-by-step help to create a service by listing the sequence of steps required to create and package a service. As you progress from one step to the next, the cheat sheet automatically launches the required tools for you. If there is a manual step in the process, the step will tell you to perform the task and click a button in the cheat sheet to move on to the next step. Relevant help information is also available to guide you.

To access the SDK Cheat Sheet:

1. Open the MESA Developer Studio SDK perspective.
2. From the Help menu, select **Cheat Sheets**.
3. In the Cheat Sheet Selection window, expand the Application Studio folder and select **MESA Developer Studio SDK** .
4. Click **OK**. The MESA Developer Studio SDK Cheat Sheet opens on the right.

Steps to Create a Service Using MESA Developer Studio SDK

The process of creating and installing a service using MESA Developer Studio SDK involves several steps. The following list provides a high-level overview of what is required.

To create and install a service:

1. Create a new SDK project.
2. Add business logic.
3. Add service parameters (optional).
4. Add any additional objects (optional).
5. Build a service package.
6. Install and run the service in an application test instance and verify that the service works as expected.
7. Install the service in the application production environment.

Start MESA Developer Studio SDK

Start the MESA Developer Studio SDK from your computer in Eclipse. The SDK can run independently from the application; that is, you do not have to be connected to the application instance in Eclipse at all times. However, the WebDAV server must be running the first time you launch the SDK and each time you want to deploy a service package to the instance.

Note: If you are creating an adapter, verify that the third-party system you will use the adapter to connect with is running and working correctly.

To start MESA Developer Studio SDK:

1. Launch Eclipse.
2. From the Window menu, select **Open Perspective > Other**.
3. From the list, select **MESA Developer Studio SDK**.
4. If this is the first time you have launched SDK, you are asked to enter licensing information.
5. Complete the following and click **OK** :
   - Hostname - Type your application server name.
   - Webdav Port - Type the WebDAV port number for your application server.
   - Name - Type a descriptive name for this instance.
   - Username - Type your application username.
   - Password - Type the password for your username.
6. Click **Finish**.

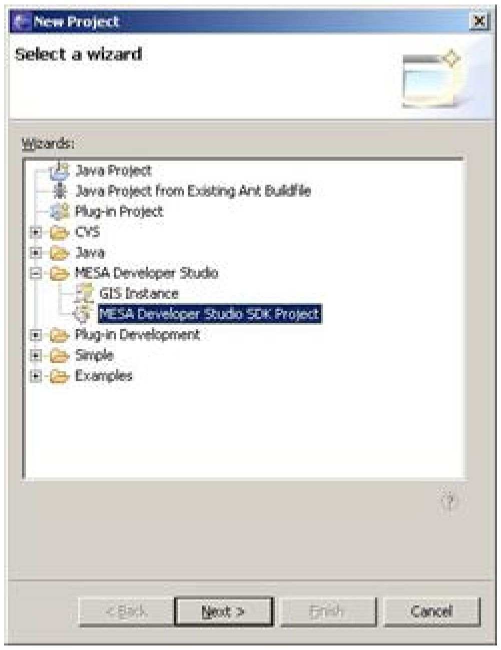
Create a MESA Developer Studio SDK Project

Note: Services developed with the SDK should be tested and deployed in a test environment before being deployed to a production instance.

You can either use the Cheat Sheet or follow these steps:

To create a project:

1. From the File menu, select **New > Project**.
2. Select **MESA Developer Studio > MESA Developer Studio SDK Project** and click **Next**.



3. Type a unique project name (for example, the name of the service you are creating) and click **Next**. Do not use spaces.
4. Complete the following Service Profile information and click **Next**:
   - Service name – Unique name for the service, using Java naming standards. Defaults to project name.
   - Service package - Name of the Java package where the service should be stored. Defaults to com.mypackage.
   - Service label – Name of the service as it should appear in the application UI. Defaults to project name. Make this name unique so that it can be easily recognized in the application.
   - Service description – Meaningful description that will appear in the application UI. Defaults to starting with "This service implements…" but that can be changed or removed.
   - Service Version – Required for the service definition file. System created.

   Note: Whenever you revise the service definition file, you must increase this number (examples: 3 to 4, 3 to 3.1). When you reinstall the service and restart the application, the higher version will overwrite the previous version in the database.

   - Service type - Service or Adapter. If you select adapter, also select whether or not it will be stateful.
5. Complete the Service build options and click **Next**.
   - Document Storage Type Options – How the service stores documents when running. This option is used in both Big A and Little a.
   - Code Generation Options – If checked, the selected Document Storage Type option will be used.
   - Create Project Folder Options – Select the optional folders to be created for the service project.
6. Select the SI Library version. The library version available depends on the instance you are connecting to. You can load additional libraries.
7. Click **Finish**. The project is created. The system creates all required fields for a deployable service.

   Edit the resource and Java files as needed to develop the service.

   Note: Saving the project regenerates the view of the project. The project should be saved anytime a change is made that affects the navigation options for the project. For example, adding a new file or folder creates a new navigation object.

Add Business Logic to a Service

For a service to perform tasks, you add business logic. In this step you extend the generated service code by adding your own business logic to the Big A portion of the service using Java:

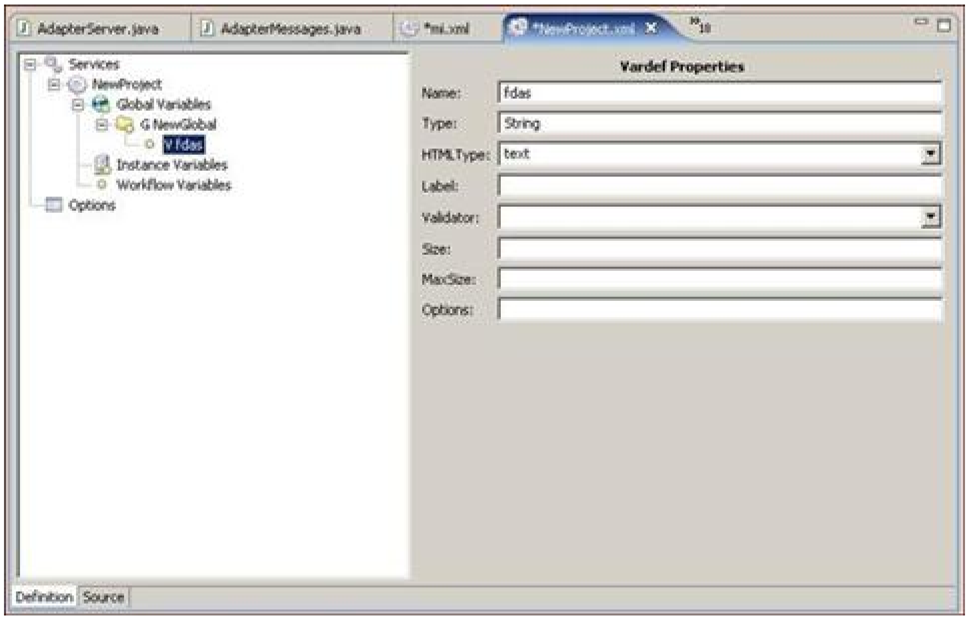
1. From the Window menu, select **Show View** > **Other** > **General** > **Tasks** .
2. In the MESA Developer Studio SDK Package Explorer, find your new service project. Expand the project so that you can see all components of the project.
3. Under **src** > **com.mypackage**, find the folder called ***yourservicename*****Impl.java**, and double-click it.
4. The code for the project will be displayed in the Eclipse editor. Find the line `// TODO: Start here to implement the service logic`.
5. Under that line, you can add logic that fits your service or adapter.
6. Save the project and regenerate the code.

Add Parameters to the Service Definition File

In this step you can define service parameters that can be used to configure and control your Service.

To add a parameter to a service:

1. In the Package Explorer, expand the **servicedefs** node and right-click on ***<service name>*.xml**.
2. Select **Open With > Service Definition Editor**. The service view opens with the parameter group types displayed.
3. Right click on a parameter group type and select **New Group**.
4. Type a title for the new group and click **OK**.
5. Click the new group title to add instructions to the group properties.
6. Add parameters to the group. Right-click on the group title and select **New Vardef**.
7. Type a name for the vardef (variable definition) and click **OK**.
8. Click the new vardef to add properties to this parameter.



Vardef Properties

Name: fdas
Type: String
HTMLType: text
Label:
Validator:
Size:
MaxSize:
Options:

9. Complete the following:
   - Name – required. Name of the parameter as it will appear to the user. System provided. Cannot contain spaces.
   - Type – required. Java type of the parameter. Default value is string.
   - HTML Type – required. HTML input type of the parameter. Valid values are: Text, Select, and Radio. Default is Text
   - Label – required. Cannot contain spaces.
   - Validator – optional. Type of validator. Select from the list.
   - Size – optional. Number of characters for the parameter display size.
   - Max Size – optional. Maximum number of characters allowed for the parameter.
   - Options – optional.

10. Click **File > Save project**.
11. The Language property file (ui/properties/lang/en/*service name*_en.properties) contains label/value pairs that allow to give labels (for example, variables) a descriptive name in the user interface. If the language property file does not contain a label for each corresponding entry you will receive an error message. Right-click on the error message and select **Quick Fix**.
12. Select the desired fix and click **OK**. The language property file is updated with the new label and an editor with the updated language property file displays.
13. Save the language property file.

Note: Each time a Service Definition file or the language property file is saved, a consistency check is performed between the files.

Adding Resources to a Service

Depending on the service you are creating, you may need to add third-party files such as BPML, scripts, databases, properties, libraries, and .jar files. In addition to adding these resources, you can also remove resources from the project file (the original files are not deleted from their original location). Additionally, you can create folders for any files you want to add to the service project.

The Service SDK adds these files in the location you specify when you create the service. If you add any of these resources, you must add the folders described in the table of optional directories.

Write Log Messages into a Message Log File

MESA Developer Studio SDK supports the user in externalizing log messages strings into separate message property files which can be used in the Java classes of the service. Service projects contain the following java classes for defining log messages:

• Message String Declaration File: *service_name*Messages
• Message Property File : *service_name*Resources.properties

To write log messages:

1. Edit the declaration file *service_name*Messages. Declare a constant string variable for each log message you want to use. The declaration has the following syntax: public static String *MessageID*. It is helpful to start the message ID with a component prefix.

   Example: `public static String ExampleImpl_NoTicketsAvail`

2. Edit the resource file *service_name*Resources.properties. The log message entries have following simple syntax: *`MessageID=Message_Text`*.

   Example: `ExampleImpl_NoTicketsAvail=No more tickets available`

3. Use the log message in the Java Source Code Editor. To write an error message with the XLogger log, the logError method is used. This takes a string parameter as an argument. To pass the name of the message string, write "ExampleMessages" then use the shortcut "ctrl-tab" to open the Eclipse Java Editor auto-complete drop-down box, select the log message ID, and press Enter.

   Example:
   log.logError(ExampleMessages.ExampleImpl_NoTicketsAvail)

Create a serviceinstances.xml File

To create a service instance:

1. Right-click on the project name and select **New** > **File**.
2. In the File Name field, type `serviceinstances.xml`.
3. Click **Finish**.

   A new file is created in the project.
4. Open the file serviceinstances.xml in an XML editor and define the instance.
5. If a serviceinstances.xml file already exists, you can import it into your project. When you deploy the service package, the service instance is created automatically.

Example (Serviceinstances.xml)

In the following example of an serviceinstance.xml file, an Adapter instance MyExample is created for the Adapter Example which is described in the service definition file below. The Adapter Example has only one instance variable, *UserName*. In the Adapter Instance MyExample, the instance variable UserName is configured with the value "Smith."

```
<?xml version="1.0" encoding="UTF-8"?>
<services>
<service parentdefname="ExampleAdapter"
name="MyExample"
      displayname="Example Adapter"
      description="Test Instance of ExampleAdapter"
targetenv="all"
activestatus="1"
systemservice="0"
parentdefid="-1">
<parm name="UserName" value="Smith"/>
</service>
</services>
```

Service Definition File

```
<SERVICES>
<SERVICE name=" ExampleAdapter "
      description="example.description"
      label="example.label"
      implementationType="CLASS"
  JNDIName="com.mycompany.example.ExampleAdapter"
          type="Adapter"
          adapterType="STATEFUL"
          adapterClass=" com.mycompany.example.ExampleAdapterImpl"
          version="3.0"
          SystemService="NO">
 <VARS type="instance">
    <GROUP title="example.group1.title"
        instructions="example.group1.instructions">

     <VARDEF varname="UserName" type="String" htmlType="text"
```

```
             validator=" ALPHANUMERIC " size="20" maxsize="40"
             label="example.username" />
    </GROUP>
  </VARS>
</SERVICE>
</SERVICES>
```

Change the SDK Library Version

If you need to use SDK Libraries other than those supplied with your version of the application (platform_library.jar and Studio-API.jar), you can switch to a different (newer) SDK Library.

To change library versions:

1. Download the required two jar files to the Windows machine where Eclipse and MESA Developer Studio SDK are installed.
2. Navigate to the directory of your Eclipse installation to the *Eclipse-root*\plugins\com.sterlingcommerce.mesa.servicesdk_3000.0.0\lib directory. The subdirectory 3000 contains the two SDK Libraries (jar files) delivered with this version.
3. On the same level, create a subdirectory containing the name of the additional or new application version and copy the two jars from step 1 into the new directory.
4. Navigate to the *Eclipse-root*\plugins\com.sterlingcommerce.mesa.servicesdk_3000.0.0\res directory. The subdirectory 3000 contains the file AntExport.xml which is used by the Export Wizard.
5. In the *Eclipse-root*\plugins\com.sterlingcommerce.mesa.servicesdk_3000.0.0\res directory, create a new subdirectory with the name of the new Mesa Developer Studio version (for example, 3000.0.1).
6. If you did not create a new AntExport.xml file, copy the existing file from the res\3000 directory to the new directory. If you created a newer AntExport.xml file, copy that one to the new directory instead.
7. Repeat steps 1 - 6 for all different versions of SDK Libraries you require in the SDK.
8. Start Eclipse and switch the perspective to MESA Developer Studio SDK.
9. If you want to change the application version of a service that is already created, select SDK Libraries [3000] from your project directory.
10. Right-click and select **Configure.**
11. From the list, select the SDK Libraries you added using steps 1 -6.
12. Click **Finish**. The new SDK library version is now available from the list in the New Project Wizard.

Export a Service for Deployment

Once you have created a service you can package it for installation into the application. In this process, you bundle all of the required service resources from your project in a package (Jar-archive) that can be deployed on your application system.

To build a service package:

1. In Eclipse, select the project you want to export in the Package Explorer. (You can package more than one service at the same time by using the **Ctrl** key when selecting.)
2. Right-click and select **Export.**
3. In the Export window, select **Mesa Studio** > **Service Packages** as the export destination and click **Next.**

4. Browse to select the destination directory.
5. Click **Finish**. The service package *service name_version*.jar is built and placed in the *selected package folder*/*service name*/dist/*service name folder*. You may be prompted to save resources before the export is executed.

   Note: The export process always exports the entire project even if you selected only one or more of its subcomponents.

   The export process writes to the AntExport.log file in the *destination directory*/*adaptername* directory with the results of the packaging process. The service is now ready to be installed into the application.

Install a Service into the Application

After you create a service and package the source code, you must install the service package into the application.

Note: Before you install the service package into a production environment, you should install and test it in a test environment.

To install a service package:

1. From the MESA Studio perspective, choose the application instance where the service will be installed.
2. Right-click and choose **Install Service Package**.
3. Browse and select the package file. Click **Open.**
4. Click **Finish** to begin the installation of the service on the application instance.
5. After the service package is installed, restart the application instance.
6. Log in to the application and ensure that the service definition can be viewed and configured by using the options available from the **Deployment > Services > Configuration** screen.

Update a Service Definition

Anytime a service definition is modified, the version must be changed in the service definition (.xml extension) file. When the updated service is exported to your application, the version numbers are compared. If the new version number is greater, the old service definition is overwrtten in the database by the updated files.

To update a service definition:

1. Make changes to files for your service as needed.
2. In the Package Explorer, expand your project and open the servicedefs folder.
3. Doubleclick the service definition (.xml extension) file. A service.xml tab appears at the top.
4. Click the Design tab at the bottom and expand SERVICE to display the attributes for your service.
5. Replace the value for the version attribute with the new version number.

   Restriction: The new version number must be incremented by a whole number or a decimal. Strings such as 2.1.1 cannot be used.

   Note: Alternately, click the Source tab at the bottom and edit the .xml file directly. Example:

   name="FileRename" type="Service" version="3.1"

6. Save and close the service definition (.xml) file.
7. Rebuild (export) the service.
8. Reinstall the service into the application. If asked whether previous files should be overwritten, click Yes.

Architecture

Introduction to Application Architecture

The application executes customer-specific business processes. An XML-based business process model directs the order of all processing activities in the application.

Business Process Definitions

Application business process definitions are based on the draft Business Process Modeling Language (BPML) specification from the Business Process Management Initiative (www.bpmi.org). Business process definitions are stored in XML and can be created iin any editor that can export the XML format recognized by the application.

Services

The application views every activity in a business process as a *service*. A service can initiate:

- Legacy programs
- ERP systems
- Perl scripts
- Java code
- Decision engines
- Most computer programs

The application supports reuse of business processes, which allows activities to be implemented as a service, a business process, or subprocess.

Reuse also enables business processes to be written with multiple reusable components or as a single large service. For example, RosettaNet™ support can be implemented as multiple activities strung together to form a business process or as a single service.

There are several basic types of services in your application, as described in the following table:

Type	Description
Internal	Services that are completely inside the application. Although internal services accept parameters and produce results, they do not directly interact with external systems (systems outside the application).

Type	Description
Input	Services that receive data from external systems.
Output	Services that send data to external systems.
Transport Adapter	Services that use communications protocols like FTP and HTTP to move data into or out of the application.
Application Adapter	Services that interact with external application systems.

Adapters

Adapters are generally defined as services that interact with external systems. They are special cases of services that interact with external systems, or that store or manage state data outside of the workflow context.

Components of a Service

Every service accepts a business process state and produces a modified business process state or workflow context (WFC). Every service also has a harness. For the service, the harness performs the following functions:

1. Receives the input WFC
2. Extracts the information from it that the service needs
3. Runs the service
4. Places the results from the service in a new WFC or output for future steps in the business process workflow

Example of a Service

A FileRename service uses the java.io.File class renameTo() method to rename files. The service takes three input parameters: sourceFile, destinationFile, and overwrite, a Boolean flag to indicate if the target file is being overwritten.

Workflow Context

The workflow context (WFC) object is the service's primary API to the engine. The WFC represents the business process state after each service has run. The WFC input to a service is written to a database. The service step is complete after the new WFC is placed in persistent storage.

If the application stops, it can be restarted from the persisted WFCs by finding the most recent WFCs and sending those requests to the appropriate services. Services can be restarted automatically. Adapters, which are put in a halting state when the application starts, require user intervention to restart them.

Basic Service Framework

The basic service framework, or harness model, enables the application to view all services in the same way. For example, both the Translation service and the File System adapter have harnesses. Although they are different services, they support the same API, which is represented by the harnesses.

Some adapters, such as the SAP and BEA Tuxedo adapters, are used to harness a system that is outside the control of the application. Although these adapters perform diffierent activities, each has a harness that presents a consistent interface to the rest of the system. The harness is generic, but the adapter itself is specific to the system it interacts with. Using the basic framework, you can start, configure, and stop an adapter for an external system in the application interface. The actual operations of the external system are separate.

The harness enhances system performance. For example, the harness wrapped around the Translation service caches and reuses translation maps. The actual Translation service is unaffected by this action. This independence is especially important when the wrapped service is outside the control of the application.

Special Service Capabilities

The application supports the following unique capabilities, which provide flexibility in managing services:

- Large file support – The ability for services to handle files larger than available memory. This can be an effective way to help manage load sharing.
- Service groups – The ability to group “like” services together and treat them as a pool of services
- Storage types – The ability to select the document storage type for a service, such as Database or File System

Relationship Between Business Processes and Services

Before you can run a business process definition, you must validate and compile it. Validation looks for and reports certain known issues.

Compilation breaks the definition into smaller chunks: a header and entries. The header specifies global properties of the business process definition. Each service within the definition has an entry. The compiled information for each service orchestrates coordination between that service and subsequent services and the parameters that they require. This coordination information uses the status of a service to determine the next step in the flow of execution from the service to the engine.

Business Process Example

Service A is orchestrated in such a way that, if it returns success, it directs processing to continue to Service B, but a failure directs processing to continue to Service X.

The application stores the compiled information for each service in the compiled ActivityInfo. ActivityInfo contains an abstract description of the service, such as "Translate the current document using map 5."

Compilation enables the application to predetermine the start node of a business process. This capability makes the business process easy to instantiate and run and prevents the application from repeatedly parsing XML. Compilation also reduces the number of database queries because the next-step pointer is stored in the current activity information.

Starting a Business Process

The application supports dynamic selection (bootstrapping) of business processes. To specify dynamic selection of a business process, configure an adapter to select a business process definition by matching one or more adapter properties.

Input data enters the application through an input adapter. An input adapter performs the following functions:

- Receives data from an external system
- Puts the data and any metadata into an initial workflow context (IWFC)
- Calls the IWFC start method to start the business process, which causes a business process definition to be found and instantiated for the input data

The application starts a new process and the following steps are taken:

1. A new WFC is created.
2. The WFC is put in persistent storage.

3. The application starts the associated business process.

Many-to-Many Relationship

The separation of business logic (BP) and the endpoint (adapter) allows for a many-to-many relationship between adapters and business process definitions. Using the metadata given to the IWFC, one adapter can start several business processes.

Conversely, several adapters can start the same business process. A many-to-many relationship between adapters and business process definitions enables the application to focus on business problems, not just on how data arrives.

Making an input adapter the first step in a business process impairs the many-to-many relationship and keeps the business process from being reused as a subprocess.

Business Process Definition Fails to Start

If an adapter tries to start a business process definition that does not exist or is disabled, the application saves the request to start the business process definition and any related documents within the application. The user can use the business process monitor to view error messages for any business process definitions that failed to execute.

- If the business process definition cannot be found, the user can do an advanced restart and select a different business process definition, which uses the same input data.
- If the business process definition is disabled, then when the user enables that business process definition, the application automatically resumes any instances of that business process definition that stopped.

To ensure that an adapter catches the InitialWorkFlowContextException, code its logic accordingly:

```
{
  iwfc.start()
}
  catch
(InitialWorkFlowContextException)
 {
//do not delete our data here if this happens
//set the appropriate response to the user
}
```

To enable an adapter to start a business process with more than one document, code the following commands:

```
//for a single document
Document doc = new Document();
etc.
iwfc.putDocument(doc)
//for more than one document
Document doc1 = new
Document();
etc.
Document doc2 = new Document();
etc.
iwfc.putDocument(name1, doc2);
iwfc.putDocument(name2, doc2);
```

name1 and *name2* are unique keys for the document within the application. When there is only one document, the application assigns the unique key of PrimaryDocument.

If a service needs to write more than one document, the service calls:

```
wfc.putDocument(name,doc);
```

For a single document, the service calls:

```
wfc.putDocument(doc))
```

To get a specific document from a set, the service calls:

```
wfc.getDocument(name)
```

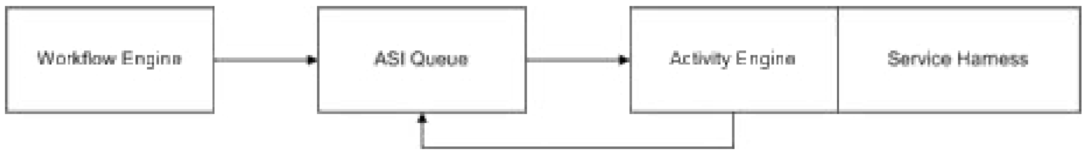
Running a Business Process

When a business process starts, the workflow engine (WFE) executes the services defined in the business process definition and the WFE creates a workflow context (WFC) from the initial workflow context (IWFC). The WFE uses the compiled ActivityInfo to get information about the first service to call. Next, the WFE puts the WFC on the ASI queue, so that the client initiating the business process does not wait for it to complete.

The WFE analyzes the compiled information to determine the current activity (service) that needs to be run. This information is stored in the compiled ActivityInfo. The ActivityInfo contains an abstract description of the service.

The WFE determines, for example, how the Translation service has been configured to run.

The following figure shows the execution cycle:



The ASI queue acts as a hand-off point. It does more than routing-it guarantees that the Java thread of execution is not interrupted during the running of the business process. The listener attached to the ASI queue is a lightweight activity engine. The activity engine takes the WFC off the ASI queue and invokes the service. Logically part of the workflow engine, the activity engine calls the service, takes the results from the service, and immediately starts the next cycle, determining the service that needs to be called and requesting that service on the ASI queue.

The activity engine can determine the next service because the harness has analyzed the service results and set the state values in the WFC. The activity engine consults these values to determine the next activity. The activity engine uses the return code from the current service to choose the next activity from a set of potential activities listed in the current ActivityInfo.

Service Harness Implementation and Service Adapter Implementation

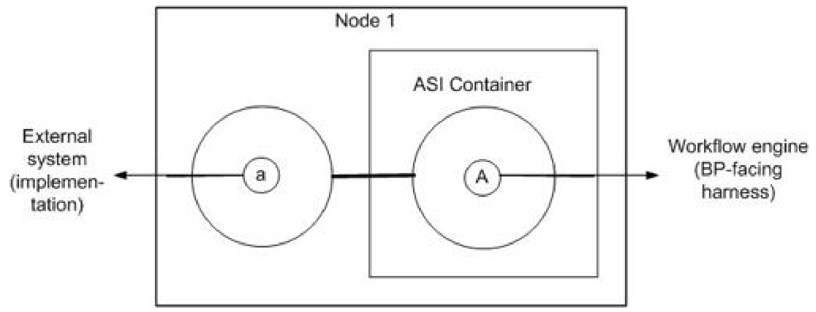
For the development of adapters, the service architecture provides a clear separation of concerns. There are two parts to each adapter implementation:

- Service harness implementation is the part of the adapter inside the ASI container. It provides interaction and interface with the workflow engine.
- Service adapter implementation is the part of the adapter outside the ASI container. It provides the interface to the external system.

This allows the service adapter implementation to run independently of the container if necessary, and gives more implementation options for the developer. Services that do not to interface beyond the process boundary of the application do not need a service adapter implementation component .

The following figure shows an adapter implemented with a service harness implementation and service adapter implementation.

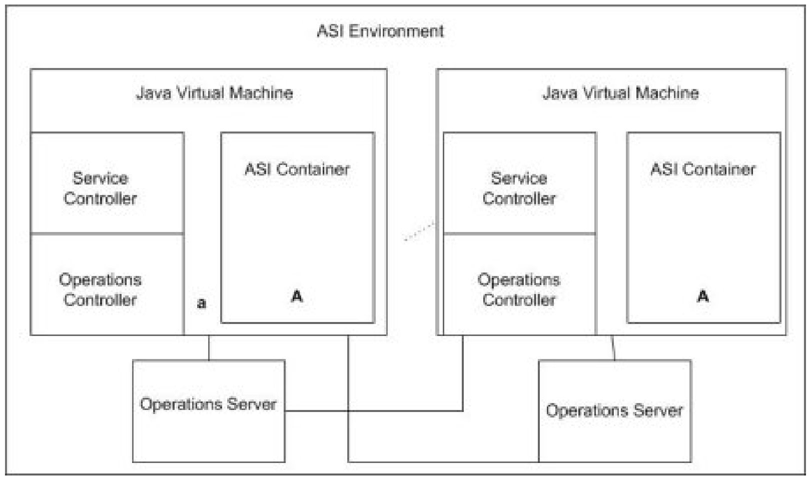
The service harness implementation automatically scales and is portable across clusters because it is instantiated from within the ASI container. The service adapter implementation is tied to a specific computer. The service harness implementation can move around from one call to the next. The service adapter implementation, however, is fixed next to the resource it is accessing.


Node 1
ASI Container
External system (implemen- tation)
a
A
Workflow engine (BP-facing harness)

Example of Service Harness Implementation and Service Adapter Implementation

A cluster of computers in a ASI environment has private disk space on one computer. The service adapter implementation portion of the File System adapter must be on the computer that can access the disk. The service harness implementation portion of the File System adapter, however, can run in any container on any computer.

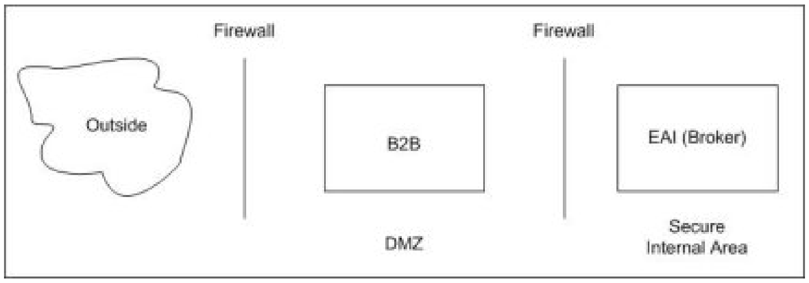
In the following figure, the service harness implementation is moved to a different container in a different Java VM:



B2B Server

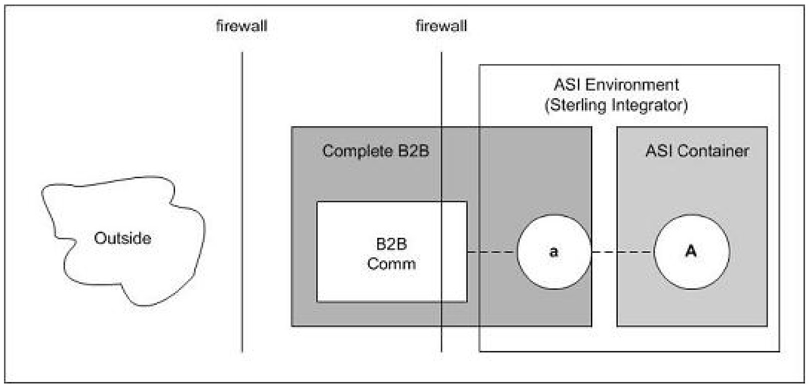
The application includes a business-to-business (B2B) server. The B2B server can be viewed as an independent system.

The following figure shows a traditional model of B2B and enterprise application integration (EAI):



However, within the application, it is more appropriate to view the B2B server as a complex adapter. The B2B server has a two-part service adapter implementation. One part runs in the DMZ and one part runs in the ASI environment.

The following figure shows a B2B server as a complex adapter:



Secure DMZ

The part of the B2B server that runs in the DMZ performs communications activities only. It stores no data. Trading profiles are stored in the secure area where the application resides. The part of the B2B server in the DMZ can run in a simple Java Virtual Machine (JVM) or a complete ASI environment.

The part of the B2B server in the DMZ and the part of the B2B server inside the secure area communicate as if they were separate systems and not part of a single ASI environment.

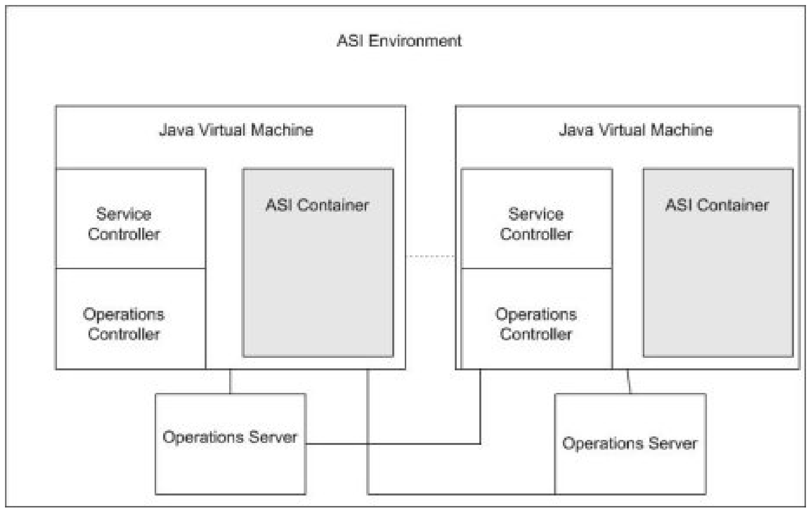
Service and Operations Controllers

The application service and operations controllers monitor and manage executing services and workflows within the application environment. These controllers free system operators and business analysts from having to attend to application-server details.

Service controllers provide a single place within a VM to manage, configure, query, and cache all service-related information. They also enable the application to scale and manage the service adapter implementation parts of adapters. There is one service controller per VM in the ASI Container.

Operations controllers manage resources across VM boundaries. You can have multiple operations servers for redundancy and several embedded components, one per VM. Operations servers provide a single point of contact for all operational questions.

The following figure shows the service and operations controllers:


ASI Environment
Java Virtual Machine
Service Controller
Operations Controller
ASI Container
Java Virtual Machine
Service Controller
Operations Controller
ASI Container
Operations Server
Operations Server

Developing a Service

Adapter Architecture Summary

An adapter interacts with external systems to get data in and out of them. Typically, an adapter consists of a harness, a Remote Method Invocation (RMI), and files that enable the adapter to be used in the application interface.

Harness

The harness part of the adapter must implement the processData() function, which the activity engine calls whenever it has work for the adapter to perform. This function can be called to push data out of the system or signal that data needs to be collected.

RMI (Service Adapter Implementation)

Preferably, most of an adapter workload should reside in the Service Harness Implementation. The service adapter implementation exists mainly to allow for a separation of concerns from the harness.

Typically, adapters do some work that would be inefficient or hinder operation completely within the ASI Container. For example, the service adapter implementation can be set up to wait for data to arrive and to periodically poll the external system for data.

Scalability

Services in the application are scalable: for adapters, the service adapter implementation creates a new thread to service each new request it receives. The service adapter implementation is multi-threaded by default. You do not need to write additional code.

About the Workflow Context Used by Adapters

The workflow context API encapsulates a basic unit of work, including all parameters required by the adapter to act on that unit of work.

The workflow context maintains the state of the business process from service to service. It contains, among other things, the document being manipulated by the business process. This is also where each service reports errors and status. The application infrastructure is designed to persist the workflow context between steps.

The workflow context contains several components:

• Input Parameters – Retrieve parameters before beginning the operation
• Workflow Document Body – Set up the document body
• Error Reporting – Set up status and error reporting

Input Parameters

A service should have all of its parameters before doing any of its core logic. The workflow context provides the getWFContent method to retrieve parameters:

```
getWFContent(String parmName);
```

The getWFContent method retrieves a named input parameter from the workflow context. It gets global (service type), copy (service configuration), and WFD (workflow definition or business process definition) level parameters.

WFD parameters override service configuration parameters, which override service type parameters. If the workflow content message contains a string value with the parmName requested in the getWFContent method call, then the value in the workflow content hash table overrides all other values.

The hash table can contain any type of object. The getWFContent method enables a parameter to override a higher-level parameter only if its value is stored as a string in the hash table.

For example, the following string would retrieve an input parameter named URL:

```
String url = getWFContent("URL");
```

The service may need to get a parameter passed at run time by a previous service. If the parameter is not a service type, service configuration, or WFD parameter being overridden, then the service calls:

```
getWFContent(parmName);
```

Workflow Document Body

A typical service or adapter operates on a document contained within the workflow context. The document contains the body of the document, information about the name of the body, and metadata that describes the document.

To retrieve the document from the workflow context, use the streaming method:

```
InputStream inputStream = document.getInputStream();
byte[] body = new byte[102400];
int read = 0;
int position = 0;
while (inputStream.available() > 0) {
    read = inputStream.read(body, position, 102400);
    position += read;
    //do something with the bytes read
}
inputStream.close()
```

After the document is retrieved, you can obtain the body of the document by using streaming APIs. The following example shows a new document body inserted into the workflow context:

```
OutputStream outputStream = null;
Document document = wfc.createDocument();
document.setBodyName("somename");
try {
    outputStream = document.getOutputStream();
    outputStream.write(someByteArray);
    /*
      A more likely scenario is a looping construct where we read
      from one stream and write to another.
    */
} catch(Exception e) {
    theAdapterLog.logError("Houston, we have a problem." e);
    wfc.setBasicStatusError();
    sci.unregisterThread();
} finally {
    //Always make sure we close out document output streams.
    try {
        if (outputStream != null) {
            outputStream.close();
        }
    }catch(Exception e) {
        theAdapterLog.logError("Error closing document in adapter EXAMPLE." e);
    }
}
wfc.putPrimaryDocument(document);
```

Error Reporting

Another important component of the workflow context is status and error reporting. The workflow engine requires an adapter to return status information at the completion of the requested activity. The requesting business process uses this information to make decisions that control business process flow. Status is reported within the workflow context.

About the Service Controller Framework Used by Adapters

The service controller is a unified framework that all adapters use to remove application-server dependencies.

The service controller is also responsible for starting and stopping the adapter.

Stateless and Stateful Adapters

Stateless and stateful adapters differ at the object level. For stateless adapters, the service controller instantiates one object that services all configured copies of the adapter. Each request to the service adapter implementation of the adapter must be a complete request, because states cannot be maintained between requests. For stateful adapters, the service controller instantiates one object for each configured copy of the adapter.

Instance variables for RMI objects are not useful because multiple threads (or, in the case of stateless adapters, multiple copies of the adapter) have access to the same instance variables, as if they were class variables.

Method variables are unique to the invocation of the method, so they are acceptable to use.

Service Controller Interface

An adapter is composed of two parts:

• A harness that is the interface to the workflow engine
• An RMI server that communicates with external systems

The code example below shows how the adapter processData method does the following:

1. Registers with the service controller
2. Finds its RMI service
3. Invokes an RMI method
4. Unregisters with the service controller
5. Returns this code to the workflow engine:

```
String svcName = wfc.getServiceName;
ServicesControllerImpl sci = ServicesControllerImpl.getInstance();
sci.harnessRegister(new Integer(wfc.getWorkFlowId()).toString(),
   svcName);
        try{
           rmi = (Yourserver)sci.getAdapter(svcName);
        }catch(Exception e){
           wfc.setBasicStatusError();
           sci.unregisterThread();
           return wfc;
        }
        if ( rmi == null ){
           wfc.setBasicStatusError();
           sci.unregisterThread();
           return wfc;
        }
        try {
        // request “Service Adapter Implementation” to do work
        rmi.someRMIMethod(parms, xmlInBytes);
        }
        catch(Exception e) {
            wfc.setBasicStatusError();
            sci.unregisterThread();
            return wfc;
        }
```

This code example uses the following methods to accomplish its work:

Method	Description
harnessRegister(workflowID, serviceName)	Assists the service controller in its monitoring and control functions. The harnessRegister should be called at the beginning of the processData method. Returns – None.
UnregisterThread()	Assists the service controller in its monitoring and control functions. It should be called before the return of the processData method. Returns – None.

Service Controller Interface – RMI

The IAdapterImpl class provides the following methods for use in the service adapter implementation of the adapter:

Method	Description
startup()	The service controller calls startup() after a stateful adapter is created. Startup() should perform all setup and initialization necessary for the adapter to function correctly. This method returns a Boolean value. A true return indicates that startup was successful.
shutdown()	The service controller calls the shutdown() method when a stateful adapter shutdown is required. shutdown() should perform all operations necessary to shut down the adapter. In the case of a multi-threaded adapter, shutdown() must ensure that all of its threads are stopped before returning to the service controller. shutdown() should wait for threads that are performing work directly for a workflow to become quiescent. The adapter can stop threads that are waiting for external input. The IAdapterImpl base class provides the methods interruptThreads() and stopThreads() to assist in shutting down errant threads. As an additional assistance to the shutdown() method, the base class provides a count of registered threads. This count can be found in the invokes variable. The shutdown() method should poll this variable no more than once a second, waiting for it to decrement to zero. Note that the shutdown thread does not appear in this count. If the adapter has threads that listen on sockets, the invokes count may never go to zero. The shutdown() method will need to account for this case. The service controller will not wait indefinitely for shutdown() to fulfill its responsibilities. The adapter can configure the time-out period by overriding getShutdownTimeout(). The service controller enables shutdown() for the amount of time returned from getShutdownTimeout(), and then the shutdown thread is terminated. This method returns a Boolean value. A true return indicates that the shutdown() method completed successfully.
refresh()	The service controller calls refresh() when the configuration changes for a stateful adapter. refresh() should perform all setup and initialization necessary for the adapter to function correctly. If an adapter maintains a connection or connections to an end system, and the connectivity configuration changes, refresh() should return a false value to the service controller. In this case, the service controller shuts down the adapter and then restarts it. This action prevents workflows from trying to invoke the adapter while connectivity changes are occurring. This method returns a Boolean value. A true return indicates that refresh() completed successfully. A false return indicates that the adapter did not refresh, in which case the service controller shuts down the adapter and then restarts it.

Method	Description
getShutdownTimeout()	The service controller calls getShutdownTimeout() to determine how long to wait for shutdown to complete before terminating the adapter threads. A default implementation provided in the base class returns 60 seconds. This method returns the period for shutdown to complete, in milliseconds.
interruptThreads()	The interruptThreads method is provided in the base class to assist the adapter shutdown() method in shutting down its active threads. interruptThreads() calls the interrupt method on each active thread and assists in a graceful shutdown where possible. Returns – None.
stopThreads()	The stopThreads method is provided in the base class to assist the adapter shutdown() method for active threads. stopThreads() terminates every active thread. Returns – None.
registerThread()	The registerThread method assists the service controller in its monitoring and control functions. registerThread() should be called at the beginning of each method called by the service harness implementation portion of the adapter. Returns – None.
unregisterThread()	The unregisterThread method assists the service controller in its monitoring and control functions. unregisterThread() should be called at the end of each method called by the service harness implementation portion of the adapter. Returns – None.

Error and Status Reporting

The workflow engine requires a service to return status information at the completion of the requested activity. The requesting business process uses this information to make decisions that control business process flow.

Three types of status information are returned to the workflow engine:

• Basic status
• Advanced status
• Exceptions

Basic Status

Basic status is the overall status of the work performed by the service.

```
wfc.setBasicStatus(status)
```

For example:

```
wfc.setBasicStatus(WorkFlowContext.ERROR);
```

Advanced Status

The advanced status is a modifier for the basic status. Reporting the advanced status requires the service writer to analyze the service error categories. Business analysts use the list of advanced errors to test for error conditions. The list should be representative, but not long.

The workflow context provides the following method for setting advanced status:

```
wfc.setAdvancedStatus(StringadvancedStatus);
```

Exceptions

An exception indicates a possible failure condition. A service may need to generate a workflow exception when a required input parameter is:

- Missing
- Invalid
- Disabled. For example, a map or a workflow definition is disabled.

Use the following syntax to construct an exception:

```
new
WorkFlowException(String errorText, int reasonCode)
```

Workflow exception reason codes are:

- public final static int GENERAL_PARM_ERROR = 0; (Default) Use this in situations that require, for example, missing properties files.
- public final static int MANDATORY_PARM_MISSING = 1;
- public final static int INVALID_VALUE_FOR_PARM = 2;
- public final static int RESOURCE_DISABLED = 3;
- public final static int NO_DOCUMENT = 4

The workflow engine places the error text string for the workflow exception into the status report.

Status Report

The service can add text describing the activity performed to the workflow context. The workflow context uses the following method for this purpose:

```
setWFStatusRpt("Status_Report",statusReportText);
```

A status report is available to view if an Info icon appears in the Report column. Click the icon to display the status report for that service. You can view the status report text from the application interface.

Successful Invocation	Use the application interface to view the progress of a business process as it executes; use the business process monitor to view details after a business process has run. When a service is successfully invoked, the Status column displays *Success*. This display is a result of calling the setBasicStatusSuccess() method in the workflow context. In this case, the advanced status does not need to be set.
Unsuccessful Invocation	An unsuccessful invocation of an adapter or service should result in the Status column displaying *Error*. This display is a result of calling the setBasicStatusFailure() method in the workflow context. The setAdvancedStatus() method is also called to give additional information about the failure condition.

Setting Up User Prompts for Configuring a Service in the UI

The console can be set up to prompt for configuration information specific to the service being developed. Simple service configuration does not require you to write any Java code. The only requirement is that the service XML file is set up to specify the information to be collected.

To set up the console, use the following files:

• Language-specific properties files - Provides screen text in the language chosen by the user
• Service XML file - Describes the information collected at configuration time

Language-Specific Properties Files

The language-specific files should exist for each service XML file. It is also customary to pick a two- or three-character service abbreviation for the service and prepend this abbreviation to each language-specific property name. For example, for a file system, *fs* would indicate file system and look like *fs.label* or *fs.description*. This convention helps to ensure that the language-specific property names are unique.

Here is an example of a language-specific properties file:

```
fs.label = File System Adapter
fs.description = Collects and Extracts files from a file system.
fs.wfd.group1.title = Workflow Properties
fs.wfd.group1.instructions = Specify the appropriate workflow settings.
fs.instance.group1.title = Collection
fs.instance.group1.instructions = Specify the appropriate settings for collecting
data using the File System Adapter.
fs.action = Action
fs.cfolder = Collection Folder name
fs.efolder = Extraction Folder name
fs.pollinterval = Poll Interval (mins)
```

Service XML File

Here is an example of the service XML file:

```
SERVICE name="FileSystem"
description="fs.description"
label="fs.label"
implementationType="CLASS"
JNDIName="FileSystemEJBHome" type="Adapter"
adapterType="STATELESS"
adapterClass=
 "com.sterlingcommerce.woodstock.services.filesystem.FileSystemServerImpl"
  version="1.0" SystemService="NO">
```

A service XML file includes these variables:

Variable	Description
name	Descriptive name of the service
description	Description in language-specific form

Variable	Description
label	Descriptive name in language-specific form
implementationType	Either RMI or CLASS
JNDIName	Lookup name of the service
type	Adapter, basic, Advanced, Split, or Join
adapterType	STATEFUL or STATELESS
adapterClass	Class name of the Service Implementation of the adapter
version	1.0 - System-assigned value when the project is created. If the service definition changes, the version number must be incremented before the new definition is exported. See Update a Service Definition.
systemService	NO for adapters

<VARS> Tags

Inside the <Service> tag are <VARS> tags. The <VARS> tags contain definitions of configuration items to collect from the user. Three types of <VARS> tags correspond to the scope of the configuration:

- global - The widest possible scope, applicable to all adapters of this type. Configuration parameters are displayed in the **Deployment > Services > Installation/Setup** section.
- instance - Limited in scope to a single instance of an adapter. Configuration parameters are displayed in the **Deployment > Services > Configuration** section.
- wfd - Workflow definition configuration for use only by the Graphical Process Modeler. The primary purpose of this tag type is to define the possible configuration that can be made in the workflow definition. Because the configuration is defined here, the Graphical Process Modeler can display the possible configuration to the user.

<VARS> Tag Example

Following is a sample <VARS> tag:

```
<VARS type="instance">
```

<GROUP> Tags

Inside the <VARS> tags are <GROUP> tags. <GROUP> tags group configuration information by page. The <GROUP> tag is part of the <VARS> tag and contains title and instructions.

- title - title of the current page
- instructions - help describing what the user is supposed to do with the current page.

The following is an example of a <Group> tag:

```
<GROUP title="fs.instance.group1.title"
instructions="fs.instance.group1.instructions">
```

<VARDEF> Tag

Inside the <GROUP> tag is the <VARDEF> tag.

The following table describes the <VARDEF> tag elements:

Tag Type	Description
varname	Name of the Java property that a service uses to retrieve data collected from the user.
type	Type of the input, normally String.
htmlType	Type of input you are retrieving from the user. Normal textual information is htmlType text, radio buttons are specified with radio, and password information can be retrieved with the htmlType Password. For text area, specify textarea. For drop-down list, specify the htmlType select.
validator	Validation types-for example, ALPHANUMERIC specifies that only alphabetic and numeric characters are accepted as input. NUMBER and NUMERIC specify only numeric validation.
label	Description of the configuration to be entered by the user.
size	Field size that displays to the user when collecting information.
maxsize	Maximum number of characters the user can enter for this variable.
required=NO	Specified only if the variable is not required.
defaultVal	Default value that is useful only if the VARS type is wfd.

The following example shows how all of the tags work together:

```
<VARS type="instance">
   <GROUP title="fs.instance.group1.title"
instructions="fs.instance.group1.instructions">
   <VARDEF varname="collectionFolder" type="String" htmlType="text"
validator="ALPHANUMERIC" size="30" maxsize="250" label="fs.cfolder" />
    <VARDEF varname="useSubFolder" type="String" htmlType="radio"
validator="ALPHANUMERIC" options="radio2" label="fs.subfolders" />
   </GROUP>
</VARS>
```

Setting Up Event Logging for a Service

This section provides general guidelines for logging from a service in the framework. The Logging service is a unified framework for logging messages.

Because services may have different needs and functions, apply the following guidelines only with a good understanding of the service logging output.

Logging Event Guidelines

The following table lists the guidelines for the types of events to log for each logging method:

Method	Guideline
Error	Log any non-exception occurrence that would cause the intended operation of the service to fail, such as: • Communication failed to external system • Invalid input data
Exception	Log any checked exception that is the result of an error or fault, such as a caught exception that is not handled by the adapter. **Note:** Exceptions that are handled by the service should not be logged with logException.
Warn	Log any system disruption that is neither fatal nor a handled exception. Also log any notifications of possible error conditions. Examples: • Connection to external resource is lost while no processing is occurring • Disk space limits • License file expirations
Debug	Log anything that is useful information for development/debugging purposes, including general processing information and functionally relevant occurrences. Examples: • Entering and exiting of major methods • Method input parameters • Method return values • Parameter values (setting and getting) • Initiation and completion of operations
Log	Log anything that should always appear in the log file. This method sends messages to the default log regardless of log level. Examples: • Adapter StartUp • Adapter ShutDown • Adapter Refresh • Connection to external systems

XLogger Logging

The XLogger class provides a unified log output, which includes the following:

• Module name
• Thread ID
• Service name

This class can also be used in stateful RMI adapters (service adapter implementations).

EJB Logging

In the service harness implementation of the adapter, the service name is available in the workflow context and can be used in the constructor when creating an XLogger copy.

For example, use this form of the constructor at the beginning of the processData method:

```
adapterLogger = new XLogger(classsName, getServiceName());
```

The adapterLogger should be an instance variable of type XLogger.

RMI Logging

For the service adapter implementation of the adapter there are two approaches, depending on whether the adapter is stateless or stateful.

For stateless adapters, there is only one object created for all copies of the adapter. Therefore, instance variables are shared by all copies of the adapter.

The service name is not available to the service adapter implementation of a stateless adapter from the infrastructure. The EJB must pass the service name to the service adapter implementation of the adapter. The service name should not be used in the constructor when creating an XLogger copy-for example, adapterLogger = new XLogger(classsName). AdapterLogger should be a class variable of type XLogger. In this case, use the logging method that includes the service name.

For stateful adapters, there is an object created for each copy of the adapter, but all the threads of that adapter copy share the same object. Therefore, all threads of an adapter copy share instance variables.

The service name is available to the service adapter implementation of a stateful adapter from the infrastructure by calling getServiceName().

The following is sample code:

```
adapterLogger = new XLogger(classsName, getServiceName());
```

The adapterLogger should be a class variable of type XLogger. The easiest way to do this is to construct a new XLogger if one does not already exist.

XLogger Logging Methods

You can use the XLogger logging methods in the EJB and RMI parts of the adapter. However, the availability of the service name varies.

Use the XLogger methods for logging from adapters. These methods format the class name and the service name provided along with a thread identifier, and add the message to be logged. The result is a log message that includes date/time, class name, service name, thread ID, and message.

The following table describes the XLogger class methods and indicates where they are used:

Method	Used by Stateful RMI and EJBs	Used by Stateless RMI
XLogger(String ClassName)		Constructor

Method	Used by Stateful RMI and EJBs	Used by Stateless RMI
XLogger(String ClassName, String ServiceName)	Constructor	
logError(String ServiceName, String message)		Logging errors
logError(String message)	Logging errors	
logException(String ServiceName, String message, Exception e)		Logging fault exceptions
logException(String message, Exception e)	Logging fault exceptions	
logWarn(String ServiceName, String message)		Logging warning messages
logWarn(String message)	Logging warning messages	
logDebug(String ServiceName, String message)		Logging debug messages
logDebug(String message)	Logging debug messages	
log(String ServiceName, String message)		General logging
log(String message)	General logging	

LogService Logging Methods

The LogService class provides the following static logging methods. Use these methods at any time in an EJB or RMI part of an adapter. Because each method implies a different logging threshold, some general usage guidelines are provided.

Each log message generated through LogService has a timestamp and logging level. You must supply the name of the originating class and a message.

The following sample code shows an example:

```
[2001-06-12 12:46:55.381] ALL [SiebelEJBBean] Hello World
```

Timestamp	Logging Level	Originating Class	Message
`[2001-06-12 12:46:55.381]`	`ALL`	`[SiebelEJBBean]`	`Hello World`

The following table lists the log methods and general guidelines for their formatting:

Note: For the BPML specifications that accepts, see the *Business Process Guide*.

Method	Format Guideline	Example
logError()	[*Class name*] General business error. Specific application error.	`[SiebelEJBBean] Error: Adapter unable to process request. Action parameter is null.`
logException()	[*Class name*] General business error. Specific application error. The logException() method requires both a message and an exception. This method logs the exception name and stack trace.	`[SiebelEJBBean] Exception: Adapter unable to process Read request. Unable to connect to the file system server.`
logWarn()	[*Class name*] General Warning. Expected results or recommended actions to be taken.	`[SiebelEJBBean] Warning: Available disk space is less than 1 GB. Delete unnecessary files.`
logDebug()	[*Class name*] Debug message	`[SiebelEJBBean] Entering ProcessData method.`
log()	[*Class name*] Log message	`[SiebelEJBBean] Adapter started.`

File Rename Service Example

Example 1: Creating the File Rename Service

To learn about creating a simple service that has parameters, you can use this step-by-step example to create a service called the FileRename service. This example walks through the following tasks in the sequence shown in the table. Note that some are done on your PC, using Eclipse or a text editor; others are done on the server where your application instance is located.

Location where the task is performed	Tool or software to use for the task	Task
On your PC	Eclipse	• Create the service using the MESA Studio SDK Wizard. • Add custom code to the service in the SDK. • Verify, save, and compile the service. • Export the service (createthe service package file).
On your PC	Text editor (NotePad, WordPad, etc.)	Create a text file to use as input for a business process you will build to test the service. This is the file that will be renamed by the service when the business process runs.
On the server	Operating system commands, file management utility, or FTP	• Copy the text file from your local system to the server. Note the location so that you can find it when you are ready to run the business process. • Copy the service package file from your local system to the server. Copy the service to the bin directory of your application installation. • Stop the application. • Install the service in the application (run InstallService.sh/cmd on the server while the application is down). • Restart the application.
-	In the application	• Create a service configuration in the application. • Add information to the service configuration in the GPM. • Create a business process to test the service. • Check in the business process. • Run the business process.

Location where the task is performed	Tool or software to use for the task	Task
		• View the business process results to see if the file was renamed correctly. • Troubleshoot errors, repeating until successful.

Create the FileRename SDK Project

This task is done on your PC using Eclipse.

1. Open the MESA Studio SDK perspective in Eclipse.
2. Select **File** > **New** > **Project**.
3. Select **MESA Developer Studio** > **MESA Developer Studio SDK Project**.
4. Enter **FileRename** as the name for the project and click **Next**.

   The project name is also the name for the service.

5. On the Service Profile screen, enter a description for the service: `The service renames a file passed in from the WFC`. Click **Next**.

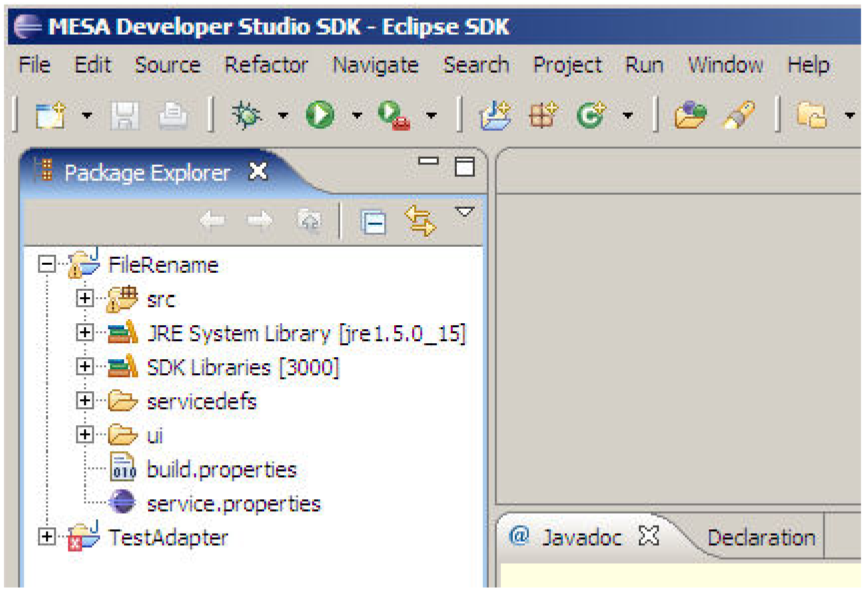
   You can also change the label for the service from the Service Profile screen.

6. On the Service Build Options screen, leave all fields set to the defaults. Click **Next**.
7. On the SI System Libraries screen, select **3000** as the SI library version. Click Finish.

   The project is created for the new service.

8. If the current perspective selected in Eclipse is not Mesa developer Studio SDK, an **Open associated perspective?** dialog displays. Click **Yes**.
9. Open the Package Explorer pane in Eclipse to see the project for the service.

   The project displays as follows:



Tip: Click **Window > Reset Perspective** at any time if needed to refresh your view in MESA Studio SDK.

About System Libraries

The application system libraries includes collections of classes that you can use in your custom services and adapters. See the following directories:

• *install*/xapidocs/api_javadocs/index.html
• *install*/studiodocs/afc_javadocs
• *install*/studiodocs/asi_javadocs
• *install*/studiodocs/ifcbase_javadocs

Jar Files Available

In MESA Developer Studio SDK, you have access to the following Java APIs. These are the jars most often needed in custom service creation. Detailed information about each API is available in the javadocs located in the *install_dir*/install/studiodocs folder in your application installation.

Java objects in the SDK include:

Class or Subclass	Pathname
BaseDocument	*install_dir*/install/studiodocs/afc_javadocs
DocumentInputStream	*install_dir*/install/studiodocs/afc_javadocs
DocumentOutputStream	*install_dir*/install/studiodocs/afc_javadocs
JDBCService	*install_dir*/install/studiodocs/afc_javadocs
JDBCOutputStreamWrapper	*install_dir*/install/studiodocs/afc_javadocs
JDBCInputStreamWrapper	*install_dir*/install/studiodocs/afc_javadocs
Manager	*install_dir*/install/studiodocs/afc_javadocs
Event	*install_dir*/install/studiodocs/ifc_javadocs
EventProcessor	*install_dir*/install/studiodocs/ifc_javadocs
EventProcessorFactory	*install_dir*/install/studiodocs/ifc_javadocs
IEventProcessor	*install_dir*/install/studiodocs/ifc_javadocs
SemaphoreManager	*install_dir*/install/studiodocs/ifc_javadocs
LockManager	*install_dir*/install/studiodocs/asi_javadocs
WorkFlowContext	*install_dir*/install/studiodocs/asi_javadocs
WFCBase	*install_dir*/install/studiodocs/asi_javadocs
InitialWorkFlowContext	*install_dir*/install/studiodocs/asi_javadocs
Document	*install_dir*/install/studiodocs/asi_javadocs

Class or Subclass	Pathname
ServicesControllerImpl	*install_dir*/install/studiodocs/asi_javadocs
ActivityData	*install_dir*/install/studiodocs/asi_javadocs

Add Custom Code to the FileRename Project

This task is done on your PC using Eclipse.

First, create the service parameters.

1. Open the FileRename project in the MESA Studio SDK in Eclipse.
2. In the Package Explorer view, expand the project and double-click **FileRenameImpl.java**.
3. Locate the import statement `import java.text.MessageFormat;` and add the following import statement below it: `import java.io.File;`
4. Locate the comment `// TODO: Start here to implement the service logic:` and add the following lines below it:

```
String sourceFileName = wfc.getParm("sourceFile");
String destinationFileName = wfc.getParm("destinationFile");
boolean overwrite = true;
String overwriteStr = wfc.getParm ("overwrite");
if (overwriteStr!=null){
overwrite = overwriteStr.equalsIgnoreCase("true") ? true:false;
}
log.logDebug("File rename source file = " + sourceFileName);
log.logDebug("File rename destination file = " + destinationFileName);
log.logDebug("File rename overwrite flag = " + overwrite);

File sourceFile = new File(sourceFileName);
if(!sourceFile.isFile()) {
    throw new WorkFlowException("Source is not a file or accessible or does
not exist: " + sourceFileName);
}
File destinationFile = new File(destinationFileName);
if(destinationFile.isFile()) {
    if(!overwrite) {
       throw new WorkFlowException("Destination file exists and overwrite flag
 is set to false: " + destinationFile);
    } else {
        if(!destinationFile.delete()) {
            throw new WorkFlowException("Unable to overwrite destination file:
 " + destinationFile);
        }
    }
}

if(!sourceFile.renameTo(destinationFile)) {
    throw new WorkFlowException("File renaming failed.  Source file = " +
sourceFileName + ", destination file = " + destinationFileName);
}
log.logDebug("File renamed successfully.");
```

5. Save the project by selecting **File** > **Save**.

   If the following warning displays, you may ignore it:`The method handleError(XLogger, WorkFlowContext, String, Object[]) from the type FileRenameImpl is never used locally.`

Create Parameters for the File Rename Service

This task is done on your PC using the MESA Studio SDK in Eclipse.

The FileRename service uses the java.io.File class renameTo() method to rename files. The service takes three input parameters:

• sourceFile: Original and target file names (including full path)
• destinationFile: Original and target file names (including full path)
• overwrite: Boolean flag to indicate if overwriting target file

The service takes the following parameter types:

• The overwrite parameter is defined at the instance level; that is, it is configured using the **Deployment** > **Services** > **Configure** option in the application. (BPML can overwrite this service instance configuration at run time.)
• The parameters Source File and Destination File are defined at the workflow level; that is, their values need to be passed in from BPML (message to service).

You will use the Service Definition Editor in the MESA Studio SDK to create the parameters.

There are two parts to the process of creating a new parameter: first, define the parameter at either the Instance or Workflow level for an option group, vardef information such as label, HTML type (radio button, select, or text) field length, and options. Second, define the parameter options. For this example, the overwrite flag will have an option with true and false choices.

Service Definition Parameters for the File Rename Service

The following tables provide the Service Definition Editor entries required for the three parameters. For parameter descriptions, see Adding Parameters to the Service Definition File.

Overwrite Flag Parameter

The Overwrite flag parameter is used to determine whether the file passed in to the service by the workflow context should be renamed or left "as is."

Field	Value
Parameter Level	Instance
Group Title	Overwrite flag
Group Instructions	select to overwrite existing files

Field	Value
Vardef Properties	
Name	overwrite
Type	String
HTML Type	radio
Label	FileRename.overwrite
Validator	ALPHANUMERIC
Size	5
Max Size	5
Options	overwriteFlag
Options	
Option Properties - Name	overwriteFlag
Choice Element Properties	
First element:	
Value	true
Display name	true
Second element:	
Value	false
Display name	false

Source File Parameters

The source file is passed into the service by the workflow context and renamed.

Field	Value
Parameter Level	Workflow
Group Title	FileNames
Group Instructions	original and new filenames
Vardef Properties	
Name	sourceFile
Type	String
HTML Type	text
Label	FileRename.OriginalFile
Validator	ALPHANUMERIC
Size	80
Max Size	80
Options	n/a

Field	Value
Options	n/a
Option Properties - Name	n/a
Choice Element Properties	n/a
Value	n/a
Display Name	n/a

Destination File Parameter

These are the parameters for the renamed file.

Field	Value
Parameter Level	Workflow
Group Title	FileNames
Group Instructions	original and new filenames
Vardef Properties	
Name	destinationFile
Type	String
HTML Type	text
Label	FileRename.TargetFile
Validator	ALPHANUMERIC
Size	80
Max Size	80
Options	n/a
Options	n/a
Option Properties - Name	n/a
Choice Element Properties	n/a
Value	n/a
Display Name	n/a

Service Definition Parameter Field Reference

When you create a new parameter for a service definition, you must provide information about the type of parameter, options, and so forth. For each service definition, you enter information for some or all of the following:

Variables

You can define a service variable at one of three levels:

- Global – has a constant value for all instances of a service. That is, you can create multiple instances of the service, but this variable will be the same in each.
- Instance – can have a different value for each instance of a service. That is, if you create multiple instances of a service, this variable value can be changed for each, as needed.
- Workflow – can have a different value each time the service is called from a business process. That is, the service gets the value for the variable from the current business process instance.

Options

Options are used with variables to provide a finite, reusable list of possible values. Examples: yes/no, true/false

- Name – Required. Name of the parameter as it will appear to the user. System provided. Cannot contain spaces. Use standard Java variable naming conventions.
- Type – Required. Java type of the parameter. Valid values: String (default), Boolean, integer.
- HTML Type – Required. HTML input type of the parameter. Valid values: Text, Select, and Radio. Default is Text.
- Label – Required. Cannot contain spaces.
- Validator – Optional. Type of validator. Select from the list.
- Size – Optional. Number of characters for the parameter display size.
- Max Size – Optional. Maximum number of characters allowed for the parameter.
- Options – Optional. Reusable lists of values used by the parameter, such as true/false and yes/no.

Add the Overwrite Parameter

This task is done on your PC in Eclipse.

1. Open and expand the FileRename SDK project in the Package Explorer.
2. Expand the servicedefs node. Right-click **FileRename.xml** and select **Open With > Service Definition Editor**.
3. In the service definition editor pane for FileRename.xml, expand the Services node and the FileRename service.
4. Right-click **Instance Variables** and select **New Group**.
5. Type `overwrite flag` in the Group Title field and click **OK**.

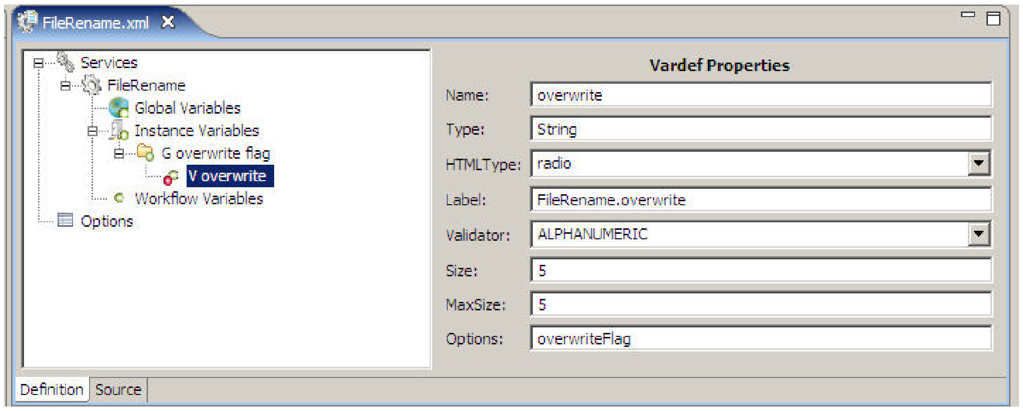
   The new overwrite flag group displays, preceded with the group designation G: `G overwrite flag`.
6. Select the **overwrite flag** group in the FileRename tree.

   The Group Properties pane displays.
7. Type `select to overwrite existing files` in the Instructions field.

   Next, add the option for the new variable definitions.
8. Right-click on the **overwrite flag** group and select **New Vardef**.

   The New Vardef pane displays.
9. Type `overwrite` in the New vardef name? field and click **OK**.

   The new field displays as `V overwrite`.
10. Select the **overwrite flag** variable and edit the values as shown:



Vardef Properties

Name:	overwrite
Type:	String
HTMLType:	radio
Label:	FileRename.overwrite
Validator:	ALPHANUMERIC
Size:	5
MaxSize:	5
Options:	overwriteFlag

Finally, you will create the option that displays the user options for the overwrite flag on the UI.

11. Right-click on **Options**.

    The **New Option** dialog displays.

12. In the **New Option name?** dialog box, enter `overwriteFlag`.
13. Right-click on the **overwriteFlag** option and select **New Element**.
14. In the **New element value?** field, enter **true**.
15. Create an additional new element with the value false.
16. Save your changes to the project by selecting **File** > **Save**.

    You will resolve errors when you add language property file names.

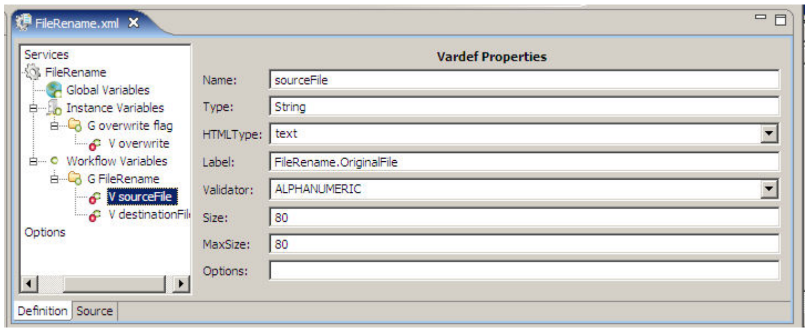
Add the Source File and Destination File Parameters

This task is done on your PC in Eclipse.

1. Open and expand the FileRename SDK project in the Package Explorer.
2. Expand the servicedefs node. Right-click **FileRename.xml** and select **Open With** > **Service Definition Editor**.
3. In the service definition editor pane for FileRename.xml, expand the **Services** node and the FileRename service.
4. Right-click **Workflow Variables** and select **New Group**.
5. Type `FileRename` in the Group Title field and click **OK.**
6. Select the **FileRename** group in the FileRename tree.

   The Group Properties pane displays.

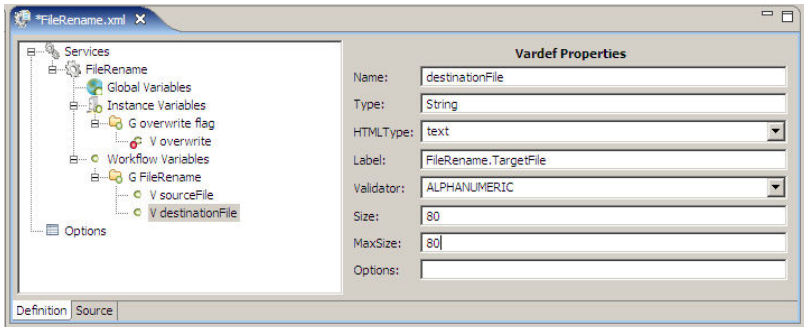
7. Type `original and new filenames` in the Instructions field.
8. Right-click the **FileRename** group and select **New Vardef**.
9. Type `sourceFile` in the Name field and click **OK**.
10. Select the **sourceFile** vardef to display the properties pane. Type or select the value shown for each field in the following example:



Vardef Properties

Name: sourceFile
Type: String
HTMLType: text
Label: FileRename.OriginalFile
Validator: ALPHANUMERIC
Size: 80
MaxSize: 80
Options:

11. Right-click the **FileRename** group and select **New Vardef**.
12. Type `destinationFile` in the Name field and click **OK**.
13. Select the **destinationFile** vardef to display the properties pane. Type or select the value shown for each field in the following example:



Vardef Properties

Name: destinationFile
Type: String
HTMLType: text
Label: FileRename.TargetFile
Validator: ALPHANUMERIC
Size: 80
MaxSize: 80
Options:

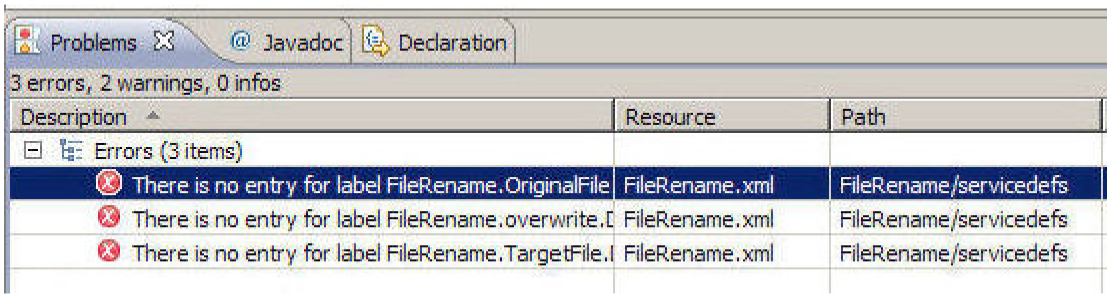
14. Save your changes to the project by selecting **File** > **Save**.

    You will resolve errors when you add language property file entries.

Add Language Property File Entries for the File Rename Service

This task is done on your PC in Eclipse after adding the Overwrite flag, Source File, and Destination File parameters.

After creating the new parameters for the File Rename service, you should see error icons next to each of the new parameters, as shown in the following example.



3 errors, 2 warnings, 0 infos

Description	Resource	Path
Errors (3 items)		
There is no entry for label FileRename.OriginalFile	FileRename.xml	FileRename/servicedefs
There is no entry for label FileRename.overwrite.I	FileRename.xml	FileRename/servicedefs
There is no entry for label FileRename.TargetFile.I	FileRename.xml	FileRename/servicedefs

The errors occur because the File Rename service language property file does not contain entries for these parameters yet. Using the Quick Fix option in Eclipse, you will add the necessary entries to the property file.

Complete the following steps for each error associated with the File Rename service parameters (Overwrite flag, Source File, and Destination File).

1. Open and expand the FileRename SDK project in the Package Explorer.
2. Expand the **servicedefs** node. Right-click **FileRename.xml** and select **Open With** > **Service Definition Editor**.
3. Select **Window** > **Show View** > **Problems**.

   An Error Log view (tab) displays.

   Tip: If Problems is not visible, select `Window > Other > General > Problems`.

4. Right-click on the error message for one of the three parameters in the Problems view and select **Quick Fix**.
5. Under Select a fix, ensure that Append a new property to the language property file is selected, and click **OK**.

   A new line for the parameter is added to the language property file for the FileRename service.

6. Repeat for the remaining errors.
7. Save your changes to the project by selecting **File** > **Save**.

   The Problems view refreshes, removing the error entries.

Build the Service Package

This task is done on your PC using the MESA Studio SDK in Eclipse.

1. Open the FileRename SDK package in the Package Explorer.
2. Right-click on the name of the package and select **Export**.
3. In the Export window, expand the MESA Studio node, select **Service Packages**, and click **Next**.
4. In the Export Service Package window, type or browse to the location on your PC where the service package should be created. Click **Finish.**
5. Open Windows Explorer and browse to the location to verify that the service package is there.

   For this example, the path and filename are:

   drive:\your_chosen_folder\FileRename\dist\FileRename_3000.0.0.jar

Create the Text File to be Renamed

You must be able to connect to the server where the application resides.

This task is done on your PC and on the server where the application resides.

1. On your PC, use a text editor to create a simple text file. Save the file as `test.class`.

   This is the file that will be renamed by your service.

2. Copy the text file to a location on the server where it is accessible to the application. Make note of the location.

   In a later task, you will create a business process that will use this file.

Install the File Rename Service into the Application

You must be able to connect to the server where the application resides.

This task is done using your PC and the server where the application resides.

After you create a service and package the source code, you must install the service package into the application. This places the .jar file on the server where the application expects to find it, and expands it. The next time the server is restarted, the service will be in effect.

Note: Before you install the service package into a production environment, install and test it in a test environment.

1. Stop the application instance.
2. In Eclipse, open the MESA Studio perspective.
3. Select the application instance where the package should be installed.
4. With the instance selected, right-click and choose **Install Service Package**.
5. On the Install Service Package window, browse to the location of the FileRename package file and click **Open**.
6. Click **Finish** to begin installing the service.

   This may take several minutes, depending on your system and network load.

7. Once the service installation is complete, start the application instance.
8. Log in to the application and ensure that the service can be viewed and configured from the options on the **Deployment > Services > Configuration** screen.

Configure the File Rename Service Instance

This task is done using the application.

Installing the service in the previous task created a FileRename service type. In this task, you create a new instance of that service type.

Tip: As a best practice, create and configure a new instance instead of editing the base configuration for your new service type.

1. Log on to the application.
2. Select **Deployment > Services > Configuration**.
3. Click **Go!** for Create New Service.
4. Select FileRename as the Service Type to configure and click **Next**.
5. Type `FileRename` in the Service name field and enter a description for this instance.

   Do not select a group.
6. Set the overwrite parameter to true.
7. Click **Confirm** to save the service configuration.

Create Test BPML

1. Using a text editor (or a BPML editor in your application), create and save a test file containing the following:

   ```
   <process name="lzTest">
     <sequence name="main">
       <operation name="FileRename">
         <participant name="lzFileRename"/>
         <output message="fnoutput">
           <assign to="sourceFile">c:\temp\test.class</assign>
           <assign to="destinationFile">c:\temp\test_fn.class</assign>
           <assign to="." from="*"></assign>
         </output>
         <input message="input">
           <assign to="." from="*"></assign>
         </input>
       </operation>
     </sequence>
   </process>
   ```
2. Accept defaults for any other editor parameters such as process levels or deadline settings.
3. Check the test file into the application.

Test the File Rename Service

Testing on custom services and adapters is generally done by encapsulating most processes in Java classes and have the custom service or adapter call them.

For the file rename example, use the your test BPML and the text file you created earlier to test your service in the application.

1. For the first test, execute the business process against the test file using the configuration you created.

   Status should be `Success` and your c:\temp directory should contain the test_fn.class file.
2. Re-run the test.

Status will be `Error` with the message `Source is not a file or accessable or doesn't exist: c:\temp\test.class`. This indicates that, after the previous test, the test.class file no longer exists after the first class and is not available to be renamed.

Additional Examples

Example 2: Basic Adapter

This example shows basic adapter operation, including:

- An example of firing off a thread
- Sleeping
- Registering activity information to show engine taking workflow off queue while waiting for an activity to be completed
- Updating activity signals to engine that activity is complete and can be put back in the queue

The files in this example can be substituted for the Java files in Eclipse.

TestAdapterImpl.java

This code invokes the desired methods of the test adapter interface and extends the generated service code by adding business logic to the big A portion.

```
package com.mypackage;

import java.util.*;
import java.sql.*;
import javax.naming.*;
import java.rmi.RemoteException;

import com.sterlingcommerce.woodstock.services.*;
import com.sterlingcommerce.woodstock.workflow.*;
import com.sterlingcommerce.woodstock.services.controller.ServicesControllerImpl;
import com.sterlingcommerce.woodstock.util.frame.log.*;

public class  TestAdapterImpl implements IService {

    public TestAdapterImpl() {}

    public WorkFlowContext processData(WorkFlowContext wfc)
        throws WorkFlowException , RemoteException{
```

```
        boolean error = false;
        //Logger logger = LogService.getLogger("TestAdapterImpl");
        wfc.harnessRegister();
        wfc.setBasicStatus(WFCBase.SUCCESS);
        try {
            String tmp= wfc.getParm("SLEEP_INTERVAL");
            int sleep_interval = 1;
            if (tmp !=null ) {
                try {
                    sleep_interval=(new Integer(tmp)).intValue();
            } catch (Exception e) {
                    //logger.logException(e);
                }
            }
            TestAdapterServer rmi = (TestAdapterServer) wfc.getAdapter
                                    (wfc.getServiceName(), wfc);
            rmi.sleepForAwhile(sleep_interval, wfc);
            wfc.setAdvancedStatus("rmi running "+tmp);
            wfc.setWFStatusRpt("Status_Report"," start rmi to sleep for
                    " + sleep_interval + " seconds\n");
        } catch (InterruptedException ie) {
            error=true;
            //logger.logException(ie);
            throw new WorkFlowException(ie);
        } catch (Exception e) {
            error=true;
            //logger.logException(e);
            throw new WorkFlowException(e);
        } finally {
            if (!error ) {
                wfc.setBasicStatus(WFCBase.WAITING_ON_IO);
            }
            if (error) {
                wfc.setBasicStatus(WFCBase.ERROR);
            }
            wfc.unregisterThread();
        }
        return wfc;
    }
}
```

TestAdapterServer.java

Method interfaces are declared here and follow a remote interface structure.

```
package com.mypackage;

import java.rmi.RemoteException;
import java.util.*;

import com.sterlingcommerce.woodstock.services.IAdapterRMI;
//import com.sterlingcommerce.woodstock.services.IAdapterImpl;
import com.sterlingcommerce.woodstock.workflow.*;
//import com.sterlingcommerce.woodstock.services.controller.*;
```

```
/**
 * Remote interface of the TestAdapter.
 */
public interface TestAdapterServer extends IAdapterRMI{
    public void sleepForAwhile(int interval, WorkFlowContext wfc)
       throws InterruptedException ,WorkFlowException, RemoteException,Exception;


    public void refreshAdapter(Properties p) throws RemoteException;

    public String message(String msg) throws RemoteException;
}
```

TestAdapterServerImpl.java

TestAdapterServerImpl.java implements methods from TestAdapterServer.

```
package com.mypackage;

import java.rmi.RemoteException;
import java.util.*;

import com.sterlingcommerce.woodstock.util.frame.log.LogService;
import com.sterlingcommerce.woodstock.services.IAdapterRMI;
import com.sterlingcommerce.woodstock.services.IAdapterImpl;
import com.sterlingcommerce.woodstock.workflow.*;
import com.sterlingcommerce.woodstock.services.controller.*;

/*
 * The implementation class of the adapter
 */
public class TestAdapterServerImpl extends IAdapterImpl implements
        TestAdapterServer {

    private static final String CLASS_NAME = "TestAdapterServerImpl";

    public TestAdapterServerImpl() {}

    /**
     * Implementation from TestAdapterServer.
     *
     * @param int How long to sleep (millis).
     * @Exception Catch all for exceptions that may occur.
     * @return void
     */
    public void sleepForAwhile(int interval,
                               WorkFlowContext wfc) throws Exception {
        /* If threading is not required or desirable, the contents of the run
           method of the TestAdapterThread can be inlined at this point.
         */
        TestAdapterThread testAdapterThread = new TestAdapterThread
                (interval, wfc);
        Thread thread = new Thread(testAdapterThread);
        thread.start();
    }

    public void refreshAdapter(Properties adapterProperties) {
        /* A shutdown and startup can be called here if this adapter should
           support alterations of properties without requiring a restart.
           These two methods must be overloaded and implemented in this
           class to do the proper work.
        */
    }

    public String message(String msg) {
        return msg;
    }
}
```

TestAdapterThread.java

TestAdapterThread.java is added as another class to the package. It contains functionality for firing off a thread, sleep, registering activity information to show the engine taken off queue while waiting for an activitiy to be completed, and update activity that signals to engine that activity is complete and can be put back in the queue.

Tip: To add the class quickly, drop the file into your workspace folder and refresh the project with **File** > **Refresh** (F5).

```
package com.mypackage;

import java.rmi.RemoteException;
import java.util.*;
import java.sql.*;

import com.sterlingcommerce.woodstock.workflow.*;
import com.sterlingcommerce.woodstock.util.frame.*;
import com.sterlingcommerce.woodstock.util.frame.jdbc.*;
import com.sterlingcommerce.woodstock.util.frame.log.*;
import com.sterlingcommerce.woodstock.services.controller.*;

public class TestAdapterThread implements Runnable {

    private int interval = 0;
    private WorkFlowContext wfc = null ;
    private Exception exception = null;

    public  TestAdapterThread(int interval, WorkFlowContext wfc) {
        this.interval = interval;
        this.wfc = wfc;
    }

    public void run() {
        ActivityData activityData = null;
        //Logger logger = LogService.getLogger("TestAdapter");
        Thread.currentThread().setName("The TestAdapter little a");
        try {
            activityData = wfc.registerActivity(wfc.getServiceName(),
                   "RMI is running",null,"going to sleep for a bit");
            Thread.currentThread().sleep(interval);
            activityData.setProgressData
                   ("RMI done sleeping for :" + interval + " seconds");
            activityData.setModTime(System.currentTimeMillis());
            wfc.updateActivity(activityData);
        } catch (Exception e) {
            //logger.logException(e);
        } finally {
            if(activityData != null) {
                wfc.unregisterActivity(activityData);
            }
        }
    }
}
```

Example 3: Bootstrap Adapter

This is example code that shows basic structure and functions for a bootstrap adapter (can launch a business process).

TestBootstrapAdapter.java

```
package com.mypackage;

import java.util.*;
import java.sql.*;
import javax.naming.*;
import java.rmi.RemoteException;

import com.sterlingcommerce.woodstock.services.*;
import com.sterlingcommerce.woodstock.workflow.WorkFlowException;
import com.sterlingcommerce.woodstock.workflow.WorkFlowContext;
import com.sterlingcommerce.woodstock.workflow.*;
import com.sterlingcommerce.woodstock.util.frame.log.*;

/**
 * Since this TestBootstrapAdapter is a bootstrap adapter, it cannot be
 * invoked from a Business Process.  Therefore it will throw an
 * exception if any BP makes this attempt.
 *
 * NOTE: This is a runtime restriction only and not something that is
 * constrained at the BPML validation level.
 */
public class  TestBootstrapAdapter implements IService {
    private final static String errorMessage = "TestBootstrapAdapter:
            Should not call the TestBootstrapAdapter directly from
            Business Process";
    private static final Logger logger = LogService.getLogger
            ("TestBootstrapAdapterLogger");

    public TestBootstrapAdapter() {}

    /**
     * This method is not implemented.
     * @param WorkFlowContext
     * @exception WorkFlowException if a Business Process tries to call
     *  the TestBootstrapAdapter directly.
     */
    public WorkFlowContext processData(WorkFlowContext wfc)
        throws WorkFlowException , RemoteException {
        logger.logError(errorMessage);
        throw new WorkFlowException(errorMessage);
    }
}
```

TestBootstrapAdapterImpl.java

```

package com.mypackage;

import java.io.*;
import java.rmi.RemoteException;
import java.util.*;

import com.sterlingcommerce.woodstock.services.IAdapterRMI;
import com.sterlingcommerce.woodstock.services.IAdapterImpl;
import com.sterlingcommerce.woodstock.workflow.*;
import com.sterlingcommerce.woodstock.services.controller.*;
import com.sterlingcommerce.woodstock.util.frame.log.*;

/*
 * The implementation class of the adapter
 */
public class TestBootstrapAdapterImpl extends IAdapterImpl implements
    IAdapterRMI {

    private static final String CLASS_NAME = "TestBootstrapAdapterImpl";
    private final Logger logger = LogService.getLogger
            ("TestBootstrapAdapterLogger");

    /*
      Instance variables need to be used with care.  In this example,
      the use of the stored FileWatcher means that this is a stateful adapter
      and the servicedef xml file must indicate this.  Otherwise we will
      only have one instantiation of this object regardless
      of how many service instances exist.
     */
    private FileWatcher fw = null;
    private Thread thread = null;
    private int interval = 60;
    private Properties adapterProperties = null;

    public TestBootstrapAdapterImpl() {}

    public void refreshAdapter(Properties p) {
        shutdownAdapter();
        startupAdapter(p);
    }

    public String message(String msg) {
        return msg;
    }

    /**
     * Start the TestBootstrapAdapter.  This method is called by the
     * ServiceController before the adapter is put into the RMI
     * registry (or JNDI) tree for future RMI requests.
     *
     * @param  Properties  Adapter configurations
     * @return void
     */
    public void startupAdapter(Properties adapterProperties) {
        if (logger.debug) {
            logger.logDebug(CLASS_NAME + ".startupAdapter() - Starting
                    [" + getName() + "]");
        }
        this.adapterProperties = adapterProperties;

        if (adapterProperties == null) {
            String errMsg = CLASS_NAME + ".startupAdapter() - The adapter
                    [" + getName() + "] cannot be started up with null
                    properties.";
            logger.logError(errMsg);
            throw new IllegalArgumentException(errMsg);
        }

        setStatus(STARTINGUP);

        try {
            interval = Integer.parseInt((String) adapterProperties.get
                    ("interval"));
            fw = new FileWatcher((String) adapterProperties.get("directory"),
                    interval, (String) adapterProperties.get("workflowName"),
                    logger);
            thread = new Thread(fw);
            thread.start();
            setStatus(RUNNING);
            logger.logDebug(CLASS_NAME + ".startupAdapter() -
                    [" + getName() + "] with configuration " + this +
                    " was started up successfully and is now running.");
        } catch (Exception e) {
            logger.logException(CLASS_NAME + ".startupAdapter() -
                    Unable to start the adapter [" + getName() + "] with
                    configuration " + this + "] due to an exception:", e);
            setStatus(NOTAVAILABLE);
        }
    }

    /**
     * Shuts down the TestBootstrapAdapter.
     *
     * @return void
     */
    public void shutdownAdapter() {
        String instanceName = this.getName();
        if (logger.debug) {
            logger.logDebug(CLASS_NAME + ".shutdownAdapter() - About to
                    shutdown [" + instanceName + "]");
        }
        setStatus(SHUTTINGDOWN);

        try {
            //Do shutdown work specific to this adapter as applicable.
            fw.stop();
           //The join here is not ideal under all conditions, but suffices here.

            thread.join();
            setStatus(SHUTDOWN);
            logger.logDebug(CLASS_NAME + ".shutdownAdapter() -
                   [" + instanceName + "] was shut down successfully.");
        } catch(Exception e) {
            logger.logException(CLASS_NAME + ".shutdownAdapter()
                   - [" + instanceName + "] failed to shutdown:", e);
            setStatus(COULDNTSTOP);
        }
    }

    /**
     * Returns string describing configuration information if applicable.
     */
    public String toString() {
        return "no configuration information";
    }
}

class FileWatcher implements Runnable {

    private String directory = null;
    private int interval = 60;
    private String workflowName = null;
    private boolean keepRunning = true;
    private boolean stopped = false;
    private Logger logger = null;

    public FileWatcher(String directory, int interval, String workflowName,
           Logger logger) {
        /*
         A real implementation should verify that the directory exists
         and can be accessed.
         */
        this.directory = directory;
        this.interval = interval;
        this.workflowName = workflowName;
        this.logger = logger;
    }

    public void stop() {
        keepRunning = false;
    }
    public boolean isStopped() {
        return stopped;
    }

    public void run() {
        /*
         Look to see if there are any files to collect.  This is a
         simplistic example and not something that is robust
         enough for a production adapter.  There is no duplicate detection
         nor does this example attempt to ensure the deletion is successful.
         The File.delete API is not guaranteed to work every time on all
         platforms.  It is also missing simple validation
         checks and possible border conditions.  This is OK since
         the intent it to show basic structure
         and operation only.  Including everything required to make
         this production ready would distract from the main goal of
         this example.
        */

        File file = null;
        while(keepRunning) {
            try {
                /*
                 Skipping the implementation of the directory listing,
                 and reading in of one or more files.
                 Assuming we have a payload of data at some point
                 the next thing to do is to pass along the payload to
                 a document and fire off a business process if
                 that is the intent.
                 */
                Document document = new Document();
                //Use the streaming APIs to write the payload.
                //Always close the stream.
                InitialWorkFlowContext iwfc = new InitialWorkFlowContext();
                iwfc.setWorkFlowName(workflowName);
                iwfc.putPrimaryDocument(document);
                iwfc.start();
                Thread.currentThread().sleep(interval * 1000);
            } catch(Exception e) {
                logger.logException(e);
            }
        }
        stopped = true;
    }
}
```

Installing MESA Studio

Overview for Installing and Configuring MESA Developer Studio

MESA Developer Studio is an Integrated Development Environment (IDE) that uses Eclipse software plug-ins. Use the MESA Developer Studio to connect with an instance of Sterling Integratorfor resource access and control of operations of the application, change the template that the application uses, and develop custom services - all from within a development environment.

In addition to MESA Developer Studio, the following plug-ins are available:

• MESA Developer Studio SDK – for developing and deploying custom services and adapters.
• MESA Developer Studio Skin Editor – for customizing the look and feel of the application interface.
• Reporting Services – a separately-licensed set of plug-ins used to create fact models and reports for Reporting Services.

Assumptions and Prerequisites for MESA Developer Studio

Read the following assumptions and prerequisites prior to installing MESA Developer Studio on the client:

• Basic knowledge of the application and its architecture.
• Basic knowledge of Eclipse is assumed. For more information, see the Eclipse online help, or go to http://eclipse.org.
• You have the required MESA Developer Studio (and if applicable, Reporting Services) product licenses.

Knowledge prerequisites for using MESA Studio Developer SDK are located in the *Creating Custom Services* section.

Steps to Set Up MESA Developer Studio

Setting up MESA Developer Studio is a multi-step process which should be completed in the order described. The following is a checklist for each stage in the process. The checklist provides an overview of the entire process. Separate instructions for completing each step are included.

1. After you have installed and configured the application, download and install Eclipse version 3.3.x. For more information, see http://eclipse.org/downloads/index.php.
   Note: MESA Developer Studio requires specific plug-in versions. We recommend that you install an instance of Eclipse for MESA Developer Studio development only. If you do use a single installation of

Eclipse with both MESA Developer Studio projects and other development projects, disable plug-ins that report conflicts.

2. Download and install Java 2 SDK Standard Edition 5.0 (JDK 1.5.0_14 or higher) on the same PC that you installed Eclipse. It is important that you have the full JDK and not just the JRE.

   After installation, additional configuration is required.
3. Verify that MESA Developer Studio uses the correct JRE in Eclipse.
4. Start the application WebDAV server (application installations on UNIX or iSeries only).
5. Install the MESA Developer Studio (and if purchased, Reporting Services) Plug-ins.
6. Set up your application instance in MESA Developer Studio.
7. Set up application resources to be used with MESA Developer Studio.

Eclipse Terms

The following Eclipse terms are used in this documentation to describe MESA Developer Studio components:

- Project - All the resources related to a particular implementation reside in a project. It can contain folders, files, and other Eclipse objects.
- Workspace - directory where work is stored.
- Workbench - UI window that contains these elements:
  - Perspective - group of views and editors in a Workbench window that correspond to a certain project.
  - View - visual component within the Workbench and dependent on the perspective that was selected. Used to navigate or display information such as properties or messages.
  - Editors - visual component in the Workbench used to create, change, or browse a resource.

Configure the Java JDK on Your PC

In order for Eclipse to work correctly, you must have Java 2 SDK Standard Edition 5.0 (within 5.x) JDK installed on the same PC where you installed Eclipse. You must have the full JDK installed; the JRE alone is not enough. Ensure that Eclipse is closed when you download and install the JDK. The JDK is available for download from the Sun website: www.sun.com. The 5.0 JDK is located in the product download archive area.

After installing the JDK, you must configure your computer to use it.
To configure your PC for the new JDK:

1. From the Windows Start menu, select **Settings > Control Panel > System**.
2. Click the Advanced tab.
3. Click **Environment Variables**.
4. Under System Variables, click **New**.
5. Complete the following and click **OK**:
   - *Variable Name* - Type `JAVA_HOME`.
   - Variable Value - Type the directory path for the location where you installed the JDK.
6. Click **OK** to exit.

Verify that MESA Developer Studio Uses the Correct JRE

In addition to adding a home directory on Windows for this JDK instance, you must also verify that MESA Developer Studio uses the correct JRE.

To verify the MESA Developer Studio JRE:

1. Open Eclipse.
2. From the Window menu, select **Preferences**.
3. Expand the Java section and select **Installed JREs**. The Installed JREs window appears.
4. If jdk1.5.0_14 is not listed (version should be as listed or higher), click **Add** and go to next step.

   If it is listed, ensure that it is selected and click **OK**. You are ready to use MESA Developer Studio.
5. Complete the following and click **OK** :
   - JRE Type - leave at default.
   - JRE Name - leave blank. This will be automatically populated once you select the JRE home directory.
   - JRE home directory - Browse and select the path to the JDK home directory you defined in the *Configuring the Java JDK on Your PC* section. For example, C:\Program Files\java\jdk1.5.0_14.
   - Default VM Arguments - leave blank.
   - JRE system libraries - this will be automatically populated once you select the JRE home directory.
6. Click **OK** to return to the Preferences window.
7. Select the checkbox for the JDK you just added. This ensures that its libraries are used for building projects.
8. Click **OK** to save your changes and exit the Preferences window.

Start the WebDAV Server

The application uses a WebDAV server to provide MESA Studio with access to the application resources, including MESA Developer Studio plug-in updates. This WebDAV server is automatically installed with the application for use with MESA Developer Studio.

While the WebDAV server starts automatically with the application on Windows, you must start the WebDAV server manually for use with the application on UNIX and on iSeries.

Using the Application WebDAV Server on Windows

The WebDAV server that is used with MESA Developer Studio is implemented as a service, and starts automatically when you start the application (startWindowsService.cmd). When the application is stopped (stopWindowsService.cmd), the WebDAV server, and, if used, MySQL, remain running. This is necessary to start and stop instances of the application from within Eclipse and MESA Developer Studio.

You can stop the WebDAV Server service using the stopWebdavWindowsService.cmd. Also, when the application and WebDAV Server service are running and the WebDAV Server service gets stopped, the application (and MySQL if used) will remain running. The logfilename of the WebDAV Server service is dav.log.

Start the Application WebDAV Server on UNIX

You do not need to have the application running to start the WebDAV server.

Note: You must start the WebDAV server on each application instance you want to work with in MESA Developer Studio.

To start the WebDAV server:

1. Open a UNIX command window on the server where your application instance is installed.
2. Go to *installDir*/install/bin.
3. Start the WebDAV server by executing the runDAVServer.sh command.
4. You are asked to enter your installation password. This is the passphrase you enter when you start the application. You must enter this information only once for each application installation because the password is written permanently to the properties file. This step is optional. However, if you do not enter the password, you will not be able to start and stop application instances from within MESA Developer Studio.
5. After the startup process is complete, the WebDAV port is listed. Make a note of this number.

   Note: The default WebDAV port is the base install port + 46. This port is assigned when you install the application and should not be changed. The WebDAV port number is used when downloading and installing the plug-ins and when adding an application instance to MESA Developer Studio.

Start the Application WebDAV Server (iSeries)

You do not need to have the application running to start the WebDAV server.

Note: You must start the WebDAV server on each application instance you want to work with in MESA Developer Studio.

1. Sign onto iSeries with your application user profile.
2. Submit a batch job by entering the following command:

   ```
   SBMJOB CMD(QSH CMD('umask 002 ; cd <install_dir>/install/bin;
   ./runDAVServer.sh'))
   JOB(SIDAV)
   ```

3. To reduce keying errors at startup, create a command language (CL) program similar to the following example:

   ```
   PGM
   SBMJOB CMD(QSH CMD('umask 002 ; cd <install_dir>/install/bin ; +
   ./runDAVServer.sh'))
   JOB(SIDAV)
   ENDPGM
   ```

   Note: Commands not supported on iSeries using the MESA Studio control editor are Start/Stop GIS, List current processes, List disk usage, and Install 3rd party files. To use these commands, execute them on the command line.

Installing MESA Developer Studio Components

You must download and install MESA Developer Studio Eclipse plug-in components from your application instance. Use this procedure to install Reporting Services plug-ins, as well. Before starting, ensure that:

- Your application instance is up and running.
- You have started the WebDAV server if the instance is on UNIX or iSeries. If your application instance is on Windows, verify that the WebDAV server service is running.

Install New Features

To install MESA Developer Studio:

1. Open Eclipse.
2. Select a default workspace folder location. You can add additional workspace folder locations at any time.
3. From the Eclipse Help menu, select **Software Updates > Find and install**.
4. Select **Search for new features to install**.
5. Click **Next**.
6. Click **New Remote Site**.
7. Complete the following and click **OK**:
   - Name - type a descriptive name for the remote application server.
   - URL - type the server name or IP address, followed by a colon and the WebDAV port number, followed by a slash (/) and the word "eclipse," in this format: *new_serverWebDAVportnumber*/eclipse
8. The Install Sites to Visit window displays. It includes all available sites for new plug-in files to install, including the remote site just added. Select the checkbox to the left of the new site. Clear all other selected checkboxes. Click **Finish**.
9. The system verifies the selected site and displays the results. On the search results page, expand the update site node and select from the following plug-ins, according to your licenses:
   - MESA Studio
   - MESA Developer Studio SDK
   - MESA Developer Studio Skin Editor
   - Reporting Services (automatically selects all three Reporting Services plug-ins: Fact Model Editor, Report Editor, and Report Format Editor)

   Caution:

   - Do not change the default installation path for the plug-ins.
   - If you are selecting Reporting Services, you must also select the MESA Studio plug-in (unless you have already installed it).
10. Click **Next**. Accept the terms of the license and click **Next** again.
11. Click **Finish**.
12. Click **Install All** to accept the feature verification.

    You must restart Eclipse for the changes to take effect.

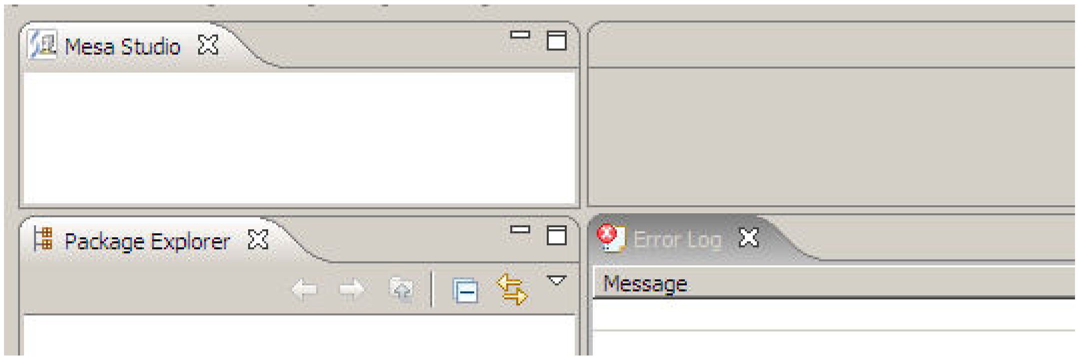
Set Up an Application Instance

To connect to an application instance, you must create a new MESA Studio instance for it. You can use multiple application instances, but each must be configured individually. To configure an application instance, complete the following task.

Note: If you are installing Reporting Services, you must complete this task (either before or after installing Reporting Services).

1. From the Window menu, select **Open Perspective** > **Other**.
2. Select MESA Studio and click **OK**.

   A MESA Studio tab displays.



3. In the MESA Studio view in the upper left, right-click and select **New instance**.
4. Complete the following information and click **Finish**:

   • Hostname - Type the name of the application server.
   • Port - Type the WebDAV port number for the application server.
   • Name - Type a descriptive name for this instance.
   • User name - Type a valid application user name (for example, admin).
   • Password - Type the password for the username entered.

   MESA Studio attempts to establish a connection to the instance using the WebDAV server. The status of the instance is displayed:

   • Red – MESA Studio is unable to connect to the instance; it is possible that the instance has not been started or the WebDAV Server is not running.
   • Yellow – MESA Studio is attempting to connect to the instance. It is possible that the instance was started, but is not yet up and running.
   • Green – The instance is running and MESA Studio has connected to it.

   Note: Refresh the workspace to see new instances or to see if a connection status has changed.

Edit Connection Information

Once you have set up Application instance for use with MESA Developer Studio, you can edit the connection information, view configuration details, test the connection, and refresh the connection.

To edit the connection information:

1. Right-click on the instance name.
2. Select **Edit**.
3. Edit the settings as needed.
4. Click **Finish**. MESA Developer Studio attempts to establish a connection to the instance using the new information. The status is displayed (green, yellow, or red) according to the status of the instance.

View Configuration Details

To view configuration details:

1. Double-click on the instance name.

   Note: The ports on the Overview window are static. Only the ports present at install are displayed. Any changes or additions made after installation are not displayed.

2. Click the **Log tab** to view log information for your application. Log information is used with the File Search Service.

   Caution: If you do not first run the log filter on the server before clicking the Log tab, you will see the following error message: `Error accessing /gisdav/<installDir>/searchResultsDir/searchResults.xml` and the log page will be empty.

Refresh the Instance

Use Refresh if you have locked or unlocked business processes and maps through your application and you want to see their current status in MESA Developer Studio.

To refresh an application instance connection:

1. Right-click on the instance name.
2. Select **Refresh** . The Progress Information window appears and closes automatically when the refresh process is complete. The status is displayed (green, yellow, or red) according to the status of the instance.

Install Additional MESA Developer Studio Components and Updates

You can install additional MESA Developer Studio components not installed at the time of the original installation at any time. To install additional components follow the steps listed in the section *Installing MESA Developer Studio Components*. The system verifies that the license file has newly licensed components and installs them.

The system verifies that the additional components are licensed in your application. If not, you are asked to provide new connection parameters to Application instance that has the appropriate license for the additional MESA Developer Studio components. Once the license check is complete, the new components are activated.

If you are updating an existing component, restart Eclipse in order for the new component to be updated.

Install Reporting Services Plug-Ins

Reporting Services works with Application MESA Developer Studio, which is an Integrated Development Environment (IDE) that uses Eclipse software plug-ins. The Reporting Services Fact Model Editor, Report Editor, and Report Format Editor are all accessed as Eclipse plug-ins.

To set up the Reporting Services plug-ins:

1. Follow the procedures for the MESA Developer Studio configuration.

   Note: When completing the procedure in *Installing MESA Developer Studio Components*, ensure that you select both the Reporting Services plug-ins and the MESA Developer Studio plug-in for download and installation in Eclipse.

   Restriction: The MESA Developer Studio plug-in is a prerequisite for the Reporting Services plug-ins. You must install it either with or prior to installing the Reporting Services plug-ins.

2. After installing the MESA Developer Studio and Reporting Services plug-ins, complete the following tasks:
   a) Start the WebDAV server for your Application instance.
   b) Start the Event Listeners.
   c) Configure your Eclipse installation to point to Application WebDAV server.
   d) Customize the Window Perspective in Eclipse to include Sterling Commerce Reporting Services. This makes the Reporting Services options available directly from the Eclipse menus. In Eclipse, select **Window > Customize Perspective**. In the Shortcuts pane on the left, select **Sterling Commerce Reporting Services** and click **OK**.

Copyright

Index